Java

Table des matières

[Installer Java 9](#_Toc472503963)

[JRE ou JDK ? 9](#_Toc472503964)

[Eclipse 9](#_Toc472503965)

[Premier programme : 10](#_Toc472503966)

[Variables et opérateurs : 10](#_Toc472503967)

[Rappel : 10](#_Toc472503968)

[Les types de variables : 10](#_Toc472503969)

[Quelques calculs : 12](#_Toc472503970)

[Les casts (ou conversion) : 12](#_Toc472503971)

[Depuis Java7, formatage des nombres : 13](#_Toc472503972)

[Lire les entrées claviers : 14](#_Toc472503973)

[Class Scanner : 14](#_Toc472503974)

[Récupérer ce que l’on veut : 14](#_Toc472503975)

[Les conditions : 15](#_Toc472503976)

[If : 15](#_Toc472503977)

[Switch : 16](#_Toc472503978)

[Ternaire : 16](#_Toc472503979)

[Les boucles : 17](#_Toc472503980)

[While : 17](#_Toc472503981)

[Do .. while 18](#_Toc472503982)

[Boucle for : 18](#_Toc472503983)

[Les tableaux : 19](#_Toc472503984)

[Tableaux à une dimension : 19](#_Toc472503985)

[Les tableaux multidimensionnels : 19](#_Toc472503986)

[Rechercher dans un tableau 19](#_Toc472503987)

[Les méthodes de classe : 21](#_Toc472503988)

[Quelques méthodes utiles : 21](#_Toc472503989)

[Créer sa propre méthode : 22](#_Toc472503990)

[La surcharge de méthode : 22](#_Toc472503991)

[Notre première classe : 24](#_Toc472503992)

[Constructeur : 24](#_Toc472503993)

[Les accesseurs et mutateurs : 25](#_Toc472503994)

[This : 26](#_Toc472503995)

[Les variables de classe : 26](#_Toc472503996)

[L’encapsulation : 27](#_Toc472503997)

[L’héritage : 27](#_Toc472503998)

[Le polymorphisme : 29](#_Toc472503999)

[Final : 31](#_Toc472504000)

[La classe object : 31](#_Toc472504001)

[Modéliser ses objets grâce à UML 32](#_Toc472504002)

[Présentation d’UML : 32](#_Toc472504003)

[Modéliser ses objets : 32](#_Toc472504004)

[Modéliser les liens : 33](#_Toc472504005)

[Les packages 34](#_Toc472504006)

[Créer un package : 34](#_Toc472504007)

[Droit d’accès entre les packages : 34](#_Toc472504008)

[Classes abstraites et interface 35](#_Toc472504009)

[Les classes abstraites : 35](#_Toc472504010)

[Les interfaces : 36](#_Toc472504011)

[Le pattern Strategy : 37](#_Toc472504012)

[Les exceptions : 42](#_Toc472504013)

[Try … catch : 42](#_Toc472504014)

[Exceptions personnalisées : 42](#_Toc472504015)

[Gestion de plusieurs exceptions : 44](#_Toc472504016)

[Depuis Java7 : multi catch : 45](#_Toc472504017)

[Les énumérations 46](#_Toc472504018)

[Avant les enum : 46](#_Toc472504019)

[Après les enum : 46](#_Toc472504020)

[Les collections d’objet 48](#_Toc472504021)

[Les différents types de collection : 48](#_Toc472504022)

[Objet List : 48](#_Toc472504023)

[LinkedList : 48](#_Toc472504024)

[L’objet ArrayList : 49](#_Toc472504025)

[Les objets Map : 50](#_Toc472504026)

[Hashtable : 50](#_Toc472504027)

[L’objet HashMap : 51](#_Toc472504028)

[Les objets set : 51](#_Toc472504029)

[HashSet : 51](#_Toc472504030)

[Généricité en Java 53](#_Toc472504031)

[Principe de base : 53](#_Toc472504032)

[Plus loin dans la généricité : 55](#_Toc472504033)

[Généricité et collection : 56](#_Toc472504034)

[Héritage et généricité : 56](#_Toc472504035)

[Les flux entrée/sortie 59](#_Toc472504036)

[Utilisation de Java.io 59](#_Toc472504037)

[L’objet File : 59](#_Toc472504038)

[Les objets FileInputStream et FileOutputStream : 60](#_Toc472504039)

[Les objets FilterInputStream et FilterOutputStream : 61](#_Toc472504040)

[ObjectInputStream / ObjectOutputStream : 65](#_Toc472504041)

[Les objets CharArray(Writer/Reader) et String(Writer/Reader) : 69](#_Toc472504042)

[File(Writer/Reader) et Print(Writer/Reader) : 70](#_Toc472504043)

[Java.nio : 71](#_Toc472504044)

[Depuis Java7 : Java.nio II : 74](#_Toc472504045)

[Le pattern decorator : 77](#_Toc472504046)

[Java et la réflexivité : 79](#_Toc472504047)

[Objet class : 79](#_Toc472504048)

[Connaitre la superclass : 79](#_Toc472504049)

[Liste des interfaces d’une class : 79](#_Toc472504050)

[Liste des méthodes de la class : 80](#_Toc472504051)

[Connaître liste des champs (variable de class ou d’instance) : 81](#_Toc472504052)

[Liste des constructeurs de la class : 81](#_Toc472504053)

[Instancianciation dynamique : 81](#_Toc472504054)

[Notre première fenêtre 84](#_Toc472504055)

[L’objet jFrame : 84](#_Toc472504056)

[Composition d’une jFrame : 86](#_Toc472504057)

[L’objet jPanel : 87](#_Toc472504058)

[Objet graphics et graphics2d : 88](#_Toc472504059)

[L’objet Graphics : 88](#_Toc472504060)

[Méthode drowOval : 89](#_Toc472504061)

[Méthode drawRoundRect() : 89](#_Toc472504062)

[drawLine() 90](#_Toc472504063)

[drawPolygon() 90](#_Toc472504064)

[drawString() 91](#_Toc472504065)

[drawImage() 92](#_Toc472504066)

[Graphics2D : 93](#_Toc472504067)

[Une animation 95](#_Toc472504068)

[Création d’une animation : 95](#_Toc472504069)

[Amélioration : 97](#_Toc472504070)

[Positionner les boutons 99](#_Toc472504071)

[Utiliser la classe jButton 99](#_Toc472504072)

[Positionner les compsants : 99](#_Toc472504073)

[L’objet BorderLayout 100](#_Toc472504074)

[L’objet GridLayout : 100](#_Toc472504075)

[Objet boxLayout : 102](#_Toc472504076)

[L’objet cardLayout : 103](#_Toc472504077)

[L’objet GridBagLayout : 104](#_Toc472504078)

[L’objet FlowLayout : 108](#_Toc472504079)

[Interagir avec un bouton 110](#_Toc472504080)

[Une classe bouton personnalisée : 110](#_Toc472504081)

[InterfaceMouseListener : 111](#_Toc472504082)

[Intéragir avec le bouton : 114](#_Toc472504083)

[Déclencher une action : ActionListener : 114](#_Toc472504084)

[Parler avec sa classe intérieur : 117](#_Toc472504085)

[Les classes anonymes : 119](#_Toc472504086)

[Contrôler l’animation : marche et arrêt : 120](#_Toc472504087)

[Être à l’écoute de ses objets : design pattern observer : 123](#_Toc472504088)

[Des objets qui parlent et qui écoutent : le pattern observer : 123](#_Toc472504089)

[Bouton personnalisé et optimisé 126](#_Toc472504090)

[TP corrigé une calculatrice 128](#_Toc472504091)

[Exécuter des taches simultanément 133](#_Toc472504092)

[Utiliser l’interface Runnable 137](#_Toc472504093)

[Synchroniser ses threads 140](#_Toc472504094)

[Contrôler son animation 141](#_Toc472504095)

[Depuis Java7, le pattern Fork/Join 142](#_Toc472504096)

[Les champs de formulaire 151](#_Toc472504097)

[Les listes, l’objet JComboBox : 151](#_Toc472504098)

[Communiquer avec l’objet : l’interface ItemListener : 152](#_Toc472504099)

[Changer la forme de l’animation : 153](#_Toc472504100)

[Les cases à cocher : objet JCheckBox : 158](#_Toc472504101)

[Première utilisation : 158](#_Toc472504102)

[Un pseudomorphing pour notre animation : 159](#_Toc472504103)

[Objet JRadioButton : 164](#_Toc472504104)

[Les champs de texte : objet JTextField : 165](#_Toc472504105)

[JFormattedTextField : 166](#_Toc472504106)

[Contrôle du clavier, interface KeyListener : 168](#_Toc472504107)

[Les menus et boites de dialogues : 172](#_Toc472504108)

[Les boîtes de dialogues : 172](#_Toc472504109)

[Les boites d’information : 172](#_Toc472504110)

[Les boîtes de confirmation : 174](#_Toc472504111)

[Les boîtes de saisie : 175](#_Toc472504112)

[Boîtes de dialogues personnalisées : 177](#_Toc472504113)

[Les menus 184](#_Toc472504114)

[Les raccourcis clavier : 191](#_Toc472504115)

[Faire un menu contextuel : 192](#_Toc472504116)

[Les barres d’outils : 199](#_Toc472504117)

[Utiliser les actions abstraites : 202](#_Toc472504118)

[TP l’ardoise magique 204](#_Toc472504119)

[Conteneurs, slider et barre de progression 209](#_Toc472504120)

[Autres conteneurs : 209](#_Toc472504121)

[Objet JScrollPane : 211](#_Toc472504122)

[L’objet JTabbedpane : 213](#_Toc472504123)

[JDesktopPane avec des JinternalFrame : 216](#_Toc472504124)

[ObjetJWindow : 217](#_Toc472504125)

[JEditorPane : 218](#_Toc472504126)

[JSlider : 219](#_Toc472504127)

[JProgressBar : 220](#_Toc472504128)

[Enjoliver ses IHM : 221](#_Toc472504129)

[Les arbres et leur structure 223](#_Toc472504130)

[La composition des arbres : 223](#_Toc472504131)

[Des arbres qui parlent : 226](#_Toc472504132)

[Décorer les arbres : 229](#_Toc472504133)

[Modifier le contenu d’un arbre : 232](#_Toc472504134)

[Les interfaces de tableau 238](#_Toc472504135)

[Premier pas : 238](#_Toc472504136)

[Gestion de l’affichage : 239](#_Toc472504137)

[Contrôler l’affichage : 243](#_Toc472504138)

[Interaction avec l’objet JTable : 246](#_Toc472504139)

[Ajouter des lignes et des colonnes : 251](#_Toc472504140)

[MVC 253](#_Toc472504141)

[Premiers pas : 253](#_Toc472504142)

[Le modèle : 254](#_Toc472504143)

[Le contrôleur : 256](#_Toc472504144)

[La vue : 258](#_Toc472504145)

[Le drag’n drop 261](#_Toc472504146)

[Présentation : 261](#_Toc472504147)

[Créer son propre handlerTransfer : 265](#_Toc472504148)

[Activer le drop sur un JTree : 268](#_Toc472504149)

[Mieux gérer les intéractions avec les composants 273](#_Toc472504150)

[Présentation : 273](#_Toc472504151)

[Utiliser l’EDT : 273](#_Toc472504152)

[La classe SwingWorker<T, V> 275](#_Toc472504153)

[JDBC : les BDD 280](#_Toc472504154)

[Préparer la BDD : 280](#_Toc472504155)

[Créer la base de donnée : 281](#_Toc472504156)

[Se connecter à la BDD : 281](#_Toc472504157)

[Connexion : 282](#_Toc472504158)

[Fouiller dans sa BDD 284](#_Toc472504159)

[Le couple Statement – ResultSet : 284](#_Toc472504160)

[Autre exemple : le nom des profs : 287](#_Toc472504161)

[La matière que chaque prof enseigne : 288](#_Toc472504162)

[Dernier exemple, chaque prof + matière pour chaque classe : 289](#_Toc472504163)

[Statement : 290](#_Toc472504164)

[Les requêtes préparées : 291](#_Toc472504165)

[ResultSet : 293](#_Toc472504166)

[Modifier des données : 296](#_Toc472504167)

[Statement, plus loin : 298](#_Toc472504168)

[Gérer les transactions manuellement : 299](#_Toc472504169)

[Limiter le nombre de connexion 302](#_Toc472504170)

[Pourquoi ne se connecter qu’une seule fois : 302](#_Toc472504171)

[Le pattern singleton : 302](#_Toc472504172)

[Le singleton : 304](#_Toc472504173)

[TP testeur de requête 307](#_Toc472504174)

[Lier ses tables avec des objets Java 311](#_Toc472504175)

[Avant toute chose : 311](#_Toc472504176)

[Le pattern DAO : 315](#_Toc472504177)

[Tests : 321](#_Toc472504178)

[Pattern factory : 322](#_Toc472504179)

[Fabriquer nos DAO : 323](#_Toc472504180)

[De l’usine à la multinationale : 325](#_Toc472504181)

# Installer Java

## JRE ou JDK ?

JRE contient tout ce qu’il faut pour exécuter des programmes dans une machine virtuelle Java. JDK contient JDE et permet de développer et compiler des programmes.

Les machines virtuelles Java ne comprennent que le **Byte Code**. C’est le rôle de l’IDE Eclipse que de renvoyer notre programme en Byte Code.

Il y a des environnements pour chaque type de programme :

* J2SE(Java 2 Standard Edition, celui qui nous intéresse dans cet ouvrage) : permet de développer des applications dites « client lourd », par exemple Word, Excel, la suite OpenOffice.org… Toutes ces applications sont des « clients lourds » . C'est ce que nous allons faire dans ce cours
* J2EE(Java 2 Enterprise Edition) : permet de développer des applications web en Java. On parle aussi de clients légers.
* J2ME(Java 2 Micro Edition) : permet de développer des applications pour appareils portables, comme des téléphones portables, des PDA…

## Eclipse

L’interface :

* File: C'est ici que nous pourrons créer de nouveaux projets Java, les enregistrer et les exporter le cas échéant.  
  Les raccourcis à retenir sont :
  + **ALT** + **SHIFT** + **N** : nouveau projet ;
  + **CTRL** + **S** : enregistrer le fichier où l'on est positionné ;
  + **CTRL** + **SHIFT** + **S** : tout sauvegarder ;
  + **CTRL** + **W** : fermer le fichier où l'on est positionné ;
  + **CTRL** + **SHIFT** + **W** : fermer tous les fichiers ouverts.
* Edit: Dans ce menu, nous pourrons utiliser les commandes «copier» , «coller», etc.
* Window: Dans celui-ci, nous pourrons configurer Eclipse selon nos besoins.

## Premier programme :

Rappel : println renvoie à la ligne, print non.

1. \rva insérer un retour chariot, parfois utilisé aussi pour les retours à la ligne ;
2. \tva faire une tabulation.

# Variables et opérateurs :

## Rappel :

| **Raccourcis** | **Traduction** | **Correspondance** |
| --- | --- | --- |
| b | Bit | C'est la plus petite valeur informatique : soit 0 soit 1 |
| o | Octet | regroupement de 8 bits, par exemple : 01011101 |
| Ko | Kilo Octet | regroupement de 1024 octets |
| Mo | Mega Octet | regroupement de 1024 ko |
| Go | Giga Octet | regroupement de 1024 Mo |
| To | Tera Octet | regroupement de 1024 Go |

## Les types de variables :

En Java, nous avons deux types de variables :

1. des variables de type simple ou « primitif » ;
2. des variables de type complexe ou des « objets ».
3. Le type byte (1 octet) peut contenir les entiers entre -128 et +127.
4. byte temperature;
5. temperature = 64;
6. Le type short (2 octets) contient les entiers compris entre -32768 et +32767.
7. short vitesseMax;
8. vitesseMax = 32000;
9. Le type int (4 octets) va de -2\*109 à 2\*109 (2 et 9 zéros derrière… ce qui fait déjà un joli nombre).
10. int temperatureSoleil;
11. temperatureSoleil = 15600000; //La température est exprimée en kelvins
12. Le type long (8 octets) peut aller de −9×1018  à 9×1018 (encore plus gros…).
13. long anneeLumiere;
14. anneeLumiere = 9460700000000000L;
15. Afin d'informer la JVM que le type utilisé est **long**, vous **DEVEZ** ajouter un "**L**" à la fin de votre nombre, sinon le compilateur essaiera d'allouer ce dernier dans une taille d'espace mémoire de type entier et votre code ne compilera pas si votre nombre est trop grand...
16. Le type float (4 octets) est utilisé pour les nombres avec une virgule flottante.
17. float pi;
18. pi = 3.141592653f;
19. Ou encore :
20. float nombre;
21. nombre = 2.0f;
22. Vous remarquerez que nous ne mettons pas une virgule, mais un point ! Et vous remarquerez aussi que même si le nombre en question est rond, on écrit « .0 » derrière celui-ci, le tout suivi de « f ».
23. Le type double (8 octets) est identique à float, si ce n'est qu'il contient plus de chiffres derrière la virgule et qu'il n'a pas de suffixe.
24. double division;
25. division = 0.333333333333333333333333333333333333333333334d;
26. Ici encore, vous devez utiliser une lettre - le « d » - pour parfaire la déclaration de votre variable.

##### Des variables stockant un caractère

1. Le type char contient un caractère stocké entre apostrophes (« ' ' »), comme ceci :
2. char caractere;
3. caractere = 'A';

##### Des variables de type booléen

1. Le type boolean, lui, ne peut contenir que deux valeurs  : true (vrai) ou false (faux), sans guillemets (ces valeurs sont natives dans le langage, il les comprend directement et sait les interpréter).
2. boolean question;
3. question = true;

String est un objet, il comence par une majuscule.

int entier = 32;

float pi = 3.1416f;

char carac = 'z';

String mot = new String("Coucou");

Convention :

* **tous vos noms de classes doivent commencer par une majuscule ;**
* **tous vos noms de variables doivent commencer par une minuscule ;**
* **si le nom d'une variable est composé de plusieurs mots, le premier commence par une minuscule, le ou les autres par une majuscule, et ce, sans séparation ;**
* **tout ceci sans accentuation !**

## Quelques calculs :

int nbre1, nbre2, nbre3; //Déclaration des variables

nbre1 = 1 + 3; //nbre1 vaut 4

nbre2 = 2 \* 6; //nbre2 vaut 12

nbre3 = nbre2 / nbre1; //nbre3 vaut 3

nbre1 = 5 % 2; //nbre1 vaut 1, car 5 = 2 \* 2 + 1

nbre2 = 99 % 8; //nbre2 vaut 3, car 99 = 8 \* 12 + 3

nbre3 = 6 % 3; //là, nbre3 vaut 0, car il n'y a pas de reste

Afficher sa variable :

double nbre1 = 10, nbre2 = 3;

int resultat = (int)(nbre1 / nbre2);

System.out.println("Le résultat est = " + resultat);

## Les casts (ou conversion) :

int i = 123;

float j = (float)i;

D'un type int en double :

int i = 123;

double j = (double)i;

Et inversement :

double i = 1.23;

double j = 2.9999999;

int k = (int)i; //k vaut 1

k = (int)j;

Exemple, on a deux int et on a un résultat double. On aura quand même un resultat tronqué car la JVM fait d’abords le calcule et ensuite fait le cast. On devra donc plutôt faire :

int nbre1 = 3, nbre2 = 2;

double resultat = (double)(nbre1) / (double)(nbre2);

System.out.println("Le résultat est = " + resultat);

//affiche : Le résultat est = 1.5

Cast avec un String :

int i = 12;

String j = new String();

j = j.valueOf(i);

int k = Integer.valueOf(j).intValue();

## Depuis Java7, formatage des nombres :

double nombre = 1000000000000d; // cast en d

//Peut s'écrire ainsi

double nombre = 1\_\_\_\_000\_\_\_\_000\_\_\_000\_000d; // cast en d

//Le nombre d'underscore n'a pas d'importance

//Voici quelques autres exemple d'utilisation

int entier = 32\_000;

double monDouble = 12\_34\_56\_78\_89\_10d; // cast en d

double monDouble2 = 1234\_5678\_8910d; // cast en d

Les underscore doivent être placés entre deux caractères numériques :

double d = 123\_.159;

int entier = \_123;

int entier2 = 123\_;

Avant Java 7, il était possible de déclarer des expressions numériques en hexadécimal, en utilisant le préfixe « 0x » :

int entier = 255; //Peut s'écrire « int entier = 0xFF; »

int entier = 20; //Peut s'écrire « int entier = 0x14; »

int entier = 5112; //Peut s'écrire « int entier = 0x13\_F8; »

Depuis java 7, vous avez aussi la possibilité d'utiliser la notation binaire, en utilisant le préfixe « 0b » :

int entier = 0b1111\_1111; //Est équivalent à : « int entier = 255; »

int entier = 0b1000\_0000\_0000; //Est équivalent à : « int entier = 2048; »

int entier = 0b100000000000; //Est équivalent à : « int entier = 2048; »

# Lire les entrées claviers :

## Class Scanner :

Scanner sc = new Scanner(System.in);

On a besoin d’importer une classe pour le faire fonctionner :

//Ceci importe la classe Scanner du package java.util

import java.util.Scanner;

//Ceci importe toutes les classes du package java.util

import java.util.\*;

## Récupérer ce que l’on veut :

Scanner sc = new Scanner(System.in);

System.out.println("Veuillez saisir un nombre :");

int str = sc.nextInt();

System.out.println("Vous avez saisi le nombre : " + str);

Ou avec un String :

Scanner sc = new Scanner(System.in);

System.out.println("Veuillez saisir un mot :");

String str = sc.nextLine();

System.out.println("Vous avez saisi : " + str);

(Attention, Char n’est pas pris en compte par Scanner 🡺 nextChar() ).

Pour fait l’équivalent, on utilise charAt().

System.out.println("Saisissez une lettre :");

Scanner sc = new Scanner(System.in);

String str = sc.nextLine();

char carac = str.charAt(0);

System.out.println("Vous avez saisi le caractère : " + carac);

Si on doit utiliser plusieurs nextXxx(), il faut penser à vider la ligne avant d’en écrire une autre. Normalement la tête de lecture est placée sur la ligne suivante, mais pas quand on demande plusieurs variables différentes !

import java.util.Scanner;

public class Main {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Saisissez un entier : ");

int i = sc.nextInt();

System.out.println("Saisissez une chaîne : ");

String str = sc.nextLine();

System.out.println("FIN ! ");

}

}

On aura dirrectement « fin » ici. Pour pallier ce problème, il suffit de vider la ligne après les instructions ne le faisant pas automatiquement :

import java.util.Scanner;

public class Main {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Saisissez un entier : ");

int i = sc.nextInt();

System.out.println("Saisissez une chaîne : ");

//On vide la ligne avant d'en lire une autre

sc.nextLine();

String str = sc.nextLine();

System.out.println("FIN ! ");

}

}

# Les conditions :

## If :

int i = 10;

if (i < 0)

System.out.println("le nombre est négatif");

else

System.out.println("le nombre est positif");

Les conditions multiples :

int i = 58;

if(i < 100 && i > 100)

System.out.println("Le nombre est bien dans l'intervalle.");

else

System.out.println("Le nombre n'est pas dans l'intervalle.");

## Switch :

int note = 10; //On imagine que la note maximale est 20

switch (note)

{

case 0:

System.out.println("Ouch !");

break;

case 10:

System.out.println("Vous avez juste la moyenne.");

break;

case 20:

System.out.println("Parfait !");

break;

default:

System.out.println("Il faut davantage travailler.");

}

## Ternaire :

int x = 10, y = 20;

int max = (x < y) ? y : x ; //Maintenant, max vaut 20

* Nous cherchons à affecter une valeur à notre variable max, mais de l'autre côté de l'opérateur d'affectation se trouve une condition ternaire…
* Ce qui se trouve entre les parenthèses est évalué : x est-il plus petit que y ? Donc, deux cas de figure se profilent à l'horizon :
  + si la condition renvoie true (vrai), qu'elle est vérifiée, la valeur qui se trouve après le ?sera affectée ;
  + sinon, la valeur se trouvant après le symbole**:** sera affectée.
* L'affectation est effective : vous pouvez utiliser votre variable max.

# Les boucles :

## While :

//Une variable vide

String prenom;

//On initialise celle-ci à O pour oui

char reponse = 'O';

//Notre objet Scanner, n'oubliez pas l'import de java.util.Scanner !

Scanner sc = new Scanner(System.in);

//Tant que la réponse donnée est égale à oui…

while (reponse == 'O')

{

//On affiche une instruction

System.out.println("Donnez un prénom : ");

//On récupère le prénom saisi

prenom = sc.nextLine();

//On affiche notre phrase avec le prénom

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

//On demande si la personne veut faire un autre essai

System.out.println("Voulez-vous réessayer ? (O/N)");

//On récupère la réponse de l'utilisateur

reponse = sc.nextLine().charAt(0);

}

System.out.println("Au revoir…");

//Fin de la boucle

On peut le forcer à mettre O ou N :

String prenom;

char reponse = 'O';

Scanner sc = new Scanner(System.in);

while (reponse == 'O')

{

System.out.println("Donnez un prénom : ");

prenom = sc.nextLine();

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

//Sans ça, nous n'entrerions pas dans la deuxième boucle

reponse = ' ';

//Tant que la réponse n'est pas O ou N, on repose la question

while(reponse != 'O' && reponse != 'N')

{

//On demande si la personne veut faire un autre essai

System.out.println("Voulez-vous réessayer ? (O/N)");

reponse = sc.nextLine().charAt(0);

}

}

System.out.println("Au revoir…");

Une autre synthaxe :

int a = 1, b = 15;

while (a++ < b)

System.out.println("coucou " +a+ " fois !!");

## Do .. while

String prenom = new String();

//Pas besoin d'initialiser : on entre au moins une fois dans la boucle !

char reponse = ' ';

Scanner sc = new Scanner(System.in);

do{

System.out.println("Donnez un prénom : ");

prenom = sc.nextLine();

System.out.println("Bonjour " +prenom+ ", comment vas-tu ?");

do{

System.out.println("Voulez-vous réessayer ? (O/N)");

reponse = sc.nextLine().charAt(0);

}while(reponse != 'O' && reponse != 'N');

}while (reponse == 'O');

System.out.println("Au revoir…");

Elle s’éxecute au moins une fois. Il y a un point virgule après le while !

## Boucle for :

for(int i = 1; i <= 10; i++)

{

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

}

# Les tableaux :

## Tableaux à une dimension :

int tableauEntier[] = new int[6];

//Ou encore

int[] tableauEntier2 = new int[6];

Ou :

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

double tableauDouble[] = {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};

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

String tableauChaine[] = {"chaine1", "chaine2", "chaine3" , "chaine4"};

## Les tableaux multidimensionnels :

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

## Rechercher dans un tableau

Recherche avec une boucle :

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

for(int i = 0; i < tableauCaractere.length; i++)

{

System.out.println("À l'emplacement " + i +" du tableau nous avons = " + tableauCaractere[i]);

}

Exemple :

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

int i = 0;

char reponse = ' ',carac = ' ';

Scanner sc = new Scanner(System.in);

do {//Boucle principale

do {//On répète cette boucle tant que l'utilisateur n'a pas rentré une lettre figurant dans le tableau

i = 0;

System.out.println("Rentrez une lettre en minuscule, SVP ");

carac = sc.nextLine().charAt(0);

//Boucle de recherche dans le tableau

while(i < tableauCaractere.length && carac != tableauCaractere[i])

i++;

//Si i < 7 c'est que la boucle n'a pas dépassé le nombre de cases du tableau

if (i < tableauCaractere.length)

System.out.println(" La lettre " +carac+ " se trouve bien dans le tableau !");

else //Sinon

System.out.println(" La lettre " +carac+ " ne se trouve pas dans le tableau !");

}while(i >= tableauCaractere.length);

//Tant que la lettre de l'utilisateur ne correspond pas à une lettre du tableau

do{

System.out.println("Voulez-vous essayer à nouveau ? (O/N)");

reponse = sc.nextLine().charAt(0);

}while(reponse != 'N' && reponse != 'O');

}while (reponse == 'O');

System.out.println("Au revoir !");

Tableau avec les String :

String tab[][]={{"toto", "titi", "tutu", "tete", "tata"}, {"1", "2", "3", "4"}};

int i = 0, j = 0;

for(String sousTab[] : tab)

{

i = 0;

for(String str : sousTab)

{

System.out.println("La valeur de la nouvelle boucle est : " + str);

System.out.println("La valeur du tableau à l'indice ["+j+"]["+i+"] est : " + tab[j][i]);

i++;

}

j++;

}

# Les méthodes de classe :

## Quelques méthodes utiles :

toLowerCase() :

String chaine = new String("COUCOU TOUT LE MONDE !"), chaine2 = new String();

chaine2 = chaine.toLowerCase(); //Donne "coucou tout le monde !"

toUpperCase() :

String chaine = new String("coucou coucou"), chaine2 = new String();

chaine2 = chaine.toUpperCase(); //Donne "COUCOU COUCOU"

length() :

String chaine = new String("coucou ! ");

int longueur = 0;

longueur = chaine.length(); //Renvoie 9

equals():

String str1 = new String("coucou"), str2 = new String("toutou");

if (str1.equals(str2))

System.out.println("Les deux chaînes sont identiques !");

else

System.out.println("Les deux chaînes sont différentes !");

La méthode substring() extrait une partie d'une chaîne de caractères. Elle prend deux entiers en arguments : le premier définit le premier caractère (inclus) de la sous-chaîne à extraire, le second correspond au dernier caractère (exclu) à extraire. Là encore, le premier caractère porte le numéro 0.

String chaine = new String("la paix niche"), chaine2 = new String();

chaine2 = chaine.substring(3,13); //Permet d'extraire "paix niche"

La méthode indexOf() explore une chaîne de caractères à la recherche d'une suite donnée de caractères, et renvoie la position (ou l'index) de la sous-chaîne passée en argument. la méthode indexOf() explore à partir du début de la chaîne, lastIndexOf() explore en partant de la fin, mais renvoie l'index à partir du début de la chaîne. Ces deux méthodes prennent un caractère ou une chaîne de caractères comme argument, et renvoient un int. Tout comme charAt() et substring(), le premier caractère porte le numéro 0. Je crois qu'ici, un exemple s'impose, plus encore que pour les autres fonctions :

String mot = new String("anticonstitutionnellement");

int n = 0;

n = mot.indexOf('t'); //n vaut 2

n = mot.lastIndexOf('t'); //n vaut 24

n = mot.indexOf("ti"); //n vaut 2

n = mot.lastIndexOf("ti"); //n vaut 12

n = mot.indexOf('x'); //n vaut -1

Les fonctions mathématiques :

double X = 0.0;

X = Math.random();

//Retourne un nombre aléatoire

//compris entre 0 et 1, comme 0.0001385746329371058

double sin = Math.sin(120); //La fonction sinus

double cos = Math.cos(120); //La fonction cosinus

double tan = Math.tan(120); //La fonction tangente

double abs = Math.abs(-120.25); //La fonction valeur absolue (retourne le nombre sans le signe)

double d = 2;

double exp = Math.pow(d, 2); //La fonction exposant

//Ici, on initialise la variable exp avec la valeur de d élevée au carré

//La méthode pow() prend donc une valeur en premier paramètre, et un exposant en second

## Créer sa propre méthode :

Les méthodes dans la classe main doivent être static :

public class Sdz1

{

public static void main(String[] args)

{

String[] tab = {"toto", "tata", "titi", "tete"};

parcourirTableau(tab);

}

static void parcourirTableau(String[] tabBis)

{

for(String str : tabBis)

System.out.println(str);

}

}

## La surcharge de méthode :

Le principe conssite à ne pas changer de nom de méthode mais à changer ses paramètres :

static void parcourirTableau(String[] tab)

{

for(String str : tab)

System.out.println(str);

}

static void parcourirTableau(int[] tab)

{

for(int str : tab)

System.out.println(str);

}

Vous pouvez faire de même avec les tableaux à deux dimensions. Voici à quoi pourrait ressembler le code d'une telle méthode (je ne rappelle pas le code des deux méthodes ci-dessus) :

static void parcourirTableau(String[][] tab)

{

for(String tab2[] : tab)

{

for(String str : tab2)

System.out.println(str);

}

}

On pourra donc faire, avec ces trois méthodes :

String[] tabStr = {"toto", "titi", "tata"};

int[] tabInt = {1, 2, 3, 4};

String[][] tabStr2 = {{"1", "2", "3", "4"}, {"toto", "titi", "tata"}};

//La méthode avec un tableau de String sera invoquée

parcourirTableau(tabStr);

//La méthode avec un tableau d'int sera invoquée

parcourirTableau(tabInt);

//La méthode avec un tableau de String à deux dimensions sera invoquée

parcourirTableau(tabStr2);

# Notre première classe :

## Constructeur :

public class Ville{

String nomVille;

String nomPays;

int nbreHabitants;

}

Avec le constructeur :

public class Ville{

//Stocke le nom de notre ville

String nomVille;

//Stocke le nom du pays de notre ville

String nomPays;

//Stocke le nombre d'habitants de notre ville

int nbreHabitants;

//Constructeur par défaut

public Ville(){

System.out.println("Création d'une ville !");

nomVille = "Inconnu";

nomPays = "Inconnu";

nbreHabitants = 0;

}

}

Création d’un objet :

public class Sdz1{

public static void main(String[] args){

Ville ville = new Ville();

}

}

Ou avec plusieurs types de constructeurs :

public class Ville {

//Stocke le nom de notre ville

String nomVille;

//Stocke le nom du pays de notre ville

String nomPays;

//Stocke le nombre d'habitants de notre ville

int nbreHabitants;

//Constructeur par défaut

public Ville(){

System.out.println("Création d'une ville !");

nomVille = "Inconnu";

nomPays = "Inconnu";

nbreHabitants = 0;

}

//Constructeur avec paramètres

//J'ai ajouté un « p » en première lettre des paramètres.

//Ce n'est pas une convention, mais ça peut être un bon moyen de les repérer.

public Ville(String pNom, int pNbre, String pPays)

{

System.out.println("Création d'une ville avec des paramètres !");

nomVille = pNom;

nomPays = pPays;

nbreHabitants = pNbre;

}

}

Il faut contrôler les variables de classe, on les met donc en private :

public class Ville {

private String nomVille;

private String nomPays;

private int nbreHabitants;

//…

}

## Les accesseurs et mutateurs :

Ils permettent de lire et modifier les variables de classe :

public class Ville {

//Les variables et les constructeurs n'ont pas changé…

//\*\*\*\*\*\*\*\*\*\*\*\*\* ACCESSEURS \*\*\*\*\*\*\*\*\*\*\*\*\*

//Retourne le nom de la ville

public String getNom() {

return nomVille;

}

//Retourne le nom du pays

public String getNomPays()

{

return nomPays;

}

// Retourne le nombre d'habitants

public int getNombreHabitants()

{

return nbreHabitants;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\* MUTATEURS \*\*\*\*\*\*\*\*\*\*\*\*\*

//Définit le nom de la ville

public void setNom(String pNom)

{

nomVille = pNom;

}

//Définit le nom du pays

public void setNomPays(String pPays)

{

nomPays = pPays;

}

//Définit le nombre d'habitants

public void setNombreHabitants(int nbre)

{

nbreHabitants = nbre;

}

}

## This :

Pour simplifier, this fait référence à l'objet courant ! Bien que la traduction anglaise exacte soit « ceci », il faut comprendre « moi ». À l'intérieur d'un objet, ce mot clé permet de désigner une de ses variables ou une de ses méthodes.

## Les variables de classe :

Les variables de classes sont disponible avec la totalité des classes.

Afin qu'une variable soit une variable de classe, elle doit être précédée du mot clé static. Cela donnerait dans notre classe Ville :

public class Ville {

//Variables publiques qui comptent les instances

public static int nbreInstances = 0;

//Variable privée qui comptera aussi les instances

private static int nbreInstancesBis = 0;

//Les autres variables n'ont pas changé

public Ville(){

//On incrémente nos variables à chaque appel aux constructeurs

nbreInstances++;

nbreInstancesBis++;

//Le reste ne change pas.

}

public Ville(String pNom, int pNbre, String pPays)

{

//On incrémente nos variables à chaque appel aux constructeurs

nbreInstances++;

nbreInstancesBis++;

//Le reste ne change pas

}

public static int getNombreInstancesBis()

{

return nbreInstancesBis;

}

//Le reste du code est le même qu'avant

}

## L’encapsulation :

Voilà, vous venez de construire votre premier objet « maison ». Cependant, sans le savoir, vous avez fait plus que ça : vous avez créé un objet dont les variables sont protégées de l'extérieur. En effet, depuis l'extérieur de la classe, elles ne sont accessibles que via les accesseurs et mutateurs que nous avons défini. C'est le principe d'encapsulation !

# L’héritage :

class Capitale extends Ville {

}

On ne peut pas utiliser les variables de la classe mère si ceux-ci sont protégés, exemple :

public class Capitale extends Ville{

public Capitale(){

this.nomVille = "toto";

}

}

Ici, erreur s’ils sont en private !

Solution : passer de *private* à *protected* :

public class Ville {

public static int nbreInstances = 0;

protected static int nbreInstancesBis = 0;

protected String nomVille;

protected String nomPays;

protected int nbreHabitants;

protected char categorie;

//Tout le reste est identique.

}

On peut accéder aux paramètres de la classe mère, on utilise *super*. Par exemple dans le constructeur :

class Capitale extends Ville {

private String monument;

//Constructeur par défaut

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

}

On peut aussi l’utiliser ainsi : cf polymorphisme :

class Capitale extends Ville {

private String monument;

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

public String decrisToi(){

String str = super.decrisToi() + "\n \t ==>>" + this.monument+ " en est un monument";

System.out.println("Invocation de super.decrisToi()");

return str;

}

}

Au complet, on obtient comme classe fille :

public class Capitale extends Ville {

private String monument;

//Constructeur par défaut

public Capitale(){

//Ce mot clé appelle le constructeur de la classe mère

super();

monument = "aucun";

}

//Constructeur d'initialisation de capitale

public Capitale(String nom, int hab, String pays, String monument){

super(nom, hab, pays);

this.monument = monument;

}

/\*\*

\* Description d'une capitale

\* @return String retourne la description de l'objet

\*/

public String decrisToi(){

String str = super.decrisToi() + "\n \t ==>>" + this.monument + "en est un monument";

return str;

}

/\*\*

\* @return le nom du monument

\*/

public String getMonument() {

return monument;

}

//Définit le nom du monument

public void setMonument(String monument) {

this.monument = monument;

}

}

Ici décritToi est une méthode **polymorphe**.

## Le polymorphisme :

Vous aurez sans doute remarqué que je n'utilise que des objets Ville dans ma boucle : on appelle ceci la **covariance des variables** ! Cela signifie qu'une variable objet peut contenir un objet qui hérite du type de cette variable. Dans notre cas, un objet de typeVillepeut contenir un objet de typeCapitale. Dans ce cas, on dit queVilleest la **superclasse** deCapitale. La covariance est efficace dans le cas où la classe héritant redéfinit certaines méthodes de sa superclasse.

Attention à ne pas confondre la surcharge de méthode avec une méthode polymorphe.

* Une méthode surchargée diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée.
* Une méthode polymorphe a un squelette identique à la méthode de base, mais traite les choses différemment. Cette méthode se trouve dans une autre classe et donc, par extension, dans une autre instance de cette autre classe.

Un exemple :

//Définition d'un tableau de villes null

Ville[] tableau = new Ville[6];

//Définition d'un tableau de noms de Villes et un autre de nombres d'habitants

String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};

int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

//Les trois premiers éléments du tableau seront des Villes

//et le reste des capitales

for(int i = 0; i < 6; i++){

if (i <3){

Ville V = new Ville(tab[i], tab2[i], "france");

tableau[i] = V;

}

else{

Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");

tableau[i] = C;

}

}

//Il ne nous reste plus qu'à décrire tout notre tableau !

for(Object obj : tableau){

System.out.println(obj.toString()+"\n");

}

Une précision s'impose : si vous avez un objetvde typeVille, par exemple, que vous n'avez pas redéfini la méthode toString()et que vous testez ce code :

System.out.println(v);

… vous appellerez automatiquement la méthode toString()de la classe Object! Mais ici, comme vous avez redéfini la méthode toString()dans votre classe Ville, ces deux instructions sont équivalentes :

System.out.println(v.toString());

//Est équivalent à

System.out.println(v);

**Une méthode n’est invocable que si elle est défini dans cet objet**. Exemple :

public class Sdz1 {

public static void main(String[] args){

Ville[] tableau = new Ville[6];

String[] tab = {"Marseille", "lille", "caen", "lyon", "paris", "nantes"};

int[] tab2 = {123456, 78456, 654987, 75832165, 1594, 213};

for(int i = 0; i < 6; i++){

if (i <3){

Ville V = new Ville(tab[i], tab2[i], "france");

tableau[i] = V;

}

else{

Capitale C = new Capitale(tab[i], tab2[i], "france", "la tour Eiffel");

tableau[i] = C;

}

}

//Il ne nous reste plus qu'à décrire tout notre tableau !

for(Object v : tableau){

System.out.println(v.decrisToi()+"\n");

}

}

}

Ne fonctionne pas.

Pour qu'il fonctionne, vous devez dire à la JVM que la référence de type Objectest en fait une référence de type Ville, comme ceci :((Ville)v).decrisToi();. Vous transtypez la référence v en Ville par cette syntaxe. Ici, l'ordre des opérations s'effectue comme ceci :

* Vous transtypez la référence v en Ville;
* Vous appliquez la méthode decrisToi()à la référence appelante, c'est-à-dire, ici, une référence Object changée en Ville.

Il y a deux autres méthodes qui sont très souvent redéfinies :

* public boolean equals(Object o), qui permet de vérifier si un objet est égal à un autre ;
* public int hashCode(), qui attribue un code de hashage à un objet. En gros, elle donne un identifiant à un objet. Notez que cet identifiant sert plus à catégoriser votre objet qu'à l'identifier formellement.

Il faut garder en tête que ce n'est pas parce que deux objets ont un même code de hashage qu'ils sont égaux (en effet, deux objets peuvent avoir la même « catégorie » et être différents…) ; par contre, deux objets égaux ont forcément le même code de hashage ! En fait, la méthodehashcode()est utilisée par certains objets (que nous verrons avec les collections) afin de pouvoir classer les objets entre eux.

Eclipse peut générer equals et hasCode avec Source/Generate hashcode and equals.

## Final :

Une classe déclarée finale est figé, elle ne peut être hérité et ne peut être modifiée. Il en va de même pour les variables.

## La classe object :

Depuis Java7 on peut utiliser :java.util.Objects. Avec on peut utiliser hasCode() et equals() :

public int hashCode() {

return Objects.hash(categorie, nbreHabitants, nomPays, nomVille);

}

Pour l’objet Ville on aurait cette méthode :

public boolean equals(Object obj) {

//On vérifie si les références d'objets sont identiques

if (this == obj)

return true;

//On s'assure que les objets sont du même type, ici de type Ville

if (getClass() != obj.getClass())

return false;

//Maintenant, on compare les attributs de nos objets

Ville other = (Ville) obj;

return Objects.equals(other.getCategorie(), this.getCategorie()) &&

Objects.equals(other.getNom(), this.getNom()) &&

Objects.equals(other.getNombreHabitants(), this.getNombreHabitants()) &&

Objects.equals(other.getNomPays(), this.getNomPays());

}

# Modéliser ses objets grâce à UML

## Présentation d’UML :

UML permet de modéliser les liens entre les objets. Il en existe plusieurs comme :

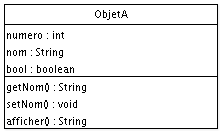
Cependant, il en existe d'autres, comme :

* boUML,
* Together,
* Poseidon,
* Pyut
* etc.

Ou encore <http://argouml.tigris.org> écrit en Java et donc multiplatform. On peut réaliser différents diagrammes comme :

* le diagramme de *use case* (cas d'utilisation) permet de déterminer les différents cas d'utilisation d'un programme informatique ;
* le diagramme de classes ; c'est de celui-là que nous allons nous servir. Il permet de modéliser des classes ainsi que les interactions entre elles ;
* les diagrammes de séquences, eux, permettent de visualiser le déroulement d'une application dans un contexte donné ;
* et d'autres encore…

## Modéliser ses objets :



Voici une classe nommée ObjetA qui a comme attributs :

* numero de type int ;
* nom de type String ;
* bool de type boolean.

Ses méthodes sont :

* getNom() qui retourne une chaîne de caractères ;
* setNom() qui ne renvoie rien ;
* afficher() qui renvoie également une chaîne de caractères.

## Modéliser les liens :



Ici on a un objet dans un autre.

# Les packages

## Créer un package :

Ceux-ci regroupent des classes pour une meilleur lisibilité.

Il existe aussi une convention de nommage pour les packages :

* ceux-ci doivent être écrits entièrement en minuscules ;
* les caractères autorisés sont alphanumériques (de a à z, de 0 à 9) et peuvent contenir des points (.) ;
* tout package doit commencer par *com*, *edu*, *gov*, *mil*, *net*, *org* ou les deux lettres identifiant un pays (ISO Standard 3166, 1981) ; « fr » correspond à la France, « en » correspond à l'Angleterre (pour England)etc.
* aucun mot clé Java ne doit être présent dans le nom, sauf si vous le faites suivre d'un underscore (« \_ »), comme ceci : com.sdz.package\_.

## Droit d’accès entre les packages :

Les classes publics sont accessibles à l’exterieur des packages, les autres ont une portée en « default ».

Exemple, on va créer deux classes :

package com.sdz.test;

class B {

public String str ="";

}

package com.sdz.test;

public class A {

public B b = new B();

}

Vous aurez remarqué que les classes contenues dans un package ont en toute première instruction la déclaration de ce package.

Maintenant que cela est fait, afin de faire le test, créez une classe contenant la méthode main, toujours dans le même package, comme ceci :

package com.sdz.test;

public class Main {

public static void main(String[] args){

A a = new A();

B b = new B();

//Aucun problème ici

}

}

Mais si on met ce code dans com.sdz.2, on aura un problème à l’importation de com.sdz.test.b.

# Classes abstraites et interface

## Les classes abstraites :

C’est une classe normale sauf qu’on ne peut pas l’instancier.

Création d’une classe et d’une méthode abstraite :

abstract class Animal{

abstract void manger(); //Une méthode abstraite

}

On pourra donc faire :

public class Test{

public static void main(String args[]){

Animal loup = new Loup();

Animal chien = new Chien();

loup.manger();

chien.crier();

}

}

Exemple :

public class Test{

public static void main(String[] args){

Object obj = new Loup();

((Loup)obj).manger();

}

}

En revanche, ceci pose problème :

public static void main(String[] args){

Object obj = new Loup();

Loup l = obj; //Problème de référence

}

Eh oui ! Nous essayons de mettre une référence de type Object dans une référence de type Loup : pour avertir la JVM que la référence que vous voulez affecter à votre objet de type Loup est un Loup, vous devez utiliser le transtypage ! Revoyons notre code :

public static void main(String[] args){

Object obj = new Loup();

Loup l = (Loup)obj;

//Vous prévenez la JVM que la référence que vous passez est de type Loup.

}

Exemple pour ce code :

* Nos objets seront probablement tous de couleur et de poids différents. Nos classes auront donc le droit de modifier ceux-ci.
* Ici, nous partons du principe que tous nos animaux mangent de la viande. La méthode manger() sera donc définie dans la classe Animal.
* Idem pour la méthode boire(). Ils boiront tous de l'eau (je vous voyais venir).
* Ils ne crieront pas et ne se déplaceront pas de la même manière. Nous emploierons donc des méthodes polymorphes et déclarerons les méthodes deplacement() et crier() abstraites dans la classe Animal.

## Les interfaces :

Si on veut utiliser notre appli pour un chenil, on ne pourra pas mettre faireCalin() ou autre dans animal parce que c’est presque impossible pour un lion par exemple.

SI on le met directement dans chien :

1. vous allez devoir mettre en place une convention de nommage entre le programmeur qui va utiliser vos objets et vous. Vous ne pourrez pas utiliser la méthode faireCalin(), alors que le programmeur oui ;
2. si vous faites cela, adieu au polymorphisme ! Vous ne pourrez pas appeler vos objets par le biais d'un supertype. Pour pouvoir accéder à ces méthodes, vous devrez obligatoirement passer par une référence à un objet Chien. Pas terrible, tout ça !

On utilise donc les interfaces :

Une interface est 100% abstraite, pas besoin d’écrire abstract :

public interface I{

public void A();

public String B();

}

public interface I2{

public void C();

public String D();

}

Et pour faire en sorte qu'une classe utilise une interface, il suffit d'utiliser le mot clé implements. Ce qui nous donnerait :

public class X implements I{

public void A(){

//…

}

public String B(){

//…

}

}

Exemple :

public static void main(String[] args){

//Avec cette référence, vous pouvez utiliser les méthodes de l'interface I

I var = new X();

//Avec cette référence, vous pouvez utiliser les méthodes de l'interface I2

I2 var2 = new X();

var.A();

var2.C();

}

Utilisation concrète :

public class Test {

public static void main(String[] args) {

//Les méthodes d'un chien

Chien c = new Chien("Gris bleuté", 20);

c.boire();

c.manger();

c.deplacement();

c.crier();

System.out.println(c.toString());

System.out.println("--------------------------------------------");

//Les méthodes de l'interface

c.faireCalin();

c.faireLeBeau();

c.faireLechouille();

System.out.println("--------------------------------------------");

//Utilisons le polymorphisme de notre interface

Rintintin r = new Chien();

r.faireLeBeau();

r.faireCalin();

r.faireLechouille();

}

}

## Le pattern Strategy :

Le fait de toucher à votre hiérarchie peut amener des erreurs indésirables, voire des absurdités : tout cela parce que vous allez changer une structure qui fonctionne à cause de contraintes que l'on vous impose. Pour remédier à ce problème, il existe un concept simple (il s'agit même d'un des fondements de la programmation orientée objet) : **l'encapsulation !**

On va utiliser un pattern design (un patron de conception) pour construire la hiérarchie.

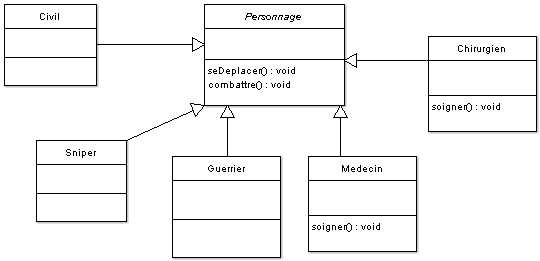
**Exemple :**

On prend le jeux z-army :



Sauf que dans le deuxième vollet on veux se battre aussi avec les médecins, on déplace combattre dans la superclasse.

On doit aussi créer beaucoup d’autres personnages :



Classe de test :

public static void main(String[] args) {

Personnage[] tPers = {new Guerrier(), new Chirurgien(), new Civil(), new Sniper(), new Medecin()};

String[] tArmes = {"pistolet", "pistolet", "couteau", "fusil à pompe", "couteau"};

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

System.out.println("\nInstance de " + tPers[i].getClass().getName());

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

tPers[i].combattre();

tPers[i].setArmes(tArmes[i]);

tPers[i].combattre();

tPers[i].seDeplacer();

tPers[i].soigner();

}

}

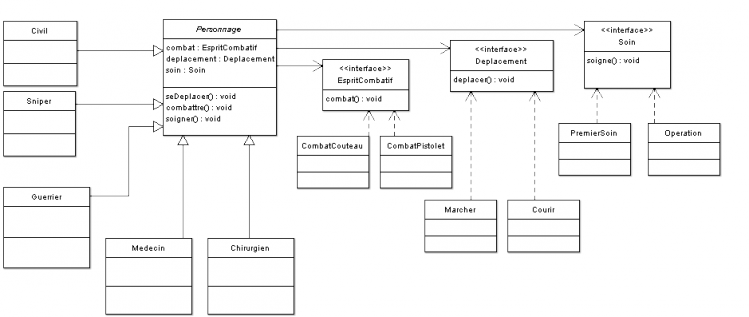
Faisons un point de la situation :

* du code dupliqué s'insinue dans votre code ;
* à chaque modification du comportement de vos personnages, vous êtes obligés de retoucher le code source de la (ou des) classe(s) concernée(s) ;
* votre code perd en « réutilisabilité » et du coup, il n'est pas extensible du tout !

La solution : le pattern strategy :

Le principe de base de ce pattern est le suivant : « isolez ce qui varie dans votre programme et encapsulez-le ! »

Nouveau diagramme :



##### Implémentations de l'interface EspritCombatif

package com.sdz.comportement;

public class Pacifiste implements EspritCombatif {

public void combat() {

System.out.println("Je ne combats pas !");

}

}

package com.sdz.comportement;

public class CombatPistolet implements EspritCombatif{

public void combat() {

System.out.println("Je combats au pitolet !");

}

}

package com.sdz.comportement;

public class CombatCouteau implements EspritCombatif {

public void combat() {

System.out.println("Je me bats au couteau !");

}

}

##### Implémentations de l'interface Deplacement

package com.sdz.comportement;

public class Marcher implements Deplacement {

public void deplacer() {

System.out.println("Je me déplace en marchant.");

}

}

package com.sdz.comportement;

public class Courir implements Deplacement {

public void deplacer() {

System.out.println("Je me déplace en courant.");

}

}

##### Implémentations de l'interface Soin

package com.sdz.comportement;

public class PremierSoin implements Soin {

public void soigne() {

System.out.println("Je donne les premiers soins.");

}

}

package com.sdz.comportement;

public class Operation implements Soin {

public void soigne() {

System.out.println("Je pratique des opérations !");

}

}

package com.sdz.comportement;

public class AucunSoin implements Soin {

public void soigne() {

System.out.println("Je ne donne AUCUN soin !");

}

}

Ainsi la classe personnage :

import com.sdz.comportement.\*;

public abstract class Personnage {

//Nos instances de comportement

protected EspritCombatif espritCombatif = new Pacifiste();

protected Soin soin = new AucunSoin();

protected Deplacement deplacement = new Marcher();

//Constructeur par défaut

public Personnage(){}

//Constructeur avec paramètres

public Personnage(EspritCombatif espritCombatif, Soin soin, Deplacement deplacement) {

this.espritCombatif = espritCombatif;

this.soin = soin;

this.deplacement = deplacement;

}

//Méthode de déplacement de personnage

public void seDeplacer(){

//On utilise les objets de déplacement de façon polymorphe

deplacement.deplacer();

}

// Méthode que les combattants utilisent

public void combattre(){

//On utilise les objets de déplacement de façon polymorphe

espritCombatif.combat();

}

//Méthode de soin

public void soigner(){

//On utilise les objets de déplacement de façon polymorphe

soin.soigne();

}

//Redéfinit le comportement au combat

public void setEspritCombatif(EspritCombatif espritCombatif) {

this.espritCombatif = espritCombatif;

}

//Redéfinit le comportement de Soin

public void setSoin(Soin soin) {

this.soin = soin;

}

//Redéfinit le comportement de déplacement

public void setDeplacement(Deplacement deplacement) {

this.deplacement = deplacement;

}

}

# Les exceptions :

## Try … catch :

Exception est une classe de Java. On peut l’utiliser par exemple, pour les divisons par 0 :

public static void main(String[] args) {

int j = 20, i = 0;

try {

System.out.println(j/i);

} catch (ArithmeticException e) {

System.out.println("Division par zéro !");

}

System.out.println("coucou toi !");

}

Finally est l’action qui sera faite quoi qu’il arrive :

public static void main(String[] args){

try {

System.out.println(" =>" + (1/0));

} catch (ClassCastException e) {

e.printStackTrace();

}

finally{

System.out.println("action faite systématiquement");

}

}

## Exceptions personnalisées :

On peut créer nos propres exceptions, par conventions ils ont Exception comme suffixe !

* throws : ce mot clé permet de signaler à la JVM qu'un morceau de code, une méthode, une classe… est potentiellement dangereux et qu'il faut utiliser un bloc try{…}catch{…}. Il est suivi du nom de la classe qui va gérer l'exception.
* throw : celui-ci permet tout simplement de lever une exception manuellement en instanciant un objet de type Exception (ou un objet hérité). Dans l'exemple de notre ArithmeticException, il y a quelque part dans les méandres de Java un throw new ArithmeticException().

Exemple :

class NombreHabitantException extends Exception{

public NombreHabitantException(){

System.out.println("Vous essayez d'instancier une classe Ville avec un nombre d'habitants négatif !");

}

}

Maintenant qu’on a créé la classe, on peut l’utiliser avec notre constructeur :

public Ville(String pNom, int pNbre, String pPays)

throws NombreHabitantException

{

if(pNbre < 0)

throw new NombreHabitantException();

else

{

nbreInstance++;

nbreInstanceBis++;

nomVille = pNom;

nomPays = pPays;

nbreHabitant = pNbre;

this.setCategorie();

}

}

A partir de maintenant, on ne pourra plus faire new Ville simplement. On devra le mettre entre try…catch dans le main avec notre Exception personnalisée.

**Attention !**

public static void main(String[] args)

{

try {

Ville v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

System.out.println(v.toString());

}

Ne fonctionnera pas parce que v est déclaré dans un sous-bloc et n’est donc défini que dans celui-ci. Il faut donc faire :

public static void main(String[] args)

{

Ville v = null;

try {

v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

System.out.println(v.toString());

}

Mais le programme plantera quand même puisque v sera null :

public static void main(String[] args)

{

Ville v = null;

try {

v = new Ville("Rennes", 12000, "France");

} catch (NombreHabitantException e) { }

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

}

Ici on a résolu le problème.

On peut créer un deuxième construceur dans notre classe Exception pour renseigner le nombre choisi :

public NombreHabitantException(int nbre)

{

System.out.println("Instanciation avec un nombre d'habitants négatif.");

System.out.println("\t => " + nbre);

}

Il suffit maintenant de modifier le constructeur de la classe Ville en conséquence :

public Ville(String pNom, int pNbre, String pPays)

throws NombreHabitantException

{

if(pNbre < 0)

throw new NombreHabitantException(pNbre);

else

{

//Le code est identique à précédemment

}

}

## Gestion de plusieurs exceptions :

Imaginons que l’on veuille lever une exception si le nom de la ville comporte moins de 3 caractères :

public class NomVilleException extends Exception {

public NomVilleException(String message){

super(message);

}

}

Vous avez remarqué que nous avons utilisé super ? Avec cette redéfinition, nous pourrons afficher notre message d'erreur en utilisant la méthode getMessage().

Dans le code suivant, nous ajoutons une condition dans le constructeur Ville :

public Ville(String pNom, int pNbre, String pPays) throws NombreHabitantException, NomVilleException

{

if(pNbre < 0)

throw new NombreHabitantException(pNbre);

if(pNom.length() < 3)

throw new NomVilleException("le nom de la ville est inférieur à 3 caractères ! nom = " + pNom);

else

{

nbreInstance++;

nbreInstanceBis++;

nomVille = pNom;

nomPays = pPays;

nbreHabitant = pNbre;

this.setCategorie();

}

}

Enfin, dans le main :

Ville v = null;

try {

v = new Ville("Re", 12000, "France");

}

//Gestion de l'exception sur le nombre d'habitants

catch (NombreHabitantException e) {

e.printStackTrace();

}

//Gestion de l'exception sur le nom de la ville

catch(NomVilleException e2){

System.out.println(e2.getMessage());

}

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

## Depuis Java7 : multi catch :

On peut séparer deux Exceptions grace à un pipe | :

public static void main(String[] args){

Ville v = null;

try {

v = new Ville("Re", 12000, "France");

}

//Gestion de plusieurs exceptions différentes

catch (NombreHabitantException | NomVilleException e2){

System.out.println(e2.getMessage());

}

finally{

if(v == null)

v = new Ville();

}

System.out.println(v.toString());

}

# Les énumérations

## Avant les enum :

On pouvait faire :

public class AvantEnumeration {

public static final int PARAM1 = 1;

public static final int PARAM2 = 2;

public void fait(int param){

if(param == PARAM1)

System.out.println("Fait à la façon N°1");

if(param == PARAM2)

System.out.println("Fait à la façon N°2");

}

public static void main(String args[]){

AvantEnumeration ae = new AvantEnumeration();

ae.fait(AvantEnumeration.PARAM1);

ae.fait(AvantEnumeration.PARAM2);

ae.fait(4);

}

}

## Après les enum :

Ils se construisent comme une classe mais on écrit enum à la place :

public enum Langage {

JAVA,

C,

CPlus,

PHP;

}

Ils s’utilisent comme une classe static déclarée en public.

public class Main {

public static void main(String args[]){

for(Langage lang : Langage.values()){

if(Langage.JAVA.equals(lang))

System.out.println("J'aime le : " + lang);

else

System.out.println(lang);

}

}

}

Exemple complet :

public enum Langage {

//Objets directement construits

JAVA("Langage JAVA", "Eclipse"),

C ("Lanage C", "Code Block"),

CPlus ("Langage C++", "Visual studio"),

PHP ("Langage PHP", "PS Pad");

private String name = "";

private String editor = "";

//Constructeur

Langage(String name, String editor){

this.name = name;

this.editor = editor;

}

public void getEditor(){

System.out.println("Editeur : " + editor);

}

public String toString(){

return name;

}

public static void main(String args[]){

Langage l1 = Langage.JAVA;

Langage l2 = Langage.PHP;

l1.getEditor();

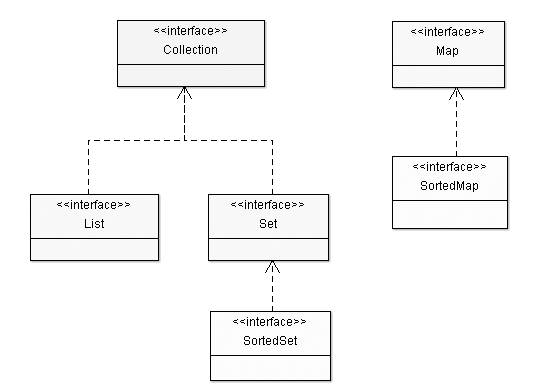
l2.getEditor();

}

}

# Les collections d’objet

## Les différents types de collection :

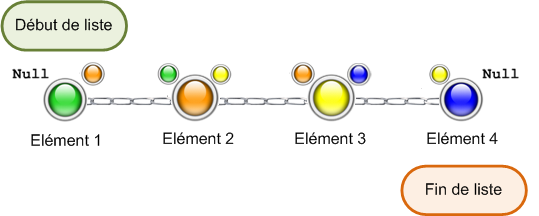


L’objet set est assez restrictif car il n’autorise qu’une seule fois le même objet à l’intérieur.

## Objet List :

On y trouve les objets Vector, ArrayList et LinkedList.

### LinkedList :



LinkedList implémente l’interface Itérator, on peut donc l’utiliser pour lister les éléments de l’objet.

import java.util.LinkedList;

import java.util.List;

import java.util.ListIterator;

public class Test {

public static void main(String[] args) {

List l = new LinkedList();

l.add(12);

l.add("toto ! !");

l.add(12.20f);

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

System.out.println("Élément à l'index " + i + " = " + l.get(i));

System.out.println("\n \tParcours avec un itérateur ");

System.out.println("-----------------------------------");

ListIterator li = l.listIterator();

while(li.hasNext())

System.out.println(li.next());

}

}

### L’objet ArrayList :

Les ArrayList acceptent tout type de donnée mais null. On peut donc y mettre n’importe quoi.

import java.util.ArrayList;

public class Test {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(12);

al.add("Une chaîne de caractères !");

al.add(12.20f);

al.add('d');

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

{

System.out.println("donnée à l'indice " + i + " = " + al.get(i));

}

}

}

Les méthodes des ArrayList :

* add() permet d'ajouter un élément ;
* get(int index) retourne l'élément à l'indice demandé ;
* remove(int index) efface l'entrée à l'indice demandé ;
* isEmpty() renvoie « vrai » si l'objet est vide ;
* removeAll() efface tout le contenu de l'objet ;
* contains(Object element) retourne « vrai » si l'élément passé en paramètre est dans l'ArrayList.

## Les objets Map :

Collection qui fonctionne avec un couple clé valeur. On y trouve les objets

* Hashtable
* HashMap
* TreeMap
* WeakHashMap

### Hashtable :

On parcourt l’objet Hashtable grace à la classe Enumeration :

import java.util.Enumeration;

import java.util.Hashtable;

public class Test {

public static void main(String[] args) {

Hashtable ht = new Hashtable();

ht.put(1, "printemps");

ht.put(10, "été");

ht.put(12, "automne");

ht.put(45, "hiver");

Enumeration e = ht.elements();

while(e.hasMoreElements())

System.out.println(e.nextElement());

}

}

Les méthodes :

* isEmpty() retourne « vrai » si l'objet est vide ;
* contains(Object value) retourne « vrai » si la valeur est présente. Identique à containsValue(Object value) ;
* containsKey(Object key) retourne « vrai » si la clé passée en paramètre est présente dans la Hashtable ;
* put(Object key, Object value) ajoute le couple key - value dans l'objet ;
* elements() retourne une énumération des éléments de l'objet ;
* keys() retourne la liste des clés sous forme d'énumération.

La classe Enumeration a deux methodes :

|  |  |
| --- | --- |
| boolean | [**hasMoreElements**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html#hasMoreElements())()  Tests if this enumeration contains more elements. |
| [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html) | [**nextElement**](https://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html#nextElement())()  Returns the next element of this enumeration if this enumeration object has at least one more element to provide. |

### L’objet HashMap :

Quasi identique sauf qu’il :

* il accepte la valeur null ;
* il n'est pas Thread Safe.

## Les objets set :

Objet qui n’accepte pas les doublons (même null).

On trouve les objets :

* HashSet
* TreeSet
* KinkedHashSet
* …

### HashSet :

Le plus utilisé, on utilise un Iterator :

import java.util.HashSet;

import java.util.Iterator;

public class Test {

public static void main(String[] args) {

HashSet hs = new HashSet();

hs.add("toto");

hs.add(12);

hs.add('d');

Iterator it = hs.iterator();

while(it.hasNext())

System.out.println(it.next());

System.out.println("\nParcours avec un tableau d'objet");

System.out.println("-----------------------------------");

Object[] obj = hs.toArray();

for(Object o : obj)

System.out.println(o);

}

}

Les méthodes :

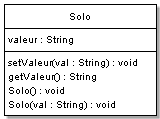
* add() ajoute un élément ;
* contains(Object value) retourne « vrai » si l'objet contient value ;
* isEmpty() retourne « vrai » si l'objet est vide ;
* iterator() renvoie un objet de type Iterator ;
* remove(Object o) retire l'objet o de la collection ;
* toArray() retourne un tableau d'Object.

# Généricité en Java

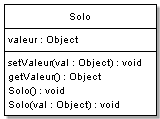
Le principe de la généricité est de faire des classes qui n’acceptent qu’un certain type d’objet ou de donnée de façon dynamique.

## Principe de base :

**Cas d’une classe qui ne l’utilise pas :**



Pour faire une classe qui accepte tout type de donnée, première idée :

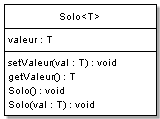


Sauf qu’avec cette configuration on est obligé de faire un cast.

Solo val = new Solo(12);

int nbre = (Integer)val.getValeur();

**Classe avec la généricité :**



Voici le code :

public class Solo<T> {

//Variable d'instance

private T valeur;

//Constructeur par défaut

public Solo(){

this.valeur = null;

}

//Constructeur avec paramètre inconnu pour l'instant

public Solo(T val){

this.valeur = val;

}

//Définit la valeur avec le paramètre

public void setValeur(T val){

this.valeur = val;

}

//Retourne la valeur déjà « castée » par la signature de la méthode !

public T getValeur(){

return this.valeur;

}

}

On définit T à l’instanciation de la classe :

public static void main(String[] args) {

Solo<Integer> val = new Solo<Integer>(12);

int nbre = val.getValeur();

}

Seulement cela ne fonctionne qu’avec le type que l’on a spécifié. Si on écrit Integer on ne pourra plus rentrer de String.

On peut aussi faire :

public static void main(String[] args) {

Solo<Integer> val = new Solo<Integer>();

Solo<String> valS = new Solo<String>("TOTOTOTO");

Solo<Float> valF = new Solo<Float>(12.2f);

Solo<Double> valD = new Solo<Double>(12.202568);

}

Quand on déclare une variable de type primitif, on peut utiliser ses classes enveloppes, aussi appelé WRAPPER. Elles ajoutent automatiquement les méthodes du type (String pas exemple), ainsi que des méthodes pour caster leurs valeurs.

Depuis Java5, Java gère l’AUTOBOXING, qui permet de transformer un type primitif en classe WRAPPER (le boxing) et inversement (unboxing). :

public static void main(String[] args){

int i = new Integer(12); //Est équivalent à int i = 12

double d = new Double(12.2586); //Est équivalent à double d = 12.2586

Double d = 12.0;

Character c = 'C';

al = new ArrayList();

//Avant Java 5 il fallait faire al.add(new Integer(12))

//Depuis Java 5 il suffit de faire

al.add(12);

//…

}

### Plus loin dans la généricité :

On pourrait très bien créer une classe avec deux variables génériques :

public class Duo<T, S> {

//Variable d'instance de type T

private T valeur1;

//Variable d'instance de type S

private S valeur2;

//Constructeur par défaut

public Duo(){

this.valeur1 = null;

this.valeur2 = null;

}

//Constructeur avec paramètres

public Duo(T val1, S val2){

this.valeur1 = val1;

this.valeur2 = val2;

}

//Méthodes d'initialisation des deux valeurs

public void setValeur(T val1, S val2){

this.valeur1 = val1;

this.valeur2 = val2;

}

//Retourne la valeur T

public T getValeur1() {

return valeur1;

}

//Définit la valeur T

public void setValeur1(T valeur1) {

this.valeur1 = valeur1;

}

//Retourne la valeur S

public S getValeur2() {

return valeur2;

}

//Définit la valeur S

public void setValeur2(S valeur2) {

this.valeur2 = valeur2;

}

}

Et ainsi les instancier :

public static void main(String[] args) {

Duo<String, Boolean> dual = new Duo<String, Boolean>("toto", true);

System.out.println("Valeur de l'objet dual : val1 = " + dual.getValeur1() + ", val2 = " + dual.getValeur2());

Duo<Double, Character> dual2 = new Duo<Double, Character>(12.2585, 'C');

System.out.println("Valeur de l'objet dual2 : val1 = " + dual2.getValeur1() + ", val2 = " + dual2.getValeur2());

}

Mais attention à ne pas redéfinir la variable que l’on a déjà instancié. Par exempel duo<Integer,Boolean> val en duo <Integer, Character> val.

## Généricité et collection :

On peut aussi utiliser la généricité sur les objets servant à gérer les collections.

Exemple : Avec une ArrayList on n’est jamais sûr du type sur lequel on va tomber en parcourant le tableau puisqu’il peut tout contenir. On va utiliser le polymorphisme.

public static void main(String[] args) {

System.out.println("Liste de String");

System.out.println("------------------------------");

List<String> listeString= new ArrayList<String>();

listeString.add("Une chaîne");

listeString.add("Une autre");

listeString.add("Encore une autre");

listeString.add("Allez, une dernière");

for(String str : listeString)

System.out.println(str);

System.out.println("\nListe de float");

System.out.println("------------------------------");

List<Float> listeFloat = new ArrayList<Float>();

listeFloat.add(12.25f);

listeFloat.add(15.25f);

listeFloat.add(2.25f);

listeFloat.add(128764.25f);

for(float f : listeFloat)

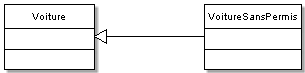
System.out.println(f);

}

Ce sont les mêmes règles que pécédement : pas de float dans une liste de String et invérsement.

### Héritage et généricité :

Imaginons que l’on ait une classe Voiture et une classe VoitureSansPermis qui hérite de Voiture :



public static void main(String[] args) {

List<Voiture> listVoiture = new ArrayList<Voiture>();

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoiture = listVoitureSP; //Interdit !

}

C’est interdit car on aura à un moment une référence de type voitureSansPermis à laquelle on va tenter de lui affecter la référence Voiture.

Une des solutions consiste à utiliser le wildcard : « ? ». Le fait de déclarer une collection avec le wildcard, comme ceci :

ArrayList<?> list;

… revient à indiquer que notre collection accepte n'importe quel type d'objet.

Par exemple, pour définir une liste qui n’accepte qu’un objet de class Voiture ou un de ses sous-classes :

public static void main(String[] args) {

List<? extends Voiture> listVoitureSP = new ArrayList<VoitureSansPermis>();

afficher(listVoitureSP);

}

//Méthode générique !

static void afficher(ArrayList<? extends Voiture> list){

for(Voiture v : list)

System.out.println(v.toString());

}

Mais en utilisant le wildcard, la liste se met en lecture seule, on ne peut plus insérer.

Du coup, ça fonctionne bien en lecture :

public static void main(String[] args){

//Liste de voiture

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

listVoiture.add(new Voiture());

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoitureSP.add(new VoitureSansPermis());

listVoitureSP.add(new VoitureSansPermis());

affiche(listVoiture);

affiche(listVoitureSP);

}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collection de VoitureSansPermis

static void affiche(List<? extends Voiture> list){

for(Voiture v : list)

System.out.print(v.toString());

}

On aura au moins une seule méthode pour lire tout ce qui déscend de Voiture.

Si on remplace extends par super, on autorise n’importe quelle classe superclass de Voiture dont Voiture elle-même :

public static void main(String[] args){

//Liste de voiture

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

listVoiture.add(new Voiture());

List<Object> listVoitureSP = new ArrayList<Object>();

listVoitureSP.add(new Object());

listVoitureSP.add(new Object());

affiche(listVoiture);

}

//Avec cette méthode, on accepte aussi bien les collections de Voiture que les collections d'Object : superclasse de toutes les classes

static void affiche(List<? super Voiture> list){

for(Object v : list)

System.out.print(v.toString());

}

En définitif, le wildcard sert surtout à retrouver le polymorphisme avec les Collections :

import java.util.ArrayList;

import java.util.List;

public class Garage {

List<Voiture> list = new ArrayList<Voiture>();

public void add(List<? extends Voiture> listVoiture){

for(Voiture v : listVoiture)

list.add(v);

System.out.println("Contenu de notre garage :");

for(Voiture v : list)

System.out.print(v.toString());

}

}

Un petit test rapide :

public static void main(String[] args){

List<Voiture> listVoiture = new ArrayList<Voiture>();

listVoiture.add(new Voiture());

List<VoitureSansPermis> listVoitureSP = new ArrayList<VoitureSansPermis>();

listVoitureSP.add(new VoitureSansPermis());

Garage garage = new Garage();

garage.add(listVoiture);

System.out.println("--------------------------");

garage.add(listVoitureSP);

}

# Les flux entrée/sortie

Pour réaliser des entrées sorties, on utilise des stream. Des objets différents pour les deux catégories de flux :

* Les objets travaillant avec des flux d’entrée : (in) pour la lecture.
* Les objets travaillant avec des flux de sortie : (out) pour l’écriture.

## Utilisation de Java.io

### L’objet File :

Avant de commencer on créer un fichier texte à la racine du projet. Exempel test.txt

//Package à importer afin d'utiliser l'objet File

import java.io.File;

public class Main {

public static void main(String[] args) {

//Création de l'objet File

File f = new File("test.txt");

System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());

System.out.println("Nom du fichier : " + f.getName());

System.out.println("Est-ce qu'il existe ? " + f.exists());

System.out.println("Est-ce un répertoire ? " + f.isDirectory());

System.out.println("Est-ce un fichier ? " + f.isFile());

System.out.println("Affichage des lecteurs à la racine du PC : ");

for(File file : f.listRoots())

{

System.out.println(file.getAbsolutePath());

try {

int i = 1;

//On parcourt la liste des fichiers et répertoires

for(File nom : file.listFiles()){

//S'il s'agit d'un dossier, on ajoute un "/"

System.out.print("\t\t" + ((nom.isDirectory()) ? nom.getName()+"/" : nom.getName()));

if((i%4) == 0){

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

}

i++;

}

System.out.println("\n");

} catch (NullPointerException e) {

//L'instruction peut générer une NullPointerException

//s'il n'y a pas de sous-fichier !

}

}

}

}

On peut aussi,

* Supprimer le fichier avec *delete()*
* Ajouter un répertoire avec *mkdir()*

### Les objets FileInputStream et FileOutputStream :

C’est grace à ces deux objets que l’on va pouvoir lire et écrire dans un fichier.

Ces classes héritent des classes abstraites *InputStream* et *OutputStream*.

Exemple pour travailler avec le fichier :

//Packages à importer afin d'utiliser les objets

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

// Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis = null;

FileOutputStream fos = null;

try {

// On instancie nos objets :

// fis va lire le fichier

// fos va écrire dans le nouveau !

fis = new FileInputStream(new File("test.txt"));

fos = new FileOutputStream(new File("test2.txt"));

// On crée un tableau de byte pour indiquer le nombre de bytes lus à

// chaque tour de boucle

byte[] buf = new byte[8];

// On crée une variable de type int pour y affecter le résultat de

// la lecture

// Vaut -1 quand c'est fini

int n = 0;

// Tant que l'affectation dans la variable est possible, on boucle

// Lorsque la lecture du fichier est terminée l'affectation n'est

// plus possible !

// On sort donc de la boucle

while ((n = fis.read(buf)) >= 0) {

// On écrit dans notre deuxième fichier avec l'objet adéquat

fos.write(buf);

// On affiche ce qu'a lu notre boucle au format byte et au

// format char

for (byte bit : buf) {

System.out.print("\t" + bit + "(" + (char) bit + ")");

}

System.out.println("");

//Nous réinitialisons le buffer à vide

//au cas où les derniers byte lus ne soient pas un multiple de 8

//Ceci permet d'avoir un buffer vierge à chaque lecture et ne pas avoir de doublon en fin de fichier

buf = new byte[8];

}

System.out.println("Copie terminée !");

} catch (FileNotFoundException e) {

// Cette exception est levée si l'objet FileInputStream ne trouve

// aucun fichier

e.printStackTrace();

} catch (IOException e) {

// Celle-ci se produit lors d'une erreur d'écriture ou de lecture

e.printStackTrace();

} finally {

// On ferme nos flux de données dans un bloc finally pour s'assurer

// que ces instructions seront exécutées dans tous les cas même si

// une exception est levée !

try {

if (fis != null)

fis.close();

} catch (IOException e) {

e.printStackTrace();

}

try {

if (fos != null)

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

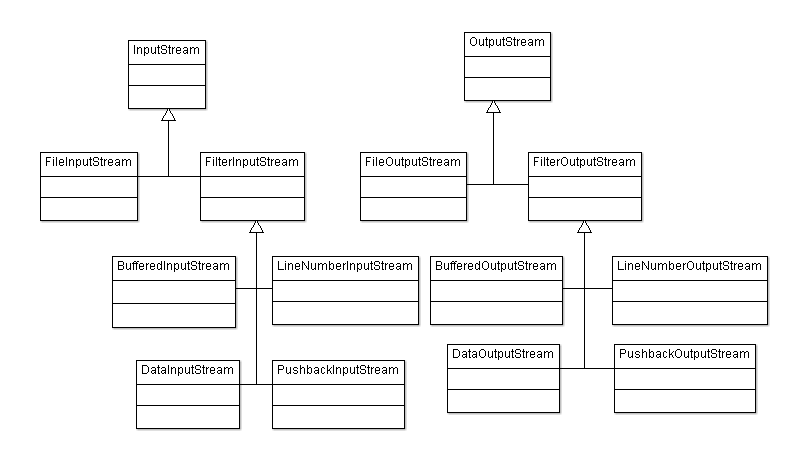
}

}

Il est impossible de lire dans un fichier inexistant, on aura une Exception *FileNotFoundException* par contre on peut écrire dans un fichier inexistant, il sera créé.

* les espaces :SPpour « SPace », code décimal 32 ;
* les sauts de lignes :LFpour « Line Feed », code décimal 13 ;
* les retours chariot :CRpour « Carriage Return », code décimal 10.

### Les objets FilterInputStream et FilterOutputStream :



Ainsi les classes Filter, étant abstraites, définissent un comportement pour leurs classes filles.

* DataInputStream : offre la possibilité de lire directement des types primitifs (double, char, int) grâce à des méthodes comme readDouble(),readInt()…
* BufferedInputStream : cette classe permet d'avoir un tampon à disposition dans la lecture du flux. En gros, les données vont tout d'abord remplir le tampon, et dès que celui-ci est plein, le programme accède aux données.
* PushbackInputStream : permet de remettre un octet déjà lu dans le flux entrant.
* LineNumberInputStream : cette classe offre la possibilité de récupérer le numéro de la ligne lue à un instant T.

Puisque ces classes acceptent une instance de leur superclasse en paramètre, vous pouvez cumuler les filtres et obtenir des choses de ce genre :

FileInputStream fis = new FileInputStream(new File("toto.txt"));

DataInputStream dis = new DataInputStream(fis);

BufferedInputStream bis = new BufferedInputStream(dis);

//Ou en condensé :

BufferedInputStream bis = new BufferredInputStream(

new DataInputStream(

new FileInputStream(

new File("toto.txt"))));

Si on met un autre fichier test, plus gros, on peut comparer le temps d’exécution en milliseconde :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.DataInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis;

BufferedInputStream bis;

try {

fis = new FileInputStream(new File("test.txt"));

bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

byte[] buf = new byte[8];

//On récupère le temps du système

long startTime = System.currentTimeMillis();

//Inutile d'effectuer des traitements dans notre boucle

while(fis.read(buf) != -1);

//On affiche le temps d'exécution

System.out.println("Temps de lecture avec FileInputStream : " + (System.currentTimeMillis() - startTime));

//On réinitialise

startTime = System.currentTimeMillis();

//Inutile d'effectuer des traitements dans notre boucle

while(bis.read(buf) != -1);

//On réaffiche

System.out.println("Temps de lecture avec BufferedInputStream : " + System.currentTimeMillis() - startTime));

//On ferme nos flux de données

fis.close();

bis.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

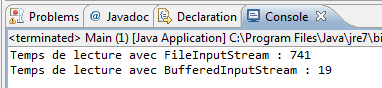
} catch (IOException e) {

e.printStackTrace();

}

}

}



Si on fait le test avec l’écriture :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

FileInputStream fis;

FileOutputStream fos;

BufferedInputStream bis;

BufferedOutputStream bos;

try {

fis = new FileInputStream(new File("test.txt"));

fos = new FileOutputStream(new File("test2.txt"));

bis = new BufferedInputStream(new FileInputStream(new File("test.txt")));

bos = new BufferedOutputStream(new FileOutputStream(new File("test3.txt")));

byte[] buf = new byte[8];

//On récupère le temps du système

long startTime = System.currentTimeMillis();

while(fis.read(buf) != -1){

fos.write(buf);

}

//On affiche le temps d'exécution

System.out.println("Temps de lecture + écriture avec FileInputStream et FileOutputStream : " + (System.currentTimeMillis() - startTime));

//On réinitialise

startTime = System.currentTimeMillis();

while(bis.read(buf) != -1){

bos.write(buf);

}

//On réaffiche

System.out.println("Temps de lecture + écriture avec BufferedInputStream et BufferedOutputStream : " + (System.currentTimeMillis() - startTime));

//On ferme nos flux de données

fis.close();

bis.close();

fos.close();

bos.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

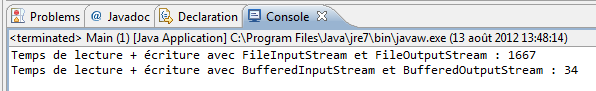
} catch (IOException e) {

e.printStackTrace();

}

}

}



**Les DataInput/OuputStream** :

Fonctionnent comme les Buffered.

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

DataInputStream dis;

DataOutputStream dos;

try {

dos = new DataOutputStream(

new BufferedOutputStream(

new FileOutputStream(

new File("sdz.txt"))));

//Nous allons écrire chaque type primitif

dos.writeBoolean(true);

dos.writeByte(100);

dos.writeChar('C');

dos.writeDouble(12.05);

dos.writeFloat(100.52f);

dos.writeInt(1024);

dos.writeLong(123456789654321L);

dos.writeShort(2);

dos.close();

//On récupère maintenant les données !

dis = new DataInputStream(

new BufferedInputStream(

new FileInputStream(

new File("sdz.txt"))));

System.out.println(dis.readBoolean());

System.out.println(dis.readByte());

System.out.println(dis.readChar());

System.out.println(dis.readDouble());

System.out.println(dis.readFloat());

System.out.println(dis.readInt());

System.out.println(dis.readLong());

System.out.println(dis.readShort());

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Travailler aves des objets :

### ObjectInputStream / ObjectOutputStream :

Ecrire un objet **dans** un fichier = SERIALISATION

Exemple de serialisation :

//Package à importer

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

**L’interface Serializable est une « interface marqueur »**

Rien qu’en implémentant, Java sait que l’on peut serialiser ses objets.

Si une superclass implémente Serializable, ses classes filles seront considérées comme telle aussi.

Exemple :

On va serialiser deux, trois objets, puis les désérialiser pour les réutiliser :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try/catch

ObjectInputStream ois;

ObjectOutputStream oos;

try {

oos = new ObjectOutputStream(

new BufferedOutputStream(

new FileOutputStream(

new File("game.txt"))));

//Nous allons écrire chaque objet Game dans le fichier

oos.writeObject(new Game("Assassin Creed", "Aventure", 45.69));

oos.writeObject(new Game("Tomb Raider", "Plateforme", 23.45));

oos.writeObject(new Game("Tetris", "Stratégie", 2.50));

//Ne pas oublier de fermer le flux !

oos.close();

//On récupère maintenant les données !

ois = new ObjectInputStream(

new BufferedInputStream(

new FileInputStream(

new File("game.txt"))));

try {

System.out.println("Affichage des jeux :");

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

System.out.println(((Game)ois.readObject()).toString());

System.out.println(((Game)ois.readObject()).toString());

System.out.println(((Game)ois.readObject()).toString());

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

ois.close();

} catch (FileNotFoundException e) {

e.printStackTrace();

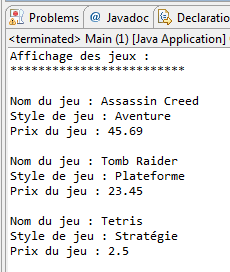
} catch (IOException e) {

e.printStackTrace();

}

}

}



**Et si notre objet contenait un autre objet ?**

Par exemple une notice :

public class Notice {

private String langue ;

public Notice(){

this.langue = "Français";

}

public Notice(String lang){

this.langue = lang;

}

public String toString() {

return "\t Langue de la notice : " + this.langue + "\n";

}

}

Nous allons maintenant implémenter une notice par défaut dans notre objetGame. Voici notre classe modifiée :

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

private Notice notice;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

this.notice = new Notice();

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

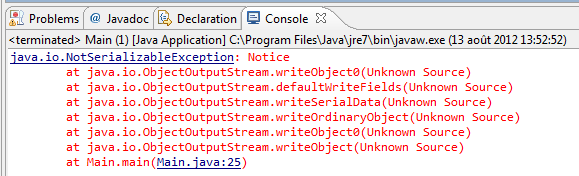
Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

On obitent :



L’erreur est due au fait que notice n’est pas serialisable. Soit :

* On fait en sorte de rendre notice serialisable
* Soit on dit dans *Game* que notice n’est pas à serialiser

Pour la première option on implémente simplement *Serializable*. Pour le second on doit déclarer l’objet notice comme étant *transient* :

import java.io.Serializable;

public class Game implements Serializable{

private String nom, style;

private double prix;

//Maintenant, cette variable ne sera pas sérialisée

//Elle sera tout bonnement ignorée !

private transient Notice notice;

public Game(String nom, String style, double prix) {

this.nom = nom;

this.style = style;

this.prix = prix;

this.notice = new Notice();

}

public String toString(){

return "Nom du jeu : " + this.nom + "\n

Style de jeu : " + this.style + "\n

Prix du jeu : " + this.prix + "\n";

}

}

**Important, on ne met pas la notice dans le toString, car celle-ci est ignorée, on aura donc un *NullPointerException*.**

### Les objets CharArray(Writer/Reader) et String(Writer/Reader) :

Font presque la même chose et ont les mêmes méthodes que leur classe mère. Ils servent juste à écrire un flux de caractère dans un buffer adaptatif.

**CharArray(Writer/Reader) :**

//Packages à importer afin d'utiliser l'objet File

import java.io.CharArrayReader;

import java.io.CharArrayWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

CharArrayWriter caw = new CharArrayWriter();

CharArrayReader car;

try {

caw.write("Coucou les Zéros");

//Appel à la méthode toString de notre objet de manière tacite

System.out.println(caw);

//caw.close() n'a aucun effet sur le flux

//Seul caw.reset() peut tout effacer

caw.close();

//On passe un tableau de caractères à l'objet qui va lire le tampon

car = new CharArrayReader(caw.toCharArray());

int i;

//On remet tous les caractères lus dans un String

String str = "";

while(( i = car.read()) != -1)

str += (char) i;

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

}

}

}

**String(Writer/Reader) :**

//Packages à importer afin d'utiliser l'objet File

import java.io.IOException;

import java.io.StringReader;

import java.io.StringWriter;

public class Main {

public static void main(String[] args) {

StringWriter sw = new StringWriter();

StringReader sr;

try {

sw.write("Coucou les Zéros");

//Appel à la méthode toString de notre objet de manière tacite

System.out.println(sw);

//caw.close() n'a aucun effet sur le flux

//Seul caw.reset() peut tout effacer

sw.close();

//On passe un tableau de caractères à l'objet qui va lire le tampon

sr = new StringReader(sw.toString());

int i ;

//On remet tous les caractères lus dans un String

String str = "";

while(( i = sr.read()) != -1)

str += (char) i;

System.out.println(str);

} catch (IOException e) {

e.printStackTrace();

}

}

}

### File(Writer/Reader) et Print(Writer/Reader) :

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

File file = new File("testFileWriter.txt");

FileWriter fw;

FileReader fr;

try {

//Création de l'objet

fw = new FileWriter(file);

String str = "Bonjour à tous, amis Zéros !\n";

str += "\tComment allez-vous ? \n";

//On écrit la chaîne

fw.write(str);

//On ferme le flux

fw.close();

//Création de l'objet de lecture

fr = new FileReader(file);

str = "";

int i = 0;

//Lecture des données

while((i = fr.read()) != -1)

str += (char)i;

//Affichage

System.out.println(str);

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

}

}

**Java.io existe depuis le jdk 1.1, alors que Java.nio est récent.**

## Java.nio :

Nio : New input output.

Rappel : java.io traite par octet, alors que java.nio traite des blocs de données 🡺 lecture accelérée !

Tout repose sur deux objets : *channels* et *buffers*

*Channels* : sont des flux, ils travaillent avec un *buffers* dont on définit la taille.

En gros, quand on ouvre un flux vers un fichier, on peut récupérer un canal vers ce fichier. Combiné à un buffer, on lira encore plus vite qu’avec *BufferedReaderStream*.

Si on reprend le gros fichier text :

//Packages à importer afin d'utiliser l'objet File

import java.io.BufferedInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.FileChannel;

public class Main {

public static void main(String[] args) {

FileInputStream fis;

BufferedInputStream bis;

FileChannel fc;

try {

//Création des objets

fis = new FileInputStream(new File("test.txt"));

bis = new BufferedInputStream(fis);

//Démarrage du chrono

long time = System.currentTimeMillis();

//Lecture

while(bis.read() != -1);

//Temps d'exécution

System.out.println("Temps d'exécution avec un buffer conventionnel : " + (System.currentTimeMillis() - time));

//Création d'un nouveau flux de fichier

fis = new FileInputStream(new File("test.txt"));

//On récupère le canal

fc = fis.getChannel();

//On en déduit la taille

int size = (int)fc.size();

//On crée un buffer correspondant à la taille du fichier

ByteBuffer bBuff = ByteBuffer.allocate(size);

//Démarrage du chrono

time = System.currentTimeMillis();

//Démarrage de la lecture

fc.read(bBuff);

//On prépare à la lecture avec l'appel à flip

bBuff.flip();

//Affichage du temps d'exécution

System.out.println("Temps d'exécution avec un nouveau buffer : " + (System.currentTimeMillis() - time));

//Puisque nous avons utilisé un buffer de byte afin de récupérer les données

//Nous pouvons utiliser un tableau de byte

//La méthode array retourne un tableau de byte

byte[] tabByte = bBuff.array();

} catch (FileNotFoundException e) {

e.printStackTrace();

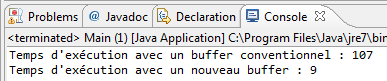
} catch (IOException e) {

e.printStackTrace();

}

}

}



Ce package offre un buffer par type primitif pour la lecture sur le channel, vous trouverez donc ces classes :

* IntBuffer;
* CharBuffer;
* ShortBuffer;
* ByteBuffer;
* DoubleBuffer;
* FloatBuffer;
* LongBuffer.

Pour s’assurer que le flux est bien fermé :

//Packages à importer afin d'utiliser l'objet File

//…

public class Main {

public static void main(String[] args) {

//Nous déclarons nos objets en dehors du bloc try / catch

ObjectInputStream ois;

ObjectOutputStream oos;

try {

//On travaille avec nos objets

} catch (FileNotFoundException e) {

//Gestion des exceptions

} catch (IOException e) {

//Gestion des exceptions

}

finally{

if(ois != null)ois.close();

if(oos != null)oos.close();

}

}

}

On met un test dans finally.

Avec Java7, plusieurs nouveauté, à part la gestion de la mémoire qui est déléguée au *Garbage Collector* (ramasse miette), le reste est à ouvrir et fermer manuellement (flux de donnée, connexion BDD …)

Depuis Java7, Java a aussi initialiser *try-with-ressources* qui ferme automatiquement ce qui a été ouvert dans le try.

try(FileInputStream fis = new FileInputStream("test.txt");

FileOutputStream fos = new FileOutputStream("test2.txt")) {

byte[] buf = new byte[8];

int n = 0;

while((n = fis.read(buf)) >= 0){

fos.write(buf);

for(byte bit : buf)

System.out.print("\t" + bit + "(" + (char)bit + ")");

System.out.println("");

}

System.out.println("Copie terminée !");

} catch (IOException e) {

e.printStackTrace();

}

**A noter que chaque paramètre est séparé d’un ; dans le try !**

Mais si on encapsule :

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"))) {

//…

}

FileInputStream ne sera pas fermer automatiquement, il faut bien penser à les déclarer un à un séparés par un point-virgule.

## Depuis Java7 : Java.nio II :

Nouveau package : java.nio.file, en remplacement à java.io.file :

* une meilleure gestion des exceptions : la plupart des méthodes de la classe File se contentent de renvoyer une valeur nulle en cas de problème, avec ce nouveau package, des exceptions seront levées permettant de mieux cibler la cause du (ou des) problème(s) ;
* un accès complet au système de fichiers (support des liens/liens symboliques, etc.) ;
* l'ajout de méthodes utilitaires tels que le déplacement/la copie de fichier, la lecture/écriture binaire ou texte…
* récupérer la liste des fichiers d'un répertoire via un flux ;
* remplacement de la classe java.io.File par l'interface java.nio.file.Path.

Path path = Paths.get("test.txt");

System.out.println("Chemin absolu du fichier : " + path.toAbsolutePath());

System.out.println("Est-ce qu'il existe ? " + Files.exists(path));

System.out.println("Nom du fichier : " + path.getFileName());

System.out.println("Est-ce un répertoire ? " + Files.isDirectory(path));

La classe Files vous permet aussi de lister le contenu d'un répertoire mais via un objet DirectoryStream qui est un itérateur. Ceci évite de charger tous les fichiers en mémoire pour récupérer leurs informations. Voici comment procéder :

//On récupère maintenant la liste des répertoires dans une collection typée

//Via l'objet FileSystem qui représente le système de fichier de l'OS hébergeant la JVM

Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();

//Maintenant, il ne nous reste plus qu'à parcourir

for(Path chemin : roots){

System.out.println(chemin);

//Pour lister un répertoire, il faut utiliser l'objet DirectoryStream

//L'objet Files permet de créer ce type d'objet afin de pouvoir l'utiliser

try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin)){

int i = 0;

for(Path nom : listing){

System.out.print("\t\t" + ((Files.isDirectory(nom)) ? nom+"/" : nom));

i++;

if(i%4 == 0)System.out.println("\n");

}

} catch (IOException e) {

e.printStackTrace();

}

}

On peut aussi filtrer pour n’afficher qu’il ne liste qu’un certain fichier.

try(DirectoryStream<Path> listing = Files.newDirectoryStream(chemin, "\*.txt")){ … }

//Ne prendra en compte que les fichier ayant l'extension .txt

Avec File :

**Copie de fichier :**

Path source = Paths.get("test.txt");

Path cible = Paths.get("test2.txt");

try {

Files.copy(source, cible, StandardCopyOption.REPLACE\_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

Le troisième argument permet de spécifier les options de copie. Voici celles qui sont disponibles :

* StandardCopyOption.REPLACE\_EXISTING: remplace le fichier cible même s'il existe déjà ;
* StandardCopyOption.COPY\_ATTRIBUTES: copie les attributs du fichier source sur le fichier cible (droits en lecture etc.) ;
* StandardCopyOption.ATOMIC\_MOVE: copie atomique ;
* LinkOption.NOFOLLOW\_LINKS: ne prendra pas en compte les liens.

**Déplacement de fichier :**

Path source = Paths.get("test2.txt");

Path cible = Paths.get("test3.txt");

try {

Files.move(source, cible, StandardCopyOption.REPLACE\_EXISTING);

} catch (IOException e) { e.printStackTrace(); }

Dans le même genre vous avez aussi :

* une méthodeFiles.delete(path)qui supprime un fichier ;
* une méthodeFiles.createFile(path)qui permet de créer un fichier vide.

**Ouvrir des flux :**

:

Path source = Paths.get("test.txt");

//Ouverture en lecture :

try ( InputStream input = Files.newInputStream(source) ) { … }

//Ouverture en écriture :

try ( OutputStream output = Files.newOutputStream(source) ) { … }

//Ouverture d'un Reader en lecture :

try ( BufferedReader reader = Files.newBufferedReader(source, StandardCharsets.UTF\_8) ) { … }

//Ouverture d'un Writer en écriture :

try ( BufferedWriter writer = Files.newBufferedWriter(source, StandardCharsets.UTF\_8) ) { … }

Pour en savoir plus sur ce que permet la nouvelle classe java.nio.file.Files, je vous invite à regarder [la documentation Java](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html).

Java 7 vous permet également de gérer les fichier ZIP grâce à l'objet FileSystem:

// Création d'un système de fichiers en fonction d'un fichier ZIP

try (FileSystem zipFS = FileSystems.newFileSystem(Paths.get("monFichier.zip"), null)) {

//Suppression d'un fichier à l'intérieur du ZIP :

Files.deleteIfExists( zipFS.getPath("test.txt") );

//Création d'un fichier à l'intérieur du ZIP :

Path path = zipFS.getPath("nouveau.txt");

String message = "Hello World !!!";

Files.write(path, message.getBytes());

//Parcours des éléments à l'intérieur du ZIP :

try (DirectoryStream<Path> stream = Files.newDirectoryStream(zipFS.getPath("/"))) {

for (Path entry : stream) {

System.out.println(entry);

}

}

//Copie d'un fichier du disque vers l'archive ZIP :

Files.copy(Paths.get("fichierSurDisque.txt"), zipFS.getPath("fichierDansZIP.txt"));

}

Il est également possible d'être averti *via* l'objetWatchServicelorsqu'un un fichier est modifié, de gérer des entrées/sorties asynchrones via les objets AsynchronousFileChannel, AsynchronousSocketChannel ou AsynchronousServerSocketChannel. Ceci permet de faire les actions en tâche de fond, sans bloquer le code pendant l'exécution. Il est aussi possible d'avoir accès aux attributs grâce à 6 vues permettant de voir plus ou moins d'informations, à savoir :

* BasicFileAttributeViewpermet un accès aux propriétés généralement communes à tous les systèmes de fichiers ;
* DosFileAttributeViewajoute le support des attributs MS-DOS (readonly,hidden,system,archive) à l'objet ci-dessus ;
* PosixFileAttributeViewajoute les permissions POSIX du monde Unix au premier objet cité ;
* FileOwnerAttributeViewpermet de manipuler le propriétaire du fichier ;
* AclFileAttributeViewpermet de manipuler les droits d'accès au fichier ;
* UserDefinedFileAttributeView: permet de définir des attributs personnalisés.

## Le pattern decorator :

Les objets que l’on utilise utilisent des instances de même supertype :

DataInputStream dis = new DataInputStream(

new BufferedInputStream(

new FileInputStream(

new File("sdz.txt"))));

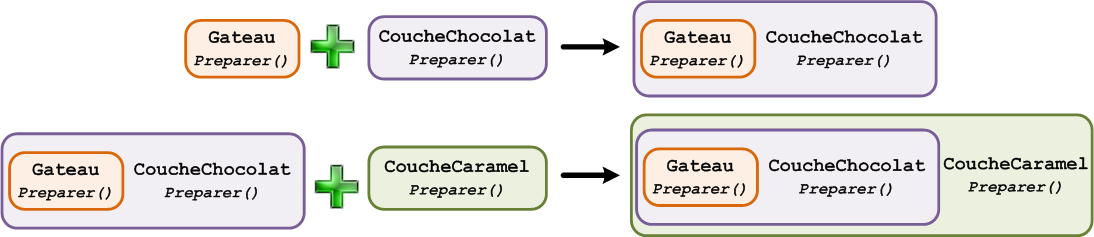
Avant de récupérer des données dans notre objet, elles vont d’abords transiter par des objets passés en paramètre et dans un certain ordre : *pattern decorator*.

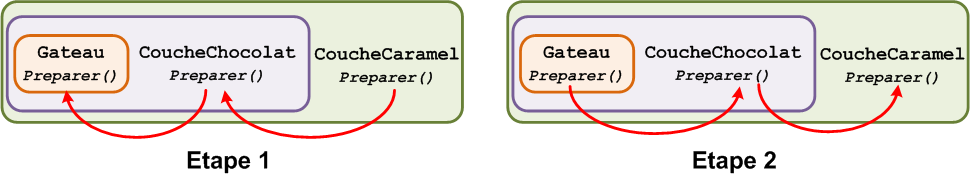
Nous allons procéder de la façon suivante :

* nous allons créer un objetGateau;
* nous allons lui ajouter uneCoucheChocolat;
* nous allons aussi lui ajouter uneCoucheCaramel;
* nous appellerons la méthode qui confectionnera notre gâteau.

Tout cela démarre avec un concept fondamental : l'objet de base et les objets qui le décorent *doivent* être du même type, et ce, toujours pour la même raison, le polymorphisme, le polymorphisme, et le polymorphisme !

En fait, les objets qui vont décorer notre gâteau posséderont la même méthode preparer()que notre objet principal, et nous allons faire fondre cet objet dans les autres. Cela signifie que nos objets qui vont servir de décorateurs comporteront une instance de type Patisserie ; ils vont englober les instances les unes après les autres et du coup, nous pourrons appeler la méthode preparer()de manière récursive !





On obtiendrait au final :

##### Patisserie.java

public abstract class Patisserie {

public abstract String preparer();

}

##### Gateau.java

public class Gateau extends Patisserie{

public String preparer() {

return "Je suis un gâteau et je suis constitué des éléments suivants. \n";

}

}

##### Couche.java

public abstract class Couche extends Patisserie{

protected Patisserie pat;

protected String nom;

public Couche(Patisserie p){

pat = p;

}

public String preparer() {

String str = pat.preparer();

return str + nom;

}

}

##### CoucheChocolat.java

public class CoucheChocolat extends Couche{

public CoucheChocolat(Patisserie p) {

super(p);

this.nom = "\t- Une couche de chocolat.\n";

}

}

##### CoucheCaramel.java

public class CoucheCaramel extends Couche{

public CoucheCaramel(Patisserie p) {

super(p);

this.nom = "\t- Une couche de caramel.\n";

}

}

##### CoucheBiscuit.java

public class CoucheBiscuit extends Couche {

public CoucheBiscuit(Patisserie p) {

super(p);

this.nom = "\t- Une couche de biscuit.\n";

}

}

Et voici un code de test ainsi que son résultat, représenté à la figure suivante.

public class Main{

public static void main(String[] args){

Patisserie pat = new CoucheChocolat(

new CoucheCaramel(

new CoucheBiscuit(

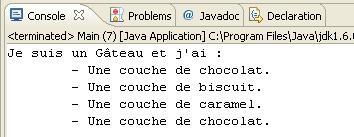
new CoucheChocolat(

new Gateau()))));

System.out.println(pat.preparer());

}

}



# Java et la réflexivité :

Aussi appelé *introspéction* consiste à découvrir de façon dynamique, des informations relatives à une classe ou un objet.

## Objet class :

Quand on crée une classe, la JVM créé un objet class pour chacune d’elle. On peut récupérer des infos sur cette classe.

public static void main(String[] args) {

Class c = String.class;

Class c2 = new String().getClass();

//La fameuse méthode finale dont je vous parlais dans le chapitre sur l'héritage

//Cette méthode vient de la classe Object

}

Maintenant qu’on a récupéré la class :

### Connaitre la superclass :

System.out.println("La superclasse de la classe " + String.class.getName() + " est : " + String.class.getSuperclass());

### Liste des interfaces d’une class :

public static void main(String[] args) {

//On récupère un objet Class

Class c = new String().getClass();

//Class c = String.class; est équivalent

//La méthode getInterfaces retourne un tableau de Class

Class[] faces = c.getInterfaces();

//Pour voir le nombre d'interfaces

System.out.println("Il y a " + faces.length + " interfaces implémentées");

//On parcourt le tableau d'interfaces

for(int i = 0; i < faces.length; i++)

System.out.println(faces[i]);

}

### Liste des méthodes de la class :

public static void main(String[] args) {

Class c = new String().getClass();

Method[] m = c.getMethods();

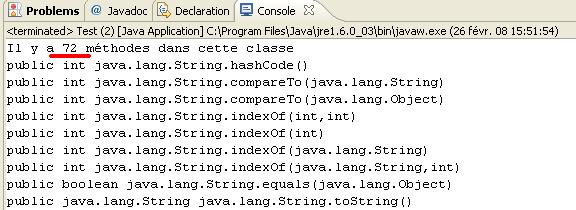
System.out.println("Il y a " + m.length + " méthodes dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

System.out.println(m[i]);

}



Method() a aussi des méthodes intéressantes :

public static void main(String[] args) {

Class c = new String().getClass();

Method[] m = c.getMethods();

System.out.println("Il y a " + m.length + " méthodes dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

{

System.out.println(m[i]);

Class[] p = m[i].getParameterTypes();

for(int j = 0; j < p.length; j++)

System.out.println(p[j].getName());

System.out.println("----------------------------------\n");

}

}

### Connaître liste des champs (variable de class ou d’instance) :

public static void main(String[] args) {

Class c = new String().getClass();

Field[] m = c.getDeclaredFields();

System.out.println("Il y a " + m.length + " champs dans cette classe");

//On parcourt le tableau de méthodes

for(int i = 0; i < m.length; i++)

System.out.println(m[i].getName());

}

Ce qui nous donne :

Il y a 7 champs dans cette classe

value

offset

count

hash

serialVersionUID

serialPersistentFields

CASE\_INSENSITIVE\_ORDER

### Liste des constructeurs de la class :

public static void main(String[] args) {

Class c = new String().getClass();

Constructor[] construc = c.getConstructors();

System.out.println("Il y a " + construc.length + " constructeurs dans cette classe");

//On parcourt le tableau des constructeurs

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

System.out.println(construc[i].getName());

Class[] param = construc[i].getParameterTypes();

for(int j = 0; j < param.length; j++)

System.out.println(param[j]);

System.out.println("-----------------------------\n");

}

}

## Instancianciation dynamique :

Si on crée une nouvelle class :

public class Paire {

private String valeur1, valeur2;

public Paire(){

this.valeur1 = null;

this.valeur2 = null;

System.out.println("Instanciation !");

}

public Paire(String val1, String val2){

this.valeur1 = val1;

this.valeur2 = val2;

System.out.println("Instanciation avec des paramètres !");

}

public String toString(){

return "Je suis un objet qui a pour valeur : " + this.valeur1 + " - " + this.valeur2;

}

public String getValeur1() {

return valeur1;

}

public void setValeur1(String valeur1) {

this.valeur1 = valeur1;

}

public String getValeur2() {

return valeur2;

}

public void setValeur2(String valeur2) {

this.valeur2 = valeur2;

}

}

Le but du jeu est de créer un objet paire sans utiliser l’opérateur new :

public static void main(String[] args) {

String nom = Paire.class.getName();

try {

//On crée un objet Class

Class cl = Class.forName(nom);

//Nouvelle instance de la classe Paire

Object o = cl.newInstance();

//On crée les paramètres du constructeur

Class[] types = new Class[]{String.class, String.class};

//On récupère le constructeur avec les deux paramètres

Constructor ct = cl.getConstructor(types);

//On instancie l'objet avec le constructeur surchargé !

Object o2 = ct.newInstance(new String[]{"valeur 1 ", "valeur 2"} );

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

Si on appelle le toString :

public static void main(String[] args) {

String nom = Paire.class.getName();

try {

//On crée un objet Class

Class cl = Class.forName(nom);

//Nouvelle instance de la classe Paire

Object o = cl.newInstance();

//On crée les paramètres du constructeur

Class[] types = new Class[]{String.class, String.class};

//On récupère le constructeur avec les deux paramètres

Constructor ct = cl.getConstructor(types);

//On instancie l'objet avec le constructeur surchargé !

Object o2 = ct.newInstance(new String[]{"valeur 1 ", "valeur 2"} );

//On va chercher la méthode toString, elle n'a aucun paramètre

Method m = cl.getMethod("toString", null);

//La méthode invoke exécute la méthode sur l'objet passé en paramètre

//Pas de paramètre, donc null en deuxième paramètre de la méthode invoke !

System.out.println("----------------------------------------");

System.out.println("Méthode " + m.getName() + " sur o2: " +m.invoke(o2, null));

System.out.println("Méthode " + m.getName() + " sur o: " +m.invoke(o, null));

} catch (SecurityException e) {

e.printStackTrace();

} catch (IllegalArgumentException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

Avec ces méthodes, on peut par exemple stocker le nom d’une variable ou d’une classe en BDD.

# Notre première fenêtre

* *IHM*: interface homme machine.
* *GUI :* Graphical User Interface

On utilisera les packages javax.swing et java.awt présents d’office dans Java.

## L’objet jFrame :

Les objets jFrame viennent de javax.swing et non awt.

Les composants awt vienent de la création de Java et son lourds (HeavyWeight) et son gérés par le système d’exploitation. Les objets swing sont dessinés dans un conteneur et sont des objets légers : (LightWeight).

Il n’est donc pas recommandé d’utiliser les deu types dans une même fenêtre !

import javax.swing.JFrame;

public class Test {

public static void main(String[] args){

JFrame fenetre = new JFrame();

}

}

Lorsque vous exécutez ce code, vous n'obtenez rien, car par défaut, votre JFrame n'est pas visible. Vous devez donc lui dire « sois visible » de cette manière :

import javax.swing.JFrame;

public class Test {

public static void main(String[] args){

JFrame fenetre = new JFrame();

fenetre.setVisible(true);

}

}

La fenêtre étant minuscule, il faut tout lui dire :

Pour obtenir une fenêtre plus conséquente, il faudrait donc :

* Qu’elle soit plus grande ;
* Qu’elle comporte un titre (ce ne serait pas du luxe !) ;
* Qu’elle figure au centre de l'écran, ce serait parfait ;
* Que notre programme s'arrête réellement lorsqu'on clique sur la croix rouge, car, pour ceux qui ne l'auraient pas remarqué, le processus Eclipse tourne encore même après la fermeture de la fenêtre.

import javax.swing.JFrame;

public class Test {

public static void main(String[] args){

JFrame fenetre = new JFrame();

//Définit un titre pour notre fenêtre

fenetre.setTitle("Ma première fenêtre Java");

//Définit sa taille : 400 pixels de large et 100 pixels de haut

fenetre.setSize(400, 100);

//Nous demandons maintenant à notre objet de se positionner au centre

fenetre.setLocationRelativeTo(null);

//Termine le processus lorsqu'on clique sur la croix rouge

fenetre.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

//Et enfin, la rendre visible

fenetre.setVisible(true);

}

}

import javax.swing.JFrame;

public class Fenetre extends JFrame {

public Fenetre(){

this.setTitle("Ma première fenêtre Java");

this.setSize(400, 500);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setVisible(true);

}

}

Pour positionner la fenêtre on peut utiliser setLocation (int x, int y) :



##### Empêcher le redimensionnement de la fenêtre

Pour cela, il suffit d'invoquer la méthode setResizable (boolean b) : false empêche le redimensionnement tandis que true l'autorise.

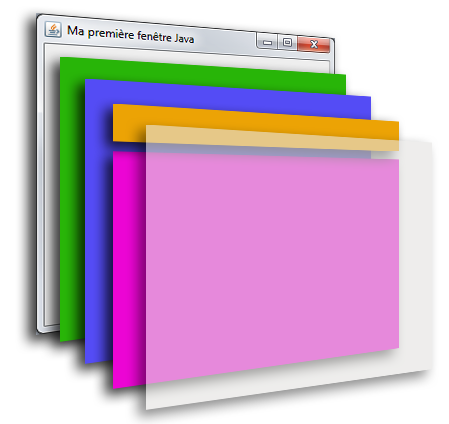
##### Garder la fenêtre au premier plan

Il s'agit là encore d'une méthode qui prend un booléen en paramètre. Passer true laissera la fenêtre au premier plan quoi qu'il advienne, false annulera cela. Cette méthode est setAlwaysOnTop (boolean b).

*Retirer les contours et boutons de contrôle :*

setUndecorated(boolean b)

### Composition d’une jFrame :



Nous avons, dans l'ordre :

* La fenêtre ;
* Le RootPane (en vert), le conteneur principal qui contient les autres composants ;
* Le LayeredPane (en violet), qui forme juste un panneau composé du conteneur global et de la barre de menu (MenuBar) ;
* La MenuBar (en orange), la barre de menu, quand il y en a une ;
* Le *content pane* (en rose) : c'est dans celui-ci que nous placerons nos composants ;
* Le GlassPane (en transparence), couche utilisée pour intercepter les actions de l'utilisateur avant qu'elles ne parviennent aux composants.

Il existe d'autres types de fenêtre : la JWindow, une JFrame sans bordure et non draggable (déplaçable), et la JDialog, une fenêtre non redimensionnable. Nous n'en parlerons toutefois pas ici.

## L’objet jPanel :

On va utiliser un jPanel, composant de type conteneur qui a pour vocation d’accueillir d’autre objets de même type ou des objets de type composant (bouton, case à cocher …)

Voici la marche à suivre :

1. Importer la classe javax.swing.JPanel dans notre classe héritée de JFrame.
2. Instancier un JPanel puis lui spécifier une couleur de fond pour mieux le distinguer.
3. Avertir notre JFrame que ce sera notre JPanel qui constituera son content pane.

import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class Fenetre extends JFrame {

public Fenetre(){

this.setTitle("Ma première fenêtre Java");

this.setSize(400, 100);

this.setLocationRelativeTo(null);

//Instanciation d'un objet JPanel

JPanel pan = new JPanel();

//Définition de sa couleur de fond

pan.setBackground(Color.ORANGE);

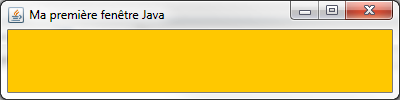
//On prévient notre JFrame que notre JPanel sera son content pane

this.setContentPane(pan);

this.setVisible(true);

}

}



## Objet graphics et graphics2d :

### L’objet Graphics :

**On ne peut l’utiliser que si et seulement si le système nous l’a donné via la méthode *getGraphics()* d’un composant swing.**

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

//Vous verrez cette phrase chaque fois que la méthode sera invoquée

System.out.println("Je suis exécutée !");

g.fillOval(20, 20, 75, 75);

}

}

C’est la méthode que l’objet appelle pour dessiner la fenêtre, même si on redimenssionne la page par exemple.

C’est automatique, on n’a pas à s’en soucier.

Mis dans jFrame :

import javax.swing.JFrame;

public class Fenetre extends JFrame {

public Fenetre(){

this.setTitle("Ma première fenêtre Java");

this.setSize(100, 150);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setContentPane(new Panneau());

this.setVisible(true);

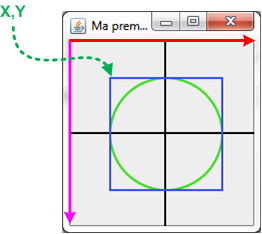
}

}

Si on devait traduire la méthode fillOval : : « Trace un rond plein en commençant à dessiner sur l'axe x à 20 pixels et sur l'axe y à 20 pixels, et fais-en sorte qu'il occupe 75 pixels de large et 75 pixels de haut. »

Il existe des méthodes quand on hérite de jPanel. Par exemple, si on hérite de jPanel on peut : avoir la largeur : *getWidth ()*, la hauteur : *getHeight ()*.

Si on veut que le cercle soit centré, il faut que le carré qui l’entoure le soit :



On obtient donc :

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

int x1 = this.getWidth()/4;

int y1 = this.getHeight()/4;

g.fillOval(x1, y1, this.getWidth()/2, this.getHeight()/2);

}

}

Le code sera donc centré en toute circonstance.

### Méthode drowOval :

Fonctionne exactement de la même manière sauf que le rond est vide.

Il en va de même pour fillRect et drowRect :

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

//x1, y1, width, height

g.drawRect(10, 10, 50, 60);

g.fillRect(65, 65, 30, 40);

}

}

### Méthode drawRoundRect() :

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

//x1, y1, width, height, arcWidth, arcHeight

g.drawRoundRect(10, 10, 30, 50, 10, 10);

g.fillRoundRect(55, 65, 55, 30, 5, 5);

}

}

### drawLine()

Trace des lignes droites, on spécifie le départ et l’arrivée.

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

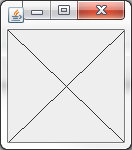
//x1, y1, x2, y2

g.drawLine(0, 0, this.getWidth(), this.getHeight());

g.drawLine(0, this.getHeight(), this.getWidth(), 0);

}

}



### drawPolygon()

drawPolygon(int[] x, int[] y, int nbrePoints);

On doit définir tous les points qui le forme dans un tableau.

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

int x[] = {20, 30, 50, 60, 60, 50, 30, 20};

int y[] = {30, 20, 20, 30, 50, 60, 60, 50};

g.drawPolygon(x, y, 8);

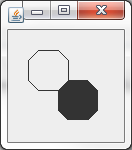
int x2[] = {50, 60, 80, 90, 90, 80, 60, 50};

int y2[] = {60, 50, 50, 60, 80, 90, 90, 80};

g.fillPolygon(x2, y2, 8);

}

}



Il existe aussi drawPolyLine() qui fait la même chose mais en traçant plusieurs lignes.

### drawString()

Permet de mettre du texte !

import java.awt.Graphics;

import javax.swing.JPanel;

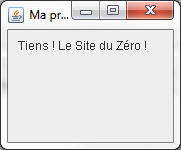
public class Panneau extends JPanel {

public void paintComponent(Graphics g){

g.drawString("Tiens ! Le Site du Zéro !", 10, 20);

}

}



Modifier la couleur :

import java.awt.Color;

import java.awt.Font;

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

Font font = new Font("Courier", Font.BOLD, 20);

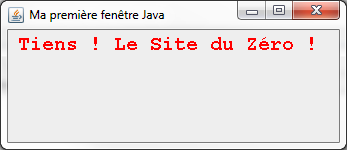
g.setFont(font);

g.setColor(Color.red);

g.drawString("Tiens ! Le Site du Zéro !", 10, 20);

}

}



### drawImage()

drawImage(Image img, int x, int y, Observer obs);

Vous devez charger votre image grâce à trois objets :

* un objet Image ;
* un objet ImageIO ;
* un objet File.

Il suffit d’initialiser un objet de type image avec Image.IO

En ce qui concerne le dernier paramètre de la méthode drawImage, il s'agit de l'objet qui est censé observer l'image. Ici, nous allons utiliser notre objet Panneau, donc this.

Cette méthode dessinera l'image avec ses propres dimensions. Si vous voulez qu'elle occupe l'intégralité de votre conteneur, utilisez le constructeur suivant : drawImage (Image img, int x, int y, int width, int height, Observer obs).

import java.awt.Graphics;

import java.awt.Image;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

try {

Image img = ImageIO.read(new File("images.jpg"));

g.drawImage(img, 0, 0, this);

//Pour une image de fond les deux 0 sont les points de départ

//g.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), this);

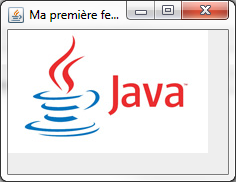
} catch (IOException e) {

e.printStackTrace();

}

}

}



### Graphics2D :

C’est une amélioration de Graphics.

**Pour utiliser Graphics2D on doit caster Graphics en Graphics2D.**

Il ne faut pas oublier d’importer java.awt.Graphics ET java.awt.Graphics2D.

Exemple : avec GradientPait :

On doit ajouter import java.awt.GradientPaint:

GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30, Color.cyan, true);

* Premier paramètre : la coordonnée x où doit commencer la première couleur ;
* Deuxième paramètre : la coordonnée y où doit commencer la première couleur ;
* Troisième paramètre : la première couleur ;
* Quatrième paramètre : la coordonnée x où doit commencer la seconde couleur ;
* Cinquième paramètre : la coordonnée y où doit commencer la seconde couleur ;
* Sixième paramètre : la seconde couleur ;
* Septième paramètre : le booléen indiquant si le dégradé doit se répéter.

import java.awt.Color;

import java.awt.Font;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

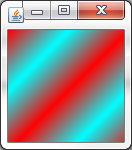
GradientPaint gp = new GradientPaint(0, 0, Color.RED, 30, 30, Color.cyan, true);

g2d.setPaint(gp);

g2d.fillRect(0, 0, this.getWidth(), this.getHeight());

}

}



Si on met x à 0, on obtient un dégradé vertical.

import java.awt.Color;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.imageio.ImageIO;

import javax.swing.JPanel;

public class Panneau extends JPanel {

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

GradientPaint gp, gp2, gp3, gp4, gp5, gp6;

gp = new GradientPaint(0, 0, Color.RED, 20, 0, Color.magenta, true);

gp2 = new GradientPaint(20, 0, Color.magenta, 40, 0, Color.blue, true);

gp3 = new GradientPaint(40, 0, Color.blue, 60, 0, Color.green, true);

gp4 = new GradientPaint(60, 0, Color.green, 80, 0, Color.yellow, true);

gp5 = new GradientPaint(80, 0, Color.yellow, 100, 0, Color.orange, true);

gp6 = new GradientPaint(100, 0, Color.orange, 120, 0, Color.red, true);

g2d.setPaint(gp);

g2d.fillRect(0, 0, 20, this.getHeight());

g2d.setPaint(gp2);

g2d.fillRect(20, 0, 20, this.getHeight());

g2d.setPaint(gp3);

g2d.fillRect(40, 0, 20, this.getHeight());

g2d.setPaint(gp4);

g2d.fillRect(60, 0, 20, this.getHeight());

g2d.setPaint(gp5);

g2d.fillRect(80, 0, 20, this.getHeight());

g2d.setPaint(gp6);

g2d.fillRect(100, 0, 40, this.getHeight());

}

}

# Une animation

## Création d’une animation :

En utilisant jFrame et jPanel on peut créer un effet de déplacement.

On va dire que notre rond part de l’exterieur de la fenêtre pour arriver à l’interieur :

import java.awt.Color;

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

private int posX = -50;

private int posY = -50;

public void paintComponent(Graphics g){

g.setColor(Color.red);

g.fillOval(posX, posY, 50, 50);

}

public int getPosX() {

return posX;

}

public void setPosX(int posX) {

this.posX = posX;

}

public int getPosY() {

return posY;

}

public void setPosY(int posY) {

this.posY = posY;

}

}

On va maintenant le déplacer :

import java.awt.Dimension;

import javax.swing.JFrame;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.setContentPane(pan);

this.setVisible(true);

go();

}

private void go(){

for(int i = -50; i < pan.getWidth(); i++){

int x = pan.getPosX(), y = pan.getPosY();

x++;

y++;

pan.setPosX(x);

pan.setPosY(y);

pan.repaint();

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

Ici repaint demande de redessiner le composant. Il demande de nouveau paintComponent ().

Pour effacer le passage du rond :

import java.awt.Color;

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel {

private int posX = -50;

private int posY = -50;

public void paintComponent(Graphics g){

//On choisit une couleur de fond pour le rectangle

g.setColor(Color.white);

//On le dessine de sorte qu'il occupe toute la surface

g.fillRect(0, 0, this.getWidth(), this.getHeight());

//On redéfinit une couleur pour le rond

g.setColor(Color.red);

//On le dessine aux coordonnées souhaitées

g.fillOval(posX, posY, 50, 50);

}

public int getPosX() {

return posX;

}

public void setPosX(int posX) {

this.posX = posX;

}

public int getPosY() {

return posY;

}

public void setPosY(int posY) {

this.posY = posY;

}

}

Ici on dessine un rectangle blanc pour ensuite faire un rond rouge.

## Amélioration :

Faire une boucle infinie :

//Exemple avec une boucle while

while(true){

//Ce code se répétera à l'infini, car la condition est toujours vraie !

}

//Exemple avec une boucle for

for(;;)

{

//Idem que précédemment : il n'y a pas d'incrémentation donc la boucle ne se terminera jamais.

}

//Exemple avec do… while

do{

//Encore une boucle que ne se terminera pas.

}while(true);

Donc si on fait une boucle infini avec notre ancien programme et qu’on le remet à 0 en cas de fin :

private void go(){

for(;;){

int x = pan.getPosX(), y = pan.getPosY();

x++;

y++;

pan.setPosX(x);

pan.setPosY(y);

pan.repaint();

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

//Si nos coordonnées arrivent au bord de notre composant

//On réinitialise

if(x == pan.getWidth() || y == pan.getHeight()){

pan.setPosX(-50);

pan.setPosY(-50);

}

}

}

On va maintenant faire en sorte que le rond du cercle ne dépasse pas de la fenêtre.

Voici la marche à suivre :

* si la valeur de la coordonnée *x* du rond est inférieure à la largeur du composant et que le rond avance, on continue d'avancer ;
* sinon, on recule.

private void go(){

//Les coordonnées de départ de notre rond

int x = pan.getPosX(), y = pan.getPosY();

//Le booléen pour savoir si l'on recule ou non sur l'axe x

boolean backX = false;

//Le booléen pour savoir si l'on recule ou non sur l'axe y

boolean backY = false;

//Dans cet exemple, j'utilise une boucle while

//Vous verrez qu'elle fonctionne très bien

while(true){

//Si la coordonnée x est inférieure à 1, on avance

if(x < 1)

backX = false;

//Si la coordonnée x est supérieure à la taille du Panneau moins la taille du rond, on recule

if(x > pan.getWidth()-50)

backX = true;

//Idem pour l'axe y

if(y < 1)

backY = false;

if(y > pan.getHeight()-50)

backY = true;

//Si on avance, on incrémente la coordonnée

//backX est un booléen, donc !backX revient à écrire

//if (backX == false)

if(!backX)

pan.setPosX(++x);

//Sinon, on décrémente

else

pan.setPosX(--x);

//Idem pour l'axe Y

if(!backY)

pan.setPosY(++y);

else

pan.setPosY(--y);

//On redessine notre Panneau

pan.repaint();

//Comme on dit : la pause s'impose ! Ici, trois millièmes de seconde

try {

Thread.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# Positionner les boutons

## Utiliser la classe jButton

jButton vient de java.swing. On crée un nouveau projet avec une classe main appelé test et une fenêtre qui hérite de jFrame.

Pour attribuer un libellé à un bouton :

//Possibilité 1 : instanciation avec le libellé

JButton bouton = new JButton("Mon premier bouton");

//Possibilité 2 : instanciation puis définition du libellé

JButton bouton2 = new JButton();

bouton2.setText("Mon deuxième bouton");

On utilise une classe main équivalente aux précedentes.

Pour faire en sorte que le bouton prend toute la palce de la fenêtre :

import javax.swing.JButton;

import javax.swing.JFrame;

public class Fenetre extends JFrame{

private JButton bouton = new JButton("Mon bouton");

public Fenetre(){

this.setTitle("Bouton");

this.setSize(300, 150);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

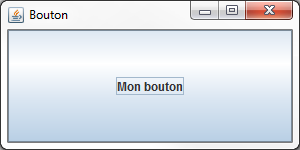
//On ajoute le bouton au content pane de la JFrame

this.getContentPane().add(bouton);

this.setVisible(true);

}

}

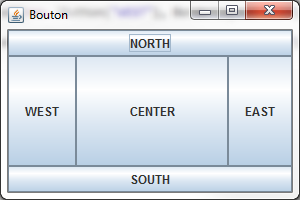


# Positionner les compsants :

Les *layout manager* se trouvent dans *java.awt*.

### L’objet BorderLayout

Il s’utilise avec *NORTH, SOUTH, EAST, WEST* et *CENTER.*



Exemple :

import java.awt.BorderLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

public class Fenetre extends JFrame{

public Fenetre(){

this.setTitle("Bouton");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

//On définit le layout à utiliser sur le content pane

this.setLayout(new BorderLayout());

//On ajoute le bouton au content pane de la JFrame

//Au centre

this.getContentPane().add(new JButton("CENTER"), BorderLayout.CENTER);

//Au nord

this.getContentPane().add(new JButton("NORTH"), BorderLayout.NORTH);

//Au sud

this.getContentPane().add(new JButton("SOUTH"), BorderLayout.SOUTH);

//À l'ouest

this.getContentPane().add(new JButton("WEST"), BorderLayout.WEST);

//À l'est

this.getContentPane().add(new JButton("EAST"), BorderLayout.EAST);

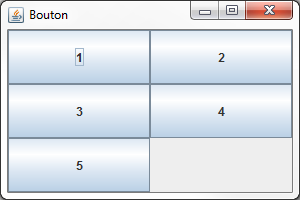
this.setVisible(true);

}

}

Par défaut, CENTER est utilisé.

### L’objet GridLayout :



On part de la partie en haut à gauche et dès qu’une ligne est remplie on passe à la suivante.

Code complet :

import java.awt.GridLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

public class Fenetre extends JFrame{

public Fenetre(){

this.setTitle("Bouton");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

//On définit le layout à utiliser sur le content pane

//Trois lignes sur deux colonnes

this.setLayout(new GridLayout(3, 2));

//On ajoute le bouton au content pane de la JFrame

this.getContentPane().add(new JButton("1"));

this.getContentPane().add(new JButton("2"));

this.getContentPane().add(new JButton("3"));

this.getContentPane().add(new JButton("4"));

this.getContentPane().add(new JButton("5"));

this.setVisible(true);

}

}

Ou alors comme ça :

GridLayout gl = new GridLayout();

gl.setColumns(2);

gl.setRows(3);

this.setLayout(gl);

Mettre de l’espace entre les boutons :

GridLayout gl = new GridLayout(3, 2);

gl.setHgap(5); //Cinq pixels d'espace entre les colonnes (H comme Horizontal)

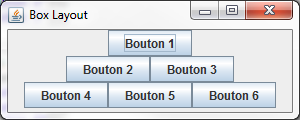
gl.setVgap(5); //Cinq pixels d'espace entre les lignes (V comme Vertical)

//Ou en abrégé : GridLayout gl = new GridLayout(3, 2, 5, 5);



### Objet boxLayout :

Grace à cet objet on pourra ranger les boutons sur une ligne ou sur une colonne.



import javax.swing.BoxLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

public Fenetre(){

this.setTitle("Box Layout");

this.setSize(300, 120);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

JPanel b1 = new JPanel();

//On définit le layout en lui indiquant qu'il travaillera en ligne

b1.setLayout(new BoxLayout(b1, BoxLayout.LINE\_AXIS));

b1.add(new JButton("Bouton 1"));

JPanel b2 = new JPanel();

//Idem pour cette ligne

b2.setLayout(new BoxLayout(b2, BoxLayout.LINE\_AXIS));

b2.add(new JButton("Bouton 2"));

b2.add(new JButton("Bouton 3"));

JPanel b3 = new JPanel();

//Idem pour cette ligne

b3.setLayout(new BoxLayout(b3, BoxLayout.LINE\_AXIS));

b3.add(new JButton("Bouton 4"));

b3.add(new JButton("Bouton 5"));

b3.add(new JButton("Bouton 6"));

JPanel b4 = new JPanel();

//On positionne maintenant ces trois lignes en colonne

b4.setLayout(new BoxLayout(b4, BoxLayout.PAGE\_AXIS));

b4.add(b1);

b4.add(b2);

b4.add(b3);

this.getContentPane().add(b4);

this.setVisible(true);

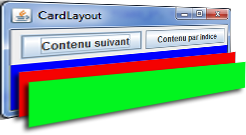
}

}

On crée plusieurs boutons que l’on range en ligne et on les met dans un autre layout qui les range en colonne.

### L’objet cardLayout :

Permet de gérer les conteneurs comme un tas de carte :



import java.awt.BorderLayout;

import java.awt.CardLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

CardLayout cl = new CardLayout();

JPanel content = new JPanel();

//Liste des noms de nos conteneurs pour la pile de cartes

String[] listContent = {"CARD\_1", "CARD\_2", "CARD\_3"};

int indice = 0;

public Fenetre(){

this.setTitle("CardLayout");

this.setSize(300, 120);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

//On crée trois conteneurs de couleur différente

JPanel card1 = new JPanel();

card1.setBackground(Color.blue);

JPanel card2 = new JPanel();

card2.setBackground(Color.red);

JPanel card3 = new JPanel();

card3.setBackground(Color.green);

JPanel boutonPane = new JPanel();

JButton bouton = new JButton("Contenu suivant");

//Définition de l'action du bouton

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

//Via cette instruction, on passe au prochain conteneur de la pile

cl.next(content);

}

});

JButton bouton2 = new JButton("Contenu par indice");

//Définition de l'action du bouton2

bouton2.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

if(++indice > 2)

indice = 0;

//Via cette instruction, on passe au conteneur correspondant au nom fourni en paramètre

cl.show(content, listContent[indice]);

}

});

boutonPane.add(bouton);

boutonPane.add(bouton2);

//On définit le layout

content.setLayout(cl);

//On ajoute les cartes à la pile avec un nom pour les retrouver

content.add(card1, listContent[0]);

content.add(card2, listContent[1]);

content.add(card3, listContent[2]);

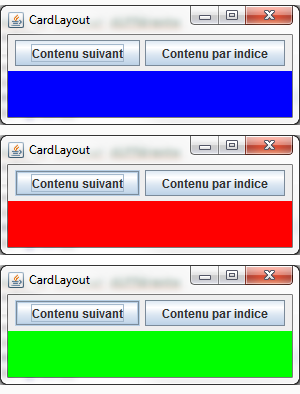
this.getContentPane().add(boutonPane, BorderLayout.NORTH);

this.getContentPane().add(content, BorderLayout.CENTER);

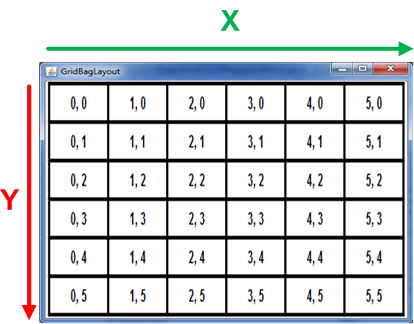
this.setVisible(true);

}

}



### L’objet GridBagLayout :



On représente une grille comme sous excel.



import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.GridBagConstraints;

import java.awt.GridBagLayout;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

public Fenetre(){

this.setTitle("GridBagLayout");

this.setSize(300, 160);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

//On crée nos différents conteneurs de couleur différente

JPanel cell1 = new JPanel();

cell1.setBackground(Color.YELLOW);

cell1.setPreferredSize(new Dimension(60, 40));

JPanel cell2 = new JPanel();

cell2.setBackground(Color.red);

cell2.setPreferredSize(new Dimension(60, 40));

JPanel cell3 = new JPanel();

cell3.setBackground(Color.green);

cell3.setPreferredSize(new Dimension(60, 40));

JPanel cell4 = new JPanel();

cell4.setBackground(Color.black);

cell4.setPreferredSize(new Dimension(60, 40));

JPanel cell5 = new JPanel();

cell5.setBackground(Color.cyan);

cell5.setPreferredSize(new Dimension(60, 40));

JPanel cell6 = new JPanel();

cell6.setBackground(Color.BLUE);

cell6.setPreferredSize(new Dimension(60, 40));

JPanel cell7 = new JPanel();

cell7.setBackground(Color.orange);

cell7.setPreferredSize(new Dimension(60, 40));

JPanel cell8 = new JPanel();

cell8.setBackground(Color.DARK\_GRAY);

cell8.setPreferredSize(new Dimension(60, 40));

//Le conteneur principal

JPanel content = new JPanel();

content.setPreferredSize(new Dimension(300, 120));

content.setBackground(Color.WHITE);

//On définit le layout manager

content.setLayout(new GridBagLayout());

//L'objet servant à positionner les composants

GridBagConstraints gbc = new GridBagConstraints();

//On positionne la case de départ du composant

gbc.gridx = 0;

gbc.gridy = 0;

//La taille en hauteur et en largeur

gbc.gridheight = 1;

gbc.gridwidth = 1;

content.add(cell1, gbc);

//---------------------------------------------

gbc.gridx = 1;

content.add(cell2, gbc);

//---------------------------------------------

gbc.gridx = 2;

content.add(cell3, gbc);

//---------------------------------------------

//Cette instruction informe le layout que c'est une fin de ligne

gbc.gridwidth = GridBagConstraints.REMAINDER;

gbc.gridx = 3;

content.add(cell4, gbc);

//---------------------------------------------

gbc.gridx = 0;

gbc.gridy = 1;

gbc.gridwidth = 1;

gbc.gridheight = 2;

//Celle-ci indique que la cellule se réplique de façon verticale

gbc.fill = GridBagConstraints.VERTICAL;

content.add(cell5, gbc);

//---------------------------------------------

gbc.gridx = 1;

gbc.gridheight = 1;

//Celle-ci indique que la cellule se réplique de façon horizontale

gbc.fill = GridBagConstraints.HORIZONTAL;

gbc.gridwidth = GridBagConstraints.REMAINDER;

content.add(cell6, gbc);

//---------------------------------------------

gbc.gridx = 1;

gbc.gridy = 2;

gbc.gridwidth = 2;

content.add(cell7, gbc);

//---------------------------------------------

gbc.gridx = 3;

gbc.gridwidth = GridBagConstraints.REMAINDER;

content.add(cell8, gbc);

//---------------------------------------------

//On ajoute le conteneur

this.setContentPane(content);

this.setVisible(true);

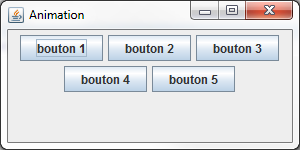
}

}

* gridx : position en x dans la grille.
* gridy : position en y dans la grille.
* gridwidth : nombre de colonnes occupées.
* gridheight : nombre de lignes occupées.
* weightx : si la grille est plus large que l'espace demandé, l'espace est redistribué proportionnellement aux valeurs de weightx des différentes colonnes.
* weighty : si la grille est plus haute que l'espace demandé, l'espace est redistribué proportionnellement aux valeurs de weighty des différentes lignes.
* anchor : ancrage du composant dans la cellule, c'est-à-dire son alignement dans la cellule (en bas à droite, en haut à gauche…). Voici les différentes valeurs utilisables :
  + FIRST\_LINE\_START : en haut à gauche ;
  + PAGE\_START : en haut au centre ;
  + FIRST\_LINE\_END : en haut à droite ;
  + LINE\_START : au milieu à gauche ;
  + CENTER : au milieu et centré ;
  + LINE\_END : au milieu à droite ;
  + LAST\_LINE\_START : en bas à gauche ;
  + PAGE\_END : en bas au centre ;
  + LAST\_LINE\_END : en bas à droite.
* fill : remplissage si la cellule est plus grande que le composant. Valeurs possibles : NONE, HORIZONTAL, VERTICAL et BOTH.
* insets : espace autour du composant. S'ajoute aux espacements définis par les propriétés ipadx et ipady ci-dessous.
* ipadx : espacement à gauche et à droite du composant.
* ipady : espacement au-dessus et au-dessous du composant.

### L’objet FlowLayout :

Il se contente de centrer l’élément. S’il y en a trop, il renvoie les autres à la ligne.



Si on reprend notre ancient projet avec la balle rouge :

import java.awt.BorderLayout;

import java.awt.Color;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("mon bouton");

private JPanel container = new JPanel();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

container.add(bouton, BorderLayout.SOUTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

private void go(){

//Les coordonnées de départ de notre rond

int x = pan.getPosX(), y = pan.getPosY();

//Le booléen pour savoir si l'on recule ou non sur l'axe x

boolean backX = false;

//Le booléen pour savoir si l'on recule ou non sur l'axe y

boolean backY = false;

//Dans cet exemple, j'utilise une boucle while

//Vous verrez qu'elle fonctionne très bien

while(true){

//Si la coordonnée x est inférieure à 1, on avance

if(x < 1)backX = false;

//Si la coordonnée x est supérieure à la taille du Panneau moins la taille du rond, on recule

if(x > pan.getWidth()-50)backX = true;

//Idem pour l'axe y

if(y < 1)backY = false;

if(y > pan.getHeight()-50)backY = true;

//Si on avance, on incrémente la coordonnée

if(!backX)

pan.setPosX(++x);

//Sinon, on décrémente

else

pan.setPosX(--x);

//Idem pour l'axe Y

if(!backY)

pan.setPosY(++y);

else

pan.setPosY(--y);

//On redessine notre Panneau

pan.repaint();

//Comme on dit : la pause s'impose ! Ici, trois millièmes de seconde

try {

Thread.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

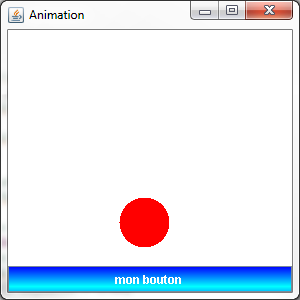
}

# Interagir avec un bouton

## Une classe bouton personnalisée :

On crée une classe qui hérite de javax.swing.jButton.

On redéfinit la méthode paintComponent.



import java.awt.Color;

import java.awt.Font;

import java.awt.FontMetrics;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JButton;

public class Bouton extends JButton {

private String name;

public Bouton(String str){

super(str);

this.name = str;

}

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20, Color.cyan, true);

g2d.setPaint(gp);

g2d.fillRect(0, 0, this.getWidth(), this.getHeight());

g2d.setColor(Color.white);

g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth()/ 2 /4), (this.getHeight() / 2) + 5);

}

}

### InterfaceMouseListener :

On veut que le design du bouton ne soit pas le même quand on clique dessus.

Pour cela Java utilise le *Design Pattern observer*.

Notre objet Bouton va s’écouter lui-même pour établir les changements.

addMouseListener(MouseListener obj).

import java.awt.Color;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{

private String name;

private Image img;

public Bouton(String str){

super(str);

this.name = str;

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

//Grâce à cette instruction, notre objet va s'écouter

//Dès qu'un événement de la souris sera intercepté, il en sera averti

this.addMouseListener(this);

}

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20, Color.cyan, true);

g2d.setPaint(gp);

g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), this);

g2d.setColor(Color.black);

g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

}

//Méthode appelée lors du clic de souris

public void mouseClicked(MouseEvent event) { }

//Méthode appelée lors du survol de la souris

public void mouseEntered(MouseEvent event) { }

//Méthode appelée lorsque la souris sort de la zone du bouton

public void mouseExited(MouseEvent event) { }

//Méthode appelée lorsque l'on presse le bouton gauche de la souris

public void mousePressed(MouseEvent event) { }

//Méthode appelée lorsque l'on relâche le clic de souris

public void mouseReleased(MouseEvent event) { }

}

Nous n'avons alors plus qu'à modifier notre image en fonction de la méthode invoquée. Notre objet comportera les caractéristiques suivantes :

* il aura une teinte jaune au survol de la souris ;
* il aura une teinte orangée lorsque l'on pressera le bouton gauche ;
* il reviendra à la normale si on relâche le clic.

import java.awt.Color;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{

private String name;

private Image img;

public Bouton(String str){

super(str);

this.name = str;

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

this.addMouseListener(this);

}

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20, Color.cyan, true);

g2d.setPaint(gp);

g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), this);

g2d.setColor(Color.black);

g2d.drawString(this.name, this.getWidth() / 2 - (this.getWidth() / 2 /4), (this.getHeight() / 2) + 5);

}

public void mouseClicked(MouseEvent event) {

//Inutile d'utiliser cette méthode ici

}

public void mouseEntered(MouseEvent event) {

//Nous changeons le fond de notre image pour le jaune lors du survol, avec le fichier fondBoutonHover.png

try {

img = ImageIO.read(new File("fondBoutonHover.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mouseExited(MouseEvent event) {

//Nous changeons le fond de notre image pour le vert lorsque nous quittons le bouton, avec le fichier fondBouton.png

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mousePressed(MouseEvent event) {

//Nous changeons le fond de notre image pour le jaune lors du clic gauche, avec le fichier fondBoutonClic.png

try {

img = ImageIO.read(new File("fondBoutonClic.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mouseReleased(MouseEvent event) {

//Nous changeons le fond de notre image pour le orange lorsque nous relâchons le clic, avec le fichier fondBoutonHover.png

try {

img = ImageIO.read(new File("fondBoutonHover.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

}

Cependant, on peut cliquer sur un bouton et relacher le clic en dehors du bouton .. pour palier le problème, il faut détecter les coordonnées de la souris.

public void mouseReleased(MouseEvent event) {

//Nous changeons le fond de notre image pour le orange lorsque nous relâchons le clic avec le fichier fondBoutonHover.png si la souris est toujours sur le bouton

if((event.getY() > 0 && event.getY() < this.getHeight()) && (event.getX() > 0 && event.getX() < this.getWidth())){

try {

img = ImageIO.read(new File("fondBoutonHover.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

//Si on se trouve à l'extérieur, on dessine le fond par défaut

else{

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

}

Vous verrez dans les chapitres qui suivent qu'il existe plusieurs interfaces pour les différentes actions possibles sur une IHM. Sachez qu'il existe aussi une convention pour ces interfaces : leur nom commence par le type de l'action, suivi du motListener. Nous avons étudié ici les actions de la souris, voyez le nom de l'interface :MouseListener.

## Intéragir avec le bouton :

### Déclencher une action : ActionListener :

public class Fenetre extends JFrame {

private Panneau pan = new Panneau();

private Bouton bouton = new Bouton("mon bouton");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

container.add(bouton, BorderLayout.SOUTH);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

//Le reste ne change pas

}

On peut changer la police, l’alignement, couleur du texte …

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

container.add(bouton, BorderLayout.SOUTH);

//Définition d'une police d'écriture

Font police = new Font("Tahoma", Font.BOLD, 16);

//On l'applique au JLabel

label.setFont(police);

//Changement de la couleur du texte

label.setForeground(Color.blue);

//On modifie l'alignement du texte grâce aux attributs statiques

//de la classe JLabel

label.setHorizontalAlignment(JLabel.CENTER);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

On peut maintenant faire en sorte que notre classe fenêtre implémente ActionListener.

(*Ctrl+shift+o* pour gérer les imports).

On va informer l’objet bouton que l’objet fenêtre l’écoute :

On va ajouter Fenetre à la liste des objets des écoutent l’objet bouton grace à addActionListener(ActionListener obj)présente dans jButton et donc dans Bouton.

Du coup dans actionPerformed() on peut mettre un compteur pour savoir combien de fois on a cliqué sur le bouton.

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{

private Panneau pan = new Panneau();

private Bouton bouton = new Bouton("mon bouton");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

//Compteur de clics

private int compteur = 0;

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

//Nous ajoutons notre fenêtre à la liste des auditeurs de notre bouton

bouton.addActionListener(this);

container.add(bouton, BorderLayout.SOUTH);

Font police = new Font("Tahoma", Font.BOLD, 16);

label.setFont(police);

label.setForeground(Color.blue);

label.setHorizontalAlignment(JLabel.CENTER);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

private void go(){

//Cette méthode ne change pas

}

public void actionPerformed(ActionEvent arg0) {

//Lorsque l'on clique sur le bouton, on met à jour le JLabel

this.compteur++;

label.setText("Vous avez cliqué " + this.compteur + " fois");

}

}



Si on veut mettre deux boutons en SOUTH grace à BorderLayout, seul le dernier apparaitra. Il faut les mettre dans un jPanel et ensuite l’insérer.

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame implements ActionListener{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("bouton 1");

private JButton bouton2 = new JButton("bouton 2");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

private int compteur = 0;

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

bouton.addActionListener(this);

bouton2.addActionListener(this);

JPanel south = new JPanel();

south.add(bouton);

south.add(bouton2);

container.add(south, BorderLayout.SOUTH);

Font police = new Font("Tahoma", Font.BOLD, 16);

label.setFont(police);

label.setForeground(Color.blue);

label.setHorizontalAlignment(JLabel.CENTER);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

//…

}

Seulement maintenant on a deux boutons pour un seul actionPerformed … Il faut donc connaitre la source de celui qui déclenche l’action avec getSource :

public void actionPerformed(ActionEvent arg0) {

if(arg0.getSource() == bouton)

label.setText("Vous avez cliqué sur le bouton 1");

if(arg0.getSource() == bouton2)

label.setText("Vous avez cliqué sur le bouton 2");

}



### Parler avec sa classe intérieur :

En Java on peut faire des classes internes, c’est-à-dire une classe dans une classe.

Elles ont accès aux attributs de la classe dans laquelle elles sont déclarées et même aux attributs privates.

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("bouton 1");

private JButton bouton2 = new JButton("bouton 2");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

private int compteur = 0;

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

//Ce sont maintenant nos classes internes qui écoutent nos boutons

bouton.addActionListener(new BoutonListener());

bouton2.addActionListener(new Bouton2Listener());

JPanel south = new JPanel();

south.add(bouton);

south.add(bouton2);

container.add(south, BorderLayout.SOUTH);

Font police = new Font("Tahoma", Font.BOLD, 16);

label.setFont(police);

label.setForeground(Color.blue);

label.setHorizontalAlignment(JLabel.CENTER);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

private void go(){

//Cette méthode ne change pas

}

//Classe écoutant notre premier bouton

class BoutonListener implements ActionListener{

//Redéfinition de la méthode actionPerformed()

public void actionPerformed(ActionEvent arg0) {

label.setText("Vous avez cliqué sur le bouton 1");

}

}

//Classe écoutant notre second bouton

class Bouton2Listener implements ActionListener{

//Redéfinition de la méthode actionPerformed()

public void actionPerformed(ActionEvent e) {

label.setText("Vous avez cliqué sur le bouton 2");

}

}

}

Plusieurs classes internes peuvent écouter le même bouton.

class Bouton3Listener implements ActionListener{

//Redéfinition de la méthode actionPerformed()

public void actionPerformed(ActionEvent e) {

System.out.println("Ma classe interne numéro 3 écoute bien !");

On peut aussi faire hériter les classes internes !

public class MaClasseExterne extends JFrame{

public MaClasseExterne(){

//...

}

class MaClassInterne extends JPanel{

public MaClassInterne(){

//…

}

}

class MaClassInterne2 extends JButton{

public MaClassInterne(){

//…

}

}

}

Ce code est donc valable.

### Les classes anonymes :

Elles sont utilisées le plus souvent pour la gestion d’évènement ponctuel.

JButton bouton = new JButton("Contenu suivant");

//Définition de l'action sur le bouton

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

//Action !

}

});

Revient à faire :

class Fenetre extends JFrame{

//…

bouton.addActionListener(new ActionListenerBis());

//…

public class ActionListenerBis implements ActionListener{

public void actionPerformed(ActionEvent event){

//Action !

}

}

}

Les classes anonymes sont soumises aux mêmes règles que les classes « normales » :

* Utilisation des méthodes non redéfinies de la classe mère ;
* Obligation de redéfinir **toutes les méthodes** d'une interface ;
* Obligation de redéfinir les méthodes abstraites d'une classe abstraite.

Cependant, ces classes possèdent des restrictions à cause de leur rôle et de leur raison d'être :

* Elles ne peuvent pas être déclaréesabstract ;
* Elles ne peuvent pas non plus être déclaréesstatic ;
* Elles ne peuvent pas définir de constructeur ;
* Elles sont automatiquement déclaréesfinal : on ne peut dériver de cette classe, l'héritage est donc impossible !

### Contrôler l’animation : marche et arrêt :

On doit modifier la classe fenêtre pour pouvoir faire ça :

* au lancement, le bouton Go ne sera pas cliquable alors que le bouton Stop oui ;
* si l'animation est interrompue, le bouton Stop ne sera plus cliquable, mais le bouton Go le sera.

On utilisera :

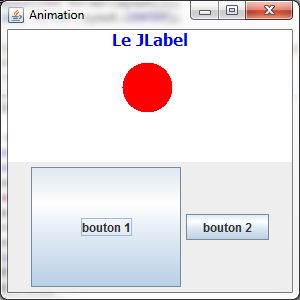
JButton bouton = new JButton("bouton");

bouton.setEnabled(false); //Le bouton n'est plus cliquable

bouton.setEnabled(true); //Le bouton est de nouveau cliquable

Pour modifier la taille on n’utilise pas setSize () mais plutôt :

bouton.setPreferredSize(new Dimension(150, 120));



Il faut maintenant arrêter notre boucle infinie :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("Go");

private JButton bouton2 = new JButton("Stop");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

private int compteur = 0;

private boolean animated = true;

private boolean backX, backY;

private int x, y;

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

bouton.addActionListener(new BoutonListener());

bouton.setEnabled(false);

bouton2.addActionListener(new Bouton2Listener());

JPanel south = new JPanel();

south.add(bouton);

south.add(bouton2);

container.add(south, BorderLayout.SOUTH);

Font police = new Font("Tahoma", Font.BOLD, 16);

label.setFont(police);

label.setForeground(Color.blue);

label.setHorizontalAlignment(JLabel.CENTER);

container.add(label, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

go();

}

private void go(){

//Les coordonnées de départ de notre rond

x = pan.getPosX();

y = pan.getPosY();

//Dans cet exemple, j'utilise une boucle while

//Vous verrez qu'elle fonctionne très bien

while(this.animated){

if(x < 1)backX = false;

if(x > pan.getWidth()-50)backX = true;

if(y < 1)backY = false;

if(y > pan.getHeight()-50)backY = true;

if(!backX)pan.setPosX(++x);

else pan.setPosX(--x);

if(!backY) pan.setPosY(++y);

else pan.setPosY(--y);

pan.repaint();

try {

Thread.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

animated = true;

bouton.setEnabled(false);

bouton2.setEnabled(true);

go();

}

}

class Bouton2Listener implements ActionListener{

public void actionPerformed(ActionEvent e) {

animated = false;

bouton.setEnabled(true);

bouton2.setEnabled(false);

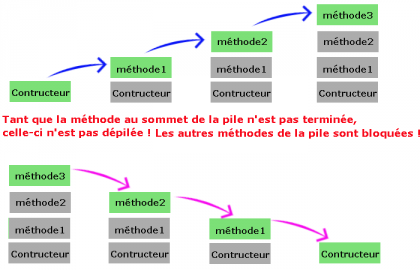
}

}

}

On remarque que :

* le boutonGon'est pas cliquable et l'autre l'est ;
* l'animation se lance ;
* l'animation s'arrête lorsque l'on clique sur le boutonStop ;
* le boutonGodevient alors cliquable ;
* lorsque vous cliquez dessus, l'animation ne se relance pas !



Dans notre programme, imaginez que la méthode actionPerformed ()soit représentée par la méthode 2, et que notre méthodevgo()soit représentée par la méthode 3. Lorsque nous entrons dans la méthode 3, nous entrons dans une boucle infinie… Conséquence directe : nous ne ressortons jamais de cette méthode et la JVM ne dépile plus !

## Être à l’écoute de ses objets : design pattern observer :

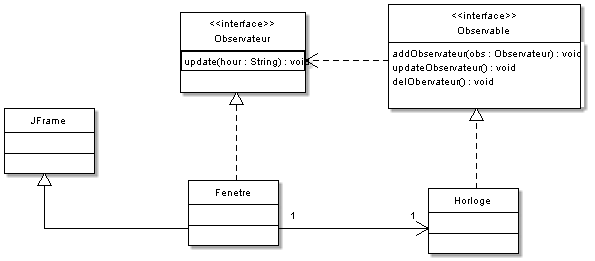
Il est utilisé pour gérer les évènements de nos IHM. C’est une technique de programmation (qui nous aidera à comprendre swing et awt).

### Des objets qui parlent et qui écoutent : le pattern observer :

Pour l’instant ce qu’on sait de cet objet :

* il fait communiquer des objets entre eux ;
* c'est un bon moyen d'éviter le couplage d'objets.

Dans notre cas, on a un ami qui veut que l’on développe une horloge pour lui. On doit mettre à jour toutes les secondes.



Grâce à cela, nos objets ne sont plus liés par leurs types, mais par leurs interfaces ! L'interface qui apportera les méthodes de mise à jour, d'ajout d'observateurs, etc. travaillera donc avec des objets de typeObservateur.

Voici comment fonctionnera l'application :

* nous instancierons la classe Horloge dans notre classeFenetre ;
* cette dernière implémentera l'interface Observateur ;
* notre objet Horloge, implémentant l'interface Observable, préviendra les objets spécifiés de ses changements ;
* nous informerons l'horloge que notre fenêtre l'observe ;
* à partir de là, notre objet Horloge fera le reste : à chaque changement, nous appellerons la méthode mettant tous les observateurs à jour.

Package com.sdz.observer).

##### Observateur.java

package com.sdz.observer;

public interface Observateur {

public void update(String hour);

}

##### Observer.java

package com.sdz.observer;

public interface Observable {

public void addObservateur(Observateur obs);

public void updateObservateur();

public void delObservateur();

}

Voici maintenant le code de nos deux classes, travaillant ensemble mais n'étant que faiblement couplées.

##### Horloge.java

package com.sdz.model;

import java.util.ArrayList;

import java.util.Calendar;

import com.sdz.observer.Observable;

import com.sdz.observer.Observateur;

public class Horloge implements Observable{

//On récupère l'instance d'un calendrier

//Elle va nous permettre de récupérer l'heure actuelle

private Calendar cal;

private String hour = "";

//Notre collection d'observateurs

private ArrayList<Observateur> listObservateur = new ArrayList<Observateur>();

public void run() {

while(true){

this.cal = Calendar.getInstance();

this.hour = //Les heures

this.cal.get(Calendar.HOUR\_OF\_DAY) + " : "

+

( //Les minutes

this.cal.get(Calendar.MINUTE) < 10

? "0" + this.cal.get(Calendar.MINUTE)

: this.cal.get(Calendar.MINUTE)

)

+ " : "

+

( //Les secondes

(this.cal.get(Calendar.SECOND)< 10)

? "0"+this.cal.get(Calendar.SECOND)

: this.cal.get(Calendar.SECOND)

);

//On avertit les observateurs que l'heure a été mise à jour

this.updateObservateur();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

//Ajoute un observateur à la liste

public void addObservateur(Observateur obs) {

this.listObservateur.add(obs);

}

//Retire tous les observateurs de la liste

public void delObservateur() {

this.listObservateur = new ArrayList<Observateur>();

}

//Avertit les observateurs que l'objet observable a changé

//et invoque la méthode update() de chaque observateur

public void updateObservateur() {

for(Observateur obs : this.listObservateur )

obs.update(this.hour);

}

}

##### Fenetre.java

package com.sdz.vue;

import java.awt.BorderLayout;

import java.awt.Font;

import javax.swing.JFrame;

import javax.swing.JLabel;

import com.sdz.model.Horloge;

import com.sdz.observer.Observateur;

public class Fenetre extends JFrame {

private JLabel label = new JLabel();

private Horloge horloge;

public Fenetre(){

//On initialise la JFrame

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.setResizable(false);

this.setSize(200, 80);

//On initialise l'horloge

this.horloge = new Horloge();

//On place un écouteur sur l'horloge

this.horloge.addObservateur(new Observateur(){

public void update(String hour) {

label.setText(hour);

}

});

//On initialise le JLabel

Font police = new Font("DS-digital", Font.TYPE1\_FONT, 30);

this.label.setFont(police);

this.label.setHorizontalAlignment(JLabel.CENTER);

//On ajoute le JLabel à la JFrame

this.getContentPane().add(this.label, BorderLayout.CENTER);

this.setVisible(true);

this.horloge.run();

}

//Méthode main() lançant le programme

public static void main(String[] args){

Fenetre fen = new Fenetre();

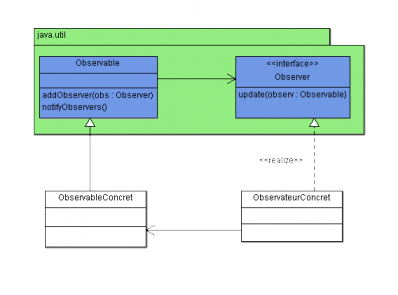
}

}

Il existe des classes Java qui permettent d’utiliser le pattern observer sans coder les interfaces :

Observer et Observable existe déjà en classe abstraite.

Hiérarchie de pattern observer dans Java :



update(Observable obs, Object obj) : sa signature a changé.  
Cette méthode prend ainsi deux paramètres :

* Un objetObservable ;
* Un Object représentant une donnée supplémentaire que vous souhaitez lui fournir.

## Bouton personnalisé et optimisé

import java.awt.Color;

import java.awt.FontMetrics;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.awt.Image;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import java.io.File;

import java.io.IOException;

import javax.imageio.ImageIO;

import javax.swing.JButton;

public class Bouton extends JButton implements MouseListener{

private String name;

private Image img;

public Bouton(String str){

super(str);

this.name = str;

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

this.addMouseListener(this);

}

public void paintComponent(Graphics g){

Graphics2D g2d = (Graphics2D)g;

GradientPaint gp = new GradientPaint(0, 0, Color.blue, 0, 20, Color.cyan, true);

g2d.setPaint(gp);

g2d.drawImage(img, 0, 0, this.getWidth(), this.getHeight(), this);

g2d.setColor(Color.black);

//Objet permettant de connaître les propriétés d'une police, dont la taille

FontMetrics fm = g2d.getFontMetrics();

//Hauteur de la police d'écriture

int height = fm.getHeight();

//Largeur totale de la chaîne passée en paramètre

int width = fm.stringWidth(this.name);

//On calcule alors la position du texte, et le tour est joué

g2d.drawString(this.name, this.getWidth() / 2 - (width / 2), (this.getHeight() / 2) + (height / 4));

}

public void mouseClicked(MouseEvent event) {

//Inutile d'utiliser cette méthode ici

}

public void mouseEntered(MouseEvent event) {

//Nous changeons le fond de notre image pour le jaune lors du survol, avec le fichier fondBoutonHover.png

try {

img = ImageIO.read(new File("fondBoutonHover.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mouseExited(MouseEvent event) {

//Nous changeons le fond de notre image pour le vert lorsque nous quittons le bouton, avec le fichier fondBouton.png

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mousePressed(MouseEvent event) {

//Nous changeons le fond de notre image pour le jaune lors du clic gauche, avec le fichier fondBoutonClic.png

try {

img = ImageIO.read(new File("fondBoutonClic.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

public void mouseReleased(MouseEvent event) {

//Nous changeons le fond de notre image pour l'orange lorsque nous relâchons le clic avec le fichier fondBoutonHover.png si la souris est toujours sur le bouton

if((event.getY() > 0 && event.getY() < this.getHeight()) && (event.getX() > 0 && event.getX() < this.getWidth())){

try {

img = ImageIO.read(new File("fondBoutonHover.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

//Si on se trouve à l'extérieur, on dessine le fond par défaut

else{

try {

img = ImageIO.read(new File("fondBouton.png"));

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

# TP corrigé une calculatrice

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.BorderFactory;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Calculatrice extends JFrame {

private JPanel container = new JPanel();

//Tableau stockant les éléments à afficher dans la calculatrice

String[] tab\_string = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", ".", "=", "C", "+", "-", "\*", "/"};

//Un bouton par élément à afficher

JButton[] tab\_button = new JButton[tab\_string.length];

private JLabel ecran = new JLabel();

private Dimension dim = new Dimension(50, 40);

private Dimension dim2 = new Dimension(50, 31);

private double chiffre1;

private boolean clicOperateur = false, update = false;

private String operateur = "";

public Calculatrice(){

this.setSize(240, 260);

this.setTitle("Calculette");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.setResizable(false);

//On initialise le conteneur avec tous les composants

initComposant();

//On ajoute le conteneur

this.setContentPane(container);

this.setVisible(true);

}

private void initComposant(){

//On définit la police d'écriture à utiliser

Font police = new Font("Arial", Font.BOLD, 20);

ecran = new JLabel("0");

ecran.setFont(police);

//On aligne les informations à droite dans le JLabel

ecran.setHorizontalAlignment(JLabel.RIGHT);

ecran.setPreferredSize(new Dimension(220, 20));

JPanel operateur = new JPanel();

operateur.setPreferredSize(new Dimension(55, 225));

JPanel chiffre = new JPanel();

chiffre.setPreferredSize(new Dimension(165, 225));

JPanel panEcran = new JPanel();

panEcran.setPreferredSize(new Dimension(220, 30));

//On parcourt le tableau initialisé

//afin de créer nos boutons

for(int i = 0; i < tab\_string.length; i++){

tab\_button[i] = new JButton(tab\_string[i]);

tab\_button[i].setPreferredSize(dim);

switch(i){

//Pour chaque élément situé à la fin du tableau

//et qui n'est pas un chiffre

//on définit le comportement à avoir grâce à un listener

case 11 :

tab\_button[i].addActionListener(new EgalListener());

chiffre.add(tab\_button[i]);

break;

case 12 :

tab\_button[i].setForeground(Color.red);

tab\_button[i].addActionListener(new ResetListener());

operateur.add(tab\_button[i]);

break;

case 13 :

tab\_button[i].addActionListener(new PlusListener());

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

case 14 :

tab\_button[i].addActionListener(new MoinsListener());

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

case 15 :

tab\_button[i].addActionListener(new MultiListener());

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

case 16 :

tab\_button[i].addActionListener(new DivListener());

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

default :

//Par défaut, ce sont les premiers éléments du tableau

//donc des chiffres, on affecte alors le bon listener

chiffre.add(tab\_button[i]);

tab\_button[i].addActionListener(new ChiffreListener());

break;

}

}

panEcran.add(ecran);

panEcran.setBorder(BorderFactory.createLineBorder(Color.black));

container.add(panEcran, BorderLayout.NORTH);

container.add(chiffre, BorderLayout.CENTER);

container.add(operateur, BorderLayout.EAST);

}

//Méthode permettant d'effectuer un calcul selon l'opérateur sélectionné

private void calcul(){

if(operateur.equals("+")){

chiffre1 = chiffre1 +

Double.valueOf(ecran.getText()).doubleValue();

ecran.setText(String.valueOf(chiffre1));

}

if(operateur.equals("-")){

chiffre1 = chiffre1 -

Double.valueOf(ecran.getText()).doubleValue();

ecran.setText(String.valueOf(chiffre1));

}

if(operateur.equals("\*")){

chiffre1 = chiffre1 \*

Double.valueOf(ecran.getText()).doubleValue();

ecran.setText(String.valueOf(chiffre1));

}

if(operateur.equals("/")){

try{

chiffre1 = chiffre1 /

Double.valueOf(ecran.getText()).doubleValue();

ecran.setText(String.valueOf(chiffre1));

} catch(ArithmeticException e) {

ecran.setText("0");

}

}

}

//Listener utilisé pour les chiffres

//Permet de stocker les chiffres et de les afficher

class ChiffreListener implements ActionListener {

public void actionPerformed(ActionEvent e){

//On affiche le chiffre additionnel dans le label

String str = ((JButton)e.getSource()).getText();

if(update){

update = false;

}

else{

if(!ecran.getText().equals("0"))

str = ecran.getText() + str;

}

ecran.setText(str);

}

}

//Listener affecté au bouton =

class EgalListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

calcul();

update = true;

clicOperateur = false;

}

}

//Listener affecté au bouton +

class PlusListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

if(clicOperateur){

calcul();

ecran.setText(String.valueOf(chiffre1));

}

else{

chiffre1 = Double.valueOf(ecran.getText()).doubleValue();

clicOperateur = true;

}

operateur = "+";

update = true;

}

}

//Listener affecté au bouton -

class MoinsListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

if(clicOperateur){

calcul();

ecran.setText(String.valueOf(chiffre1));

}

else{

chiffre1 = Double.valueOf(ecran.getText()).doubleValue();

clicOperateur = true;

}

operateur = "-";

update = true;

}

}

//Listener affecté au bouton \*

class MultiListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

if(clicOperateur){

calcul();

ecran.setText(String.valueOf(chiffre1));

}

else{

chiffre1 = Double.valueOf(ecran.getText()).doubleValue();

clicOperateur = true;

}

operateur = "\*";

update = true;

}

}

//Listener affecté au bouton /

class DivListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

if(clicOperateur){

calcul();

ecran.setText(String.valueOf(chiffre1));

}

else{

chiffre1 = Double.valueOf(ecran.getText()).doubleValue();

clicOperateur = true;

}

operateur = "/";

update = true;

}

}

//Listener affecté au bouton de remise à zéro

class ResetListener implements ActionListener {

public void actionPerformed(ActionEvent arg0){

clicOperateur = false;

update = true;

chiffre1 = 0;

operateur = "";

ecran.setText("");

}

}

}

public class Main {

public static void main(String[] args) {

Calculatrice calculette = new Calculatrice();

}

}

# Exécuter des taches simultanément

Les threads sont des fils d'exécution de notre programme. Lorsque nous en créons plusieurs, nous pouvons exécuter des tâches simultanément.

Nous en étions restés à notre animation qui bloque, et je vous avais dit que la solution était d'utiliser un deuxième Thread. Dans ce chapitre, nous allons voir comment créer une (ou plusieurs) nouvelle(s) pile(s) de fonctions grâce à ces fameux threads. Il existe une classe Thread dans Java permettant leur gestion. Vous allez voir qu'il existe deux façons de créer un nouveau thread.

**Quand on lance un programme, un thread est lancé. Le thread correspond à la pile et que chaque nouveau thread créé génère une pile d’exécution.**

Voyez un thread comme une machine bien huilée capable d'effectuer les tâches que vous lui spécifiez. Une fois instancié, un thread attend son lancement. Dès que c'est fait, il invoque sa méthode run()qui va lui permettre de connaître les tâches qu'il a à effectuer.

Deux moyens de créer un nouveau thread.

* créer une classe héritant de la classe Thread ;
* créer une implémentation de l'interface Runnable et instancier un objet Thread avec l'implémentation de cette interface.

public class TestThread extends Thread {

public TestThread(String name){

super(name);

}

public void run(){

for(int i = 0; i < 10; i++)

System.out.println(this.getName());

}

}

Testez maintenant ce code plusieurs fois :

public class Test {

public static void main(String[] args) {

TestThread t = new TestThread("A");

TestThread t2 = new TestThread(" B");

t.start();

t2.start();

}

}

Essai de plusieurs Thread

Java utilise un **ordonnanceur**. Il gère les threads de manière aléatoire, il en met en eveil et d’autre en sommeil.

Notez qu'avec les processeurs multi-coeurs aujourd'hui, il est désormais possible d'exécuter deux tâches exactement en même temps. Tout dépend donc de votre ordinateur.

Un thread peut présenter plusieurs états :

* NEW : lors de sa création.
* RUNNABLE : lorsqu'on invoque la méthode start(), le thread est prêt à travailler.
* TERMINATED : lorsque le thread a effectué toutes ses tâches ; on dit aussi qu'il est « mort ». Vous ne pouvez alors plus le relancer par la méthode start().
* TIMED\_WAITING : lorsque le thread est en pause (quand vous utilisez la méthode sleep(), par exemple).
* WAITING : lorsque le thread est en attente indéfinie.
* BLOCKED : lorsque l'ordonnanceur place un thread en sommeil pour en utiliser un autre, il lui impose cet état.

Un thread est considéré comme terminé lorsque la méthode run() est ôtée de sa pile d'exécution. En effet, une nouvelle pile d'exécution contient à sa base la méthode run() de notre thread. Une fois celle-ci dépilée, notre nouvelle pile est détruite !

En fait, le thread principal crée un second thread qui se lance et construit une pile dont la base est sa méthode run() ; celle-ci appelle une méthode, l'empile, effectue toutes les opérations demandées, et une fois qu'elle a terminé, elle dépile cette dernière. La méthode run() prend fin, la pile est alors détruite.

public class TestThread extends Thread {

Thread t;

public TestThread(String name){

super(name);

System.out.println("statut du thread " + name + " = " +this.getState());

this.start();

System.out.println("statut du thread " + name + " = " +this.getState());

}

public TestThread(String name, Thread t){

super(name);

this.t = t;

System.out.println("statut du thread " + name + " = " +this.getState());

this.start();

System.out.println("statut du thread " + name + " = " +this.getState());

}

public void run(){

for(int i = 0; i < 10; i++){

System.out.println("statut " + this.getName() + " = " +this.getState());

if(t != null)

System.out.println("statut de " + t.getName() + " pendant le thread " + this.getName() +" = " +t.getState());

}

}

public void setThread(Thread t){

this.t = t;

}

}

Ainsi que notre main :

public class Test {

public static void main(String[] args) {

TestThread t = new TestThread("A");

TestThread t2 = new TestThread(" B", t);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

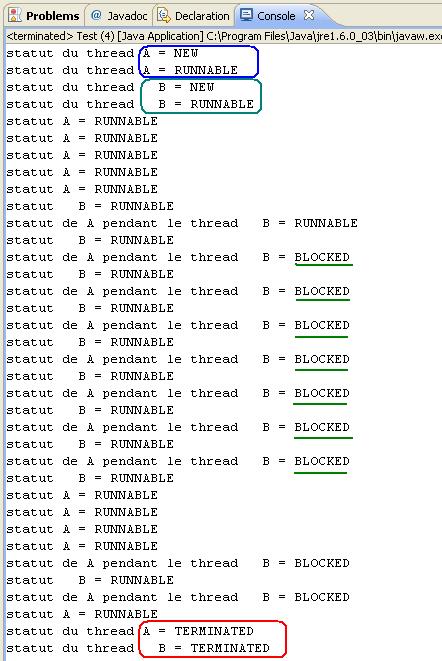
}

System.out.println("statut du thread " + t.getName() + " = " + t.getState());

System.out.println("statut du thread " + t2.getName() + " = " +t2.getState());

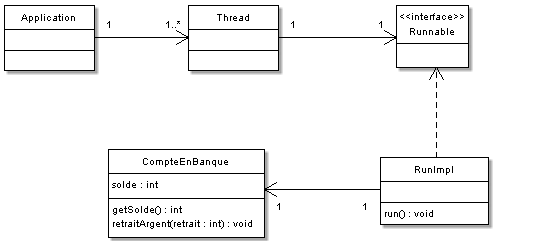
}

}



# Utiliser l’interface Runnable

Exemple d’un compte bancaire, on ne peut retirer que si l’on n’est pas à découvert.



Résumé :

* notre application peut contenir un ou plusieurs objets Thread ;
* ceux-ci ne peuvent être constitués que d'un objet de type Runnable ;
* dans notre cas, les objets Thread contiendront une implémentation de Runnable : RunImpl ;
* cette implémentation possède un objet CompteEnBanque.

##### RunImpl.java

public class RunImpl implements Runnable {

private CompteEnBanque cb;

public RunImpl(CompteEnBanque cb){

this.cb = cb;

}

public void run() {

for(int i = 0; i < 25; i++){

if(cb.getSolde() > 0){

cb.retraitArgent(2);

System.out.println("Retrait effectué");

}

}

}

}

##### CompteEnBanque.java

public class CompteEnBanque {

private int solde = 100;

public int getSolde(){

if(this.solde < 0)

System.out.println("Vous êtes à découvert !");

return this.solde;

}

public void retraitArgent(int retrait){

solde = solde - retrait;

System.out.println("Solde = " + solde);

}

}

##### Test.java

public class Test {

public static void main(String[] args) {

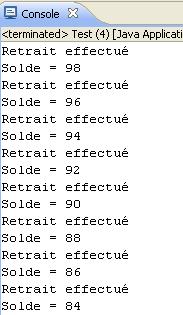
CompteEnBanque cb = new CompteEnBanque();

Thread t = new Thread(new RunImpl(cb));

t.start();

}

}



Pour le moment rien de fou, une boucle fait la même chose.

Il faut penser à modifier l'implémentation afin que nous puissions connaître le thread qui travaille :

public class RunImpl implements Runnable {

private CompteEnBanque cb;

private String name;

public RunImpl(CompteEnBanque cb, String name){

this.cb = cb;

this.name = name;

}

public void run() {

for(int i = 0; i < 50; i++){

if(cb.getSolde() > 0){

cb.retraitArgent(2);

System.out.println("Retrait effectué par " + this.name);

}

}

}

}

public class Test {

public static void main(String[] args) {

CompteEnBanque cb = new CompteEnBanque();

CompteEnBanque cb2 = new CompteEnBanque();

Thread t = new Thread(new RunImpl(cb, "Cysboy"));

Thread t2 = new Thread(new RunImpl(cb2, "Zéro"));

t.start();

t2.start();

}

}

Jusqu'ici, rien de perturbant : nous avons utilisé deux instances distinctes de RunImpl utilisant elles-mêmes deux instances distinctes de CompteEnBanque. Mais que se passerait-il si nous utilisions la même instance de CompteEnBanque pour deux threads différents ? Testez plusieurs fois le code que voici :

public class Test {

public static void main(String[] args) {

CompteEnBanque cb = new CompteEnBanque();

Thread t = new Thread(new RunImpl(cb, "Cysboy"));

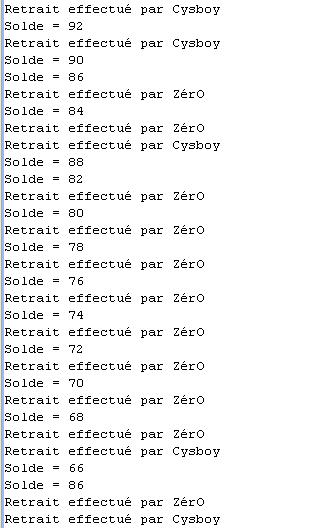
Thread t2 = new Thread(new RunImpl(cb, "Zéro"));

t.start();

t2.start();

}

}



On voit plein d’incohérence parce que quand l’ordonnanceur met un thread en veille, il le reprend là où il l’avait laissé.

# Synchroniser ses threads

Ce qu’il faut faire, c’est indiquer à la JVM qu’un thread en dépend d’un autre, ils doivent donc être synchronisé.

On utilise le mot clé *synchronised*.

public class CompteEnBanque {

//Le début du code ne change pas

public synchronized void retraitArgent(int retrait){

solde = solde - retrait;

System.out.println("Solde = " + solde);

}

}

Il vous suffit d'ajouter dans la déclaration de la méthode le mot clé synchronized, grâce auquel la méthode est inaccessible à un thread si elle est déjà utilisée par un autre thread. Ainsi, les threads cherchant à utiliser des méthodes déjà prises en charge par un autre thread sont placés dans une « liste d'attente ».

Je récapitule une nouvelle fois, en me servant d'un exemple simple. Je serai représenté par le thread A, vous par le thread B, et notre boulangerie favorite par la méthode synchronisée M. Voici ce qu'il se passe :

* le thread A (moi) appelle la méthode M ;
* je commence par demander une baguette : la boulangère me la pose sur le comptoir et commence à calculer le montant ;
* c'est là que le thread B (vous) cherche aussi à utiliser la méthode M ; cependant, elle est déjà occupée par un thread (moi) ;
* vous êtes donc mis en attente ;
* l'action revient sur moi (thread A) ; au moment de payer, je dois chercher de la monnaie dans ma poche ;
* au bout de quelques instants, je m'endors ;
* l'action revient sur le thread B (vous)… mais la méthode M n'est toujours pas libérée du thread A, vous êtes donc remis en attente ;
* on revient sur le thread A qui arrive enfin à payer et à quitter la boulangerie : la méthode M est maintenant libérée ;
* le thread B (vous) peut enfin utiliser la méthode M ;
* et là, les threads C, D, E et F entrent dans la boulangerie ;
* et ainsi de suite.

# Contrôler son animation

Retour sur notre application avec la balle :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("Go");

private JButton bouton2 = new JButton("Stop");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Le JLabel");

private int compteur = 0;

private boolean animated = true;

private boolean backX, backY;

private int x, y;

private Thread t;

public Fenetre(){

//Le constructeur n'a pas changé

}

private void go(){

//La méthode n'a pas changé

}

public class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

animated = true;

t = new Thread(new PlayAnimation());

t.start();

bouton.setEnabled(false);

bouton2.setEnabled(true);

}

}

class Bouton2Listener implements ActionListener{

public void actionPerformed(ActionEvent e) {

animated = false;

bouton.setEnabled(true);

bouton2.setEnabled(false);

}

}

class PlayAnimation implements Runnable{

public void run() {

go();

}

}

}

# Depuis Java7, le pattern Fork/Join

Le pattern Fork/Join c’est « diviser pour mieux reigner ».

Avant de commencer il faut préciser qu'il y a un certain nombre de prérequis à cela :

* la machine qui exécutera la tâche devra posséder un processeur à plusieurs cœurs (2, 4 ou plus) ;
* la tâche doit pouvoir être découpée en plusieurs sous-tâches ;
* s'assurer qu'il y a un réel gain de performance ! Dans certains cas, découper une tâche rend le traitement plus long.

On va coder une recherche de fichier sans Fork/Join pour le moment.

##### ScanException.java

public class ScanException extends Exception{

public ScanException(String message){super(message);}

}

##### FolderScanner.java

import java.io.IOException;

import java.nio.file.DirectoryStream;

import java.nio.file.Files;

import java.nio.file.Path;

public class FolderScanner{

private Path path = null;

private String filter = "\*";

private long result = 0;

public FolderScanner(){ }

public FolderScanner(Path p, String f){

path = p;

filter = f;

}

/\*\*

\* Méthode qui se charge de scanner les dossiers de façon récursive

\* @throws ScanException

\*/

public long sequentialScan() throws ScanException{

//Si le chemin n'est pas valide, on lève une exception

if(path == null || path.equals(""))

throw new ScanException("Chemin à scanner non valide (vide ou null) !");

System.out.println("Scan du dossier : " + path + " à la recherche des fichiers portant l'extension " + this.filter);

//On liste maintenant le contenu du répertoire pour traiter les sous-dossiers

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path)){

for(Path nom : listing){

//S'il s'agit d'un dossier, on le scanne grâce à notre objet

if(Files.isDirectory(nom.toAbsolutePath())){

FolderScanner f = new FolderScanner(nom.toAbsolutePath(), this.filter);

result += f.sequentialScan();

}

}

} catch (IOException e) { e.printStackTrace();}

//Maintenant, on filtre le contenu de ce même dossier sur le filtre défini

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path, this.filter)){

for(Path nom : listing){

//Pour chaque fichier correspondant, on incrémente notre compteur

result++;

}

} catch (IOException e) { e.printStackTrace(); }

return result;

}

}

##### Et la classe de test : Main.java

import java.nio.file.Path;

import java.nio.file.Paths;

public class Main {

public static void main(String[] args) {

Path chemin = Paths.get("E:\\Mes Documents");

String filtre = "\*.psd";

FolderScanner fs = new FolderScanner(chemin, filtre);

try {

Long start = System.currentTimeMillis();

Long resultat = fs.sequentialScan();

Long end = System.currentTimeMillis();

System.out.println("Il y a " + resultat + " fichier(s) portant l'extension " + filtre);

System.out.println("Temps de traitement : " + (end - start));

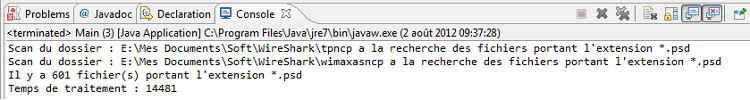
} catch (ScanException e) {

e.printStackTrace();

}

}

}



La plateforme Java 7 nous met à disposition deux classes qui héritent de la classe abstraite ForkJoinTask<V> :

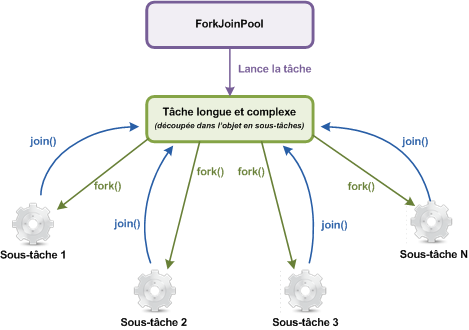
* RecursiveAction : classe permettant de découper une tâche ne renvoyant aucune valeur particulière. Elle hérite de ForkJoinTask<Void> ;
* RecursiveTask<V> : identique à la classe précédente mais retourne une valeur, de type <V>, en fin de traitement. C'est cette classe que nous allons utiliser pour pouvoir nous retourner le nombre de fichiers trouvés.

Nous allons devoir utiliser, en plus de l'objet de découpage, un objet qui aura pour rôle de superviser l'exécution des tâches et sous-tâches afin de pouvoir fusionner les threads en fin de traitement : ForkJoinPool.

**Fonctionnement :**

* compute() : méthode abstraite à redéfinir dans l'objet héritant afin de définir le traitement à effectuer ;
* fork() : méthode qui crée un nouveau thread dans le pool de thread (ForkJoinPool) ;
* join() : méthode qui permet de récupérer le résultat de la méthode compute().

Ces classes nécessitent que vous redéfinissiez la méthode compute() afin de définir ce qu'il y a à faire. La figure suivante est un schéma représentant la façon dont les choses se passent.

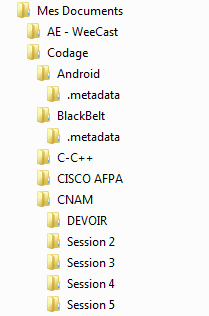


Plusieurs sous-tâches s'exécutent

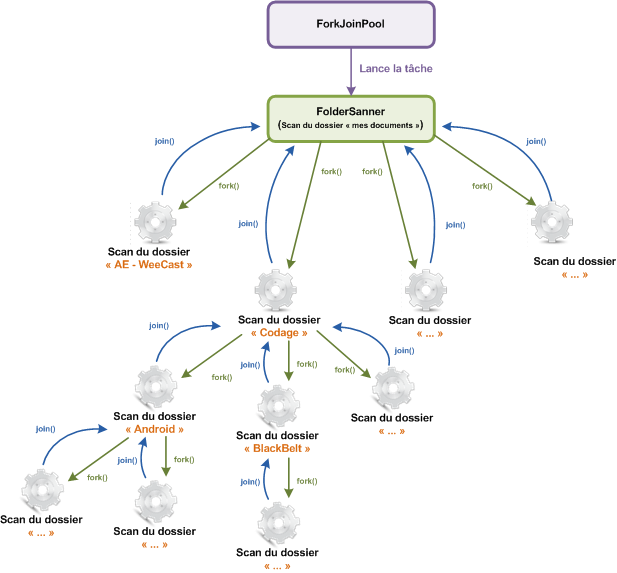
Concrètement, avec notre exemple, voici ce qu'il va se passer :

* Nous allons lancer le scan de notre dossier ;
* Notre objet qui sert à scanner le contenu va vérifier le contenu pour voir s'il n'y a pas de sous-dossiers ;
* Pour chaque sous-dossier, nous allons créer une nouvelle tâche et la lancer ;
* Nous allons compter le nombre de fichiers qui correspond à nos critères dans le dossier en cours de scan ;
* Nous allons récupérer le nombre de fichiers trouvés par les exécutions en tâche de fond ;
* Nous allons retourner le résultat final.

Pour que vous compreniez bien, voici une partie de mon dossier Mes Documents :



Et voici concrètement ce qu'il va se passer :



##### FolderScanner.java

public class FolderScanner extends RecursiveTask<Long>{

private Path path = null;

private String filter = "\*";

private long result = 0;

public FolderScanner(){ }

public FolderScanner(Path p, String f){

path = p;

filter = f;

}

/\*\*

\* Notre méthode de scan en mode mono thread

\* @throws ScanException

\*/

public long sequentialScan() throws ScanException{

//Si le chemin n'est pas valide, on lève une exception

if(path == null || path.equals(""))

throw new ScanException("Chemin à scanner non valide (vide ou null) !");

System.out.println("Scan du dossier : " + path + " à la recherche des fichiers portant l'extension " + this.filter);

//On liste maintenant le contenu du répertoire pour traiter les sous-dossiers

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path)){

for(Path nom : listing){

//S'il s'agit d'un dossier, on le scan grâce à notre objet

if(Files.isDirectory(nom.toAbsolutePath())){

FolderScanner f = new FolderScanner(nom.toAbsolutePath(), this.filter);

result += f.sequentialScan();

}

}

} catch (IOException e) { e.printStackTrace(); }

//Maintenant, on filtre le contenu de ce même dossier sur le filtre défini

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path, this.filter)){

for(Path nom : listing){

//Pour chaque fichier correspondant, on incrémente notre compteur

result++;

}

} catch (IOException e) { e.printStackTrace(); }

return result;

}

/\*\*

\* Méthode que nous allons utiliser pour les traitements

\* en mode parallèle.

\* @throws ScanException

\*/

public long parallelScan() throws ScanException{

//List d'objet qui contiendra les sous-tâches créées et lancées

List<FolderScanner> list = new ArrayList<>();

//Si le chemin n'est pas valide

if(path == null || path.equals(""))

throw new ScanException("Chemin à scanner non valide (vide ou null) !");

System.out.println("Scan du dossier : " + path + " a la recherche des fichiers portant l'extension " + this.filter);

//Nous listons, comme précédemment, le contenu du répertoire

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path)){

//On parcourt le contenu

for(Path nom : listing){

//S'il s'agit d'un dossier, on crée une sous-tâche

if(Files.isDirectory(nom.toAbsolutePath())){

//Nous créons donc un nouvel objet FolderScanner

//Qui se chargera de scanner ce dossier

FolderScanner f = new FolderScanner(nom.toAbsolutePath(), this.filter);

//Nous l'ajoutons à la liste des tâches en cours pour récupérer le résultat plus tard

list.add(f);

//C'est cette instruction qui lance l'action en tâche de fond

f.fork();

}

}

} catch (IOException e) {

e.printStackTrace();

}

//On compte maintenant les fichiers, correspondant au filtre, présents dans ce dossier

try(DirectoryStream<Path> listing = Files.newDirectoryStream(path, this.filter)){

for(Path nom : listing){

result++;

}

} catch (IOException e) {

e.printStackTrace();

}

//Et, enfin, nous récupérons le résultat de toutes les tâches de fond

for(FolderScanner f : list)

result += f.join();

//Nous renvoyons le résultat final

return result;

}

/\*\*

\* Méthode qui défini l'action à faire

\* dans notre cas, nous lan çons le scan en mode parallèles

\*/

protected Long compute() {

long resultat = 0;

try {

resultat = this.parallelScan();

} catch (ScanException e) {

e.printStackTrace();

}

return resultat;

}

public long getResultat(){ return this.result; }

}

Et voici maintenant notre classe de test :

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.concurrent.ForkJoinPool;

public class Main {

public static void main(String[] args) {

Path chemin = Paths.get("E:\\Mes Documents");

String filtre = "\*.psd";

//Création de notre tâche principale qui se charge de découper son travail en sous-tâches

FolderScanner fs = new FolderScanner(chemin, filtre);

//Nous récupérons le nombre de processeurs disponibles

int processeurs = Runtime.getRuntime().availableProcessors();

//Nous créons notre pool de thread pour nos tâches de fond

ForkJoinPool pool = new ForkJoinPool(processeurs);

Long start = System.currentTimeMillis();

//Nous lançons le traitement de notre tâche principale via le pool

pool.invoke(fs);

Long end = System.currentTimeMillis();

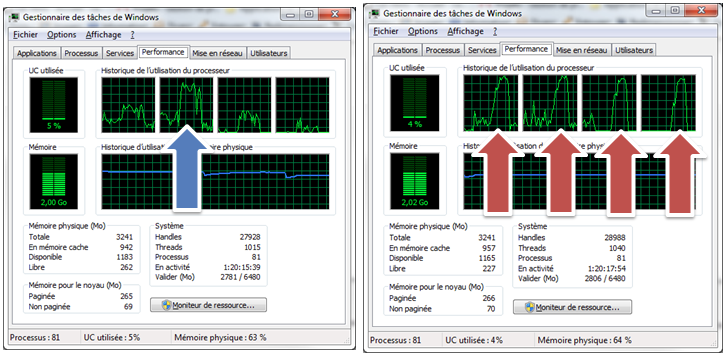
System.out.println("Il y a " + fs.getResultat() + " fichier(s) portant l'extension " + filtre);

System.out.println("Temps de traitement : " + (end - start));

}

}

La figure suivante représente l'utilisation de mes processeurs.

À gauche en mode normal, à droite en mode Fork/Join

Vous constaterez que l'utilisation de ce mode est très gourmand en ressource processeurs. Il est donc à utiliser avec parcimonie.

Dans cet exemple nous avons créé dynamiquement autant de threads que nécessaires pour traiter nos tâches. Vous n'aurez peut-être pas besoin de faire ceci pour des problèmes où seulement 2 ou 3 sous-tâches suffisent, surtout si vous le savez à l'avance. L'idée maîtresse revient à définir un seuil au delà duquel le traitement se fera en mode Fork/join, sinon, il se fera dans un seul thread (je vous rappelle qu'il se peut que ce mode de fonctionnement soit plus lent et consommateur qu'en mode normal). Voici comment procéder dans ce genre de cas :

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.RecursiveTask;

public class CalculSuite extends RecursiveTask<Long> {

private long debut = 0, fin = 0, resultat;

private int SEUIL = 1\_000;

public CalculSuite(long debut, long fin){

this.debut = debut;

this.fin = fin;

}

protected Long compute() {

long nombreDeChoseAFaire = fin - debut;

if(nombreDeChoseAFaire < SEUIL){

System.out.println("Passage en mode MonoThread ou le découpage calcul le résultat");

resultat = calculer();

}

else{

System.out.println("Passage en mode Fork/Join");

//On découpe la tâche en deux

long milieu = nombreDeChoseAFaire/2;

CalculSuite calcul1 = new CalculSuite(debut, (debut+milieu)-1);

calcul1.fork();

CalculSuite calcul2 = new CalculSuite(debut + milieu, fin);

resultat = calcul2.compute() + calcul1.join();

}

return resultat;

}

public long calculer(){

for(long i = debut; i <= fin; i++){

System.out.println(resultat + " + " + i);

resultat += i;

}

return resultat;

}

public long getResultat(){ return resultat; }

public static void main(String[] args){

ForkJoinPool pool = new ForkJoinPool();

CalculSuite calcul = new CalculSuite(0, 100\_000);

pool.invoke(calcul);

System.out.println("Résultat du calcul : " + calcul.getResultat());

}

}

# Les champs de formulaire

## Les listes, l’objet JComboBox :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JComboBox combo = new JComboBox();

private JLabel label = new JLabel("Une ComboBox");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

combo.setPreferredSize(new Dimension(100, 20));

JPanel top = new JPanel();

top.add(label);

top.add(combo);

container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

}

L’objet est vide, il faut ajouter des item :

//Les imports restent inchangés

public class Fenetre extends JFrame {

//Les variables d'instance restent inchangées

public Fenetre(){

//…

combo.setPreferredSize(new Dimension(100, 20));

combo.addItem("Option 1");

combo.addItem("Option 2");

combo.addItem("Option 3");

combo.addItem("Option 4");

//…

}

}

On peut aussi utiliser un tableau pour initialiser une combo box :

String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};

combo = new JComboBox(tab);

Depuis Java 7, l'objet JComboBox peut être paramétré avec un type générique, comme ceci : JComboBox<String> combo = new JComboBox<String>(); ce qui permet de mieux gérer le contenu de nos listes et ainsi mieux récupérer les valeurs de ces dernières.

### Communiquer avec l’objet : l’interface ItemListener :

Elle est appelée lorqu’un élément a changé d’état :

//Les autres imports

import java.awt.event.ItemEvent;

import java.awt.event.ItemListener;

public class Fenetre extends JFrame {

//Les variables d'instance restent inchangées

public Fenetre(){

//Le début ne change pas

//Ici, nous changeons juste la façon d'initialiser la JComboBox

String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};

combo = new JComboBox(tab);

//Ajout du listener

combo.addItemListener(new ItemState());

combo.setPreferredSize(new Dimension(100, 20));

combo.setForeground(Color.blue);

//La fin reste inchangée

}

//Classe interne implémentant l'interface ItemListener

class ItemState implements ItemListener{

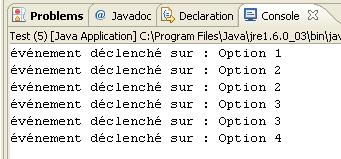
public void itemStateChanged(ItemEvent e) {

System.out.println("événement déclenché sur : " + e.getItem());

}

}

}



Java modifie en réalité l’état des otpion de DESELECTED à SELECTED et inversement.

On utilise ActionListener pour plus de simplicité :

//Les autres imports

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class Fenetre extends JFrame {

//Les variables d'instance restent inchangées

public Fenetre(){

//Le début ne change pas

String[] tab = {"Option 1", "Option 2", "Option 3", "Option 4"};

combo = new JComboBox(tab);

//Ajout du listener

combo.addItemListener(new ItemState());

combo.addActionListener(new ItemAction());

combo.setPreferredSize(new Dimension(100, 20));

combo.setForeground(Color.blue);

//La fin reste inchangée

}

//La classe interne ItemState reste inchangée

class ItemAction implements ActionListener{

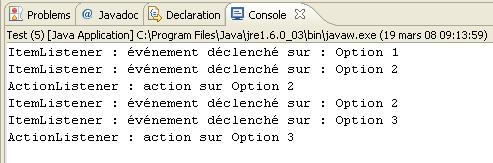
public void actionPerformed(ActionEvent e) {

System.out.println("ActionListener : action sur " + combo.getSelectedItem());

}

}

}



### Changer la forme de l’animation :

On veut faire en sorte qu’en fonction de ce qu’on choisit dans la liste, on modifie la forme.

On doit modifier notre Panneau du coup.

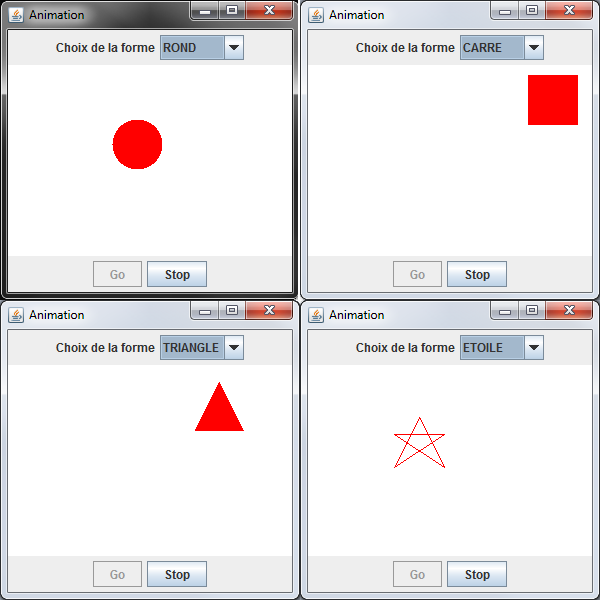
On aura donc une variable de type String qui contiendra le nom de la forme.

* Soit nous intégrons les instructions if dans cette méthode et l'objet Graphics dessinera en fonction de la variable ;
* Soit nous développons une méthode privée appelée dans la méthode paintComponent() et qui dessinera la forme demandée ;
* Soit nous utilisons le pattern strategy afin d'encapsuler la façon dont nous dessinerons nos formes dans notre animation.

Le pattern strategy est de loin la meilleure solution, mais afin de ne pas alourdir nos exemples, nous travaillerons « à l'ancienne ».

On crée une méthode privée draw(Graphics g) on passera l’objet Graphics dans paintComponent ().

On cherche à avoir ça :



##### Classe Panneau

import java.awt.Color;

import java.awt.Font;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

private int posX = -50;

private int posY = -50;

private String forme = "ROND";

public void paintComponent(Graphics g){

//On choisit une couleur de fond pour le rectangle

g.setColor(Color.white);

//On le dessine de sorte qu'il occupe toute la surface

g.fillRect(0, 0, this.getWidth(), this.getHeight());

//On redéfinit une couleur pour le rond

g.setColor(Color.red);

//On délègue la méthode de dessin à la méthode draw()

draw(g);

}

private void draw(Graphics g){

if(this.forme.equals("ROND")){

g.fillOval(posX, posY, 50, 50);

}

if(this.forme.equals("CARRE")){

g.fillRect(posX, posY, 50, 50);

}

if(this.forme.equals("TRIANGLE")){

//Calcul des sommets

//Le sommet 1 se situe à la moitié du côté supérieur du carré

int s1X = posX + 25;

int s1Y = posY;

//Le sommet 2 se situe en bas à droite

int s2X = posX + 50;

int s2Y = posY + 50;

//Le sommet 3 se situe en bas à gauche

int s3X = posX;

int s3Y = posY + 50;

//Nous créons deux tableaux de coordonnées

int[] ptsX = {s1X, s2X, s3X};

int[] ptsY = {s1Y, s2Y, s3Y};

//Nous utilisons la méthode fillPolygon()

g.fillPolygon(ptsX, ptsY, 3);

}

if(this.forme.equals("ETOILE")){

//Pour l'étoile, on se contente de tracer des lignes dans le carré

//correspondant à peu près à une étoile...

//Mais ce code peut être amélioré !

int s1X = posX + 25;

int s1Y = posY;

int s2X = posX + 50;

int s2Y = posY + 50;

g.drawLine(s1X, s1Y, s2X, s2Y);

int s3X = posX;

int s3Y = posY + 17;

g.drawLine(s2X, s2Y, s3X, s3Y);

int s4X = posX + 50;

int s4Y = posY + 17;

g.drawLine(s3X, s3Y, s4X, s4Y);

int s5X = posX;

int s5Y = posY + 50;

g.drawLine(s4X, s4Y, s5X, s5Y);

g.drawLine(s5X, s5Y, s1X, s1Y);

}

}

public void setForme(String form){

this.forme = form;

}

public int getPosX() {

return posX;

}

public void setPosX(int posX) {

this.posX = posX;

}

public int getPosY() {

return posY;

}

public void setPosY(int posY) {

this.posY = posY;

}

}

##### Classe Fenetre

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.ItemEvent;

import java.awt.event.ItemListener;

import javax.swing.JButton;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("Go");

private JButton bouton2 = new JButton("Stop");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Choix de la forme");

private int compteur = 0;

private boolean animated = true;

private boolean backX, backY;

private int x, y;

private Thread t;

private JComboBox combo = new JComboBox();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

bouton.addActionListener(new BoutonListener());

bouton2.addActionListener(new Bouton2Listener());

bouton2.setEnabled(false);

JPanel south = new JPanel();

south.add(bouton);

south.add(bouton2);

container.add(south, BorderLayout.SOUTH);

combo.addItem("ROND");

combo.addItem("CARRE");

combo.addItem("TRIANGLE");

combo.addItem("ETOILE");

combo.addActionListener(new FormeListener());

JPanel top = new JPanel();

top.add(label);

top.add(combo);

container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

private void go(){

x = pan.getPosX();

y = pan.getPosY();

while(this.animated){

if(x < 1) backX = false;

if(x > pan.getWidth() - 50) backX = true;

if(y < 1) backY = false;

if(y > pan.getHeight() - 50) backY = true;

if(!backX) pan.setPosX(++x);

else pan.setPosX(--x);

if(!backY) pan.setPosY(++y);

else pan.setPosY(--y);

pan.repaint();

try {

Thread.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

//Classe écoutant notre bouton

public class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

animated = true;

t = new Thread(new PlayAnimation());

t.start();

bouton.setEnabled(false);

bouton2.setEnabled(true);

}

}

class Bouton2Listener implements ActionListener{

public void actionPerformed(ActionEvent e) {

animated = false;

bouton.setEnabled(true);

bouton2.setEnabled(false);

}

}

class PlayAnimation implements Runnable{

public void run() {

go();

}

}

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//La méthode retourne un Object puisque nous passons des Object dans une liste

//Il faut donc utiliser la méthode toString() pour retourner un String (ou utiliser un cast)

pan.setForme(combo.getSelectedItem().toString());

}

}

}

## Les cases à cocher : objet JCheckBox :

### Première utilisation :

Nous créerons directement une implémentation de l'interface ActionListener, vous connaissez bien la démarche. Contrôlons également que notre objet est coché à l'aide de la méthode isSelected ()qui retourne un booléen. Voici un code mettant tout cela en œuvre :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JCheckBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JCheckBox check1 = new JCheckBox("Case 1");

private JCheckBox check2 = new JCheckBox("Case 2");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

JPanel top = new JPanel();

check1.addActionListener(new StateListener());

check2.addActionListener(new StateListener());

top.add(check1);

top.add(check2);

container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

class StateListener implements ActionListener{

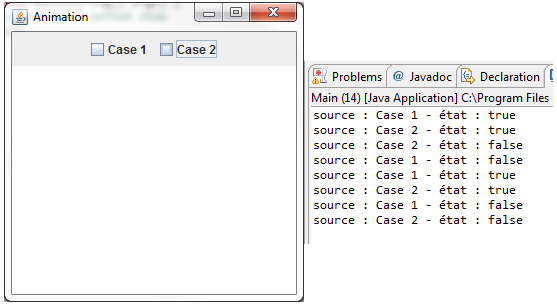
public void actionPerformed(ActionEvent e) {

System.out.println("source : " + ((JCheckBox)e.getSource()).getText() + " - état : " + ((JCheckBox)e.getSource()).isSelected());

}

}

}



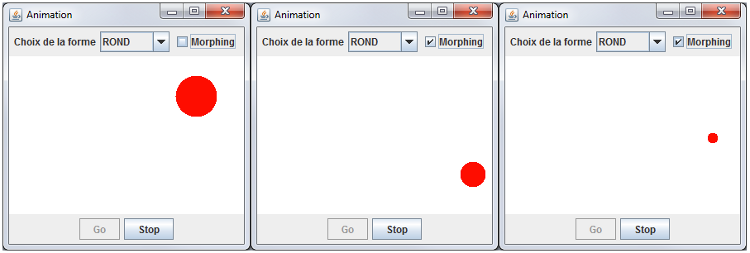
### Un pseudomorphing pour notre animation :

Premièrement, la taille de notre forme est fixe, il nous faut changer cela. Allez, hop, une variable de type int dans notre classe Panneau - disons drawSize - initialisée à 50. Tout comme avec le déplacement, nous devons savoir lorsqu'il faut augmenter ou réduire la taille de notre forme : nous utiliserons donc la même méthode que celle que nous avions développée à ce moment-là.

Un JCheckBox sera nécessaire pour savoir si le « mode morphing » est activé.

Pour bien séparer les deux cas de figure, nous insérerons une deuxième méthode de dessin dans la classe Panneau qui aura pour rôle de dessiner le morphing ; appelons-la drawMorph(Graphics g).

Lorsque nous cocherons la case, le morphing s'activera, et il se désactivera une fois décochée. La classe Panneau devra donc disposer d'un mutateur pour le booléen de morphing.



##### **Fichier** Panneau.java

import java.awt.Color;

import java.awt.Font;

import java.awt.GradientPaint;

import java.awt.Graphics;

import java.awt.Graphics2D;

import javax.swing.JPanel;

public class Panneau extends JPanel {

private int posX = -50;

private int posY = -50;

private int drawSize = 50;

//Un booléen pour le mode morphing

//Un autre pour savoir si la taille doit être réduite

private boolean morph = false, reduce = false;

private String forme = "ROND";

//Le compteur de rafraîchissements

private int increment = 0;

public void paintComponent(Graphics g){

g.setColor(Color.white);

g.fillRect(0, 0, this.getWidth(), this.getHeight());

g.setColor(Color.red);

//Si le mode morphing est activé, on peint le morphing

if(this.morph)

drawMorph(g);

//Sinon, on peint le mode normal

else

draw(g);

}

private void draw(Graphics g){

if(this.forme.equals("ROND")){

g.fillOval(posX, posY, 50, 50);

}

if(this.forme.equals("CARRE")){

g.fillRect(posX, posY, 50, 50);

}

if(this.forme.equals("TRIANGLE")){

int s1X = posX + 50/2;

int s1Y = posY;

int s2X = posX + 50;

int s2Y = posY + 50;

int s3X = posX;

int s3Y = posY + 50;

int[] ptsX = {s1X, s2X, s3X};

int[] ptsY = {s1Y, s2Y, s3Y};

g.fillPolygon(ptsX, ptsY, 3);

}

if(this.forme.equals("ETOILE")){

int s1X = posX + 50/2;

int s1Y = posY;

int s2X = posX + 50;

int s2Y = posY + 50;

g.drawLine(s1X, s1Y, s2X, s2Y);

int s3X = posX;

int s3Y = posY + 50/3;

g.drawLine(s2X, s2Y, s3X, s3Y);

int s4X = posX + 50;

int s4Y = posY + 50/3;

g.drawLine(s3X, s3Y, s4X, s4Y);

int s5X = posX;

int s5Y = posY + 50;

g.drawLine(s4X, s4Y, s5X, s5Y);

g.drawLine(s5X, s5Y, s1X, s1Y);

}

}

//Méthode qui peint le morphing

private void drawMorph(Graphics g){

//On incrémente

increment++;

//On regarde si on doit réduire ou non

if(drawSize >= 50) reduce = true;

if(drawSize <= 10) reduce = false;

if(reduce)

drawSize = drawSize - getUsedSize();

else

drawSize = drawSize + getUsedSize();

if(this.forme.equals("ROND")){

g.fillOval(posX, posY, drawSize, drawSize);

}

if(this.forme.equals("CARRE")){

g.fillRect(posX, posY, drawSize, drawSize);

}

if(this.forme.equals("TRIANGLE")){

int s1X = posX + drawSize/2;

int s1Y = posY;

int s2X = posX + drawSize;

int s2Y = posY + drawSize;

int s3X = posX;

int s3Y = posY + drawSize;

int[] ptsX = {s1X, s2X, s3X};

int[] ptsY = {s1Y, s2Y, s3Y};

g.fillPolygon(ptsX, ptsY, 3);

}

if(this.forme.equals("ETOILE")){

int s1X = posX + drawSize/2;

int s1Y = posY;

int s2X = posX + drawSize;

int s2Y = posY + drawSize;

g.drawLine(s1X, s1Y, s2X, s2Y);

int s3X = posX;

int s3Y = posY + drawSize/3;

g.drawLine(s2X, s2Y, s3X, s3Y);

int s4X = posX + drawSize;

int s4Y = posY + drawSize/3;

g.drawLine(s3X, s3Y, s4X, s4Y);

int s5X = posX;

int s5Y = posY + drawSize;

g.drawLine(s4X, s4Y, s5X, s5Y);

g.drawLine(s5X, s5Y, s1X, s1Y);

}

}

//Retourne le nombre à retrancher ou à ajouter pour le morphing

private int getUsedSize(){

int res = 0;

//Si le nombre de tours est de dix, on réinitialise l'incrément et on retourne 1

if(increment / 10 == 1){

increment = 0;

res = 1;

}

return res;

}

public int getDrawSize(){

return drawSize;

}

public boolean isMorph(){

return morph;

}

public void setMorph(boolean bool){

this.morph = bool;

//On réinitialise la taille

drawSize = 50;

}

public void setForme(String form){

this.forme = form;

}

public int getPosX() {

return posX;

}

public void setPosX(int posX) {

this.posX = posX;

}

public int getPosY() {

return posY;

}

public void setPosY(int posY) {

this.posY = posY;

}

}

##### **Fichier** Fenetre.java

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JButton bouton = new JButton("Go");

private JButton bouton2 = new JButton("Stop");

private JPanel container = new JPanel();

private JLabel label = new JLabel("Choix de la forme");

private int compteur = 0;

private boolean animated = true;

private boolean backX, backY;

private int x, y;

private Thread t;

private JComboBox combo = new JComboBox();

private JCheckBox morph = new JCheckBox("Morphing");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

bouton.addActionListener(new BoutonListener());

bouton2.addActionListener(new Bouton2Listener());

bouton2.setEnabled(false);

JPanel south = new JPanel();

south.add(bouton);

south.add(bouton2);

container.add(south, BorderLayout.SOUTH);

combo.addItem("ROND");

combo.addItem("CARRE");

combo.addItem("TRIANGLE");

combo.addItem("ETOILE");

combo.addActionListener(new FormeListener());

morph.addActionListener(new MorphListener());

JPanel top = new JPanel();

top.add(label);

top.add(combo);

top.add(morph);

container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

private void go(){

x = pan.getPosX();

y = pan.getPosY();

while(this.animated){

//Si le mode morphing est activé, on utilise la taille actuelle de la forme

if(pan.isMorph()){

if(x < 1)backX = false;

if(x > pan.getWidth() - pan.getDrawSize()) backX = true;

if(y < 1)backY = false;

if(y > pan.getHeight() - pan.getDrawSize()) backY = true;

}

//Sinon, on fait comme d'habitude

else{

if(x < 1)backX = false;

if(x > pan.getWidth()-50) backX = true;

if(y < 1)backY = false;

if(y > pan.getHeight()-50) backY = true;

}

if(!backX) pan.setPosX(++x);

else pan.setPosX(--x);

if(!backY) pan.setPosY(++y);

else pan.setPosY(--y);

pan.repaint();

try {

Thread.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

animated = true;

t = new Thread(new PlayAnimation());

t.start();

bouton.setEnabled(false);

bouton2.setEnabled(true);

}

}

class Bouton2Listener implements ActionListener{

public void actionPerformed(ActionEvent e) {

animated = false;

bouton.setEnabled(true);

bouton2.setEnabled(false);

}

}

class PlayAnimation implements Runnable{

public void run() {

go();

}

}

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

pan.setForme(combo.getSelectedItem().toString());

}

}

class MorphListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Si la case est cochée, on active le mode morphing

if(morph.isSelected())pan.setMorph(true);

//Sinon, on ne fait rien

else pan.setMorph(false);

}

}

}

## Objet JRadioButton :

De la même manière :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JCheckBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JRadioButton;

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JRadioButton jr1 = new JRadioButton("Radio 1");

private JRadioButton jr2 = new JRadioButton("Radio 2");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

JPanel top = new JPanel();

jr1.addActionListener(new StateListener());

jr2.addActionListener(new StateListener());

top.add(jr1);

top.add(jr2);

container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

class StateListener implements ActionListener{

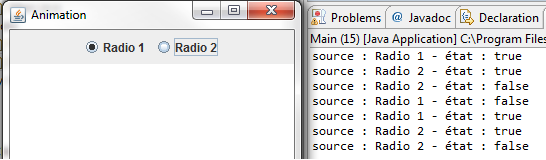
public void actionPerformed(ActionEvent e) {

System.out.println("source : " + ((JRadioButton)e.getSource()).getText() + " - état : " + ((JRadioButton)e.getSource()).isSelected());

}

}

}



## Les champs de texte : objet JTextField :

//Les imports habituels

import javax.swing.JTextField;

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JTextField jtf = new JTextField("Valeur par défaut");

private JLabel label = new JLabel("Un JTextField");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

JPanel top = new JPanel();

Font police = new Font("Arial", Font.BOLD, 14);

jtf.setFont(police);

jtf.setPreferredSize(new Dimension(150, 30));

jtf.setForeground(Color.BLUE);

top.add(label);

top.add(jtf);

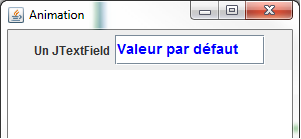
container.add(top, BorderLayout.NORTH);

this.setContentPane(container);

this.setVisible(true);

}

}



Nous pouvons initialiser le contenu avec la méthode setText(String str) ou le récupérer grâce à la méthode getText().

### JFormattedTextField :

Permet de formater ce que l’on accete dans le champ pour éviter de faire des casts.

Exemple, on attend des entiers, donc si on fait getText () on n’aura rien.

//Les imports habituels

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JFormattedTextField jtf = new JFormattedTextField(NumberFormat.getIntegerInstance());

private JFormattedTextField jtf2 = new JFormattedTextField(NumberFormat.getPercentInstance());

private JLabel label = new JLabel("Un JTextField");

private JButton b = new JButton ("OK");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

JPanel top = new JPanel();

Font police = new Font("Arial", Font.BOLD, 14);

jtf.setFont(police);

jtf.setPreferredSize(new Dimension(150, 30));

jtf.setForeground(Color.BLUE);

jtf2.setPreferredSize(new Dimension(150, 30));

b.addActionListener(new BoutonListener());

top.add(label);

top.add(jtf);

top.add(jtf2);

top.add(b);

this.setContentPane(top);

this.setVisible(true);

}

class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

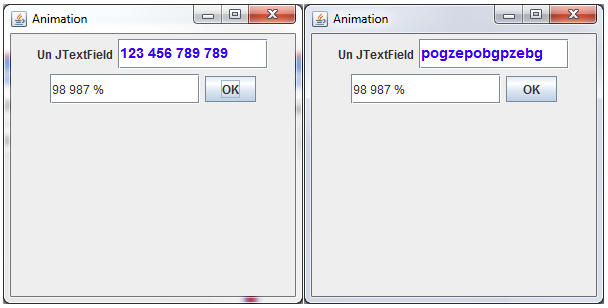
System.out.println("TEXT : jtf " + jtf.getText());

System.out.println("TEXT : jtf2 " + jtf2.getText());

}

}

}



Voici ce qu’on peut utiliser dans ce genre de champ :

* NumberFormat avec :
  + getIntegerInstance()
  + getPercentInstance()
  + getNumberInstance()
* DateFormat avec
  + getTimeInstance()
  + getDateInstance()
* MessageFormat

Vous devez définir ce format avec un paramètre lors de l'instanciation du masque à l'aide de **métacaractères**. Ceux-ci indiquent à votre objet MaskFormatter ce que le contenu de votre zone de texte contiendra. Voici la liste de ces métacaractères :

* # : indique un chiffre ;
* ' : indique un caractère d'échappement ;
* U : indique une lettre (les minuscules sont automatiquement changées en majuscules) ;
* L : indique une lettre (les majuscules sont automatiquement changées en minuscules) ;
* A : indique un chiffre ou une lettre ;
* ? : indique une lettre ;
* \* : indique que tous les caractères sont acceptés ;
* H : indique que tous les caractères hexadécimaux sont acceptés (0 à 9, a à f et A à F).

L'instanciation d'un tel objet peut lever une ParseException. Vous devez donc l'entourer d'un bloc try{…}catch(ParseException e){…}.

Voici à quoi ressemblerait un format téléphonique :

try{

MaskFormatter tel = new MaskFormatter("## ## ## ## ##");

//Ou encore

MaskFormatter tel2 = new MaskFormatter("##-##-##-##-##");

//Vous pouvez ensuite le passer à votre zone de texte

JFormattedTextField jtf = new JFormattedTextField(tel2);

}catch(ParseException e){e.printStackTrace();}

Comme le numéro de téléphone peut ne pas ête valide, on doit faire des vérif …:

* tester chaque élément du numéro ;
* tester le numéro en entier ;
* dans le cas où nous n'utilisons pas de MaskFormatter, vérifier en plus que les saisies sont numériques ;
* utiliser une expression régulière ;
* empêcher la saisie d'un type de caractères ;
* etc.

## Contrôle du clavier, interface KeyListener :

Nous connaissons déjà :

* l'interface MouseListener qui interagit avec la souris ;
* l'interface ActionListener qui interagit lors d'un clic sur un composant ;
* l'interface ItemListener qui écoute les événements sur une liste déroulante.

Voici à présent l'interface KeyListener. Comme l'indique le titre, elle nous permet d'intercepter les événements clavier lorsque l'on :

* presse une touche ;
* relâche une touche ;
* tape sur une touche.

Vous constatez qu'il y en a trois :

* keyPressed(KeyEvent event), appelée lorsqu'on presse une touche ;
* keyReleased(KeyEvent event), appelée lorsqu'on relâche une touche (c'est à ce moment que le composant se voit affecter la valeur de la touche) ;
* keyTyped(KeyEvent event), appelée entre les deux méthodes citées ci-dessus.

Comme vous vous en doutez, l'objet KeyEvent nous permettra d'obtenir des informations sur les touches qui ont été utilisées. Parmi celles-ci, nous utiliserons :

* getKeyCode() : retourne le code de la touche ;
* getKeyChar() : retourne le caractère correspondant à la touche.

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.Font;

import java.awt.event.KeyEvent;

import java.awt.event.KeyListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JTextField;

public class Fenetre extends JFrame {

private JPanel container = new JPanel();

private JTextField jtf;

private JLabel label = new JLabel("Téléphone FR");

private JButton b = new JButton ("OK");

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 150);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

jtf = new JTextField();

JPanel top = new JPanel();

Font police = new Font("Arial", Font.BOLD, 14);

jtf.setFont(police);

jtf.setPreferredSize(new Dimension(150, 30));

jtf.setForeground(Color.BLUE);

//On ajoute l'écouteur à notre composant

jtf.addKeyListener(new ClavierListener());

top.add(label);

top.add(jtf);

top.add(b);

this.setContentPane(top);

this.setVisible(true);

}

class ClavierListener implements KeyListener{

public void keyPressed(KeyEvent event) {

System.out.println("Code touche pressée : " + event.getKeyCode() + " - caractère touche pressée : " + event.getKeyChar());

pause();

}

public void keyReleased(KeyEvent event) {

System.out.println("Code touche relâchée : " + event.getKeyCode() + " - caractère touche relâchée : " + event.getKeyChar());

pause();

}

public void keyTyped(KeyEvent event) {

System.out.println("Code touche tapée : " + event.getKeyCode() + " - caractère touche tapée : " + event.getKeyChar());

pause();

}

}

private void pause(){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

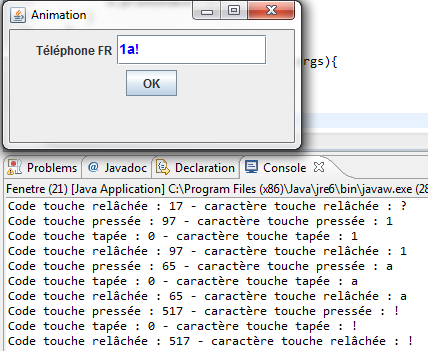
}

public static void main(String[] args){

new Fenetre();

}

}



Dans le cas qui nous intéresse, nous souhaitons que lorsque l'utilisateur saisit un caractère interdit, celui-ci soit automatiquement retiré de la zone de saisie. Pour cela, nous procéderons à un traitement spécifique dans la méthode keyReleased(KeyEvent event).

À partir de là, c'est simple : il nous suffit de supprimer le caractère tapé de la zone de saisie si son code n'est pas compris dans cet intervalle. Toutefois, un problème se pose avec cette méthode : ceux qui possèdent un ordinateur portable sans pavé numérique ne pourront rien saisir alors qu'il est possible d'obtenir des chiffres en appuyant sur **MAJ** + **&**, **é**, **'**, **(** ou **-**.

Ce souci nous amène à opter pour une autre solution : nous créerons une méthode dont le type de retour sera un booléen nous indiquant si la saisie est numérique ou non. Comment ? Tout simplement en exécutant un Integer.parseInt(value), le tout enveloppé dans un try{…}catch(NumberFormatException ex){}. Si nous essayons de convertir un caractère « a » en entier, l'exception sera levée et nous retournerons alors false (true dans le cas contraire).

La méthode parseInt() prend un String en paramètre. La méthode getKeyChar(), elle, renvoie un char. Il faudra donc penser à faire la conversion.

Voici notre implémentation quelque peu modifiée :

class ClavierListener implements KeyListener{

public void keyReleased(KeyEvent event) {

if(!isNumeric(event.getKeyChar()))

jtf.setText(jtf.getText().replace(String.valueOf(event.getKeyChar()), ""));

}

//Inutile de redéfinir ces méthodes, ous n'en avons plus l'utilité !

public void keyPressed(KeyEvent event) {}

public void keyTyped(KeyEvent event) {}

//Retourne true si le paramètre est numérique, false dans le cas contraire

private boolean isNumeric(char carac){

try {

Integer.parseInt(String.valueOf(carac));

}

catch (NumberFormatException e) {

return false;

}

return true;

}

}

# Les menus et boites de dialogues :

## Les boîtes de dialogues :

Ce sont de petites boites qui servent à :

* afficher une information (message d'erreur, d'avertissement…) ;
* demander une validation, une réfutation ou une annulation ;
* demander à l'utilisateur de saisir une information dont le système a besoin ;
* etc.

### Les boites d’information :

On utilise JOptionpane ().

JOptionPane jop1, jop2, jop3;

//Boîte du message d'information

jop1 = new JOptionPane();

jop1.showMessageDialog(null, "Message informatif", "Information", JOptionPane.INFORMATION\_MESSAGE);

//Boîte du message préventif

jop2 = new JOptionPane();

jop2.showMessageDialog(null, "Message préventif", "Attention", JOptionPane.WARNING\_MESSAGE);

//Boîte du message d'erreur

jop3 = new JOptionPane();

jop3.showMessageDialog(null, "Message d'erreur", "Erreur", JOptionPane.ERROR\_MESSAGE);

Les boîtes n’apparaissent pas en même temps parce qu’une boite de dialogue en programmation (tout langage confondu) est dite modale, c’est-à-dire que son apparition bloque le programme.

Maintenant, voyons de plus près comment construire un tel objet. Ici, nous avons utilisé la méthode showMessageDialog(Component parentComponent, String message, String title, int messageType); où :

* Component parentComponent : correspond au composant parent ; ici, il n'y en a aucun, nous mettons donc null.
* String message : permet de renseigner le message à afficher dans la boîte de dialogue.
* String title : permet de donner un titre à l'objet.
* int messageType : permet de savoir s'il s'agit d'un message d'information, de prévention ou d'erreur. Vous avez sans doute remarqué que, mis à part le texte et le titre, seul ce champ variait entre nos trois objets !



On peut redéfinir ces images :

import javax.swing.ImageIcon;

import javax.swing.JOptionPane;

public class Test {

public static void main(String[] args) {

JOptionPane jop1, jop2, jop3;

jop1 = new JOptionPane();

ImageIcon img = new ImageIcon("images/information.png");

jop1.showMessageDialog(null, "Message informatif", "Information", JOptionPane.INFORMATION\_MESSAGE, img);

jop2 = new JOptionPane();

img = new ImageIcon("images/warning.png");

jop2.showMessageDialog(null, "Message préventif", "Attention", JOptionPane.WARNING\_MESSAGE, img);

jop3 = new JOptionPane();

img = new ImageIcon("images/erreur.png");

jop3.showMessageDialog(null, "Message d'erreur", "Erreur", JOptionPane.ERROR\_MESSAGE, img);

}

}



Voici les types de boîtes que vous pouvez afficher (ces types restent valables pour tout ce qui suit) :

* JOptionPane.ERROR\_MESSAGE
* JOptionPane.INFORMATION\_MESSAGE
* JOptionPane.PLAIN\_MESSAGE
* JOptionPane.QUESTION\_MESSAGE
* JOptionPane.WARNING\_MESSAGE

### Les boîtes de confirmation :

Nous utiliserons toujours l'objet JOptionPane, mais ce sera cette fois avec la méthode showConfirmDialog(), une méthode qui retourne un entier correspondant à l'option que vous aurez choisie dans cette boîte :

* Yes ;
* No ;
* Cancel.

Exemple :

//…

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null, "Voulez-vous lancer l'animation ?", "Lancement de l'animation", JOptionPane.YES\_NO\_OPTION, JOptionPane.QUESTION\_MESSAGE);

if(option == JOptionPane.OK\_OPTION){

animated = true;

t = new Thread(new PlayAnimation());

t.start();

bouton.setEnabled(false);

bouton2.setEnabled(true);

}

//…

//…

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null, "Voulez-vous arrêter l'animation ?", "Arrêt de l'animation", JOptionPane.YES\_NO\_CANCEL\_OPTION, JOptionPane.QUESTION\_MESSAGE);

if(option != JOptionPane.NO\_OPTION &&

option != JOptionPane.CANCEL\_OPTION &&

option != JOptionPane.CLOSED\_OPTION){

animated = false;

bouton.setEnabled(true);

bouton2.setEnabled(false);

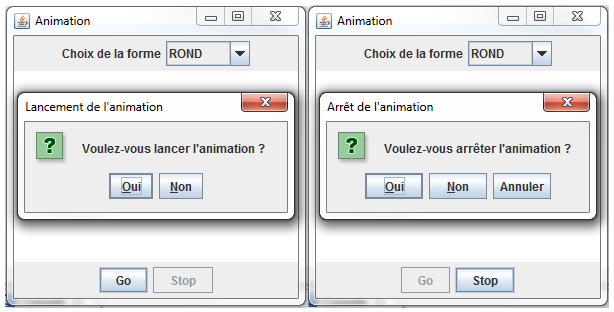
}

Voyons ce qu'il se passe ici :

* nous initialisons notre objet JOptionPane : rien d'étonnant ;
* en revanche, plutôt que d'afficher directement la boîte, nous affectons le résultat que renvoie la méthode showConfirmDialog() à une variable de type int ;
* nous nous servons de cette variable afin de savoir quel bouton a été cliqué (oui ou non).

En fait, lorsque vous cliquez sur l'un des deux boutons présents dans cette boîte, vous pouvez affecter une valeur de type int :

* correspondant à l'entier JOptionPane.OK\_OPTION, qui vaut 0 (JOptionPane.YES\_OPTION a la même valeur);
* correspondant à l'entier JOptionPane.NO\_OPTION, qui vaut 1 ;
* correspondant à l'entier JOptionPane.CANCEL\_OPTION pour la boîte apparaissant lors du clic sur « **Stop** », qui vaut 2 ;
* correspondant à l'entier JOptionPane.CLOSED\_OPTION pour la même boîte que ci-dessus et qui vaut -1.



### Les boîtes de saisie :

Ici, nous allons utiliser la méthode showInputDialog(Component parent, String message, String title, int messageType), qui retourne une chaîne de caractères.

import javax.swing.JOptionPane;

public class Test {

public static void main(String[] args) {

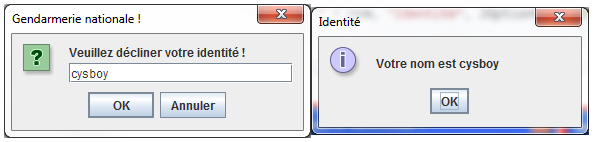
JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();

String nom = jop.showInputDialog(null, "Veuillez décliner votre identité !", "Gendarmerie nationale !", JOptionPane.QUESTION\_MESSAGE);

jop2.showMessageDialog(null, "Votre nom est " + nom, "Identité", JOptionPane.INFORMATION\_MESSAGE);

}

}



Maintenant, voyons comment on intègre une liste dans une boîte de ce genre. Vous allez voir, c'est simplissime !

import javax.swing.JOptionPane;

public class Test {

public static void main(String[] args) {

String[] sexe = {"masculin", "féminin", "indéterminé"};

JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();

String nom = (String)jop.showInputDialog(null,

"Veuillez indiquer votre sexe !",

"Gendarmerie nationale !",

JOptionPane.QUESTION\_MESSAGE,

null,

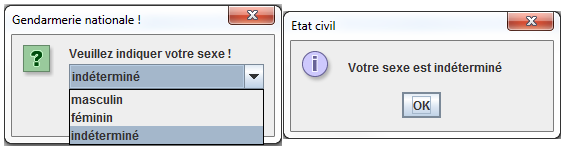
sexe,

sexe[2]);

jop2.showMessageDialog(null, "Votre sexe est " + nom, "Etat civil", JOptionPane.INFORMATION\_MESSAGE);

}

}



Voici un petit détail des paramètres utilisés dans cette méthode :

* les quatre premiers, vous connaissez ;
* le deuxième null correspond à l'icône que vous souhaitez passer ;
* ensuite, vous devez passer un tableau de String afin de remplir la combo (l'objet JComboBox) de la boîte ;
* le dernier paramètre correspond à la valeur par défaut de la liste déroulante.

Une variante avec showOptionDialog ().

Ce type de boîte propose un choix de boutons correspondant aux éléments passés en paramètres (tableau de String) au lieu d'une combo ; elle prend aussi une valeur par défaut, mais retourne l'indice de l'élément dans la liste au lieu de l'élément lui-même.

import javax.swing.JOptionPane;

public class Test {

public static void main(String[] args) {

String[] sexe = {"masculin", "féminin", "indéterminé"};

JOptionPane jop = new JOptionPane(), jop2 = new JOptionPane();

int rang = jop.showOptionDialog(null,

"Veuillez indiquer votre sexe !",

"Gendarmerie nationale !",

JOptionPane.YES\_NO\_CANCEL\_OPTION,

JOptionPane.QUESTION\_MESSAGE,

null,

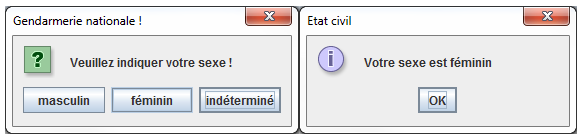
sexe,

sexe[2]);

jop2.showMessageDialog(null, "Votre sexe est " + sexe[rang], "Etat civil", JOptionPane.INFORMATION\_MESSAGE);

}

}



### Boîtes de dialogues personnalisées :

Les boîtes de dialogues héritent de la classe *JDialog*.

Commençons par créer un nouveau projet. Créez une nouvelle classe dans Eclipse, appelons-la ZDialog, faites-la hériter de la classe citée précédemment, et mettez-y le code suivant :

import javax.swing.JDialog;

import javax.swing.JFrame;

public class ZDialog extends JDialog {

public ZDialog(JFrame parent, String title, boolean modal){

//On appelle le construteur de JDialog correspondant

super(parent, title, modal);

//On spécifie une taille

this.setSize(200, 80);

//La position

this.setLocationRelativeTo(null);

//La boîte ne devra pas être redimensionnable

this.setResizable(false);

//Enfin on l'affiche

this.setVisible(true);

//Tout ceci ressemble à ce que nous faisons depuis le début avec notre JFrame.

}

}

Maintenant, créons une classe qui va tester notre ZDialog :

import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

public class Fenetre extends JFrame {

private JButton bouton = new JButton("Appel à la ZDialog");

public Fenetre(){

this.setTitle("Ma JFrame");

this.setSize(300, 100);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.getContentPane().setLayout(new FlowLayout());

this.getContentPane().add(bouton);

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

ZDialog zd = new ZDialog(null, "Coucou les ZérOs", true);

}

});

this.setVisible(true);

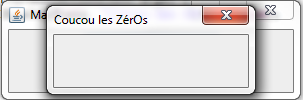
}

public static void main(String[] main){

Fenetre fen = new Fenetre();

}

}



* JFrame Parent correspond à l'objet parent ;
* String title correspond au titre de notre boîte ;
* boolean modal correspond à la modalité ; true : boîte modale, false : boîte non modale.

Partons du fait que notre boîte comprendra un bouton OK et un bouton Annuler : dans le cas où l'utilisateur clique sur OK, on récupère les informations, si l'utilisateur clique sur Annuler, on ne récupère rien. Et il faudra aussi tenir compte de la modalité de notre boîte : la méthode setVisible(false); met fin au dialogue ! Ceci signifie également que le dialogue s'entame au moment où l'instruction setVisible(true); est exécutée. C'est pourquoi nous allons sortir cette instruction du constructeur de l'objet et la mettre dans une méthode à part.

Maintenant, il faut que l'on puisse indiquer à notre boîte de renvoyer les informations ou non. C'est pour cela que nous allons utiliser un booléen - appelons-le sendData - initialisé à false, mais qui passera à true si on clique sur OK.

//Cas où notre ZDialog renverra le contenu

//D'un JTextField nommé jtf

public String showZDialog(){

this.sendData = false;

//Début du dialogue

this.setVisible(true);

//Le dialogue prend fin

//Si on a cliqué sur OK, on envoie, sinon on envoie une chaîne vide !

return (this.sendData)? jtf.getText() : "";

}

Comment récupérer les infos avec plusieurs informations à saisir ?

C'est vrai qu'on ne peut retourner qu'une valeur à la fois. Mais il peut y avoir plusieurs solutions à ce problème :

* Dans le cas où nous n'avons qu'un composant, nous pouvons adapter la méthode showZDialog() au type de retour du composant utilisé.
* Dans notre cas, nous voulons plusieurs composants, donc plusieurs valeurs. Vous pouvez :
  + retourner une collection de valeurs (ArrayList) ;
  + faire des accesseurs dans votre ZDialog ;
  + créer un objet dont le rôle est de collecter les informations dans votre boîte et de retourner cet objet ;
  + etc.

Nous allons opter pour un objet qui collectera les informations et que nous retournerons à la fin de la méthode showZDialog(). Avant de nous lancer dans sa création, nous devons savoir ce que nous allons mettre dans notre boîte… J'ai choisi de vous faire programmer une boîte permettant de spécifier les caractéristiques d'un personnage de jeu vidéo :

* son nom (un champ de saisie) ;
* son sexe (une combo) ;
* sa taille (un champ de saisie) ;
* sa couleur de cheveux (une combo) ;
* sa tranche d'âge (des boutons radios).

public class ZDialogInfo {

private String nom, sexe, age, cheveux, taille;

public ZDialogInfo(){}

public ZDialogInfo(String nom, String sexe, String age, String cheveux, String taille){

this.nom = nom;

this.sexe = sexe;

this.age = age;

this.cheveux = cheveux;

this.taille = taille;

}

public String toString(){

String str;

if(this.nom != null && this.sexe != null && this.taille != null && this.age != null && this.cheveux != null){

str = "Description de l'objet InfoZDialog";

str += "Nom : " + this.nom + "\n";

str += "Sexe : " + this.sexe + "\n";

str += "Age : " + this.age + "\n";

str += "Cheveux : " + this.cheveux + "\n";

str += "Taille : " + this.taille + "\n";

}

else{

str = "Aucune information !";

}

return str;

}

}

Voici le code source de notre boîte perso :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.BorderFactory;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JComboBox;

import javax.swing.JDialog;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JRadioButton;

import javax.swing.ButtonGroup;

import javax.swing.JTextField;

public class ZDialog extends JDialog {

private ZDialogInfo zInfo = new ZDialogInfo();

private boolean sendData;

private JLabel nomLabel, sexeLabel, cheveuxLabel, ageLabel, tailleLabel,taille2Label, icon;

private JRadioButton tranche1, tranche2, tranche3, tranche4;

private JComboBox sexe, cheveux;

private JTextField nom, taille;

public ZDialog(JFrame parent, String title, boolean modal){

super(parent, title, modal);

this.setSize(550, 270);

this.setLocationRelativeTo(null);

this.setResizable(false);

this.setDefaultCloseOperation(JDialog.DO\_NOTHING\_ON\_CLOSE);

this.initComponent();

}

public ZDialogInfo showZDialog(){

this.sendData = false;

this.setVisible(true);

return this.zInfo;

}

private void initComponent(){

//Icône

icon = new JLabel(new ImageIcon("images/icone.jpg"));

JPanel panIcon = new JPanel();

panIcon.setBackground(Color.white);

panIcon.setLayout(new BorderLayout());

panIcon.add(icon);

//Le nom

JPanel panNom = new JPanel();

panNom.setBackground(Color.white);

panNom.setPreferredSize(new Dimension(220, 60));

nom = new JTextField();

nom.setPreferredSize(new Dimension(100, 25));

panNom.setBorder(BorderFactory.createTitledBorder("Nom du personnage"));

nomLabel = new JLabel("Saisir un nom :");

panNom.add(nomLabel);

panNom.add(nom);

//Le sexe

JPanel panSexe = new JPanel();

panSexe.setBackground(Color.white);

panSexe.setPreferredSize(new Dimension(220, 60));

panSexe.setBorder(BorderFactory.createTitledBorder("Sexe du personnage"));

sexe = new JComboBox();

sexe.addItem("Masculin");

sexe.addItem("Féminin");

sexe.addItem("Indéterminé");

sexeLabel = new JLabel("Sexe : ");

panSexe.add(sexeLabel);

panSexe.add(sexe);

//L'âge

JPanel panAge = new JPanel();

panAge.setBackground(Color.white);

panAge.setBorder(BorderFactory.createTitledBorder("Age du personnage"));

panAge.setPreferredSize(new Dimension(440, 60));

tranche1 = new JRadioButton("15 - 25 ans");

tranche1.setSelected(true);

tranche2 = new JRadioButton("26 - 35 ans");

tranche3 = new JRadioButton("36 - 50 ans");

tranche4 = new JRadioButton("+ de 50 ans");

ButtonGroup bg = new ButtonGroup();

bg.add(tranche1);

bg.add(tranche2);

bg.add(tranche3);

bg.add(tranche4);

panAge.add(tranche1);

panAge.add(tranche2);

panAge.add(tranche3);

panAge.add(tranche4);

//La taille

JPanel panTaille = new JPanel();

panTaille.setBackground(Color.white);

panTaille.setPreferredSize(new Dimension(220, 60));

panTaille.setBorder(BorderFactory.createTitledBorder("Taille du personnage"));

tailleLabel = new JLabel("Taille : ");

taille2Label = new JLabel(" cm");

taille = new JTextField("180");

taille.setPreferredSize(new Dimension(90, 25));

panTaille.add(tailleLabel);

panTaille.add(taille);

panTaille.add(taille2Label);

//La couleur des cheveux

JPanel panCheveux = new JPanel();

panCheveux.setBackground(Color.white);

panCheveux.setPreferredSize(new Dimension(220, 60));

panCheveux.setBorder(BorderFactory.createTitledBorder("Couleur de cheveux du personnage"));

cheveux = new JComboBox();

cheveux.addItem("Blond");

cheveux.addItem("Brun");

cheveux.addItem("Roux");

cheveux.addItem("Blanc");

cheveuxLabel = new JLabel("Cheveux");

panCheveux.add(cheveuxLabel);

panCheveux.add(cheveux);

JPanel content = new JPanel();

content.setBackground(Color.white);

content.add(panNom);

content.add(panSexe);

content.add(panAge);

content.add(panTaille);

content.add(panCheveux);

JPanel control = new JPanel();

JButton okBouton = new JButton("OK");

okBouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

zInfo = new ZDialogInfo(nom.getText(), (String)sexe.getSelectedItem(), getAge(), (String)cheveux.getSelectedItem() ,getTaille());

setVisible(false);

}

public String getAge(){

return (tranche1.isSelected()) ? tranche1.getText() :

(tranche2.isSelected()) ? tranche2.getText() :

(tranche3.isSelected()) ? tranche3.getText() :

(tranche4.isSelected()) ? tranche4.getText() :

tranche1.getText();

}

public String getTaille(){

return (taille.getText().equals("")) ? "180" : taille.getText();

}

});

JButton cancelBouton = new JButton("Annuler");

cancelBouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

setVisible(false);

}

});

control.add(okBouton);

control.add(cancelBouton);

this.getContentPane().add(panIcon, BorderLayout.WEST);

this.getContentPane().add(content, BorderLayout.CENTER);

this.getContentPane().add(control, BorderLayout.SOUTH);

}

}

J'ai ajouté une image, mais vous n'y êtes nullement obligés ! Voici le code source permettant de tester cette boîte :

import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JOptionPane;

public class Fenetre extends JFrame {

private JButton bouton = new JButton("Appel à la ZDialog");

public Fenetre(){

this.setTitle("Ma JFrame");

this.setSize(300, 100);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.getContentPane().setLayout(new FlowLayout());

this.getContentPane().add(bouton);

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

ZDialog zd = new ZDialog(null, "Coucou les ZérOs", true);

ZDialogInfo zInfo = zd.showZDialog();

JOptionPane jop = new JOptionPane();

jop.showMessageDialog(null, zInfo.toString(), "Informations personnage", JOptionPane.INFORMATION\_MESSAGE);

}

});

this.setVisible(true);

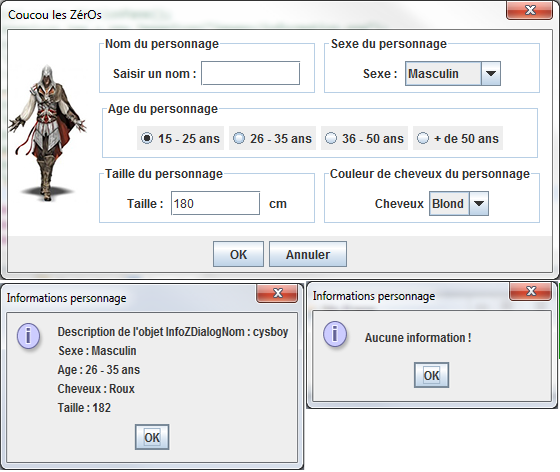
}

public static void main(String[] main){

Fenetre fen = new Fenetre();

}

}



## Les menus

Vous vous rappelez que j'ai mentionné qu'une MenuBar fait partie de la composition de l'objet JFrame. Le moment est venu pour vous d'utiliser un composant de ce genre. Néanmoins, celui-ci appartient au package java.awt. Dans ce chapitre nous utiliserons son homologue, l'objet JMenuBar, issu dans le package javax.swing. Pour travailler avec des menus, nous aurons besoin :

* de l'objet JMenu, le titre principal d'un point de menu (Fichier, Édition…) ;
* d'objets JMenuItem, les éléments composant nos menus.

On utilisera ActionListener pour executer ce que l’on selectionne dans les JMenuItem.

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;

import javax.swing.JCheckBoxMenuItem;

import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JRadioButtonMenuItem;

public class ZFenetre extends JFrame {

private JMenuBar menuBar = new JMenuBar();

private JMenu test1 = new JMenu("Fichier");

private JMenu test1\_2 = new JMenu("Sous ficher");

private JMenu test2 = new JMenu("Edition");

private JMenuItem item1 = new JMenuItem("Ouvrir");

private JMenuItem item2 = new JMenuItem("Fermer");

private JMenuItem item3 = new JMenuItem("Lancer");

private JMenuItem item4 = new JMenuItem("Arrêter");

private JCheckBoxMenuItem jcmi1 = new JCheckBoxMenuItem("Choix 1");

private JCheckBoxMenuItem jcmi2 = new JCheckBoxMenuItem("Choix 2");

private JRadioButtonMenuItem jrmi1 = new JRadioButtonMenuItem("Radio 1");

private JRadioButtonMenuItem jrmi2 = new JRadioButtonMenuItem("Radio 2");

public static void main(String[] args){

ZFenetre zFen = new ZFenetre();

}

public ZFenetre(){

this.setSize(400, 200);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

//On initialise nos menus

this.test1.add(item1);

//On ajoute les éléments dans notre sous-menu

this.test1\_2.add(jcmi1);

this.test1\_2.add(jcmi2);

//Ajout d'un séparateur

this.test1\_2.addSeparator();

//On met nos radios dans un ButtonGroup

ButtonGroup bg = new ButtonGroup();

bg.add(jrmi1);

bg.add(jrmi1);

//On présélectionne la première radio

jrmi1.setSelected(true);

this.test1\_2.add(jrmi1);

this.test1\_2.add(jrmi2);

//Ajout du sous-menu dans notre menu

this.test1.add(this.test1\_2);

//Ajout d'un séparateur

this.test1.addSeparator();

item2.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

System.exit(0);

}

});

this.test1.add(item2);

this.test2.add(item3);

this.test2.add(item4);

//L'ordre d'ajout va déterminer l'ordre d'apparition dans le menu de gauche à droite

//Le premier ajouté sera tout à gauche de la barre de menu et inversement pour le dernier

this.menuBar.add(test1);

this.menuBar.add(test2);

this.setJMenuBar(menuBar);

this.setVisible(true);

}

}

L'action attachée au JMenutItemFermer permet de quitter l'application. Ce que donne le code est affiché à la figure suivante.



Application à notre programme avec la balle :

import java.awt.BorderLayout;

import java.awt.Color;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.ButtonGroup;

import javax.swing.JButton;

import javax.swing.JCheckBox;

import javax.swing.JCheckBoxMenuItem;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JRadioButtonMenuItem;

public class Fenetre extends JFrame{

private Panneau pan = new Panneau();

private JPanel container = new JPanel();

private int compteur = 0;

private boolean animated = true;

private boolean backX, backY;

private int x,y ;

private Thread t;

private JMenuBar menuBar = new JMenuBar();

private JMenu animation = new JMenu("Animation"),

forme = new JMenu("Forme"),

typeForme = new JMenu("Type de forme"),

aPropos = new JMenu("À propos");

private JMenuItem lancer = new JMenuItem("Lancer l'animation"),

arreter = new JMenuItem("Arrêter l'animation"),

quitter = new JMenuItem("Quitter"),

aProposItem = new JMenuItem("?");

private JCheckBoxMenuItem morph = new JCheckBoxMenuItem("Morphing");

private JRadioButtonMenuItem carre = new JRadioButtonMenuItem("Carré"),

rond = new JRadioButtonMenuItem("Rond"),

triangle = new JRadioButtonMenuItem("Triangle"),

etoile = new JRadioButtonMenuItem("Etoile");

private ButtonGroup bg = new ButtonGroup();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

container.add(pan, BorderLayout.CENTER);

this.setContentPane(container);

this.initMenu();

this.setVisible(true);

}

private void initMenu(){

//Menu animation

animation.add(lancer);

arreter.setEnabled(false);

animation.add(arreter);

animation.addSeparator();

//Pour quitter l'application

quitter.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

System.exit(0);

}

});

animation.add(quitter);

//Menu forme

bg.add(carre);

bg.add(triangle);

bg.add(rond);

bg.add(etoile);

typeForme.add(rond);

typeForme.add(carre);

typeForme.add(triangle);

typeForme.add(etoile);

rond.setSelected(true);

forme.add(typeForme);

forme.add(morph);

//Menu À propos

aPropos.add(aProposItem);

//Ajout des menus dans la barre de menus

menuBar.add(animation);

menuBar.add(forme);

menuBar.add(aPropos);

//Ajout de la barre de menus sur la fenêtre

this.setJMenuBar(menuBar);

}

private void go(){

//Rien n'a changé ici

}

public class BoutonListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null,

"Voulez-vous lancer l'animation ?",

"Lancement de l'animation",

JOptionPane.YES\_NO\_OPTION,

JOptionPane.QUESTION\_MESSAGE);

if(option == JOptionPane.OK\_OPTION){

lancer.setEnabled(false);

arreter.setEnabled(true);

animated = true;

t = new Thread(new PlayAnimation());

t.start();

}

}

}

class Bouton2Listener implements ActionListener{

public void actionPerformed(ActionEvent e) {

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null,

"Voulez-vous arrêter l'animation ?",

"Arrêt de l'animation",

JOptionPane.YES\_NO\_CANCEL\_OPTION,

JOptionPane.QUESTION\_MESSAGE);

if(option != JOptionPane.NO\_OPTION && option != JOptionPane.CANCEL\_OPTION && option != JOptionPane.CLOSED\_OPTION){

animated = false;

//On remplace nos boutons par nos JMenuItem

lancer.setEnabled(true);

arreter.setEnabled(false);

}

}

}

class PlayAnimation implements Runnable{

public void run() {

go();

}

}

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//On commente cette ligne pour l'instant

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//pan.setForme(combo.getSelectedItem().toString());

}

}

class MorphListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Si la case est cochée, activation du mode morphing

if(morph.isSelected())pan.setMorph(true);

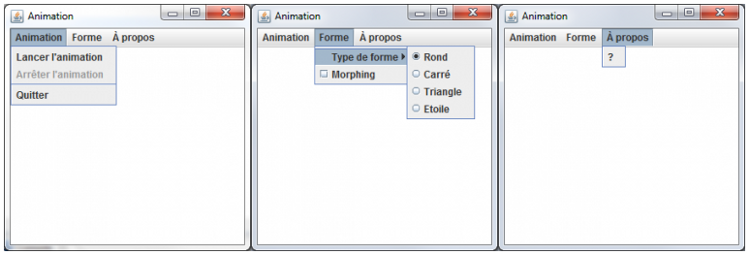
//Sinon rien !

else pan.setMorph(false);

}

}

}



On doit indiquer à nos Menuitem qu’on les écoute pour qu’ils agissent sur le programme.

Afin que l'application fonctionne bien, j'ai apporté deux modifications mineures dans la classe Panneau. J'ai ajouté une instruction dans une condition :

//J'ai ajouté : || this.forme.equals("CARRÉ")

if(this.forme.equals("CARRE") || this.forme.equals("CARRÉ")){

g.fillRect(posX, posY, 50, 50);

}

… ainsi, on accepte les deux graphies ! J'ai également ajouté un toUpperCase() :

public void setForme(String form){

this.forme = form.toUpperCase();

}

Ainsi, on s'assure que cette chaîne de caractères est en majuscules.

Voici le code de notre animation avec un beau menu pour tout contrôler :

//Les imports

public class Fenetre extends JFrame{

//La déclaration des variables reste inchangée

public Fenetre(){

//Le constructeur est inchangé

}

private void initMenu(){

//Menu Animation

//Ajout du listener pour lancer l'animation

lancer.addActionListener(new StartAnimationListener());

animation.add(lancer);

//Ajout du listener pour arrêter l'animation

arreter.addActionListener(new StopAnimationListener());

arreter.setEnabled(false);

animation.add(arreter);

animation.addSeparator();

quitter.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

System.exit(0);

}

});

animation.add(quitter);

//Menu Forme

bg.add(carre);

bg.add(triangle);

bg.add(rond);

bg.add(etoile);

//On crée un nouvel écouteur, inutile de créer 4 instances différentes

FormeListener fl = new FormeListener();

carre.addActionListener(fl);

rond.addActionListener(fl);

triangle.addActionListener(fl);

etoile.addActionListener(fl);

typeForme.add(rond);

typeForme.add(carre);

typeForme.add(triangle);

typeForme.add(etoile);

rond.setSelected(true);

forme.add(typeForme);

//Ajout du listener pour le morphing

morph.addActionListener(new MorphListener());

forme.add(morph);

//Menu À propos

//Ajout de ce que doit faire le "?"

aProposItem.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

JOptionPane jop = new JOptionPane();

ImageIcon img = new ImageIcon("images/cysboy.gif");

String mess = "Merci ! \n J'espère que vous vous amusez bien !\n";

mess += "Je crois qu'il est temps d'ajouter des accélérateurs et des "+" mnémoniques dans tout ça…\n";

mess += "\n Allez, GO les ZérOs !";

jop.showMessageDialog(null, mess, "À propos", JOptionPane.INFORMATION\_MESSAGE, img);

}

});

aPropos.add(aProposItem);

//Ajout des menus dans la barre de menus

menuBar.add(animation);

menuBar.add(forme);

menuBar.add(aPropos);

//Ajout de la barre de menus sur la fenêtre

this.setJMenuBar(menuBar);

}

private void go(){

//Idem

}

public class StartAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

//Idem

}

}

/\*\*

\* Écouteur du menu Quitter

\* @author CHerby

\*/

class StopAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Idem

}

}

class PlayAnimation implements Runnable{

public void run() {

go();

}

}

/\*\*

\* Écoute les menus Forme

\* @author CHerby

\*/

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());

}

}

/\*\*

\* Écoute le menu Morphing

\* @author CHerby

\*/

class MorphListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Si la case est cochée, activation du mode morphing

if(morph.isSelected()) pan.setMorph(true);

//Sinon rien !

else pan.setMorph(false);

}

}

}

### Les raccourcis clavier :

On utilie un Mnemonic pour simuler un clique à un endroit du menu : *setMnemonic ()*.

Ce qui nous donne, dans notre cas :

private void initMenu(){

//Menu animation

//Le début de la méthode reste inchangé

//Ajout des menus dans la barre de menus et ajout de mnémoniques !

animation.setMnemonic('A');

menuBar.add(animation);

forme.setMnemonic('F');

menuBar.add(forme);

aPropos.setMnemonic('P');

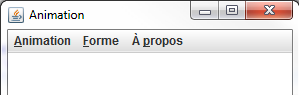
menuBar.add(aPropos);

//Ajout de la barre de menus sur la fenêtre

this.setJMenuBar(menuBar);

}

Nous avons à présent les lettres correspondant au mnémonique soulignées dans nos menus. Et il y a mieux : si vous tapez **ALT** + **<la lettre>**, le menu correspondant se déroule ! La figure suivante correspond à ce que j'obtiens.



Il existe une autre façon d’appliquer une Mnemonic mais uniquement pour les JMenu :

JMenu menu = new JMenu("Fichier", 'F'); //Ici, ce menu aura le mnémonique F

On les passe en paramètre constructeur.

**Accelerateur :** *KeyStroke***.**

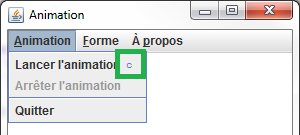
Cet objet permet de déterminer la touche utilisée ou à utiliser. C'est grâce à cet objet que nous allons pouvoir construire des combinaisons de touches pour nos accélérateurs ! Nous allons commencer par attribuer un simple caractère comme accélérateur à notre JMenuItemLancer en utilisant la méthode getKeyStroke(char caracter); de l'objet KeyStroke.

Ajoutez cette ligne de code au début de la méthode initMenu() (vous aurez besoin des packages javax.swing.KeyStroke et java.awt.event.ActionEvent) :

//Cette instruction ajoute l'accélérateur 'c' à notre objet

lancer.setAccelerator(KeyStroke.getKeyStroke('c'));

Testez votre application, un petit « c » est apparu à côté du menu Lancer. La figure suivante illustre le phénomène.



Les accélerateurs réagissent directement à la touche clavier, c’est pourquoi la plupart du temps on utilise des CTRL+C ou autre.

Dans le code qui suit, je crée un accélérateur **CTRL** + **L** pour le menu Lancer et un accélérateur **CTRL** + **SHIFT** + **A** pour le menu Arrêter :

lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_L, KeyEvent.CTRL\_MASK));

animation.add(lancer);

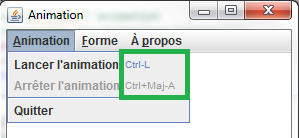
//Ajout du listener pour arrêter l'animation

arreter.addActionListener(new StopAnimationListener());

arreter.setEnabled(false);

arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_A, KeyEvent.CTRL\_DOWN\_MASK + KeyEvent.SHIFT\_DOWN\_MASK));

animation.add(arreter);



Une grande majorité d'entre eux sont sous la forme VK\_<le caractère ou le nom de la touche>. Lisez-le ainsi : **V**alue of **K**ey <nom de la touche>.

### Faire un menu contextuel :

**Les points important du menu contextuel :**

* Dans le cas d'opérations identiques à celles accessibles par le menu, nous devrons créer des objets qui s'étendent à ces deux menus.
* Le menu contextuel ne doit s'afficher que dans la zone où l'animation s'exécute, pas dans le menu !
* Il ne doit s'afficher que lorsqu'on fait un clic droit, et rien d'autre !

Nous allons mettre dans notre menu contextuel les actions « Lancer l'animation », « Arrêter l'animation » ainsi que deux nouveautés :

* changer la couleur du fond de notre animation ;
* changer la couleur de notre forme.

Pour préciser où afficher le menu contextuel, nous allons utiliser la méthode show(Component invoker, int x, int y); de la classe JPopupMenu :

* Component invoker : désigne l'objet invoquant le menu contextuel, dans notre cas, l'instance de Panneau.
* int x : coordonnée x du menu.
* int y : coordonnée y du menu.

Pour afficher le menu, on doit détecter le clic droit :  En effet, il possède une méthode isPopupTrigger() qui renvoie vrai s'il s'agit d'un clic droit.

//Les imports habituels

import javax.swing.JPopupMenu;

public class Fenetre extends JFrame{

//Nos variables habituelles

//La déclaration pour le menu contextuel

private JPopupMenu jpm = new JPopupMenu();

private JMenu background = new JMenu("Couleur de fond");

private JMenu couleur = new JMenu("Couleur de la forme");

private JMenuItem launch = new JMenuItem("Lancer l'animation");

private JMenuItem stop = new JMenuItem("Arrêter l'animation");

private JMenuItem rouge = new JMenuItem("Rouge"),

bleu = new JMenuItem("Bleu"),

vert = new JMenuItem("Vert"),

rougeBack = new JMenuItem("Rouge"),

bleuBack = new JMenuItem("Bleu"),

vertBack = new JMenuItem("Vert");

//On crée des listeners globaux

private StopAnimationListener stopAnimation=new StopAnimationListener();

private StartAnimationListener startAnimation=new StartAnimationListener();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

//On initialise le menu stop

stop.setEnabled(false);

//On affecte les écouteurs

stop.addActionListener(stopAnimation);

launch.addActionListener(startAnimation);

//On crée et on passe l'écouteur pour afficher le menu contextuel

//Création d'une implémentation de MouseAdapter

//avec redéfinition de la méthode adéquate

pan.addMouseListener(new MouseAdapter(){

public void mouseReleased(MouseEvent event){

//Seulement s'il s'agit d'un clic droit

//if(event.getButton() == MouseEvent.BUTTON3)

if(event.isPopupTrigger()){

background.add(rougeBack);

background.add(bleuBack);

background.add(vertBack);

couleur.add(rouge);

couleur.add(bleu);

couleur.add(vert);

jpm.add(launch);

jpm.add(stop);

jpm.add(couleur);

jpm.add(background);

//La méthode qui va afficher le menu

jpm.show(pan, event.getX(), event.getY());

}

}

});

container.add(pan, BorderLayout.CENTER);

this.setContentPane(container);

this.initMenu();

this.setVisible(true);

}

private void initMenu(){

//Ajout du listener pour lancer l'animation

//ATTENTION, LE LISTENER EST GLOBAL !!!

lancer.addActionListener(startAnimation);

//On attribue l'accélerateur c

lancer.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_L, KeyEvent.CTRL\_MASK));

animation.add(lancer);

//Ajout du listener pour arrêter l'animation

//LISTENER A CHANGER ICI AUSSI

arreter.addActionListener(stopAnimation);

arreter.setEnabled(false);

arreter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_A, KeyEvent.CTRL\_DOWN\_MASK + KeyEvent.SHIFT\_DOWN\_MASK));

animation.add(arreter);

//Le reste est inchangé

}

private void go(){

//La méthode n'a pas changé

}

public class StartAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null,

"Voulez-vous lancer l'animation ?",

"Lancement de l'animation",

JOptionPane.YES\_NO\_OPTION,

JOptionPane.QUESTION\_MESSAGE);

if(option == JOptionPane.OK\_OPTION){

lancer.setEnabled(false);

arreter.setEnabled(true);

//On ajoute l'instruction pour le menu contextuel

launch.setEnabled(false);

stop.setEnabled(true);

animated = true;

t = new Thread(new PlayAnimation());

t.start();

}

}

}

/\*\*

\* Écouteur du menu Quitter

\* @author CHerby

\*/

class StopAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

JOptionPane jop = new JOptionPane();

int option = jop.showConfirmDialog(null,

"Voulez-vous arrêter l'animation ?",

"Arrêt de l'animation",

JOptionPane.YES\_NO\_CANCEL\_OPTION,

JOptionPane.QUESTION\_MESSAGE);

if(option != JOptionPane.NO\_OPTION && option != JOptionPane.CANCEL\_OPTION && option != JOptionPane.CLOSED\_OPTION){

animated = false;

//On remplace nos boutons par nos JMenuItem

lancer.setEnabled(true);

arreter.setEnabled(false);

//On ajoute l'instruction pour le menu contextuel

launch.setEnabled(true);

stop.setEnabled(false);

}

}

}

class PlayAnimation implements Runnable{

//Inchangé

}

class FormeListener implements ActionListener{

//Inchangé

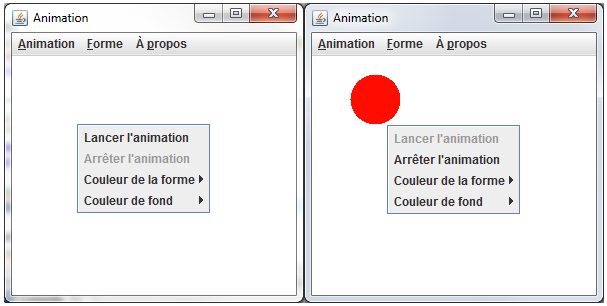
}

class MorphListener implements ActionListener{

//Inchangé

}

}



Mettre des couleurs :

##### Panneau.java

import java.awt.Color;

//Les autres imports

public class Panneau extends JPanel {

//Les variables définies auparavant ne changent pas

//On y ajoute nos deux couleurs

private Color couleurForme = Color.red;

private Color couleurFond = Color.white;

public void paintComponent(Graphics g){

//Affectation de la couleur de fond

g.setColor(couleurFond);

g.fillRect(0, 0, this.getWidth(), this.getHeight());

//Affectation de la couleur de la forme

g.setColor(couleurForme);

//Si le mode morphing est activé, on peint le morphing

if(this.morph)

drawMorph(g);

//Sinon, mode normal

else

draw(g);

}

//Méthode qui redéfinit la couleur du fond

public void setCouleurFond(Color color){

this.couleurFond = color;

}

//Méthode qui redéfinit la couleur de la forme

public void setCouleurForme(Color color){

this.couleurForme = color;

}

//Les autres méthodes sont inchangées

}

##### Fenetre.java

//Nos imports habituels

public class Fenetre extends JFrame{

//Nos variables n'ont pas changé

//On crée des listeners globaux

private StopAnimationListener stopAnimation = new StopAnimationListener();

private StartAnimationListener startAnimation = new StartAnimationListener()

//Avec des listeners pour les couleurs

private CouleurFondListener bgColor = new CouleurFondListener();

private CouleurFormeListener frmColor = new CouleurFormeListener();

public Fenetre(){

this.setTitle("Animation");

this.setSize(300, 300);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

container.setBackground(Color.white);

container.setLayout(new BorderLayout());

//On initialise le menu stop

stop.setEnabled(false);

//On affecte les écouteurs

stop.addActionListener(stopAnimation);

launch.addActionListener(startAnimation);

//On affecte les écouteurs aux points de menu

rouge.addActionListener(frmColor);

bleu.addActionListener(frmColor);

vert.addActionListener(frmColor);

blanc.addActionListener(frmColor);

rougeBack.addActionListener(bgColor);

bleuBack.addActionListener(bgColor);

vertBack.addActionListener(bgColor);

blancBack.addActionListener(bgColor);

//On crée et on passe l'écouteur pour afficher le menu contextuel

//Création d'une implémentation de MouseAdapter

//avec redéfinition de la méthode adéquate

pan.addMouseListener(new MouseAdapter(){

public void mouseReleased(MouseEvent event){

//Seulement s'il s'agit d'un clic droit

if(event.isPopupTrigger()){

background.add(blancBack);

background.add(rougeBack);

background.add(bleuBack);

background.add(vertBack);

couleur.add(blanc);

couleur.add(rouge);

couleur.add(bleu);

couleur.add(vert);

jpm.add(launch);

jpm.add(stop);

jpm.add(couleur);

jpm.add(background);

//La méthode qui va afficher le menu

jpm.show(pan, event.getX(), event.getY());

}

}

});

container.add(pan, BorderLayout.CENTER);

this.setContentPane(container);

this.initMenu();

this.setVisible(true);

}

private void initMenu(){

//Le menu n'a pas changé

}

private void go(){

//La méthode go() est identique

}

//Les classes internes :

// -> StartAnimationListener

// -> StopAnimationListener

// -> PlayAnimation

// -> FormeListener

// -> MorphListener

//sont inchangées !

//Écoute le changement de couleur du fond

class CouleurFondListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource() == vertBack)

pan.setCouleurFond(Color.green);

else if (e.getSource() == bleuBack)

pan.setCouleurFond(Color.blue);

else if(e.getSource() == rougeBack)

pan.setCouleurFond(Color.red);

else

pan.setCouleurFond(Color.white);

}

}

//Écoute le changement de couleur du fond

class CouleurFormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource() == vert)

pan.setCouleurForme(Color.green);

else if (e.getSource() == bleu)

pan.setCouleurForme(Color.blue);

else if(e.getSource() == rouge)

pan.setCouleurForme(Color.red);

else

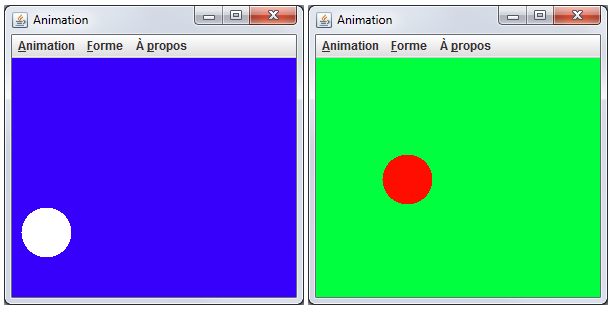
pan.setCouleurForme(Color.white);

}

}

}

la figure suivante représente deux résultats ainsi obtenus.



### Les barres d’outils :

On utilise *JToolBar* et prend des *JButton* en argument.

On doit cependant savoir si l’information vient du menu ou de la barre d’outil, on donne cette tâche à ActionListener.

import javax.swing.JToolBar;

//Nos imports habituels

public class Fenetre extends JFrame{

//Les variables declarées précédemment

//Création de notre barre d'outils

private JToolBar toolBar = new JToolBar();

//Les boutons de la barre d'outils

private JButton play = new JButton(new ImageIcon("images/play.jpg")),

cancel = new JButton(new ImageIcon("images/stop.jpg")),

square = new JButton(new ImageIcon("images/carré.jpg")),

tri = new JButton(new ImageIcon("images/triangle.jpg")),

circle = new JButton(new ImageIcon("images/rond.jpg")),

star = new JButton(new ImageIcon("images/étoile.jpg"));

private Color fondBouton = Color.white;

private FormeListener fListener = new FormeListener();

public Fenetre(){

//La seule nouveauté est la méthode ci-dessous

this.initToolBar();

this.setVisible(true);

}

private void initToolBar(){

this.cancel.setEnabled(false);

this.cancel.addActionListener(stopAnimation);

this.cancel.setBackground(fondBouton);

this.play.addActionListener(startAnimation);

this.play.setBackground(fondBouton);

this.toolBar.add(play);

this.toolBar.add(cancel);

this.toolBar.addSeparator();

//Ajout des Listeners

this.circle.addActionListener(fListener);

this.circle.setBackground(fondBouton);

this.toolBar.add(circle);

this.square.addActionListener(fListener);

this.square.setBackground(fondBouton);

this.toolBar.add(square);

this.tri.setBackground(fondBouton);

this.tri.addActionListener(fListener);

this.toolBar.add(tri);

this.star.setBackground(fondBouton);

this.star.addActionListener(fListener);

this.toolBar.add(star);

this.add(toolBar, BorderLayout.NORTH);

}

private void initMenu(){

//Méthode inchangée

}

private void go(){

//Méthode inchangée

}

public class StartAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

//Toujours la même boîte de dialogue…

if(option == JOptionPane.OK\_OPTION){

lancer.setEnabled(false);

arreter.setEnabled(true);

//ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

launch.setEnabled(false);

stop.setEnabled(true);

play.setEnabled(false);

cancel.setEnabled(true);

animated = true;

t = new Thread(new PlayAnimation());

t.start();

}

}

}

/\*\*

\* Écouteur du menu Quitter

\* @author CHerby

\*/

class StopAnimationListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Toujours la même boîte de dialogue…

if(option != JOptionPane.NO\_OPTION && option != JOptionPane.CANCEL\_OPTION && option != JOptionPane.CLOSED\_OPTION){

animated = false;

//On remplace nos boutons par nos MenuItem

lancer.setEnabled(true);

arreter.setEnabled(false);

//ON AJOUTE L'INSTRUCTION POUR LE MENU CONTEXTUEL

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

launch.setEnabled(true);

stop.setEnabled(false);

play.setEnabled(true);

cancel.setEnabled(false);

}

}

}

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//Si l'action vient d'un bouton radio du menu

if(e.getSource().getClass().getName().equals("javax.swing.JRadioButtonMenuItem"))

pan.setForme(((JRadioButtonMenuItem)e.getSource()).getText());

else{

if(e.getSource() == square){

carre.doClick();

}

else if(e.getSource() == tri){

triangle.doClick();

}

else if(e.getSource() == star){

etoile.doClick();

}

else{

rond.doClick();

}

}

}

}

//Les classes internes :

// -> CouleurFondListener

// -> CouleurFormeListener

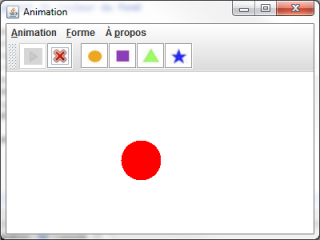
// -> PlayAnimation

// -> MorphListener

//sont inchangées !

}

Vous devez obtenir une IHM semblable à la figure suivante.



### Utiliser les actions abstraites :

Le rôle de cette classe est d'attribuer automatiquement une action à un ou plusieurs composants. Le principal avantage de ce procédé est que plusieurs composants travaillent avec une implémentation de la classe AbstractAction, mais son gros inconvénient réside dans le fait que vous devrez programmer une implémentation par action :

* une action pour la couleur de la forme en rouge ;
* une action pour la couleur de la forme en bleu ;
* une action pour la couleur de la forme en vert ;
* une action pour la couleur du fond en rouge ;
* une action pour la couleur du fond en bleu ;
* etc.

Voici comment s'implémente cette classe :

public class Fenetre extends JFrame{

//Nous pouvons utiliser les actions abstraites directement dans un JButton

private JButton bouton1 = new JButton(new RougeAction("ActionRouge", new ImageIcon("images/rouge.jpg"));

//Ou créer une instance concrète

private RougeAction rAct = new RougeAction("ActionRouge", new ImageIcon("images/rouge.jpg"));

private JToolBar toolBar = new JToolBar();

//…

//Utiliser une action directement dans une barre d'outils

private void initToolBar(){

toolBar.add(rAct);

}

//…

class RougeAction extends AbstractAction{

//Constructeur avec le nom uniquement

public RougeAction(String name){super(name);}

//Le constructeur prend le nom et une icône en paramètre

public RougeAction(String name, ImageIcon){super(name, img);}

public void actionPerformed(ActionEvent){

//Vous connaissez la marche à suivre

}

}

}

# TP l’ardoise magique

##### Point.java

// CTRL + SHIFT + O pour générer les imports

public class Point {

//Couleur du point

private Color color = Color.red;

//Taille

private int size = 10;

//Position sur l'axe X : initialisé au dehors du conteneur

private int x = -10;

//Position sur l'axe Y : initialisé au dehors du conteneur

private int y = -10;

//Type de point

private String type = "ROND";

// Constructeur par défaut

public Point(){}

public Point(int x, int y, int size, Color color, String type){

this.size = size;

this.x = x;

this.y = y;

this.color = color;

this.type = type;

}

//ACCESSEURS

public Color getColor() {

return color;

}

public void setColor(Color color) {

this.color = color;

}

public int getSize() {

return size;

}

public void setSize(int size) {

this.size = size;

}

public int getX() {

return x;

}

public void setX(int x) {

this.x = x;

}

public int getY() {

return y;

}

public void setY(int y) {

this.y = y;

}

public String getType() {

return type;

}

public void setType(String type) {

this.type = type;

}

}

##### DrawPanel.java

// CTRL + SHIFT + O pour générer les imports

public class DrawPanel extends JPanel{

//Couleur du pointeur

private Color pointerColor = Color.red;

//Forme du pointeur

private String pointerType = "CIRCLE";

//Position X du pointeur

private int posX = -10, oldX = -10;

//Position Y du pointeur

private int posY = -10, oldY = -10;

//Pour savoir si on doit dessiner ou non

private boolean erasing = true;

//Taille du pointeur

private int pointerSize = 15;

//Collection de points !

private ArrayList<Point> points = new ArrayList<Point>();

public DrawPanel(){

this.addMouseListener(new MouseAdapter(){

public void mousePressed(MouseEvent e){

points.add(new Point(e.getX() - (pointerSize / 2), e.getY() - (pointerSize / 2), pointerSize, pointerColor, pointerType));

repaint();

}

});

this.addMouseMotionListener(new MouseMotionListener(){

public void mouseDragged(MouseEvent e) {

//On récupère les coordonnées de la souris et on enlève la moitié de la taille du pointeur pour centrer le tracé

points.add(new Point(e.getX() - (pointerSize / 2), e.getY() - (pointerSize / 2), pointerSize, pointerColor, pointerType));

repaint();

}

public void mouseMoved(MouseEvent e) {}

});

}

// Vous la connaissez maintenant, celle-là

public void paintComponent(Graphics g) {

g.setColor(Color.white);

g.fillRect(0, 0, this.getWidth(), this.getHeight());

//Si on doit effacer, on ne passe pas dans le else => pas de dessin

if(this.erasing){

this.erasing = false;

}

else{

//On parcourt notre collection de points

for(Point p : this.points)

{

//On récupère la couleur

g.setColor(p.getColor());

//Selon le type de point

if(p.getType().equals("SQUARE")){

g.fillRect(p.getX(), p.getY(), p.getSize(), p.getSize());

}

else{

g.fillOval(p.getX(), p.getY(), p.getSize(), p.getSize());

}

}

}

}

//Efface le contenu

public void erase(){

this.erasing = true;

this.points = new ArrayList<Point>();

repaint();

}

//Définit la couleur du pointeur

public void setPointerColor(Color c){

this.pointerColor = c;

}

//Définit la forme du pointeur

public void setPointerType(String str){

this.pointerType = str;

}

}

##### Fenetre.java

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

//LE MENU

private JMenuBar menuBar = new JMenuBar();

JMenu fichier = new JMenu("Fichier"),

edition = new JMenu("Edition"),

forme = new JMenu("Forme du pointeur"),

couleur = new JMenu("Couleur du pointeur");

JMenuItem nouveau = new JMenuItem("Effacer"),

quitter = new JMenuItem("Quitter"),

rond = new JMenuItem("Rond"),

carre = new JMenuItem("Carré"),

bleu = new JMenuItem("Bleu"),

rouge = new JMenuItem("Rouge"),

vert = new JMenuItem("Vert");

//LA BARRE D'OUTILS

JToolBar toolBar = new JToolBar();

JButton square = new JButton(new ImageIcon("images/carré.jpg")),

circle = new JButton(new ImageIcon("images/rond.jpg")),

red = new JButton(new ImageIcon("images/rouge.jpg")),

green = new JButton(new ImageIcon("images/vert.jpg")),

blue = new JButton(new ImageIcon("images/bleu.jpg"));

//LES ÉCOUTEURS

private FormeListener fListener = new FormeListener();

private CouleurListener cListener = new CouleurListener();

//Notre zone de dessin

private DrawPanel drawPanel = new DrawPanel();

public Fenetre(){

this.setSize(700, 500);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

//On initialise le menu

this.initMenu();

//Idem pour la barre d'outils

this.initToolBar();

//On positionne notre zone de dessin

this.getContentPane().add(drawPanel, BorderLayout.CENTER);

this.setVisible(true);

}

//Initialise le menu

private void initMenu(){

nouveau.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

drawPanel.erase();

}

});

nouveau.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_N, KeyEvent.CTRL\_DOWN\_MASK));

quitter.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

System.exit(0);

}

});

quitter.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK\_W, KeyEvent.CTRL\_DOWN\_MASK));

fichier.add(nouveau);

fichier.addSeparator();

fichier.add(quitter);

fichier.setMnemonic('F');

carre.addActionListener(fListener);

rond.addActionListener(fListener);

forme.add(rond);

forme.add(carre);

rouge.addActionListener(cListener);

vert.addActionListener(cListener);

bleu.addActionListener(cListener);

couleur.add(rouge);

couleur.add(vert);

couleur.add(bleu);

edition.setMnemonic('E');

edition.add(forme);

edition.addSeparator();

edition.add(couleur);

menuBar.add(fichier);

menuBar.add(edition);

this.setJMenuBar(menuBar);

}

//Initialise la barre d'outils

private void initToolBar(){

JPanel panneau = new JPanel();

square.addActionListener(fListener);

circle.addActionListener(fListener);

red.addActionListener(cListener);

green.addActionListener(cListener);

blue.addActionListener(cListener);

toolBar.add(square);

toolBar.add(circle);

toolBar.addSeparator();

toolBar.add(red);

toolBar.add(blue);

toolBar.add(green);

this.getContentPane().add(toolBar, BorderLayout.NORTH);

}

//ÉCOUTEUR POUR LE CHANGEMENT DE FORME

class FormeListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){

if(e.getSource()==carre) drawPanel.setPointerType("SQUARE");

else drawPanel.setPointerType("CIRCLE");

}

else{

if(e.getSource()==square)drawPanel.setPointerType("SQUARE");

else drawPanel.setPointerType("CIRCLE");

}

}

}

//ÉCOUTEUR POUR LE CHANGEMENT DE COULEUR

class CouleurListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

System.out.println(e.getSource().getClass().getName());

if(e.getSource().getClass().getName().equals("javax.swing.JMenuItem")){

System.out.println("OK !");

if(e.getSource()==vert)drawPanel.setPointerColor(Color.green);

else if(e.getSource()==bleu)drawPanel.setPointerColor(Color.blue);

else drawPanel.setPointerColor(Color.red);

}

else{

if(e.getSource()==green)drawPanel.setPointerColor(Color.green);

else if(e.getSource()==blue)drawPanel.setPointerColor(Color.blue);

else drawPanel.setPointerColor(Color.red);

}

}

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

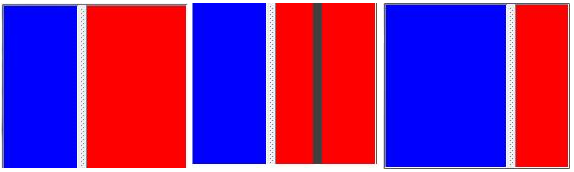
}

# Conteneurs, slider et barre de progression

Les objets sont survolés, il faut les approffondir.

## Autres conteneurs :

Fenêtre avec un JSplitPanel :



Quand on manipule la barre centrale, l’un ou l’autre devient plus petit ou plus grand en fonction de comment on bouge cette barre :

import java.awt.BorderLayout;

import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JSplitPane;

public class Fenetre extends JFrame {

//On déclare notre objet JSplitPane

private JSplitPane split;

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneur");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(200, 200);

//On crée deux conteneurs de couleurs différentes

JPanel pan = new JPanel();

pan.setBackground(Color.blue);

JPanel pan2 = new JPanel();

pan2.setBackground(Color.red);

//On construit enfin notre séparateur

split = new JSplitPane(JSplitPane.HORIZONTAL\_SPLIT, pan, pan2);

//On le passe ensuite au content pane de notre objet Fenetre

//placé au centre pour qu'il utilise tout l'espace disponible

this.getContentPane().add(split, BorderLayout.CENTER);

this.setVisible(true);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

On pourrait aussi utiliser JSplitPane.VERTICAL\_SPLIT.

Autre point, les deux autres paramètres ne sont pas nécessairement des JPanel. Ici, j'ai utilisé des JPanel, mais vous pouvez en fait utiliser n'importe quelle classe dérivant de JComponent (conteneur, bouton, case à cocher…) : elle n'est pas belle, la vie ?

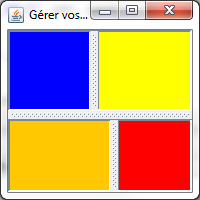
On peut aussi avoir une barre intelligente :



split.setOneTouchExpandable(true);

Ou Augmenter la taille de la barre : split.setDividerSize(int size) .

Cet objet peut accéder aux paramètres de sous-classe JComponent.



Voici le code correspondant :

import java.awt.BorderLayout;

import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JSplitPane;

public class Fenetre extends JFrame {

private JSplitPane split, split2, split3;

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneur");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(200, 200);

//On crée deux conteneurs de couleurs différentes

JPanel pan = new JPanel();

pan.setBackground(Color.blue);

JPanel pan2 = new JPanel();

pan2.setBackground(Color.red);

JPanel pan3 = new JPanel();

pan3.setBackground(Color.orange);

JPanel pan4 = new JPanel();

pan4.setBackground(Color.YELLOW);

//On construit enfin notre séparateur

split = new JSplitPane(JSplitPane.HORIZONTAL\_SPLIT, pan, pan4);

//On place le premier séparateur

split.setDividerLocation(80);

split2 = new JSplitPane(JSplitPane.HORIZONTAL\_SPLIT, pan3, pan2);

//On place le deuxième séparateur

split2.setDividerLocation(100);

//On passe les deux précédents JSplitPane à celui-ci

split3 = new JSplitPane(JSplitPane.VERTICAL\_SPLIT, split, split2);

//On place le troisième séparateur

split3.setDividerLocation(80);

//On le passe ensuite au content pane de notre objet Fenetre

//placé au centre pour qu'il utilise tout l'espace disponible

this.getContentPane().add(split3, BorderLayout.CENTER);

this.setVisible(true);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

### Objet JScrollPane :

On va voir l’objet JTextArea : une sorte de JTextField mais en plus grand mais qui ne retourne pas à la ligne automatiquement.

On peut récupérer le texte avec getText ().

import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JTextArea;

public class Fenetre extends JFrame {

private JTextArea textPane = new JTextArea();

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneur");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(200, 200);

JButton bouton = new JButton("Bouton");

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("Texte écrit dans le JTextArea : ");

System.out.println("--------------------------------");

System.out.println(textPane.getText());

}

});

//On ajoute l'objet au content pane de notre fenêtre

this.getContentPane().add(textPane, BorderLayout.CENTER);

this.getContentPane().add(bouton, BorderLayout.SOUTH);

this.setVisible(true);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

Seulement on ne voit pas le texte qui dépasse de la limite de notre fenêtre … On rajoute des scrolls :

import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JScrollPane;

import javax.swing.JTextArea;

public class Fenetre extends JFrame {

private JTextArea textPane = new JTextArea();

private JScrollPane scroll = new JScrollPane(textPane);

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneur");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(200, 200);

JButton bouton = new JButton("Bouton");

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("Texte écrit dans le JTextArea : ");

System.out.println("--------------------------------");

System.out.println(textPane.getText());

}

});

//On ajoute l'objet au content pane de notre fenêtre

this.getContentPane().add(scroll, BorderLayout.CENTER);

//On aurait pu aussi écrire

//this.getContentPane().add(new JScrollPane(textPane), BorderLayout.CENTER);

this.getContentPane().add(bouton, BorderLayout.SOUTH);

this.setVisible(true);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

L'objet utilisé afin d'avoir un ascenseur s'appelle donc un JScrollPane. Désormais, vous pourrez écrire aussi loin que vous le voulez, vers le bas et vers la droite ! Les ascenseurs apparaissent automatiquement lorsque vous dépassez les limites autorisées. De plus, vous pouvez redéfinir leurs comportements grâce aux méthodes :

* scroll.setHorizontalScrollBarPolicy(int policy), qui permet de définir le comportement du scroll en bas de votre fenêtre ;
* scroll.setVerticalScrollBarPolicy(int policy), qui permet de définir le comportement du scroll à droite de votre fenêtre.

Le paramètre de ces méthodes est un entier défini dans la classe JScrollPane, il peut prendre les valeurs suivantes :

* JScrollPane.VERTICAL\_SCROLLBAR\_AS\_NEEDED : le scroll vertical n'est visible que s'il est nécessaire, donc s'il y a dépassement de la taille en hauteur ;
* JScrollPane.VERTICAL\_SCROLLBAR\_NEVER : le scroll vertical n'est jamais visible, même si vous dépassez ; en revanche, le conteneur s'allonge tout de même ;
* JScrollPane.VERTICAL\_SCROLLBAR\_ALWAYS : le scroll vertical est toujours visible, même si vous ne dépassez pas.

### L’objet JTabbedpane :

Permet de créer des onglets dans la fenêtre :

import java.awt.Color;

import java.awt.Font;

import java.awt.Graphics;

import javax.swing.JPanel;

public class Panneau extends JPanel{

private Color color = Color.white;

private static int COUNT = 0;

private String message = "";

public Panneau(){}

public Panneau(Color color){

this.color = color;

this.message = "Contenu du panneau N°" + (++COUNT);

}

public void paintComponent(Graphics g){

g.setColor(this.color);

g.fillRect(0, 0, this.getWidth(), this.getHeight());

g.setColor(Color.white);

g.setFont(new Font("Arial", Font.BOLD, 15));

g.drawString(this.message, 10, 20);

}

}

import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JTabbedPane;

public class Fenetre extends JFrame {

private JTabbedPane onglet;

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneurs");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(400, 200);

//Création de plusieurs Panneau

Panneau[] tPan = { new Panneau(Color.RED), new Panneau(Color.GREEN), new Panneau(Color.BLUE)};

//Création de notre conteneur d'onglets

onglet = new JTabbedPane();

int i = 0;

for(Panneau pan : tPan){

//Méthode d'ajout d'onglet

onglet.add("Onglet n° "+(++i), pan);

//Vous pouvez aussi utiliser la méthode addTab

//onglet.addTab("Onglet n° "+(++i), pan);

}

//On passe ensuite les onglets au content pane

this.getContentPane().add(onglet);

this.setVisible(true);

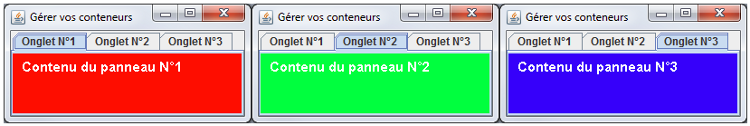
}

public static void main(String[] args){

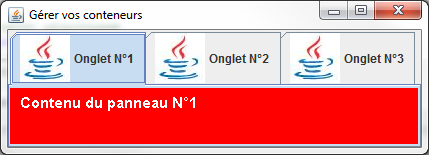
Fenetre fen = new Fenetre();

}

}



On peut rajouter une image à côté des titres d’onglet par exemple :



for(Panneau pan : tPan){

//Méthode d'ajout d'onglet

onglet.add("Onglet n° "+(++i), pan);

//On ajoute l'image à l'onglet en cours

//Les index d'onglets fonctionnent comme les tableaux : ils commencent à 0

onglet.setIconAt((i - 1), new ImageIcon("java.jpg"));

Changer l’emplacement des objets :

//Affiche les onglets en bas de la fenêtre

JTabbedPane onglet = new JTabbedPane(JTabbedPane.BOTTOM);

//Affiche les onglets à gauche de la fenêtre

JTabbedPane onglet = new JTabbedPane(JTabbedPane.LEFT);

//Affiche les onglets à droite de la fenêtre

JTabbedPane onglet = new JTabbedPane(JTabbedPane.RIGHT);



On peut enlever des objets avec remove.

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTabbedPane onglet;

//Compteur pour le nombre d'onglets

private int nbreTab = 0;

public Fenetre(){

this.setLocationRelativeTo(null);

this.setTitle("Gérer vos conteneurs");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(400, 200);

//Création de plusieurs Panneau

Panneau[] tPan = { new Panneau(Color.RED), new Panneau(Color.GREEN), new Panneau(Color.BLUE)};

//Création de notre conteneur d'onglets

onglet = new JTabbedPane();

for(Panneau pan : tPan){

//Méthode d'ajout d'onglets

onglet.addTab("Onglet N°"+(++nbreTab), pan);

}

//On passe ensuite les onglets au content pane

this.getContentPane().add(onglet, BorderLayout.CENTER);

//Ajout du bouton pour ajouter des onglets

JButton nouveau = new JButton("Ajouter un onglet");

nouveau.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

onglet.add("Onglet N°"+(++nbreTab), new Panneau(Color.DARK\_GRAY));

}

});

//Ajout du bouton pour retirer l'onglet sélectionné

JButton delete = new JButton("Effacer l'onglet");

delete.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

//On récupère l'index de l'onglet sélectionné

int selected = onglet.getSelectedIndex();

//S'il n'y a plus d'onglet, la méthode ci-dessus retourne -1

if(selected > -1)onglet.remove(selected);

}

});

JPanel pan = new JPanel();

pan.add(nouveau);

pan.add(delete);

this.getContentPane().add(pan, BorderLayout.SOUTH);

this.setVisible(true);

}

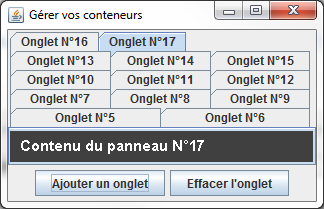
public static void main(String[] args){

Fenetre fen = new Fenetre();

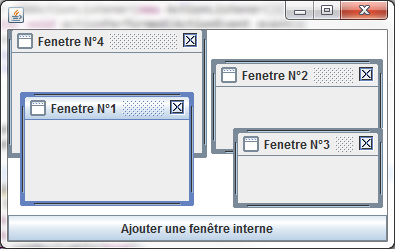
}

}

Ce qui peut vous donner la même chose que la figure suivante.



### JDesktopPane avec des JinternalFrame :



//CTRL + SHIFT + O pour générer les imports nécessaires

public class Bureau extends JFrame{

private static int nbreFenetre = 0;

private JDesktopPane desktop = new JDesktopPane();

private static int xy = 10;

public Bureau(){

this.setSize(400, 300);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JButton ajouter = new JButton("Ajouter une fenêtre interne");

ajouter.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

++nbreFenetre;

xy += 2;

desktop.add(new MiniFenetre(nbreFenetre), nbreFenetre);

}

});

this.getContentPane().add(desktop, BorderLayout.CENTER);

this.getContentPane().add(ajouter, BorderLayout.SOUTH);

}

class MiniFenetre extends JInternalFrame{

public MiniFenetre(int nbre){

this.setTitle("Fenetre N°"+nbre);

this.setClosable(true);

this.setResizable(true);

this.setSize(150, 80);

this.setLocation(xy, xy);

this.setVisible(true);

}

}

public static void main(String[] args){

Bureau bureau = new Bureau();

bureau.setVisible(true);

}

}

### ObjetJWindow :

C’est un objet JFrame mais sans les contours fermer agrandir et réduire.

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Window extends JWindow{

public static void main(String[] args){

Window wind = new Window();

wind.setVisible(true);

}

public Window(){

setSize(220, 165);

setLocationRelativeTo(null);

JPanel pan = new JPanel();

JLabel img = new JLabel(new ImageIcon("planète.jpeg"));

img.setVerticalAlignment(JLabel.CENTER);

img.setHorizontalAlignment(JLabel.CENTER);

pan.setBorder(BorderFactory.createLineBorder(Color.blue));

pan.add(img);

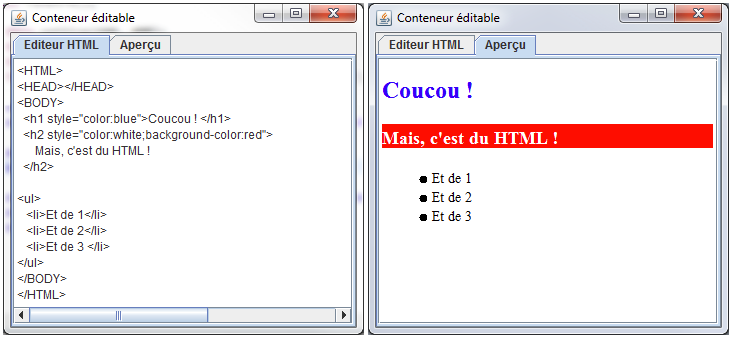
getContentPane().add(pan);

}

}

### JEditorPane :

Gère du texte (et contenu HTML).



//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JEditorPane editorPane, apercu;

private JTabbedPane onglet = new JTabbedPane();

public Fenetre(){

this.setSize(600, 400);

this.setTitle("Conteneur éditable");

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

editorPane = new JEditorPane();

editorPane.setText(" <HTML><HEAD></HEAD><BODY></BODY></HTML> ");

apercu = new JEditorPane();

apercu.setEditable(false);

onglet.addTab("Editeur HTML", new JScrollPane(editorPane));

onglet.addTab("Aperçu", new JScrollPane(apercu));

onglet.addChangeListener(new ChangeListener(){

public void stateChanged(ChangeEvent e) {

FileWriter fw = null;

try {

fw = new FileWriter(new File("tmp/tmp.html"));

fw.write(editorPane.getText());

fw.close();

} catch (FileNotFoundException e1) {

e1.printStackTrace();

} catch (IOException e1) {

e1.printStackTrace();

}

try {

File file = new File("tmp/tmp.html");

apercu.setEditorKit(new HTMLEditorKit());

apercu.setPage(file.toURL());

} catch (IOException e1) {

e1.printStackTrace();

}

}

});

this.getContentPane().add(onglet, BorderLayout.CENTER);

this.setVisible(true);

}

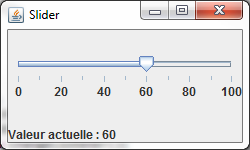
public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

### JSlider :



//CTRL + SHIFT + O pour générer les imports nécessaires

public class Slide extends JFrame{

private JLabel label = new JLabel("Valeur actuelle : 30");

public Slide(){

this.setSize(250, 150);

this.setTitle("Slider");

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JSlider slide = new JSlider();

slide.setMaximum(100);

slide.setMinimum(0);

slide.setValue(30);

slide.setPaintTicks(true);

slide.setPaintLabels(true);

slide.setMinorTickSpacing(10);

slide.setMajorTickSpacing(20);

slide.addChangeListener(new ChangeListener(){

public void stateChanged(ChangeEvent event){

label.setText("Valeur actuelle : " + ((JSlider)event.getSource()).getValue());

}

});

this.getContentPane().add(slide, BorderLayout.CENTER);

this.getContentPane().add(label, BorderLayout.SOUTH);

}

public static void main(String[] args){

Slide slide = new Slide();

slide.setVisible(true);

}

}

### JProgressBar :



//CTRL + SHIFT + O pour générer les imports nécessaires

public class Progress extends JFrame{

private Thread t;

private JProgressBar bar;

private JButton launch ;

public Progress(){

this.setSize(300, 80);

this.setTitle("\*\*\* JProgressBar \*\*\*");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

t = new Thread(new Traitement());

bar = new JProgressBar();

bar.setMaximum(500);

bar.setMinimum(0);

bar.setStringPainted(true);

this.getContentPane().add(bar, BorderLayout.CENTER);

launch = new JButton("Lancer");

launch.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

t = new Thread(new Traitement());

t.start();

}

});

this.getContentPane().add(launch, BorderLayout.SOUTH);

t.start();

this.setVisible(true);

}

class Traitement implements Runnable{

public void run(){

launch.setEnabled(false);

for(int val = 0; val <= 500; val++){

bar.setValue(val);

try {

t.sleep(10);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

launch.setEnabled(true);

}

}

public static void main(String[] args){

Progress p = new Progress();

}

}

On doit le même dans un nouveau thread sinon on aura une barre vide, un temps d’attente et enfin une barre qui se remplie, elle ne se fera donc aps en même temps que le chargement.

## Enjoliver ses IHM :

//CTRL + SHIFT + O pour générer les imports nécessaires

public class BorderDemo extends JFrame{

private String[] list = {

"Bevel Border",

"Etched Border",

"Line Border",

"Matted Border",

"Raised Bevel Border",

"Title Border",

"Compound Border"

};

private Border[] listBorder = {

BorderFactory.createBevelBorder(BevelBorder.LOWERED, Color.black, Color.red),

BorderFactory.createEtchedBorder(Color.BLUE, Color.GRAY),

BorderFactory.createLineBorder(Color.green),

BorderFactory.createMatteBorder(5, 2, 5, 2, Color.MAGENTA),

BorderFactory.createRaisedBevelBorder(),

BorderFactory.createTitledBorder("Titre"),

BorderFactory.createCompoundBorder(

BorderFactory.createBevelBorder(BevelBorder.LOWERED, Color.black, Color.blue),

BorderFactory.createMatteBorder(5, 2, 5, 2, Color.MAGENTA)

)

};

public BorderDemo(){

this.setTitle("Les bordures font la fête !");

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setSize(550, 200);

JPanel pan = new JPanel();

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

JLabel lib = new JLabel(list[i]);

lib.setPreferredSize(new Dimension(150, 50));

lib.setBorder(listBorder[i]);

lib.setAlignmentX(JLabel.CENTER);

lib.setHorizontalAlignment(JLabel.CENTER);

pan.add(lib);

}

this.getContentPane().add(pan);

}

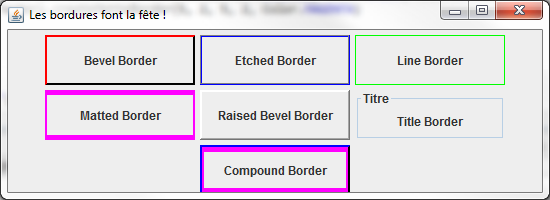
public static void main(String[] args){

BorderDemo demo = new BorderDemo();

demo.setVisible(true);

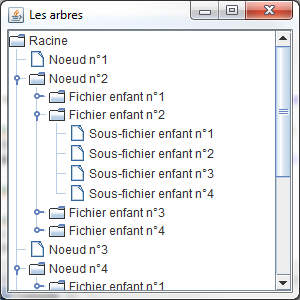
}

}



# Les arbres et leur structure

## La composition des arbres :



Se comose comme un arbre :

* Une racine (en haut)
* Des branches (les nœuds)
* Des feuilles (les enfants).

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre;

public Fenetre(){

this.setSize(300, 300);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("Les arbres");

//On invoque la méthode de construction de notre arbre

buildTree();

this.setVisible(true);

}

private void buildTree(){

//Création d'une racine

DefaultMutableTreeNode racine = new DefaultMutableTreeNode("Racine");

//Nous allons ajouter des branches et des feuilles à notre racine

for(int i = 1; i < 12; i++){

DefaultMutableTreeNode rep = new DefaultMutableTreeNode("Noeud n°"+i);

//S'il s'agit d'un nombre pair, on rajoute une branche

if((i%2) == 0){

//Et une branche en plus ! Une !

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

DefaultMutableTreeNode rep2 = new DefaultMutableTreeNode("Fichier enfant n°" + j);

//Cette fois, on ajoute les feuilles

for(int k = 1; k < 5; k++)

rep2.add(new DefaultMutableTreeNode("Sous-fichier enfant n°" + k));

rep.add(rep2);

}

}

//On ajoute la feuille ou la branche à la racine

racine.add(rep);

}

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree(racine);

//Que nous plaçons sur le ContentPane de notre JFrame à l'aide d'un scroll

this.getContentPane().add(new JScrollPane(arbre));

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

Si vous avez du mal à vous y retrouver, essayez cette version de la méthode buildTree() :

private void buildTree(){

//Création d'une racine

DefaultMutableTreeNode racine = new DefaultMutableTreeNode("Racine");

//Nous allons ajouter des branches et des feuilles à notre racine

for(int i = 1; i < 6; i++){

DefaultMutableTreeNode rep = new DefaultMutableTreeNode("Noeud n°"+i);

//On rajoute 4 branches

if(i < 4){

DefaultMutableTreeNode rep2 = new DefaultMutableTreeNode("Fichier enfant");

rep.add(rep2);

}

//On ajoute la feuille ou la branche à la racine

racine.add(rep);

}

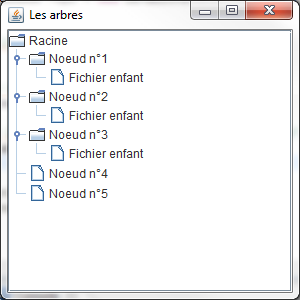
//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree(racine);

//Que nous plaçons sur le ContentPane de notre JFrame à l'aide d'un scroll

this.getContentPane().add(new JScrollPane(arbre));

}



On peut faire de même avec l’arborescence fichier :

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre;

private DefaultMutableTreeNode racine;

public Fenetre(){

this.setSize(300, 300);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("Les arbres");

//On invoque la méthode de construction de l'arbre

listRoot();

this.setVisible(true);

}

private void listRoot(){

this.racine = new DefaultMutableTreeNode();

int count = 0;

for(File file : File.listRoots()){

DefaultMutableTreeNode lecteur =

new DefaultMutableTreeNode(file.getAbsolutePath());

try {

for(File nom : file.listFiles()){

DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName()+"\\");

lecteur.add(this.listFile(nom, node));

}

} catch (NullPointerException e) {}

this.racine.add(lecteur);

}

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree(this.racine);

//Que nous plaçons sur le ContentPane de notre JFrame à l'aide d'un scroll

this.getContentPane().add(new JScrollPane(arbre));

}

private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node){

int count = 0;

if(file.isFile())

return new DefaultMutableTreeNode(file.getName());

else{

File[] list = file.listFiles();

if(list == null)

return new DefaultMutableTreeNode(file.getName());

for(File nom : list){

count++;

//Pas plus de 5 enfants par noeud

if(count < 5){

DefaultMutableTreeNode subNode;

if(nom.isDirectory()){

subNode = new DefaultMutableTreeNode(nom.getName()+"\\");

node.add(this.listFile(nom, subNode));

}else{

subNode = new DefaultMutableTreeNode(nom.getName());

}

node.add(subNode);

}

}

return node;

}

}

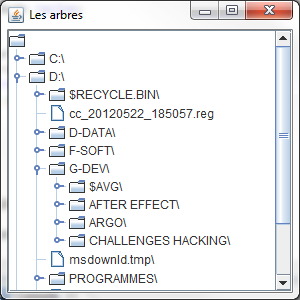
public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

Ce type de code ne devrait plus vous faire peur. La figure suivante montre ce que ça me donne, après quelques secondes…



## Des arbres qui parlent :

On va implémenter une interface. Celle-ci se nomme TreeSelectionListener. Elle ne contient qu'une méthode à redéfinir : valueChanged(TreeSelectionEvent event).

Voici un code utilisant une implémentation de cette interface :

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre;

private DefaultMutableTreeNode racine;

public Fenetre(){

this.setSize(300, 200);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("Les arbres");

//On invoque la méthode de construction de l'arbre

listRoot();

this.setVisible(true);

}

private void listRoot(){

this.racine = new DefaultMutableTreeNode();

int count = 0;

for(File file : File.listRoots()){

DefaultMutableTreeNode lecteur = new DefaultMutableTreeNode(file.getAbsolutePath());

try {

for(File nom : file.listFiles()){

DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName()+"\\");

lecteur.add(this.listFile(nom, node));

}

} catch (NullPointerException e) {}

this.racine.add(lecteur);

}

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree(this.racine);

arbre.setRootVisible(false);

arbre.addTreeSelectionListener(new TreeSelectionListener(){

public void valueChanged(TreeSelectionEvent event) {

if(arbre.getLastSelectedPathComponent() != null){

System.out.println(arbre.getLastSelectedPathComponent().toString());

}

}

});

//Que nous plaçons sur le ContentPane de notre JFrame à l'aide d'un scroll

this.getContentPane().add(new JScrollPane(arbre));

}

private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node){

int count = 0;

if(file.isFile())

return new DefaultMutableTreeNode(file.getName());

else{

File[] list = file.listFiles();

if(list == null)

return new DefaultMutableTreeNode(file.getName());

for(File nom : list){

count++;

//Pas plus de 5 enfants par noeud

if(count < 5){

DefaultMutableTreeNode subNode;

if(nom.isDirectory()){

subNode = new DefaultMutableTreeNode(nom.getName()+"\\");

node.add(this.listFile(nom, subNode));

}else{

subNode = new DefaultMutableTreeNode(nom.getName());

}

node.add(subNode);

}

}

return node;

}

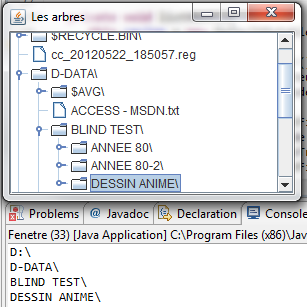
}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}



On souhaite obtenir plus de renseignement sur les nœuds : comme le chemin d’accès par exemple :

arbre.addTreeSelectionListener(new TreeSelectionListener(){

public void valueChanged(TreeSelectionEvent event) {

if(arbre.getLastSelectedPathComponent() != null){

//La méthode getPath retourne un objet TreePath

System.out.println(getAbsolutePath(event.getPath()));

}

}

private String getAbsolutePath(TreePath treePath){

String str = "";

//On balaie le contenu de l'objet TreePath

for(Object name : treePath.getPath()){

//Si l'objet a un nom, on l'ajoute au chemin

if(name.toString() != null)

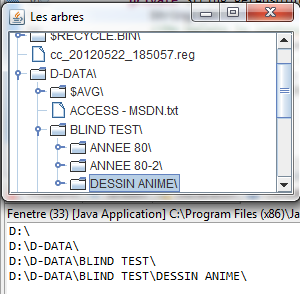
str += name.toString();

}

return str;

}

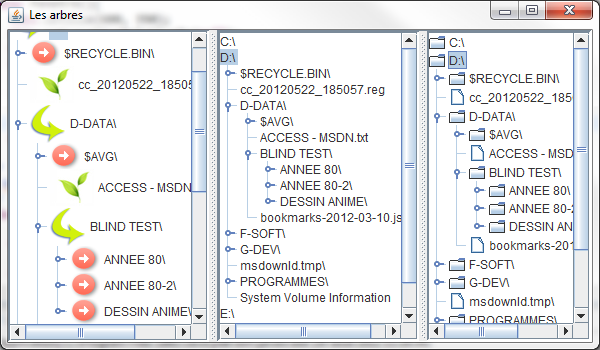
});



Exemple plus complet :

<https://openclassrooms.com/codeweb/623139>

## Décorer les arbres :



On utilise un objet DefaultTreeCellRenderer et dire à l’arbre qu’on va utiliser ce modèle avec la méthode *setCellRenderer(DefaultTreeRenderer cellRenderer)*.

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre, arbre2, arbre3;

private DefaultMutableTreeNode racine;

//On va créer deux modèles d'affichage

private DefaultTreeCellRenderer[] tCellRenderer = new DefaultTreeCellRenderer[3];

public Fenetre(){

this.setSize(600, 350);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("Les arbres");

//On invoque la méthode de construction de l'arbre

initRenderer();

listRoot();

this.setVisible(true);

}

private void initRenderer(){

//Instanciation

this.tCellRenderer[0] = new DefaultTreeCellRenderer();

//Initialisation des images pour les actions fermer, ouvrir et pour les feuilles

this.tCellRenderer[0].setClosedIcon(new ImageIcon("img/ferme.jpg"));

this.tCellRenderer[0].setOpenIcon(new ImageIcon("img/ouvert.jpg"));

this.tCellRenderer[0].setLeafIcon(new ImageIcon("img/feuille.jpg"));

this.tCellRenderer[1] = new DefaultTreeCellRenderer();

this.tCellRenderer[1].setClosedIcon(null);

this.tCellRenderer[1].setOpenIcon(null);

this.tCellRenderer[1].setLeafIcon(null);

}

private void listRoot(){

this.racine = new DefaultMutableTreeNode();

int count = 0;

for(File file : File.listRoots()){

DefaultMutableTreeNode lecteur = new DefaultMutableTreeNode(file.getAbsolutePath());

try {

for(File nom : file.listFiles()){

DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName()+"\\");

lecteur.add(this.listFile(nom, node));

}

} catch (NullPointerException e) {}

this.racine.add(lecteur);

}

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree(this.racine);

arbre.setRootVisible(false);

//On définit le rendu pour cet arbre

arbre.setCellRenderer(this.tCellRenderer[0]);

arbre2 = new JTree(this.racine);

arbre2.setRootVisible(false);

arbre2.setCellRenderer(this.tCellRenderer[1]);

arbre3 = new JTree(this.racine);

arbre3.setRootVisible(false);

JSplitPane split = new JSplitPane( JSplitPane.HORIZONTAL\_SPLIT,

new JScrollPane(arbre2),

new JScrollPane(arbre3));

split.setDividerLocation(200);

JSplitPane split2 = new JSplitPane( JSplitPane.HORIZONTAL\_SPLIT,

new JScrollPane(arbre),

split);

split2.setDividerLocation(200);

this.getContentPane().add(split2);

}

private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node){

int count = 0;

if(file.isFile())

return new DefaultMutableTreeNode(file.getName());

else{

File[] list = file.listFiles();

if(list == null)

return new DefaultMutableTreeNode(file.getName());

for(File nom : list){

count++;

//Pas plus de 5 enfants par noeud

if(count < 5){

DefaultMutableTreeNode subNode;

if(nom.isDirectory()){

subNode = new DefaultMutableTreeNode(nom.getName()+"\\");

node.add(this.listFile(nom, subNode));

}else{

subNode = new DefaultMutableTreeNode(nom.getName());

}

node.add(subNode);

}

}

return node;

}

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

}

}

On peut forcer le designe en fonction de son OS :

J'ai rajouté ces lignes de code dans le constructeur de mon objet, avant l'instruction setVisible(true) :

try {

//On force à utiliser le « look and feel » du système

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

//Ici on force tous les composants de notre fenêtre (this) à se redessiner avec le « look and feel » du système

SwingUtilities.updateComponentTreeUI(this);

}

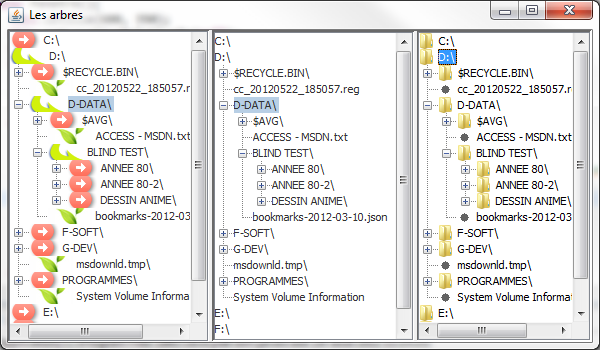
catch (InstantiationException e) {}

catch (ClassNotFoundException e) {}

catch (UnsupportedLookAndFeelException e) {}

catch (IllegalAccessException e) {}

Cela me donne la figure suivante.



## Modifier le contenu d’un arbre :

Notre arbre est constitué de plusieurs objets :

* TreeModel : c'est lui qui contient les nœuds de votre arbre ;
* TreeNode : nœuds et structure de votre arbre ;
* TreeSelectionModel : modèle de sélection de tous vos nœuds ;
* TreePath : objet qui vous permet de connaître le chemin d'un nœud dans l'arbre. La voilà, notre classe concrète ;
* TreeCellRenderer : interface permettant de modifier l'apparence d'un nœud ;
* TreeCellEditor : éditeur utilisé lorsqu'un nœud est éditable.

Voici un exemple de code utilisant un DefaultTreeModel (classe implémentant l'interface TreeModel) ainsi qu'une implémentation de l'interface TreeModelListener afin d'écouter cet objet :

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre;

private DefaultMutableTreeNode racine;

//Notre objet modèle

private DefaultTreeModel model;

public Fenetre(){

this.setSize(200, 300);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTree");

listRoot();

this.setVisible(true);

}

private void listRoot(){

this.racine = new DefaultMutableTreeNode();

int count = 0;

for(File file : File.listRoots())

{

DefaultMutableTreeNode lecteur = new DefaultMutableTreeNode(file.getAbsolutePath());

try {

for(File nom : file.listFiles()){

DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName()+"\\");

lecteur.add(this.listFile(nom, node));

}

} catch (NullPointerException e) {}

this.racine.add(lecteur);

}

//Nous créons notre modèle

this.model = new DefaultTreeModel(this.racine);

//Et nous allons écouter ce que notre modèle a à nous dire !

this.model.addTreeModelListener(new TreeModelListener() {

/\*\*

\* Méthode appelée lorsqu'un noeud a changé

\*/

public void treeNodesChanged(TreeModelEvent evt) {

System.out.println("Changement dans l'arbre");

Object[] listNoeuds = evt.getChildren();

int[] listIndices = evt.getChildIndices();

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

System.out.println("Index " + listIndices[i] + ", nouvelle valeur :" + listNoeuds[i]);

}

}

/\*\*

\* Méthode appelée lorsqu'un noeud est inséré

\*/

public void treeNodesInserted(TreeModelEvent event) {

System.out.println("Un noeud a été inséré !");

}

/\*\*

\* Méthode appelée lorsqu'un noeud est supprimé

\*/

public void treeNodesRemoved(TreeModelEvent event) {

System.out.println("Un noeud a été retiré !");

}

/\*\*

\* Méthode appelée lorsque la structure d'un noeud a été modifiée

\*/

public void treeStructureChanged(TreeModelEvent event) {

System.out.println("La structure d'un noeud a changé !");

}

});

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree();

//Nous passons notre modèle à notre arbre

//==> On pouvait aussi passer l'objet TreeModel au constructeur du JTree

arbre.setModel(model);

arbre.setRootVisible(false);

//On rend notre arbre éditable

arbre.setEditable(true);

this.getContentPane().add(new JScrollPane(arbre), BorderLayout.CENTER);

}

private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node){

int count = 0;

if(file.isFile())

return new DefaultMutableTreeNode(file.getName());

else{

File[] list = file.listFiles();

if(list == null)

return new DefaultMutableTreeNode(file.getName());

for(File nom : list){

count++;

//Pas plus de 5 enfants par noeud

if(count < 3){

DefaultMutableTreeNode subNode;

if(nom.isDirectory()){

subNode = new DefaultMutableTreeNode(nom.getName()+"\\");

node.add(this.listFile(nom, subNode));

}else{

subNode = new DefaultMutableTreeNode(nom.getName());

}

node.add(subNode);

}

}

return node;

}

}

public static void main(String[] args){

//Nous allons créer des fenêtres avec des looks différents

//Cette instruction permet de récupérer tous les looks du système

try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

}

catch (InstantiationException e) {}

catch (ClassNotFoundException e) {}

catch (UnsupportedLookAndFeelException e) {}

catch (IllegalAccessException e) {}

Fenetre fen = new Fenetre();

}

}

Maintenant, je vous propose d’examiner la manière d'ajouter des nœuds à notre arbre. Pour ce faire, nous allons utiliser un bouton qui va nous demander de spécifier le nom du nouveau nœud, via un JOptionPane.

Voici un code d'exemple :

//CTRL + SHIFT + O pour générer les imports nécessaires

public class Fenetre extends JFrame {

private JTree arbre;

private DefaultMutableTreeNode racine;

private DefaultTreeModel model;

private JButton bouton = new JButton("Ajouter");

public Fenetre(){

this.setSize(200, 300);

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTree");

//On invoque la méthode de construction de l'arbre

listRoot();

bouton.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event) {

if(arbre.getLastSelectedPathComponent() != null){

JOptionPane jop = new JOptionPane();

String nodeName = jop.showInputDialog("Saisir un nom de noeud");

if(nodeName != null && !nodeName.trim().equals("")){

DefaultMutableTreeNode parentNode = (DefaultMutableTreeNode)arbre.getLastSelectedPathComponent();

DefaultMutableTreeNode childNode = new DefaultMutableTreeNode(nodeName);

parentNode.add(childNode);

model.insertNodeInto(childNode, parentNode, parentNode.getChildCount()-1);

model.nodeChanged(parentNode);

}

}

else{

System.out.println("Aucune sélection !");

}

}

});

this.getContentPane().add(bouton, BorderLayout.SOUTH);

this.setVisible(true);

}

private void listRoot(){

this.racine = new DefaultMutableTreeNode();

int count = 0;

for(File file : File.listRoots())

{

DefaultMutableTreeNode lecteur = new DefaultMutableTreeNode(file.getAbsolutePath());

try {

for(File nom : file.listFiles()){

DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName()+"\\");

lecteur.add(this.listFile(nom, node));

}

} catch (NullPointerException e) {}

this.racine.add(lecteur);

}

//Nous créons, avec notre hiérarchie, un arbre

arbre = new JTree();

this.model = new DefaultTreeModel(this.racine);

arbre.setModel(model);

arbre.setRootVisible(false);

arbre.setEditable(true);

arbre.getModel().addTreeModelListener(new TreeModelListener() {

public void treeNodesChanged(TreeModelEvent evt) {

System.out.println("Changement dans l'arbre");

Object[] listNoeuds = evt.getChildren();

int[] listIndices = evt.getChildIndices();

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

System.out.println("Index " + listIndices[i] + ", noeud déclencheur : " + listNoeuds[i]);

}

}

public void treeNodesInserted(TreeModelEvent event) {

System.out.println("Un noeud a été inséré !");

}

public void treeNodesRemoved(TreeModelEvent event) {

System.out.println("Un noeud a été retiré !");

}

public void treeStructureChanged(TreeModelEvent event) {

System.out.println("La structure d'un noeud a changé !");

}

});

this.getContentPane().add(new JScrollPane(arbre), BorderLayout.CENTER);

}

private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node){

int count = 0;

if(file.isFile())

return new DefaultMutableTreeNode(file.getName());

else{

File[] list = file.listFiles();

if(list == null)

return new DefaultMutableTreeNode(file.getName());

for(File nom : list){

count++;

//Pas plus de 5 enfants par noeud

if(count < 3){

DefaultMutableTreeNode subNode;

if(nom.isDirectory()){

subNode = new DefaultMutableTreeNode(nom.getName()+"\\");

node.add(this.listFile(nom, subNode));

}else{

subNode = new DefaultMutableTreeNode(nom.getName());

}

node.add(subNode);

}

}

return node;

}

}

public static void main(String[] args){

//Nous allons créer des fenêtres avec des look and feel différents

//Cette instruction permet de récupérer tous les look and feel du système

try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

}

catch (InstantiationException e) {}

catch (ClassNotFoundException e) {}

catch (UnsupportedLookAndFeelException e) {}

catch (IllegalAccessException e) {}

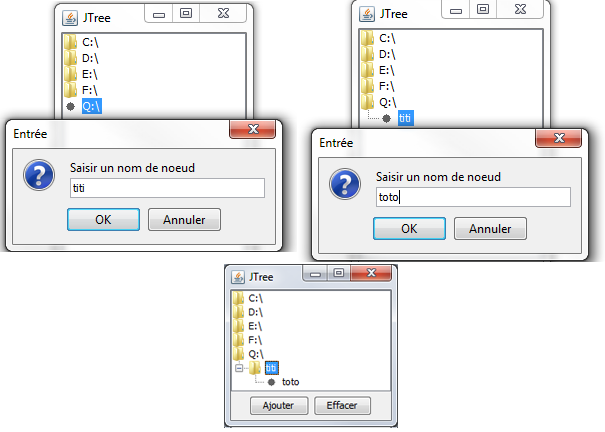
Fenetre fen = new Fenetre();

}

}

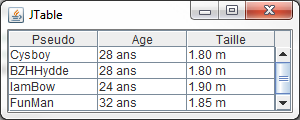
Vous remarquerez que nous avons ajouté des variables d'instances afin d'y avoir accès dans toute notre classe !

La figure suivante nous montre différentes étapes de création de nœuds.



# Les interfaces de tableau

## Premier pas :



Le code source de ce programme est le suivant :

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(300, 120);

//Les données du tableau

Object[][] data = {

{"Cysboy", "28 ans", "1.80 m"},

{"BZHHydde", "28 ans", "1.80 m"},

{"IamBow", "24 ans", "1.90 m"},

{"FunMan", "32 ans", "1.85 m"}

};

//Les titres des colonnes

String title[] = {"Pseudo", "Age", "Taille"};

JTable tableau = new JTable(data, title);

//Nous ajoutons notre tableau à notre contentPane dans un scroll

//Sinon les titres des colonnes ne s'afficheront pas !

this.getContentPane().add(new JScrollPane(tableau));

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

Si on avait pas mis le tableau dans un JScrollPane on aurait dû spécifier où mettre les titres, ils n’auraient pas automatiquement été mis en haut :

//On indique que l'en-tête doit être au nord, donc au-dessus

this.getContentPane().add(tableau.getTableHeader(), BorderLayout.NORTH);

//Et le corps au centre !

this.getContentPane().add(tableau, BorderLayout.CENTER);

## Gestion de l’affichage :



Vous allez voir que le code utilisé est simple comme tout, encore fallait-il que vous sachiez quelles méthodes et quels objets utiliser… Voici le code permettant d'obtenir ce résultat :

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

private JButton retablir = new JButton("Rétablir");

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(300, 240);

Object[][] data = {

{"Cysboy", "28 ans", "1.80 m"},

{"BZHHydde", "28 ans", "1.80 m"},

{"IamBow", "24 ans", "1.90 m"},

{"FunMan", "32 ans", "1.85 m"}

};

String title[] = {"Pseudo", "Age", "Taille"};

this.tableau = new JTable(data, title);

JPanel pan = new JPanel();

change.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

changeSize(200, 80);

change.setEnabled(false);

retablir.setEnabled(true);

}

});

retablir.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent arg0) {

changeSize(75, 16);

change.setEnabled(true);

retablir.setEnabled(false);

}

});

retablir.setEnabled(false);

pan.add(change);

pan.add(retablir);

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

this.getContentPane().add(pan, BorderLayout.SOUTH);

}

/\*\*

\* Change la taille d'une ligne et d'une colonne

\* J'ai mis deux boucles afin que vous puissiez voir

\* comment parcourir les colonnes et les lignes

\*/

public void changeSize(int width, int height){

//Nous créons un objet TableColumn afin de travailler sur notre colonne

TableColumn col;

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

if(i == 1){

//On récupère le modèle de la colonne

col = tableau.getColumnModel().getColumn(i);

//On lui affecte la nouvelle valeur

col.setPreferredWidth(width);

}

}

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

//On affecte la taille de la ligne à l'indice spécifié !

if(i == 1)

tableau.setRowHeight(i, height);

}

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

On a un modèle (*getColumnModel*) puisqu’en réalité il y aun objet entre le tableau et les données, c’est lui qui stock réellement les données. Il peut contenir tout ce qui hérite de Object.

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(600, 140);

Object[][] data = {

{"Cysboy", new JButton("6boy"), new Double(1.80), new Boolean(true)},

{"BZHHydde", new JButton("BZH"), new Double(1.78), new Boolean(false)},

{"IamBow", new JButton("BoW"), new Double(1.90), new Boolean(false)},

{"FunMan", new JButton("Year"), new Double(1.85), new Boolean(true)}

};

String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

this.tableau = new JTable(data, title);

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

On peut créer son propre modèle dans une table abstraite :

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(600, 140);

Object[][] data = {

{"Cysboy", new JButton("6boy"), new Double(1.80), new Boolean(true)},

{"BZHHydde", new JButton("BZH"), new Double(1.78), new Boolean(false)},

{"IamBow", new JButton("BoW"), new Double(1.90), new Boolean(false)},

{"FunMan", new JButton("Year"), new Double(1.85), new Boolean(true)}

};

String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

ZModel model = new ZModel(data, title);

System.out.println("Nombre de colonne : " + model.getColumnCount());

System.out.println("Nombre de ligne : " + model.getRowCount());

this.tableau = new JTable(model);

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

}

//Classe modèle personnalisée

class ZModel extends AbstractTableModel{

private Object[][] data;

private String[] title;

//Constructeur

public ZModel(Object[][] data, String[] title){

this.data = data;

this.title = title;

}

//Retourne le nombre de colonnes

public int getColumnCount() {

return this.title.length;

}

//Retourne le nombre de lignes

public int getRowCount() {

return this.data.length;

}

//Retourne la valeur à l'emplacement spécifié

public Object getValueAt(int row, int col) {

return this.data[row][col];

}

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}



Pour réobtenir les titres, on doit redéfinir aussi les autres méthodes :

/\*\*

\* Retourne le titre de la colonne à l'indice spécifié

\*/

public String getColumnName(int col) {

return this.title[col];

}

Pour avoir encore un meilleur rendu avec des checkbox au lieu d’avoir true ouf alse :

//Retourne la classe de la donnée de la colonne

public Class getColumnClass(int col){

//On retourne le type de la cellule à la colonne demandée

//On se moque de la ligne puisque les types de données sont les mêmes quelle que soit la ligne

//On choisit donc la première ligne

return this.data[0][col].getClass();

}

Rendre nos cellules éditables :

//Retourne vrai si la cellule est éditable : celle-ci sera donc éditable

public boolean isCellEditable(int row, int col){

return true;

}

Et préciser que le bouton lui, n’est pas éditable :

//Retourne vrai si la cellule est éditable : celle-ci sera donc éditable

public boolean isCellEditable(int row, int col){

//On appelle la méthode getValueAt qui retourne la valeur d'une cellule

//Et on effectue un traitement spécifique si c'est un JButton

if(getValueAt(0, col) instanceof JButton)

return false;

return true;

}

### Contrôler l’affichage :

Il faut bien distinguer un TableModel d'un DefaultTableCellRenderer. Le premier fait le lien entre les données et le tableau tandis que le second s'occupe de gérer l'affichage dans les cellules !

Le but du jeu est de définir une nouvelle façon de dessiner les composants dans les cellules. En définitive, nous n'allons pas vraiment faire cela, mais dire à notre tableau que la valeur contenue dans une cellule donnée est un composant (bouton ou autre). Il suffit de créer une classe héritant de DefaultTableCellRenderer et de redéfinir la méthode public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column).

Il y en a, des paramètres ! Mais, dans le cas qui nous intéresse, nous n'avons besoin que d'un seul d'entre eux : value. Remarquez que cette méthode retourne un objet Component. Nous allons seulement spécifier le type d'objet dont il s'agit suivant le cas.

Regardez notre classe héritée :

//CTRL + SHIFT + O pour générer les imports

public class TableComponent extends DefaultTableCellRenderer {

public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {

//Si la valeur de la cellule est un JButton, on transtype cette valeur

if (value instanceof JButton)

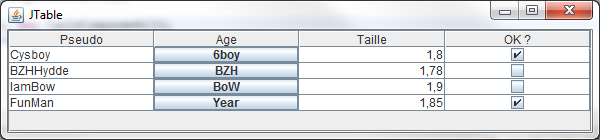
return (JButton) value;

else

return this;

}

}



Il en va de même pour les JCombo :

Exemple, si on veut mettre dans le tableau :

Object[][] data = {

{"Cysboy", new JButton("6boy"), new JComboBox(new String[]{"toto", "titi", "tata"}), new Boolean(true)},

{"BZHHydde", new JButton("BZH"), new JComboBox(new String[]{"toto", "titi", "tata"}), new Boolean(false)},

{"IamBow", new JButton("BoW"), new JComboBox(new String[]{"toto", "titi", "tata"}), new Boolean(false)},

{"FunMan", new JButton("Year"), new JComboBox(new String[]{"toto", "titi", "tata"}), new Boolean(true)}

};

Les boutons s'affichent toujours, mais pas les combos ! Je sais que certains d'entre vous ont presque trouvé la solution. Vous n'auriez pas ajouté ce qui suit dans votre objet de rendu de cellule ?

//CTRL + SHIFT + O pour générer les imports

public class TableComponent extends DefaultTableCellRenderer {

public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column) {

if (value instanceof JButton)

return (JButton) value;

//Lignes ajoutées

else if(value instanceof JComboBox)

return (JComboBox) value;

else

return this;

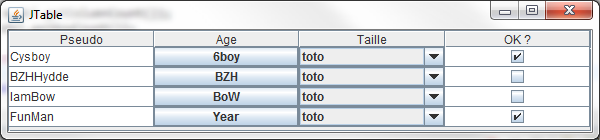
}

}

Ceux qui ont fait cela ont trouvé la première partie de la solution ! Vous avez bien spécifié à votre objet de retourner une valeur **castée** en cas de rencontre avec une combo. Seulement, et j'en suis quasiment sûr, vous avez dû oublier de dire à votre tableau de faire attention aux boutons et aux combos ! Rappelez-vous cette instruction :

this.tableau.setDefaultRenderer(JButton.class, new TableComponent());

Votre tableau ne fait attention qu'aux boutons ! Pour corriger le tir, il faut simplement changer JButton.class en JComponent.class. Après avoir fait ces deux modifications, vous devriez parvenir à la figure suivante.



Faire réagir nos composants : Méthode type !

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(600, 180);

Object[][] data = {

{"Cysboy", "6boy", "Combo", new Boolean(true)},

{"BZHHydde", "BZH", "Combo", new Boolean(false)},

{"IamBow", "BoW", "Combo", new Boolean(false)},

{"FunMan", "Year", "Combo", new Boolean(true)}

};

String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

this.tableau = new JTable(data, title);

this.tableau.setRowHeight(30);

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

De là, nous allons créer une classe qui affichera un bouton dans les cellules de la seconde colonne et une combo dans les cellules de la troisième colonne… Le principe est simple : créer une classe qui héritera de la classe JButton et qui implémentera l'interface TableCellRenderer. Nous allons ensuite dire à notre tableau qu'il doit utiliser utiliser ce type de rendu pour la seconde colonne.

Voici notre classe ButtonRenderer :

//CTRL + SHIFT + O pour générer les imports

public class ButtonRenderer extends JButton implements TableCellRenderer{

public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean isFocus, int row, int col) {

//On écrit dans le bouton ce que contient la cellule

setText((value != null) ? value.toString() : "");

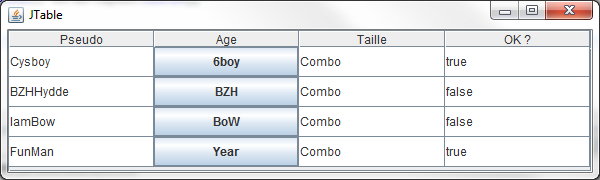
//On retourne notre bouton

return this;

}

}

Il nous suffit maintenant de mettre à jour le tableau grâce à l'instruction this.tableau.getColumn("Age").setCellRenderer(new ButtonRenderer()); : on récupère la colonne grâce à son titre, puis on applique le rendu !  
Résultat en figure suivante.



Votre bouton est de nouveau éditable, mais ce problème sera réglé par la suite… Pour le rendu de la cellule numéro 3, je vous laisse un peu chercher, ce n'est pas très difficile maintenant que vous avez appris cette méthode.

Voici le code ; la figure suivante vous montre le résultat.

//CTRL + SHIFT + O pour générer les imports

public class ComboRenderer extends JComboBox implements TableCellRenderer {

public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean isFocus, int row, int col) {

this.addItem("Très bien");

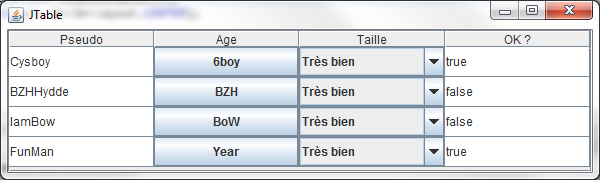
this.addItem("Bien");

this.addItem("Mal");

return this;

}

}



## Interaction avec l’objet JTable :

Dire au tableau de faire des actions spécifiques grace au bouton :

//CTRL + SHIFT + O pour générer les imports

public class ButtonEditor extends DefaultCellEditor {

protected JButton button;

private boolean isPushed;

private ButtonListener bListener = new ButtonListener();

//Constructeur avec une CheckBox

public ButtonEditor(JCheckBox checkBox) {

//Par défaut, ce type d'objet travaille avec un JCheckBox

super(checkBox);

//On crée à nouveau un bouton

button = new JButton();

button.setOpaque(true);

//On lui attribue un listener

button.addActionListener(bListener);

}

public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column) {

//On précise le numéro de ligne à notre listener

bListener.setRow(row);

//Idem pour le numéro de colonne

bListener.setColumn(column);

//On passe aussi le tableau en paramètre pour des actions potentielles

bListener.setTable(table);

//On réaffecte le libellé au bouton

button.setText( (value == null) ? "" : value.toString() );

//On renvoie le bouton

return button;

}

//Notre listener pour le bouton

class ButtonListener implements ActionListener{

private int column, row;

private JTable table;

private int nbre = 0;

public void setColumn(int col){this.column = col;}

public void setRow(int row){this.row = row;}

public void setTable(JTable table){this.table = table;}

public void actionPerformed(ActionEvent event) {

//On affiche un message, mais vous pourriez effectuer les traitements que vous voulez

System.out.println("coucou du bouton : " + ((JButton)event.getSource()).getText());

//On affecte un nouveau libellé à une autre cellule de la ligne

table.setValueAt("New Value " + (++nbre), this.row, (this.column -1));

}

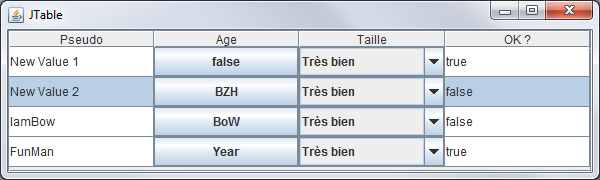
}

}

Maintenant, afin d'utiliser cet objet avec notre tableau, nous allons lui indiquer l'existence de cet éditeur dans la colonne correspondante avec cette instruction :

this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new JCheckBox()));

Le rendu, visible à la figure suivante, est très probant.



Voici le nouveau code du constructeur de notre fenêtre :

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

//Contenu de notre combo

private String[] comboData = {"Très bien", "Bien", "Mal"};

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(600, 180);

//Données de notre tableau

Object[][] data = {

{"Cysboy", "6boy", comboData[0], new Boolean(true)},

{"BZHHydde", "BZH", comboData[0], new Boolean(false)},

{"IamBow", "BoW", comboData[0], new Boolean(false)},

{"FunMan", "Year", comboData[0], new Boolean(true)}

};

//Titre du tableau

String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

//Combo à utiliser

JComboBox combo = new JComboBox(comboData);

this.tableau = new JTable(data, title);

this.tableau.setRowHeight(30);

this.tableau.getColumn("Age").setCellRenderer(new ButtonRenderer());

this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new JCheckBox()));

//On définit l'éditeur par défaut pour la cellule en lui spécifiant quel type d'affichage prendre en compte

this.tableau.getColumn("Taille").setCellEditor(new DefaultCellEditor(combo));

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

}

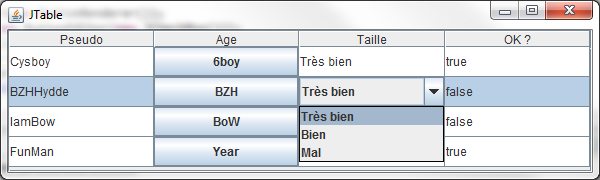
public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}



Nous allons donc utiliser un modèle de tableau personnalisé où les actions seront définies par nos soins. Voici la classe Fenetre modifiée en conséquence :

//CTRL + SHIFT + O pour générer les imports

public class Fenetre extends JFrame {

private JTable tableau;

private JButton change = new JButton("Changer la taille");

//Contenu de notre combo

private String[] comboData = {"Très bien", "Bien", "Mal"};

public Fenetre(){

this.setLocationRelativeTo(null);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setTitle("JTable");

this.setSize(600, 180);

//Données de notre tableau

Object[][] data = {

{"Cysboy", "6boy", comboData[0], new Boolean(true)},

{"BZHHydde", "BZH", comboData[0], new Boolean(false)},

{"IamBow", "BoW", comboData[0], new Boolean(false)},

{"FunMan", "Year", comboData[0], new Boolean(true)}

};

String title[] = {"Pseudo", "Age", "Taille", "OK ?"};

JComboBox combo = new JComboBox(comboData);

//Nous devons utiliser un modèle d'affichage spécifique pour pallier les bugs d'affichage !

ZModel zModel = new ZModel(data, title);

this.tableau = new JTable(zModel);

this.tableau.setRowHeight(30);

this.tableau.getColumn("Age").setCellRenderer(new ButtonRenderer());

this.tableau.getColumn("Age").setCellEditor(new ButtonEditor(new JCheckBox()));

//On définit l'éditeur par défaut pour la cellule

//En lui spécifiant quel type d'affichage prendre en compte

this.tableau.getColumn("Taille").setCellEditor(new DefaultCellEditor(combo));

this.getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

}

class ZModel extends AbstractTableModel{

private Object[][] data;

private String[] title;

//Constructeur

public ZModel(Object[][] data, String[] title){

this.data = data;

this.title = title;

}

//Retourne le titre de la colonne à l'indice spécifié

public String getColumnName(int col) {

return this.title[col];

}

//Retourne le nombre de colonnes

public int getColumnCount() {

return this.title.length;

}

//Retourne le nombre de lignes

public int getRowCount() {

return this.data.length;

}

//Retourne la valeur à l'emplacement spécifié

public Object getValueAt(int row, int col) {

return this.data[row][col];

}

//Définit la valeur à l'emplacement spécifié

public void setValueAt(Object value, int row, int col) {

//On interdit la modification sur certaines colonnes !

if(!this.getColumnName(col).equals("Age") && !this.getColumnName(col).equals("Suppression"))

this.data[row][col] = value;

}

//Retourne la classe de la donnée de la colonne

public Class getColumnClass(int col){

//On retourne le type de la cellule à la colonne demandée

//On se moque de la ligne puisque les types de données sont les mêmes quelle que soit la ligne

//On choisit donc la première ligne

return this.data[0][col].getClass();

}

public boolean isCellEditable(int row, int col){

return true;

}

}

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

Vous aurez remarqué que nous construisons notre tableau par le biais de notre modèle, ce qui implique que nous devrons également passer par le modèle pour modifier le tableau ! Conséquence directe : il va falloir modifier un peu notre classe ButtonEditor.

Voici la classe ButtonEditor utilisant le modèle de tableau pour gérer la modification des valeurs :

//CTRL + SHIFT + O pour générer les imports

public class ButtonEditor extends DefaultCellEditor {

protected JButton button;

private boolean isPushed;

private ButtonListener bListener = new ButtonListener();

public ButtonEditor(JCheckBox checkBox) {

super(checkBox);

button = new JButton();

button.setOpaque(true);

button.addActionListener(bListener);

}

public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column) {

//On affecte le numéro de ligne au listener

bListener.setRow(row);

//Idem pour le numéro de colonne

bListener.setColumn(column);

//On passe aussi le tableau en paramètre pour des actions potentielles

bListener.setTable(table);

//On réaffecte le libellé au bouton

button.setText( (value == null) ? "" : value.toString() );

//On renvoie le bouton

return button;

}

//Notre listener pour le bouton

class ButtonListener implements ActionListener{

private int column, row;

private JTable table;

private int nbre = 0;

private JButton button;

public void setColumn(int col){this.column = col;}

public void setRow(int row){this.row = row;}

public void setTable(JTable table){this.table = table;}

public JButton getButton(){return this.button;}

public void actionPerformed(ActionEvent event) {

System.out.println("coucou du bouton : " + ((JButton)event.getSource()).getText());

//On affecte un nouveau libellé à une celulle de la ligne

((AbstractTableModel)table.getModel()).setValueAt("New Value " + (++nbre), this.row, (this.column -1));

//Permet de dire à notre tableau qu'une valeur a changé à l'emplacement déterminé par les valeurs passées en paramètres

((AbstractTableModel)table.getModel()).fireTableCellUpdated(this.row, this.column - 1);

this.button = ((JButton)event.getSource());

}

}

}

## Ajouter des lignes et des colonnes :

Une autre façon d'initialiser un tableau :

//data et title sont toujours nos tableaux d'objets !

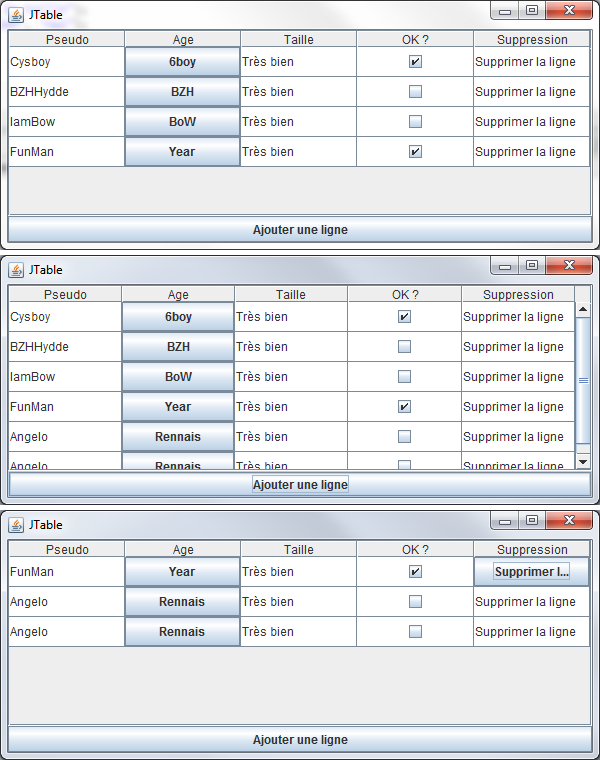
JTable tableau = new JTable(new DefaultTableModel(data, title));

Le modèle par défaut défini lors de la création du tableau nous donne accès à deux méthodes fort utiles :

* addRow(Object[] lineData) : ajoute une ligne au modèle et met automatiquement à jour le tableau ;
* removeRow(int row) : efface une ligne du modèle et met automatiquement à jour le tableau.

Avant de pouvoir utiliser ce modèle, nous allons devoir le récupérer. En fait, c'est notre tableau qui va nous le fournir en invoquant la méthode getModel() qui retourne un objet TableModel. Attention, un cast sera nécessaire afin de pouvoir utiliser l'objet récupéré ! Par exemple : ((ZModel)table.getModel()).removeRow().

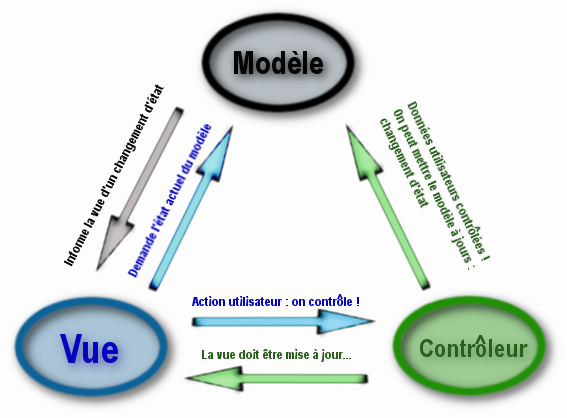
Au final, la figure suivante nous montre ce que nous obtiendrons.



Code complet du cours <https://openclassrooms.com/codeweb/433848>

# MVC

## Premiers pas :



**La vue :** Une IHM.

**Le modèle :** Les données

**Le contrôleur :** Contrôle les données.

Dans une application structurée en MVC, voici ce qu'il peut se passer :

* l'utilisateur effectue une action sur votre calculatrice (un clic sur un bouton) ;
* l'action est captée par le contrôleur, qui va vérifier la cohérence des données et éventuellement les transformer afin que le modèle les comprenne. Le contrôleur peut aussi demander à la vue de changer ;
* le modèle reçoit les données et change d'état (une variable qui change, par exemple) ;
* le modèle notifie la vue (ou les vues) qu'il faut se mettre à jour ;
* l'affichage dans la vue (ou les vues) est modifié en conséquence en allant chercher l'état du modèle.

## Le modèle :

Notre logiciel devra :

* récupérer et stocker au moins un nombre ;
* stocker l'opérateur de calcul ;
* calculer le résultat ;
* renvoyer le résultat ;
* tout remettre à zéro.

Très bien : voila donc la liste des méthodes que nous trouverons dans notre classe abstraite.  
Comme vous le savez, nous allons utiliser le pattern observer afin de faire communiquer notre modèle avec d'autres objets. Il nous faudra donc une implémentation de ce pattern ; la voici, dans un package com.sdz.observer.

##### Observable.java

package com.sdz.observer;

public interface Observable {

public void addObserver(Observer obs);

public void removeObserver();

public void notifyObserver(String str);

}

##### Observer.java

package com.sdz.observer;

public interface Observer {

public void update(String str);

}

Notre classe abstraite devra donc implémenter ce pattern afin de centraliser les implémentations. Puisque notre supertype implémente le pattern observer, les classes héritant de cette dernière hériteront aussi des méthodes de ce pattern !

Voici donc le code de notre classe abstraite que nous placerons dans le package  
com.sdz.model.

##### **AbstractModel.java**

package com.sdz.model;

import java.util.ArrayList;

import com.sdz.observer.Observable;

import com.sdz.observer.Observer;

public abstract class AbstractModel implements Observable{

protected double result = 0;

protected String operateur = "", operande = "";

private ArrayList<Observer> listObserver = new ArrayList<Observer>();

//Efface

public abstract void reset();

//Effectue le calcul

public abstract void calcul();

//Affichage forcé du résultat

public abstract void getResultat();

//Définit l'opérateur de l'opération

public abstract void setOperateur(String operateur);

//Définit le nombre à utiliser pour l'opération

public abstract void setNombre(String nbre) ;

//Implémentation du pattern observer

public void addObserver(Observer obs) {

this.listObserver.add(obs);

}

public void notifyObserver(String str) {

if(str.matches("^0[0-9]+"))

str = str.substring(1, str.length());

for(Observer obs : listObserver)

obs.update(str);

}

public void removeObserver() {

listObserver = new ArrayList<Observer>();

}

}

Ce code est clair et simple à comprendre. Maintenant, nous allons créer une classe concrète héritant de AbstractModel.

Voici la classe concrète que j'ai créée.

##### Calculator.java

package com.sdz.model;

import com.sdz.observer.Observable;

public class Calculator extends AbstractModel{

//Définit l'opérateur

public void setOperateur(String ope){

//On lance le calcul

calcul();

//On stocke l'opérateur

this.operateur = ope;

//Si l'opérateur n'est pas =

if(!ope.equals("=")){

//On réinitialise l'opérande

this.operande = "";

}

}

//Définit le nombre

public void setNombre(String result){

//On concatène le nombre

this.operande += result;

//On met à jour

notifyObserver(this.operande);

}

//Force le calcul

public void getResultat() {

calcul();

}

//Réinitialise tout

public void reset(){

this.result = 0;

this.operande = "0";

this.operateur = "";

//Mise à jour !

notifyObserver(String.valueOf(this.result));

}

//Calcul

public void calcul(){

//S'il n'y a pas d'opérateur, le résultat est le nombre saisi

if(this.operateur.equals("")){

this.result = Double.parseDouble(this.operande);

}

else{

//Si l'opérande n'est pas vide, on calcule avec l'opérateur de calcul

if(!this.operande.equals("")){

if(this.operateur.equals("+"))

this.result += Double.parseDouble(this.operande);

if(this.operateur.equals("-"))

this.result -= Double.parseDouble(this.operande);

if(this.operateur.equals("\*"))

this.result \*= Double.parseDouble(this.operande);

if(this.operateur.equals("/")){

try{

this.result /= Double.parseDouble(this.operande);

}catch(ArithmeticException e){

this.result = 0;

}

}

}

}

this.operande = "";

//On lance aussi la mise à jour !

notifyObserver(String.valueOf(this.result));

}

}

## Le contrôleur :

Que doit faire notre contrôleur? C'est lui qui va intercepter les actions de l'utilisateur, qui va modeler les données et les envoyer au modèle. Il devra donc :

* agir lors d'un clic sur un chiffre ;
* agir lors d'un clic sur un opérateur ;
* avertir le modèle pour qu'il se réinitialise dans le cas d'un clic sur le bouton reset ;
* contrôler les données.

Voilà donc notre liste de méthodes pour cet objet. Cependant, puisque notre contrôleur doit interagir avec le modèle, il faudra qu'il possède une instance de notre modèle.

Voici donc le code source de notre superclasse de contrôle.

##### AbstractControler.java

package com.sdz.controler;

import java.util.ArrayList;

import com.sdz.model.AbstractModel;

public abstract class AbstractControler {

protected AbstractModel calc;

protected String operateur = "", nbre = "";

protected ArrayList<String> listOperateur = new ArrayList<String>();

public AbstractControler(AbstractModel cal){

this.calc = cal;

//On définit la liste des opérateurs

//Afin de s'assurer qu'ils sont corrects

this.listOperateur.add("+");

this.listOperateur.add("-");

this.listOperateur.add("\*");

this.listOperateur.add("/");

this.listOperateur.add("=");

}

//Définit l'opérateur

public void setOperateur(String ope){

this.operateur = ope;

control();

}

//Définit le nombre

public void setNombre(String nombre){

this.nbre = nombre;

control();

}

//Efface

public void reset(){

this.calc.reset();

}

//Méthode de contrôle

abstract void control();

}

Nous avons défini les actions globales de notre objet de contrôle et vous constatez aussi qu'à chaque action dans notre contrôleur, celui-ci invoque la méthode control(). Celle-ci va vérifier les données et informer le modèle en conséquence.

Nous allons voir maintenant ce que doit effectuer notre instance concrète. Voici donc, sans plus tarder, notre classe.

##### CalculetteControler.java

package com.sdz.controler;

import com.sdz.model.AbstractModel;

public class CalculetteControler extends AbstractControler {

public CalculetteControler(AbstractModel cal) {

super(cal);

}

public void control() {

//On notifie le modèle d'une action si le contrôle est bon

//--------------------------------------------------------

//Si l'opérateur est dans la liste

if(this.listOperateur.contains(this.operateur)){

//Si l'opérateur est =

if(this.operateur.equals("="))

this.calc.getResultat(); //On ordonne au modèle d'afficher le résultat

//Sinon, on passe l'opérateur au modèle

else

this.calc.setOperateur(this.operateur);

}

//Si le nombre est conforme

if(this.nbre.matches("^[0-9.]+$"))

this.calc.setNombre(this.nbre);

this.operateur = "";

this.nbre = "";

}

}

## La vue :

##### Calculette.java

package com.sdz.vue;

//CTRL + SHIFT + O pour générer les imports

public class Calculette extends JFrame implements Observer{

private JPanel container = new JPanel();

String[] tab\_string = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "0", ".", "=", "C", "+", "-", "\*", "/"};

JButton[] tab\_button = new JButton[tab\_string.length];

private JLabel ecran = new JLabel();

private Dimension dim = new Dimension(50, 40);

private Dimension dim2 = new Dimension(50, 31);

private double chiffre1;

private boolean clicOperateur = false, update = false;

private String operateur = "";

//L'instance de notre objet contrôleur

private AbstractControler controler;

public Calculette(AbstractControler controler){

this.setSize(240, 260);

this.setTitle("Calculette");

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setLocationRelativeTo(null);

this.setResizable(false);

initComposant();

this.controler = controler;

this.setContentPane(container);

this.setVisible(true);

}

private void initComposant(){

Font police = new Font("Arial", Font.BOLD, 20);

ecran = new JLabel("0");

ecran.setFont(police);

ecran.setHorizontalAlignment(JLabel.RIGHT);

ecran.setPreferredSize(new Dimension(220, 20));

JPanel operateur = new JPanel();

operateur.setPreferredSize(new Dimension(55, 225));

JPanel chiffre = new JPanel();

chiffre.setPreferredSize(new Dimension(165, 225));

JPanel panEcran = new JPanel();

panEcran.setPreferredSize(new Dimension(220, 30));

//Nous utiliserons le même listener pour tous les opérateurs

OperateurListener opeListener = new OperateurListener();

for(int i = 0; i < tab\_string.length; i++)

{

tab\_button[i] = new JButton(tab\_string[i]);

tab\_button[i].setPreferredSize(dim);

switch(i){

case 11 :

tab\_button[i].addActionListener(opeListener);

chiffre.add(tab\_button[i]);

break;

case 12 :

tab\_button[i].setForeground(Color.red);

tab\_button[i].addActionListener(new ResetListener());

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

case 13 :

case 14 :

case 15 :

case 16 :

tab\_button[i].setForeground(Color.red);

tab\_button[i].addActionListener(opeListener);

tab\_button[i].setPreferredSize(dim2);

operateur.add(tab\_button[i]);

break;

default :

chiffre.add(tab\_button[i]);

tab\_button[i].addActionListener(new ChiffreListener());

break;

}

}

panEcran.add(ecran);

panEcran.setBorder(BorderFactory.createLineBorder(Color.black));

container.add(panEcran, BorderLayout.NORTH);

container.add(chiffre, BorderLayout.CENTER);

container.add(operateur, BorderLayout.EAST);

}

//Les listeners pour nos boutons

class ChiffreListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

//On affiche le chiffre en plus dans le label

String str = ((JButton)e.getSource()).getText();

if(!ecran.getText().equals("0"))

str = ecran.getText() + str;

controler.setNombre(((JButton)e.getSource()).getText());

}

}

class OperateurListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

controler.setOperateur(((JButton)e.getSource()).getText());

}

}

class ResetListener implements ActionListener{

public void actionPerformed(ActionEvent arg0) {

controler.reset();

}

}

//Implémentation du pattern observer

public void update(String str) {

ecran.setText(str);

}

}

Vous devez être à même de comprendre ce code, puisqu'il ressemble beaucoup à notre calculette réalisée dans le TP du chapitre correspondant. Vous constaterez que la vue contient le contrôleur (juste avant le constructeur de la classe).

Toutes nos classes sont à présent opérationnelles. Il ne nous manque plus qu'une classe de test afin d'observer le résultat. Elle crée les trois composants qui vont dialoguer entre eux : le modèle (données), la vue (fenêtre) et le contrôleur qui lie les deux.  
La voici :

import com.sdz.controler.\*;

import com.sdz.model.\*;

import com.sdz.vue.Calculette;

public class Main {

public static void main(String[] args) {

//Instanciation de notre modèle

AbstractModel calc = new Calculator();

//Création du contrôleur

AbstractControler controler = new CalculetteControler(calc);

//Création de notre fenêtre avec le contrôleur en paramètre

Calculette calculette = new Calculette(controler);

//Ajout de la fenêtre comme observer de notre modèle

calc.addObserver(calculette);

}

}

# Le drag’n drop

## Présentation :

//CTRL + SHIFT + O pour générer les imports

public class Test1 extends JFrame{

public Test1(){

super("Test de Drag'n Drop");

setSize(300, 200);

JPanel pan = new JPanel();

pan.setBackground(Color.white);

pan.setLayout(new BorderLayout());

//Notre textearea avec son contenu déplaçable

JTextArea label = new JTextArea("Texte déplaçable !");

label.setPreferredSize(new Dimension(300, 130));

//--------------------------------------------------

//C'est cette instruction qui permet le déplacement de son contenu

label.setDragEnabled(true);

//--------------------------------------------------

pan.add(new JScrollPane(label), BorderLayout.NORTH);

JPanel pan2 = new JPanel();

pan2.setBackground(Color.white);

pan2.setLayout(new BorderLayout());

//On crée le premier textfield avec contenu déplaçable

JTextField text = new JTextField();

//--------------------------------------------------

text.setDragEnabled(true);

//--------------------------------------------------

//Et le second, sans

JTextField text2 = new JTextField();

pan2.add(text2, BorderLayout.SOUTH);

pan2.add(text, BorderLayout.NORTH);

pan.add(pan2, BorderLayout.SOUTH);

add(pan, BorderLayout.CENTER);

setVisible(true);

}

public static void main(String[] args){

new Test1();

}

}

Par défaut, le drag'n drop n'est disponible que pour certains composants. D'abord, il ne faut pas confondre l'action « drag » et l'option « drop ». Certains composants autorisent les deux alors que d'autres n'autorisent que le drag. Voici un tableau récapitulatif des actions autorisées par composant :

|  |  |  |
| --- | --- | --- |
| Composant | Drag | Drop |
| JEditorPane | X | X |
| JColorChooser | X | X |
| JFileChooser | X | . |
| JTextPane | X | X |
| JTextField | X | X |
| JTextArea | X | X |
| JFormattedTextField | X | X |
| JPasswordTextField | . | X |
| JLabel | . | . |
| JTable | X | . |
| JTree | X | . |
| JList | X | . |

En fait, le drag'n drop peut avoir plusieurs effets :

* la copie ;
* le déplacement.

Afin d'activer le drag'n drop sur un composant graphique qui ne le permet pas par défaut, nous devons utiliser la méthodesetTransferHandler(TransferHandler newHandler)de l'objetJComponent. Cette méthode prend un objetTransferHandleren paramètre : c'est celui-ci qui lance le mécanisme de drag'n drop.

Les composants du tableau récapitulatif (hormis leJLabel) ont tous un objetTransferHandlerpar défaut. Le drag'n drop s'active par la méthodesetDragEnabled(true)sur la plupart des composants, mais comme vous avez pu le constater, pas sur leJLabel… Afin de contourner cela, nous devons lui spécifier un objetTransferHandlerréalisé par nos soins.

Voici un code permettant de déplacer le texte d'unJLabeldans unJTextField:

//CTRL + SHIFT + O pour générer les imports

public class LabelContentDemo extends JFrame{

public LabelContentDemo(){

setTitle("Drag'n Drop avec un JLabel !");

setSize(300, 100);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JPanel pan = new JPanel();

pan.setLayout(new GridLayout(2,2));

pan.setBackground(Color.white);

JLabel srcLib = new JLabel("Source de drag : ", JLabel.RIGHT);

JLabel src = new JLabel("Texte à déplacer !");

//-------------------------------------------------------------------

//On crée le nouvel objet pour activer le drag'n drop

src.setTransferHandler(new TransferHandler("text"));

//On spécifie au composant qu'il doit envoyer ses données via son objet TransferHandler

src.addMouseListener(new MouseAdapter(){

//On utilise cet événement pour que les actions soient visibles dès le clic de souris…

//Nous aurions pu utiliser mouseReleased, mais, niveau IHM, nous n'aurions rien vu

public void mousePressed(MouseEvent e){

//On récupère le JComponent

JComponent lab = (JComponent)e.getSource();

//Du composant, on récupère l'objet de transfert : le nôtre

TransferHandler handle = lab.getTransferHandler();

//On lui ordonne d'amorcer la procédure de drag'n drop

handle.exportAsDrag(lab, e, TransferHandler.COPY);

}

});

//-------------------------------------------------------------------

JLabel destLib = new JLabel("Destination de drag : ", JLabel.RIGHT);

JTextField dest = new JTextField();

//On active le comportement par défaut de ce composant

dest.setDragEnabled(true);

pan.add(srcLib);

pan.add(src);

pan.add(destLib);

pan.add(dest);

setContentPane(pan);

setVisible(true);

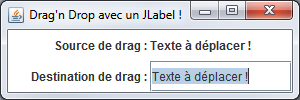
}

public static void main(String[] args){

new LabelContentDemo();

}

}



L'objet de transfert n'a pas de constructeur sans argument ! Cette instruction ne compilera pas :TransferHandler trans = new TransferHandler();. Par contre, le constructeur utilisé fonctionne parfaitement pour unJLabelTransferHandler trans = new TransferHandler("text");. Pourquoi ? Tout simplement parce que la chaîne de caractères passée en paramètre correspond à une propriétéJavaBeanutilisable par l'objet.

UnJavaBeanest un objet Java répondant à certains critères de construction :

* la classe doit êtreSerializablepour pouvoir sauvegarder et restaurer l'état des instances de cette classe ;
* la classe doit posséder un constructeur sans arguments (constructeur par défaut) ;
* les propriétés privées de la classe (variables d'instance) doivent être accessibles publiquement via des méthodes accesseurs (getouset) suivies du nom de la propriété avec la première lettre transformée en majuscule ;
* la classe doit contenir les méthodes d'interception d'événements nécessaires.

Là où les choses deviennent intéressantes, c'est lorsque nous invoquons la méthode handle.exportAsDrag(lab, e, TransferHandler.COPY);. C'est cette instruction qui amorce réellement le drag'n drop. Les trois paramètres servent à initialiser les actions à effectuer et à déterminer quand et sur qui les faire :

* le premier paramètre indique le composant qui contient les données à déplacer ;
* le second paramètre indique à notre objet l'événement sur lequel il doit déclencher le transfert ;
* le dernier indique l'action qui doit être effectuée : copie, déplacement, rien…

Comme je vous l'avais dit, il existe plusieurs types d'actions qui peuvent être effectuées lors du drop, celles-ci sont paramétrables via l'objetTransferHandle:

* TransferHandler.COPY: n'autorise que la copie des données vers le composant cible ;
* TransferHandler.MOVE: n'autorise que le déplacement des données vers le composant cible ;
* TransferHandler.LINK: n'autorise que l'action lien sur les données du composant cible ; cela revient à créer un raccourci ;
* TransferHandler.COPY\_OR\_MOVE: autorise la copie ou le déplacement ;
* TransferHandler.NONE: n'autorise rien.

Attention, l'objetTransferHandlern'accepte que les actionsCOPYlorsqu'il est instancié avec le paramètre « text » : si vous modifiez la valeur ici, votre drag'n drop ne fonctionnera plus.

## Créer son propre handlerTransfer :

Pour faire un couper coller, on doit redéfinir certaine méthode :

Voici la liste des méthodes que nous allons utiliser pour arriver à faire ce que nous cherchons :

import javax.swing.TransferHandler;

public class MyTransferHandler extends TransferHandler{

/\*\*

\* Méthode permettant à l'objet de savoir si les données reçues

\* via un drop sont autorisées à être importées

\* @param info

\* @return boolean

\*/

public boolean canImport(TransferHandler.TransferSupport info) {}

/\*\*

\* C'est ici que l'insertion des données dans notre composant est réalisée

\* @param support

\* @return boolean

\*/

public boolean importData(TransferHandler.TransferSupport support){}

/\*\*

\* Cette méthode est invoquée à la fin de l'action DROP

\* Si des actions sont à faire ensuite, c'est ici qu'il faudra coder le comportement désiré

\* @param c

\* @param t

\* @param action

\*/

protected void exportDone(JComponent c, Transferable t, int action){}

/\*\*

\* Dans cette méthode, nous allons créer l'objet utilisé par le système de drag'n drop

\* afin de faire circuler les données entre les composants

\* Vous pouvez voir qu'il s'agit d'un objet de type Transferable

\* @param c

\* @return

\*/

protected Transferable createTransferable(JComponent c) {}

/\*\*

\* Cette méthode est utilisée afin de déterminer le comportement

\* du composant vis-à-vis du drag'n drop : nous retrouverons

\* nos variables statiques COPY, MOVE, COPY\_OR\_MOVE, LINK ou NONE

\* @param c

\* @return int

\*/

public int getSourceActions(JComponent c) {}

}

Le déplacement se faire grace à cette méthode :

public int getSourceActions(JComponent c) {

//Nous n'autorisons donc que le déplacement ici

return MOVE;

}

Rappelez-vous les classes internes : la classe TransferSupport est à l'intérieur de la classe TransferHandler. Cet objet a un rôle très important : la communication entre les composants. C'est lui qui véhicule l'objet encapsulant nos données. C'est aussi lui, pour des composants plus complexes tels qu'un tableau, un arbre ou une liste, qui fournit l'emplacement où a eu lieu l'action drop.

Voici ce que vont contenir nos méthodes :

public boolean canImport(TransferHandler.TransferSupport info) {

//Nous contrôlons si les données reçues sont d'un type autorisé, ici String

if (!info.isDataFlavorSupported(DataFlavor.stringFlavor)) {

return false;

}

return true;

}

L'objetTransferSupportnous offre une méthode permettant de contrôler le type de données supportées par notre drag'n drop. Une liste de « type MIME » (signifie *Multipurpose Internet Mail Extensions*. C'est une façon de typer certains fichiers comme les images, les PDF, etc.) est disponible dans l'objetDataFlavor. Ici, nous avons utiliséDataFlavor.stringFlavor, qui signifie « chaîne de caractères », comme vous avez pu le deviner. Voici la liste des types d'éléments disponibles via l'objetDataFlavor:

* DataFlavor.javaSerializedObjectMimeType: autorise un objet Java sérialisé correspondant au type MIME « application/x-java-serialized-object » ;
* DataFlavor.imageFlavor: autorise une image, soit la classejava.awt.Imagecorrespondant au type MIME « image/x-java-image » ;
* DataFlavor.javaFileListFlavor: autorise un objetjava.util.Listcontenant des objetsjava.io.File;
* DataFlavor.javaJVMLocalObjectMimeType: autorise n'importe quel objet Java ;
* DataFlavor.javaRemoteObjectMimeType: autorise un objet distant utilisant l'interfaceRemote;
* DataFlavor.stringFlavor: autorise soit une chaîne de caractères, soit la classejava.lang.Stringcorrespondant au type MIME « application/x-java-serialized-object ».

La seconde étape de notre démarche consiste à autoriser l'import de données vers notre composant grâce à la méthodepublic boolean importData(TransferHandler.TransferSupport support):

public boolean importData(TransferHandler.TransferSupport support){

//Nous contrôlons si les données reçues sont d'un type autorisé

if(!canImport(support))

return false;

//On récupère notre objet Transferable, celui qui contient les données en transit

Transferable data = support.getTransferable();

String str = "";

try {

//Nous récupérons nos données en spécifiant ce que nous attendons

str = (String)data.getTransferData(DataFlavor.stringFlavor);

} catch (UnsupportedFlavorException e){

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

//Via le TRansferSupport, nous pouvons récupérer notre composant

JLabel lab = (JLabel)support.getComponent();

//Afin de lui affecter sa nouvelle valeur

lab.setText(str);

return true;

}

Et maintenant, le plus dur : effacer le contenu de notre objet une fois la copie des données effectuée.

//CTRL + SHIFT + O pour générer les imports

public class MyTransferHandler extends TransferHandler{

public boolean canImport(TransferHandler.TransferSupport info) {

if (!info.isDataFlavorSupported(DataFlavor.stringFlavor)) {

return false;

}

return true;

}

public boolean importData(TransferHandler.TransferSupport support){

if(!canImport(support))

return false;

Transferable data = support.getTransferable();

String str = "";

try {

str = (String)data.getTransferData(DataFlavor.stringFlavor);

} catch (UnsupportedFlavorException e){

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

JLabel lab = (JLabel)support.getComponent();

lab.setText(str);

return false;

}

protected void exportDone(JComponent c, Transferable t, int action){

//Une fois le drop effectué nous effaçons le contenu de notre JLabel

if(action == MOVE)

((JLabel)c).setText("");

}

protected Transferable createTransferable(JComponent c) {

//On retourne un nouvel objet implémentant l'interface Transferable

//StringSelection implémente cette interface, nous l'utilisons donc

return new StringSelection(((JLabel)c).getText());

}

public int getSourceActions(JComponent c) {

return MOVE;

}

}

## Activer le drop sur un JTree :

Grace au TransferSupport, on peut dire où positionner le nouvel élément dans l’arbre.

Avant de poursuivre dans cette voie, rappelez-vous qu'il faut définir l'action que doit effectuer notre composant lors du dépôt de nos données. C'est possible grâce à l'objetDropModeque nous pouvons utiliser via la méthodesetDropMode(DropMode dropMode). Voici la liste des modes disponibles :

* USE\_SELECTION
* ON
* INSERT
* ON\_OR\_INSERT
* INSERT\_COLS
* INSERT\_ROWS
* ON\_OR\_INSERT\_COLS
* ON\_OR\_INSERT\_ROWS

TransferSupport permet de récupérer un objetDropLocationcontenant toutes les informations nécessaires au bon positionnement des données dans le composant cible.

En fait, par l'objetTransfertSupport, vous pourrez déduire un objetDropLocationpropre à votre composant, par exemple :

//Pour récupérer les infos importantes sur un JTree

JTree.DropLocation dl = (JTree.DropLocation)myTransfertSupport.getDropLocation();

//Pour récupérer les infos importantes sur un JTable

JTable.DropLocation dl = (JTable.DropLocation)myTransfertSupport.getDropLocation();

//Pour récupérer les infos importantes sur un JList

JList.DropLocation dl = (JList.DropLocation)myTransfertSupport.getDropLocation();

Avec, on a accès à :

|  |  |  |
| --- | --- | --- |
| JList.DropLocation | JTree.DropLocation | JTable.DropLocation |
| isInsert getIndex | getChildIndex getPath | isInsertRow isInsertColumn getRow getColumn |

##### MyTransferHandler.java

//CTRL + SHIFT + O pour générer les imports

public class MyTransferHandler extends TransferHandler{

public boolean canImport(TransferHandler.TransferSupport info) {

if (!info.isDataFlavorSupported(DataFlavor.stringFlavor)) {

return false;

}

return true;

}

public boolean importData(TransferHandler.TransferSupport support){

if(!canImport(support))

return false;

Transferable data = support.getTransferable();

String str = "";

try {

str = (String)data.getTransferData(DataFlavor.stringFlavor);

} catch (UnsupportedFlavorException e){

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

JLabel lab = (JLabel)support.getComponent();

lab.setText(str);

return false;

}

protected void exportDone(JComponent c, Transferable t, int action){

if(action == MOVE){

JLabel lab = (JLabel)c;

String text = lab.getText();

int indice = Integer.parseInt(text.substring(text.length()-1, text.length()));

lab.setText(text.substring(0, text.length()-1) + (++indice));

}

}

protected Transferable createTransferable(JComponent c) {

return new StringSelection(((JLabel)c).getText());

}

public int getSourceActions(JComponent c) {

return MOVE;

}

}

##### TreeTransferHandler.java

//CTRL + SHIFT + O pour générer les imports

public class TreeTransferHandler extends TransferHandler{

JTree tree;

public TreeTransferHandler(JTree tree){

this.tree = tree;

}

public boolean canImport(TransferHandler.TransferSupport info) {

if (!info.isDataFlavorSupported(DataFlavor.stringFlavor))

return false;

return true;

}

public boolean importData(TransferHandler.TransferSupport support){

if(!canImport(support))

return false;

//On récupère l'endroit du drop via un objet approprié

JTree.DropLocation dl = (JTree.DropLocation)support.getDropLocation();

//Les informations afin de pouvoir créer un nouvel élément

TreePath path = dl.getPath();

int index = dl.getChildIndex();

//Comme pour le JLabel, on récupère les données

Transferable data = support.getTransferable();

String str = "";

try {

str = (String)data.getTransferData(DataFlavor.stringFlavor);

} catch (UnsupportedFlavorException e){

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

//On peut maintenant ajouter le nœud

DefaultMutableTreeNode nouveau = new DefaultMutableTreeNode(str);

//On déduit le nœud parent via le chemin

DefaultMutableTreeNode parent = (DefaultMutableTreeNode)path.getLastPathComponent();

DefaultTreeModel model = (DefaultTreeModel)this.tree.getModel();

index = (index == -1) ? model.getChildCount(path.getLastPathComponent()) : index ;

model.insertNodeInto(nouveau, parent, index);

tree.makeVisible(path.pathByAddingChild(nouveau));

tree.scrollPathToVisible(path);

return true;

}

public int getSourceActions(JComponent c) {

return COPY\_OR\_MOVE;

}

}

##### TreeDragDemo.java

//CTRL + SHIFT + O pour générer les imports

public class TreeDragDemo extends JFrame{

JTree tree;

public TreeDragDemo(){

setTitle("Drag'n Drop avec un JLabel !");

setSize(400, 200);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JPanel pan = new JPanel();

pan.setLayout(new GridLayout(1, 1));

pan.setBackground(Color.white);

JLabel srcLib = new JLabel("Source de drag : ", JLabel.RIGHT);

JLabel src = new JLabel("Noeud 1");

//------------------------------------------------------

//On utilise notre nouvel objet MyTransferHandle

src.setTransferHandler(new MyTransferHandler());

src.addMouseListener(new MouseAdapter(){

public void mousePressed(MouseEvent e){

JComponent lab = (JComponent)e.getSource();

TransferHandler handle = lab.getTransferHandler();

handle.exportAsDrag(lab, e, TransferHandler.MOVE);

}

});

//------------------------------------------------------

JLabel destLib = new JLabel("Destination de drag : ", JLabel.RIGHT);

JTextField dest = new JTextField();

dest.setDragEnabled(true);

tree = new JTree(getModel());

tree.setDragEnabled(true);

tree.setTransferHandler(new TreeTransferHandler(tree));

pan.add(src);

pan.add(new JScrollPane(tree));

//Pour le choix des actions

JComboBox combo = new JComboBox();

combo.addItem("USE\_SELECTION");

combo.addItem("ON");

combo.addItem("INSERT");

combo.addItem("ON\_OR\_INSERT");

combo.addItemListener(new ItemListener(){

public void itemStateChanged(ItemEvent event) {

String value = event.getItem().toString();

if(value.equals("USE\_SELECTION"))

tree.setDropMode(DropMode.USE\_SELECTION);

if(value.equals("ON"))

tree.setDropMode(DropMode.ON);

if(value.equals("INSERT"))

tree.setDropMode(DropMode.INSERT);

if(value.equals("ON\_OR\_INSERT"))

tree.setDropMode(DropMode.ON\_OR\_INSERT);

}

});

add(pan, BorderLayout.CENTER);

add(combo, BorderLayout.SOUTH);

setVisible(true);

}

private TreeModel getModel(){

DefaultMutableTreeNode root = new DefaultMutableTreeNode("SDZ");

DefaultMutableTreeNode forum = new DefaultMutableTreeNode("Forum");

forum.add(new DefaultMutableTreeNode("C++"));

forum.add(new DefaultMutableTreeNode("Java"));

forum.add(new DefaultMutableTreeNode("PHP"));

DefaultMutableTreeNode tuto = new DefaultMutableTreeNode("Tutoriel");

tuto.add(new DefaultMutableTreeNode("Tutoriel"));

tuto.add(new DefaultMutableTreeNode("Programmation"));

tuto.add(new DefaultMutableTreeNode("Mapping"));

root.add(tuto);

root.add(forum);

return new DefaultTreeModel(root);

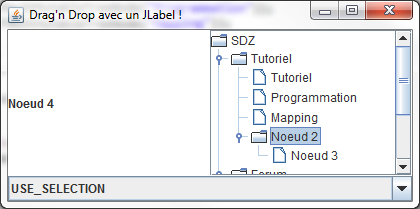
}

public static void main(String[] args){

new TreeDragDemo();

}

}

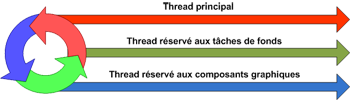


# Mieux gérer les intéractions avec les composants

*EDT :* Event Dispatch Thread. Il s’occupe de toute les modifications sur un composant graphique.

* le redimensionnement ;
* le changement de couleur ;
* le changement de valeur ;
* …

## Présentation :



La politique Java est, tout ce qui touche au graphique doit se faire dans un seul et unique Thread, l’EDT.

Par contre si on a une méthode assez longue dans actionPerformed, ce sera toute l’interface graphique qui sera figée.

Seulement voilà, nous cliquons sur un bouton engendrant un long, un très long traitement dans l'EDT (dernier bloc) : du coup, toute notre IHM est figée ! Non pas parce que Java est lent, mais parce que nous avons exécuté un traitement au mauvais endroit. Il existe toutefois quelques méthodes thread-safe :

* paint() et repaint() ;
* validate(), invalidate() et revalidate().

## Utiliser l’EDT :

Java vous fournit la classe SwingUtilities qui offre plusieurs méthodes statiques permettant d'insérer du code dans l'EDT :

* invokeLater(Runnable doRun) : exécute le thread en paramètre dans l'EDT et rend immédiatement la main au thread principal ;
* invokeAndWait(Runnable doRun) : exécute le thread en paramètre dans l'EDT et attend la fin de celui-ci pour rendre la main au thread principal ;
* isEventDispatchThread() : retourne vrai si le thread dans lequel se trouve l'instruction est dans l'EDT.

//CTRL + SHIFT + O pour générer les imports

public class Test1 {

static int count = 0, count2 = 0;

static JButton bouton = new JButton("Pause");

public static void main(String[] args){

JFrame fen = new JFrame("EDT");

fen.getContentPane().add(bouton);

fen.setSize(200, 100);

fen.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fen.setLocationRelativeTo(null);

fen.setVisible(true);

updateBouton();

System.out.println("Reprise du thread principal");

}

public static void updateBouton(){

for(int i = 0; i < 5; i++){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

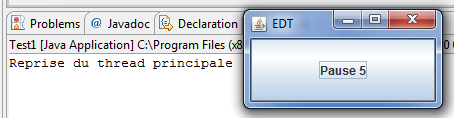
}

bouton.setText("Pause " + ++count);

}

}

}



Ici le thread principal ne reprend la main qu’à la fin de l’instruction.

Solution :

//CTRL + SHIFT + O pour générer les imports

public class Test1 {

static int count = 0;

static JButton bouton = new JButton("Pause");

public static void main(String[] args){

JFrame fen = new JFrame("EDT");

fen.getContentPane().add(bouton);

fen.setSize(200, 100);

fen.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fen.setLocationRelativeTo(null);

fen.setVisible(true);

updateBouton();

System.out.println("Reprise du thread principal");

}

public static void updateBouton(){

//Le second thread

new Thread(new Runnable(){

public void run(){

for(int i = 0; i < 5; i++){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

//Modification de notre composant dans l'EDT

Thread t = new Thread(new Runnable(){

public void run(){

bouton.setText("Pause " + ++count);

}

});

if(SwingUtilities.isEventDispatchThread())

t.start();

else{

System.out.println("Lancement dans l' EDT");

SwingUtilities.invokeLater(t);

}

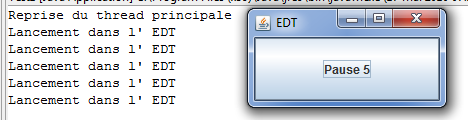
}

}

}).start();

}

}



## La classe SwingWorker<T, V>

Cette dernière est une classe abstraite permettant de réaliser des traitements en tâche de fond tout en dialoguant avec les composants graphiques via l'EDT, aussi bien en cours de traitement qu'en fin de traitement. Dès que vous aurez un traitement prenant pas mal de temps et devant interagir avec votre IHM, pensez aux SwingWorker.

Voici un exemple d'utilisation :

//CTRL + SHIFT + O pour générer les imports

public class Test1 {

static int count = 0;

static JButton bouton = new JButton("Pause");

public static void main(String[] args){

JFrame fen = new JFrame("EDT");

fen.getContentPane().add(bouton);

fen.setSize(200, 100);

fen.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fen.setLocationRelativeTo(null);

fen.setVisible(true);

updateBouton();

System.out.println("Reprise du thread principal");

}

public static void updateBouton(){

//On crée le SwingWorker

SwingWorker sw = new SwingWorker(){

protected Object doInBackground() throws Exception {

for(int i = 0; i < 5; i++){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

return null;

}

public void done(){

if(SwingUtilities.isEventDispatchThread())

System.out.println("Dans l'EDT ! ");

bouton.setText("Traitement terminé");

}

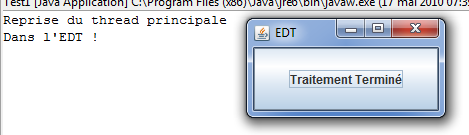
};

//On lance le SwingWorker

sw.execute();

}

}



Je vous disais plus haut que vous pouviez interagir avec l'EDT pendant le traitement. Pour ce faire, il suffit d'utiliser la méthode setProgress(int progress) combinée avec l'événement PropertyChangeListener, qui sera informé du changement d'état de la propriété progress.

Voici un code d'exemple :

//CTRL + SHIFT + O pour générer les imports

public class Test1 {

static int count = 0;

static JButton bouton = new JButton("Pause");

public static void main(String[] args){

JFrame fen = new JFrame("EDT");

fen.getContentPane().add(bouton);

fen.setSize(200, 100);

fen.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fen.setLocationRelativeTo(null);

fen.setVisible(true);

updateBouton();

System.out.println("Reprise du thread principal");

}

public static void updateBouton(){

SwingWorker sw = new SwingWorker(){

protected Object doInBackground() throws Exception {

for(int i = 0; i < 5; i++){

try {

//On change la propriété d'état

setProgress(i);

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

return null;

}

public void done(){

if(SwingUtilities.isEventDispatchThread())

System.out.println("Dans l'EDT ! ");

bouton.setText("Traitement terminé");

}

};

//On écoute le changement de valeur pour la propriété

sw.addPropertyChangeListener(new PropertyChangeListener(){

//Méthode de l'interface

public void propertyChange(PropertyChangeEvent event) {

//On vérifie tout de même le nom de la propriété

if("progress".equals(event.getPropertyName())){

if(SwingUtilities.isEventDispatchThread())

System.out.println("Dans le listener donc dans l'EDT ! ");

//On récupère sa nouvelle valeur

bouton.setText("Pause " + (Integer) event.getNewValue());

}

}

});

//On lance le SwingWorker

sw.execute();

}

}

Les méthodes que vous avez vues jusqu'ici sont issues de la classe SwingWorker, qui implémente l'interface java.util.concurrent.Future, offrant les méthodes suivantes :

* get() : permet à la méthode doInBackground() de renvoyer son résultat à d'autres threads ;
* cancel() : essaie d'interrompre la tâche de doInBackground() en cours ;
* isCancelled() : retourne vrai si l'action a été interrompue ;
* isDone() : retourne vrai si l'action est terminée

Afin de gérer les résultats intermédiaires, vous pouvez utiliser les méthodes suivantes :

* publish(V value) : publie le résultat intermédiaire pour la méthode progress(List<V> list) ;
* progress(List<V> list) : permet d'utiliser le résultat intermédiaire pour un traitement spécifique.

Voici l'exemple utilisé jusqu'ici avec les compléments :

//CTRL + SHIFT + O pour générer les imports

public class Test1 {

static int count = 0;

static JButton bouton = new JButton("Pause");

public static void main(String[] args){

JFrame fen = new JFrame("EDT");

fen.getContentPane().add(bouton);

fen.setSize(200, 100);

fen.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fen.setLocationRelativeTo(null);

fen.setVisible(true);

updateBouton();

System.out.println("Reprise du thread principal");

}

public static void updateBouton(){

//On crée un Worker générique, cette fois

SwingWorker sw = new SwingWorker<Integer, String>(){

protected Integer doInBackground() throws Exception {

int i;

for(i = 0; i < 5; i++){

try {

//On change la propriété d'état

setProgress(i);

//On publie un résultat intermédiaire

publish("Tour de boucle N° " + (i+1));

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

return i;

}

public void done(){

if(SwingUtilities.isEventDispatchThread())

System.out.println("Dans l'EDT ! ");

try {

//On utilise la méthode get() pour récupérer le résultat

//de la méthode doInBackground()

bouton.setText("Traitement terminé au bout de "+get()+" fois !");

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

}

//La méthode gérant les résultats intermédiaires

public void process(List<String> list){

for(String str : list)

System.out.println(str);

}

};

//On écoute le changement de valeur pour la propriété

sw.addPropertyChangeListener(new PropertyChangeListener(){

//Méthode de l'interface

public void propertyChange(PropertyChangeEvent event) {

//On vérifie tout de même le nom de la propriété

if("progress".equals(event.getPropertyName())){

if(SwingUtilities.isEventDispatchThread())

System.out.println("Dans le listener donc dans l'EDT ! ");

//On récupère sa nouvelle valeur

bouton.setText("Pause " + (Integer) event.getNewValue());

}

}

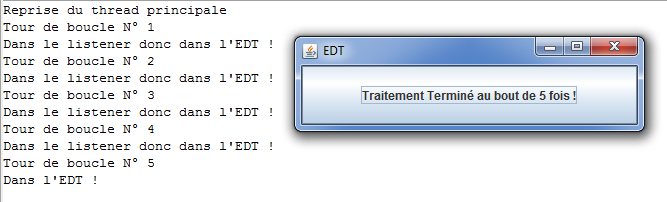
});

//On lance le SwingWorker

sw.execute();

}

}



**Au final :**

* Au lancement d'un programme Java, trois threads se lancent : le thread principal, celui gérant les tâches de fond et l'EDT.
* Java préconise que toute modification des composants graphiques se fasse dans l'EDT.
* Si vos IHM se figent, c'est peut-être parce que vous avez lancé un traitement long dans l'EDT.
* Afin d'améliorer la réactivité de vos applications, vous devez choisir au mieux dans quel thread vous allez traiter vos données.
* Java offre la classe SwingUtilities, qui permet de lancer des actions dans l'EDT depuis n'importe quel thread.
* Depuis Java 6, la classe SwingWorker(<T, V>) vous offre la possibilité de lancer des traitements dans un thread en vous assurant que les mises à jour des composants se feront dans l'EDT.

# JDBC : les BDD

*JDBC* : Java DataBase Connectivity.

Pour utiliser une BDD on a besoin de deux éléments :

* La base de donnée
* Un SGBD (système de gestion de base de donnée)

## Préparer la BDD :

Pour notre base de données, nous allons donc gérer une école dont voici les caractéristiques :

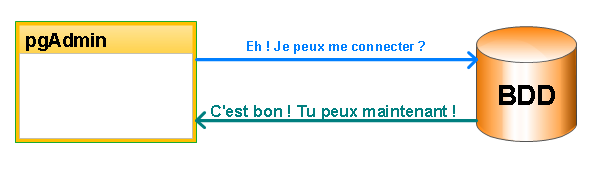
* cette école est composée de classes ;
* chaque classe est composée d'élèves ;
* à chaque classe est attribué un professeur pour chacune des matières dispensées ;
* un professeur peut enseigner plusieurs matières et exercer ses fonctions dans plusieurs classes.



### Créer la base de donnée :

On crée la BDD en SQL, ici on utilise un fichier zip stocké dans JAVA et on l’execute directement dans la BDD pour créer les tables.

## Se connecter à la BDD :



pgAdmin utilise un driver pour se connecter à la BDD. Il nous faut donc les pilotes pour se connecter depuis Java vers une base PostreSQL. C’est un fichier .jar

Tout à fait, il existe un fichier .jar pour se connecter à :

* MySQL ;
* SQL Server ;
* Oracle ;
* d'autres bases.

Sur la page de téléchargement des pilotes pour PostgreSQL, choisissez la dernière version disponible ; pour ma part, j'ai opté pour la version JDBC4. La version JDBC4 offre des nouveautés et une souplesse d'utilisation accrue de JDBC, mais vous devez savoir qu'il existe trois autres types de drivers JDBC ; au total, il en existe donc quatre :

* des drivers JDBC de type 1 : JDBC-ODBC, ce type utilise l'interface ODBC pour se connecter à une base de données (on en a déjà parlé) ; au niveau de la portabilité, on trouve mieux ;
* des drivers JDBC de type 2 : ils intègrent les pilotes natifs et les pilotes Java ; en fait, la partie Java traduit les instructions en natif afin d'être comprises et interprétées par les pilotes natifs ;
* des drivers JDBC de type 3 : écrit entièrement en Java, ce type convertit les appels en un langage totalement indépendant du SGBD ; un serveur intégré traduit ensuite les instructions dans le langage souhaité par le SGBD ;
* des drivers JDBC de type 4 : des pilotes convertissant directement les appels JDBC en instructions compréhensibles par le SGBD ; ce type de drivers est codé et proposé par les éditeurs de BDD.

Où placer l'archive ? Deux solutions :

* l'inclure dans votre projet et l'ajouter au CLASSPATH ;
* la placer dans le dossier lib/ext présent dans le dossier d'installation du JRE.

### Connexion :

Cette connexion est un exemple par rapport à aux identifiants de connexions du cours.

//CTRL + SHIFT + O pour générer les imports

public class Connect {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

System.out.println("Driver O.K.");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

System.out.println("Connexion effective !");

} catch (Exception e) {

e.printStackTrace();

}

}

}

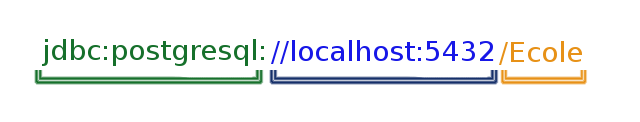
Dans un premier temps, nous avons créé une instance de l'objet Driver présent dans le fichier .jarque nous avons téléchargé. Il est inutile de créer une véritable instance de ce type d'objet ; j'entends par là que l'instruction org.postgres.Driver driver = new org.postgres.Driver() n'est pas nécessaire. Nous utilisons alors la réflexivité afin d'instancier cet objet.

À ce stade, il existe comme un pont entre votre programme Java et votre BDD, mais le trafic routier n'y est pas encore autorisé : il faut qu'une connexion soit effective afin que le programme et la base de données puissent communiquer. Cela se réalise grâce à cette ligne de code :

Connection conn = DriverManager.getConnection(url, user, passwd);

Nous avons défini au préalable trois String contenant respectivement :

* l'URL de connexion ;
* le nom de l'utilisateur ;
* le mot de passe utilisateur.



Pour Java, on commence toujours par jdbc : ensuite ça dépend de la BDD à laquelle on souhaite se connecter et donc à quel driver on a dû télécharger.

Ensuite : //adresseServer :portServer.

Puis : /nomBDD.

L'avantage d'utiliser les fichiers .jar comme drivers de connexion est que vous n'êtes pas tenus d'initialiser le driver par une méthode telle que la réflexivité, tout se passe dans Java. Puisqu'un rappel du protocole à utiliser est présent dans l'URL de connexion, tout est optimal et Java s'en sort tout seul ! Ne vous étonnez donc pas si vous ne voyez plus l'instruction Class.forName("org.postgresql.Driver") par la suite.

# Fouiller dans sa BDD

## Le couple Statement – ResultSet :

Exemple :

//CTRL + SHIFT + O pour générer les imports

public class Connect {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

//Création d'un objet Statement

Statement state = conn.createStatement();

//L'objet ResultSet contient le résultat de la requête SQL

ResultSet result = state.executeQuery("SELECT \* FROM classe");

//On récupère les MetaData

ResultSetMetaData resultMeta = result.getMetaData();

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

//On affiche le nom des colonnes

for(int i = 1; i <= resultMeta.getColumnCount(); i++)

System.out.print("\t" + resultMeta.getColumnName(i).toUpperCase() + "\t \*");

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

while(result.next()){

for(int i = 1; i <= resultMeta.getColumnCount(); i++)

System.out.print("\t" + result.getObject(i).toString() + "\t |");

System.out.println("\n---------------------------------");

}

result.close();

state.close();

} catch (Exception e) {

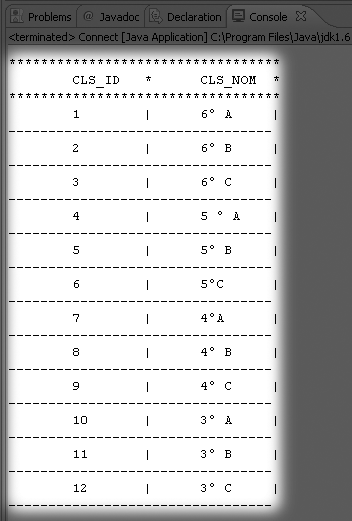
e.printStackTrace();

}

}

}

On obtient comme résultat (avec UTF-8).



Les choses se sont déroulées en quatre étapes distinctes :

* création de l'objet Statement ;
* exécution de la requête SQL ;
* récupération et affichage des données via l'objet ResultSet ;
* fermeture des objets utilisés (bien que non obligatoire, c'est recommandé).

L'objet Statement permet d'exécuter des requêtes SQL. Ces dernières peuvent être de différents types :

* CREATE ;
* INSERT ;
* UPDATE ;
* SELECT ;
* DELETE.

L'objet Statement est fourni par l'objet Connection grâce à l'instruction conn.createStatement().

**Il existe aussi un objet DataBaseMetaData qui fournit des informations sur la base de données.**

Je suis parti du principe que le type de données de mes colonnes était inconnu, mais étant donné que je les connais, le code suivant aurait tout aussi bien fonctionné :

while(result.next()){

System.out.print("\t" + result.getInt("cls\_id") + "\t |");

System.out.print("\t" + result.getString("cls\_nom") + "\t |");

System.out.println("\n---------------------------------");

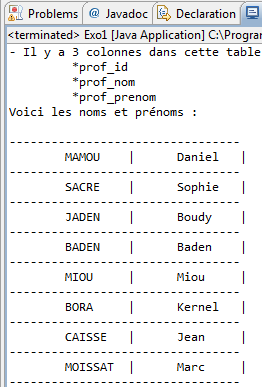
}

Il existe une méthode getXXX() par type primitif ainsi que quelques autres correspondant aux types SQL :

* getArray(int colummnIndex) ;
* getAscii(int colummnIndex) ;
* getBigDecimal(int colummnIndex) ;
* getBinary(int colummnIndex) ;
* getBlob(int colummnIndex) ;
* getBoolean(int colummnIndex) ;
* getBytes(int colummnIndex) ;
* getCharacter(int colummnIndex) ;
* getDate(int colummnIndex) ;
* getDouble(int colummnIndex) ;
* getFloat(int colummnIndex) ;
* getInt(int colummnIndex) ;
* getLong(int colummnIndex) ;
* getObject(int colummnIndex) ;
* getString(int colummnIndex).

Pour finir, je n'ai plus qu'à fermer mes objets à l'aide des instructions result.close() et state.close().

### Autre exemple : le nom des profs :



//CTRL + SHIFT + O pour générer les imports

public class Exo1 {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

Statement state = conn.createStatement();

ResultSet result = state.executeQuery("SELECT \* FROM professeur");

ResultSetMetaData resultMeta = result.getMetaData();

System.out.println("- Il y a " + resultMeta.getColumnCount() + " colonnes dans cette table");

for(int i = 1; i <= resultMeta.getColumnCount(); i++)

System.out.println("\t \*" + resultMeta.getColumnName(i));

System.out.println("Voici les noms et prénoms : ");

System.out.println("\n---------------------------------");

while(result.next()){

System.out.print("\t" + result.getString("prof\_nom") + "\t |");

System.out.print("\t" + result.getString("prof\_prenom") + "\t |");

System.out.println("\n---------------------------------");

}

result.close();

state.close();

} catch (Exception e) {

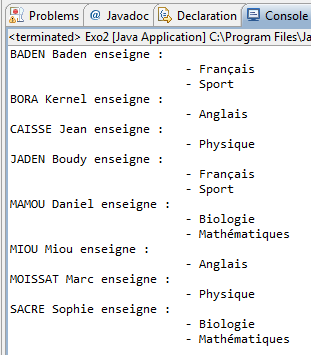
e.printStackTrace();

}

}

}

### La matière que chaque prof enseigne :



//CTRL + SHIFT + O pour générer les imports

public class Exo2 {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

Statement state = conn.createStatement();

String query = "SELECT prof\_nom, prof\_prenom, mat\_nom FROM professeur";

query += " INNER JOIN j\_mat\_prof ON jmp\_prof\_k = prof\_id";

query += " INNER JOIN matiere ON jmp\_mat\_k = mat\_id ORDER BY prof\_nom";

ResultSet result = state.executeQuery(query);

String nom = "";

while(result.next()){

if(!nom.equals(result.getString("prof\_nom"))){

nom = result.getString("prof\_nom");

System.out.println(nom + " " + result.getString("prof\_prenom") + " enseigne : ");

}

System.out.println("\t\t\t - " + result.getString("mat\_nom"));

}

result.close();

state.close();

} catch (Exception e) {

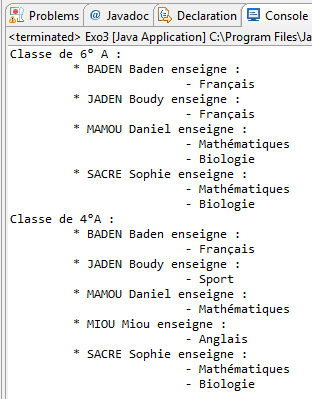
e.printStackTrace();

}

}

}

### Dernier exemple, chaque prof + matière pour chaque classe :



//CTRL + SHIFT + O pour générer les imports

public class Exo3 {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

Statement state = conn.createStatement();

String query = "SELECT prof\_nom, prof\_prenom, mat\_nom, cls\_nom FROM professeur";

query += " INNER JOIN j\_mat\_prof ON jmp\_prof\_k = prof\_id";

query += " INNER JOIN matiere ON jmp\_mat\_k = mat\_id";

query += " INNER JOIN j\_cls\_jmp ON jcm\_jmp\_k = jmp\_id";

query += " INNER JOIN classe ON jcm\_cls\_k = cls\_id AND cls\_id IN(1, 7)";

query += " ORDER BY cls\_nom DESC, prof\_nom";

ResultSet result = state.executeQuery(query);

String nom = "";

String nomClass = "";

while(result.next()){

if(!nomClass.equals(result.getString("cls\_nom"))){

nomClass = result.getString("cls\_nom");

System.out.println("Classe de " + nomClass + " :");

}

if(!nom.equals(result.getString("prof\_nom"))){

nom = result.getString("prof\_nom");

System.out.println("\t \* " + nom + " " + result.getString("prof\_prenom") + " enseigne : ");

}

System.out.println("\t\t\t - " + result.getString("mat\_nom"));

}

result.close();

state.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

### Statement :

Vous avez vu comment obtenir un objet Statement. Mais je ne vous ai pas tout dit… Vous savez déjà que pour récupérer un objet Statement, vous devez le demander gentiment à un objet Connection en invoquant la méthode createStatement(). Ce que vous ne savez pas, c'est que vous pouvez spécifier des paramètres pour la création de l'objet Statement. Ces paramètres permettent différentes actions lors du parcours des résultats via l'objet ResultSet.

Le premier paramètre est utile pour la lecture du jeu d'enregistrements :

* TYPE\_FORWARD\_ONLY : le résultat n'est consultable qu'en avançant dans les données renvoyées, il est donc *impossible* de revenir en arrière lors de la lecture ;
* TYPE\_SCROLL\_SENSITIVE : le parcours peut se faire vers l'avant ou vers l'arrière et le curseur peut se positionner n'importe où, et si des changements surviennent dans la base pendant la lecture, ils seront directement visibles lors du parcours des résultats ;
* TYPE\_SCROLL\_INSENSITIVE : à la différence du précédent, les changements ne seront pas visibles.

Le second concerne la possibilité de mise à jour du jeu d'enregistrements :

* CONCUR\_READONLY : les données sont consultables en lecture seule, c'est-à-dire que l'on ne peut modifier des valeurs pour mettre la base à jour ;
* CONCUR\_UPDATABLE : les données sont modifiables ; lors d'une modification, la base est mise à jour.

Voici comment créer un Statement permettant à l'objet ResultSet de pouvoir être lu d'avant en arrière avec possibilité de modification :

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);

## Les requêtes préparées :

De tels objets sont créés exactement de la même façon que des Statement classiques, sauf qu'au lieu de cette instruction :

Statement stm = conn.createStatement();

… nous devons écrire ceci :

PreparedStatement stm = conn.prepareStatement("SELECT \* FROM classe");

On peut renseigner des requêtes à trou et les compléter ensuite, comme suit :

Exemple :

//CTRL + SHIFT + O pour générer les imports

public class Prepare {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);

//On crée la requête

String query = "SELECT prof\_nom, prof\_prenom FROM professeur";

//Premier trou pour le nom du professeur

query += " WHERE prof\_nom = ?";

//Deuxième trou pour l'identifiant du professeur

query += " OR prof\_id = ?";

//On crée l'objet avec la requête en paramètre

PreparedStatement prepare = conn.prepareStatement(query);

//On remplace le premier trou par le nom du professeur

prepare.setString(1, "MAMOU");

//On remplace le deuxième trou par l'identifiant du professeur

prepare.setInt(2, 2);

//On affiche la requête exécutée

System.out.println(prepare.toString());

//On modifie le premier trou

prepare.setString(1, "TOTO");

//On affiche à nouveau la requête exécutée

System.out.println(prepare.toString());

//On modifie le deuxième trou

prepare.setInt(2, 159753);

//On affiche une nouvelle fois la requête exécutée

System.out.println(prepare.toString());

prepare.close();

state.close();

} catch (Exception e) {

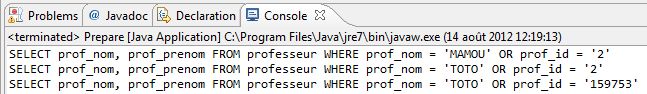
e.printStackTrace();

}

}

}

Cela nous donne la figure suivante.



Vxous vous souvenez de la liste des méthodes de l'objet ResultSet récupérant des données ? Ici, elle est identique, sauf que l'on trouve setXXX() à la place de getXXX(). Tout comme son homologue sans trou, cet objet peut prendre les mêmes types de paramètres pour la lecture et pour la modification des données lues :

PreparedStatement prepare = conn.prepareStatement(

query,

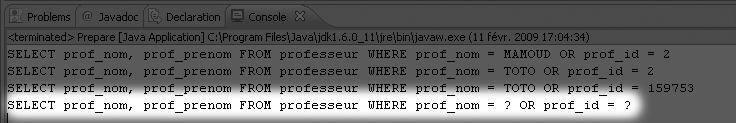
ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY

);

Sachez enfin qu'il existe aussi une méthode retournant un objet ResultSetMetaData : il s'agit de getMetaData().

prepare.clearParameters() permet de réinitialiser la requête préparée afin de retirer toutes les valeurs renseignées.



### ResultSet :

Maintenant que nous avons vu comment procéder, nous allons apprendre à nous promener dans nos objets ResultSet. En fait, l'objet ResultSet offre beaucoup de méthodes permettant d'explorer les résultats, à condition que vous ayez bien préparé l'objet Statement.

Vous avez la possibilité de :

* vous positionner avant la première ligne de votre résultat : res.beforeFirst() ;
* savoir si vous vous trouvez avant la première ligne : res.isBeforeFirst() ;
* vous placer sur la première ligne de votre résultat : res.first() ;
* savoir si vous vous trouvez sur la première ligne : res.isFirst() ;
* vous retrouver sur la dernière ligne : res.last() ;
* savoir si vous vous trouvez sur la dernière ligne : res.isLast() ;
* vous positionner après la dernière ligne de résultat : res.afterLast() ;
* savoir si vous vous trouvez après la dernière ligne : res.isAfterLast() ;
* aller de la première ligne à la dernière : res.next() ;
* aller de la dernière ligne à la première : res.previous() ;
* vous positionner sur une ligne précise de votre résultat : res.absolute(5) ;
* vous positionner sur une ligne par rapport à votre emplacement actuel : res.relative(-3).

//CTRL + SHIFT + O pour générer les imports

public class Resultset {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

String query = "SELECT prof\_nom, prof\_prenom FROM professeur";

ResultSet res = state.executeQuery(query);

int i = 1;

System.out.println("\n\t---------------------------------------");

System.out.println("\tLECTURE STANDARD.");

System.out.println("\t---------------------------------------");

while(res.next()){

System.out.println("\tNom : "+res.getString("prof\_nom") +" \t prénom : "+res.getString("prof\_prenom"));

//On regarde si on se trouve sur la dernière ligne du résultat

if(res.isLast())

System.out.println("\t\t\* DERNIER RESULTAT !\n");

i++;

}

//Une fois la lecture terminée, on contrôle si on se trouve bien à l'extérieur des lignes de résultat

if(res.isAfterLast())

System.out.println("\tNous venons de terminer !\n");

System.out.println("\t---------------------------------------");

System.out.println("\tLecture en sens contraire.");

System.out.println("\t---------------------------------------");

//On se trouve alors à la fin

//On peut parcourir le résultat en sens contraire

while(res.previous()){

System.out.println("\tNom : "+res.getString("prof\_nom") +" \t prénom : "+res.getString("prof\_prenom"));

//On regarde si on se trouve sur la première ligne du résultat

if(res.isFirst())

System.out.println("\t\t\* RETOUR AU DEBUT !\n");

}

//On regarde si on se trouve avant la première ligne du résultat

if(res.isBeforeFirst())

System.out.println("\tNous venons de revenir au début !\n");

System.out.println("\t---------------------------------------");

System.out.println("\tAprès positionnement absolu du curseur à la place N° "+ i/2 + ".");

System.out.println("\t---------------------------------------");

//On positionne le curseur sur la ligne i/2

//Peu importe où on se trouve

res.absolute(i/2);

while(res.next())

System.out.println("\tNom : "+res.getString("prof\_nom") +" \t prénom : "+ res.getString("prof\_prenom"));

System.out.println("\t---------------------------------------");

System.out.println("\tAprès positionnement relatif du curseur à la place N° "+(i-(i-2)) + ".");

System.out.println("\t---------------------------------------");

//On place le curseur à la ligne actuelle moins i-2

//Si on n'avait pas mis de signe moins, on aurait avancé de i-2 lignes

res.relative(-(i-2));

while(res.next())

System.out.println("\tNom : "+res.getString("prof\_nom") +" \t prénom : "+res.getString("prof\_prenom"));

res.close();

state.close();

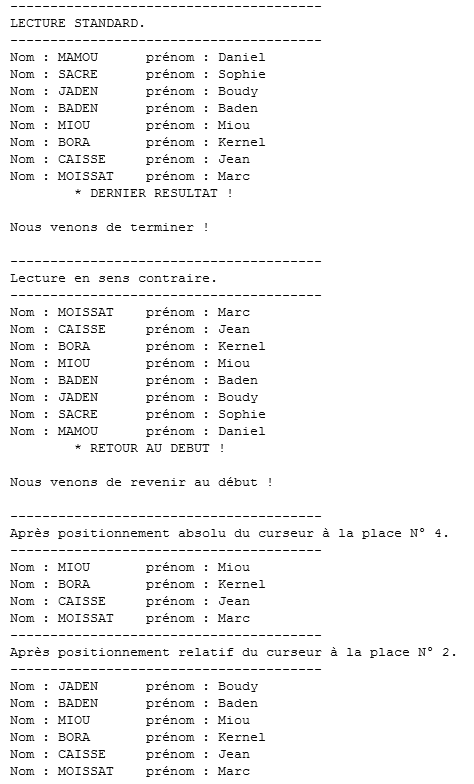
} catch (Exception e) {

e.printStackTrace();

}

}

}



Lorsque vous souhaitez placer le curseur sur la première ligne, vous devez utiliser absolute(1) quel que soit l'endroit où vous vous trouvez ! En revanche, cela nécessite que le ResultSet soit de type TYPE\_SCROLL\_SENSITIVE ou TYPE\_SCROLL\_INSENSITIVE, sans quoi vous aurez une exception.

## Modifier des données :

* res.getAscii() ;
* res.getBytes() ;
* res.getInt() ;
* res.getString() ;
* etc.

Ici, vous devez remplacer getXXX() par updateXXX(). Ces méthodes de mise à jour des données prennent deux paramètres :

* le nom de la colonne (String) ;
* la valeur à attribuer à la place de la valeur existante (cela dépend de la méthode utilisée).

Exemples :

* updateFloat(String nomColonne, float value) prend un float en paramètre ;
* updateString(String nomColonne, String value) prend une chaîne de caractères en paramètre

Pour que les changements soient effectifs, on peut les valider : updateRow(), ou même les annuler : cancelRowUpdates(). L’annulation doit se faire avant la confirmation.

mise à jour :

//CTRL + SHIFT + O pour générer les imports

public class Modif {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

//On autorise la mise à jour des données

//Et la mise à jour de l'affichage

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);

//On va chercher une ligne dans la base de données

String query = "SELECT prof\_id, prof\_nom, prof\_prenom FROM professeur " + "WHERE prof\_nom = 'MAMOU'";

ResultSet res = state.executeQuery(query);

res.first();

//On affiche ce que l'on trouve

System.out.println("NOM : " + res.getString("prof\_nom") + " - PRENOM : " + res.getString("prof\_prenom"));

//On met à jour les champs

res.updateString("prof\_nom", "COURTEL");

res.updateString("prof\_prenom", "Angelo");

//On valide

res.updateRow();

//On affiche les modifications

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

System.out.println("APRES MODIFICATION : ");

System.out.println("\tNOM : " + res.getString("prof\_nom") + " - PRENOM : " + res.getString("prof\_prenom") + "\n");

//On remet les informations de départ

res.updateString("prof\_nom", "MAMOU");

res.updateString("prof\_prenom", "Daniel");

//On valide à nouveau

res.updateRow();

//Et voilà !

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

System.out.println("APRES REMODIFICATION : ");

System.out.println("\tNOM : " + res.getString("prof\_nom") + " - PRENOM : " + res.getString("prof\_prenom") + "\n");

res.close();

state.close();

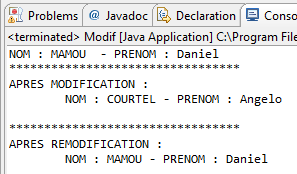
} catch (Exception e) {

e.printStackTrace();

}

}

}



## Statement, plus loin :

Par conséquent, vous devez avoir deviné que les requêtes de type INSERT, UPDATE, DELETE et CREATE sont également exécutées par ces objets. Voici un code d'exemple :

//CTRL + SHIFT + O pour générer les imports

public class State {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);

//On autorise la mise à jour des données

//Et la mise à jour de l'affichage

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,ResultSet.CONCUR\_UPDATABLE);

PreparedStatement prepare = conn.prepareStatement("UPDATE professeur set prof\_prenom = ? "+"WHERE prof\_nom = 'MAMOU'");

//On va chercher une ligne dans la base de données

String query = "SELECT prof\_nom, prof\_prenom FROM professeur "+"WHERE prof\_nom ='MAMOU'";

//On exécute la requête

ResultSet res = state.executeQuery(query);

res.first();

//On affiche

System.out.println("\n\tDONNEES D'ORIGINE : ");

System.out.println("\t-------------------");

System.out.println("\tNOM : " + res.getString("prof\_nom") + " - PRENOM : " + res.getString("prof\_prenom"));

//On paramètre notre requête préparée

prepare.setString(1, "Gérard");

//On exécute

prepare.executeUpdate();

res = state.executeQuery(query);

res.first();

//On affiche à nouveau

System.out.println("\n\t\t APRES MAJ : ");

System.out.println("\t\t \* NOM : " + res.getString("prof\_nom") + " - PRENOM :" + res.getString("prof\_prenom"));

//On effectue une mise à jour

prepare.setString(1, "Daniel");

prepare.executeUpdate();

res = state.executeQuery(query);

res.first();

//On affiche une nouvelle fois

System.out.println("\n\t\t REMISE A ZERO : ");

System.out.println("\t\t \* NOM : " + res.getString("prof\_nom") + " - PRENOM :" + res.getString("prof\_prenom"));

prepare.close();

res.close();

state.close();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

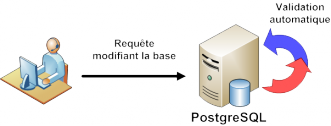
Pour les autres types de requêtes, il suffit d'invoquer la même méthode que pour la mise à jour. En fait, celle-ci retourne un booléen indiquant si le traitement a réussi ou échoué. Voici quelques exemples :

state.executeUpdate("INSERT INTO professeur (prof\_nom, prof\_prenom) VALUES('SALMON', 'Dylan')");

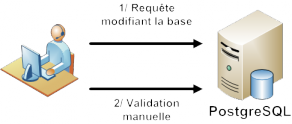
state.executeUpdate("DELETE FROM professeur WHERE prof\_nom = 'MAMOU'");

## Gérer les transactions manuellement :

Quand on envoie une requête, le moteur valide la requête :



Mais on peut avoir la main dessus et valider soit même :



Pour gérer manuellement les transactions, on spécifie au moteur SQL de ne pas valider automatiquement les requêtes SQL grâce à une méthode (qui ne concernera toutefois pas l'objet Statement, mais l'objet Connection) prenant un booléen en paramètre :

//CTRL + SHIFT + O pour générer les imports

public class Transact {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "batterie";

Connection conn = DriverManager.getConnection(url, user, passwd);

conn.setAutoCommit(false);

} catch (Exception e) {

e.printStackTrace();

}

}

}

Quand on veut que nos requêtes soient prises en compte, il faut les valider avec conn.commit().

En mode setAutoCommit(false), si vous ne validez pas vos requêtes, elles ne seront pas prises en compte.

Exemple :

//CTRL + SHIFT + O pour générer les imports

public class Transact {

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

String url = "jdbc:postgresql://localhost:5432/Ecole";

String user = "postgres";

String passwd = "batterie";

Connection conn = DriverManager.getConnection(url, user, passwd);

conn.setAutoCommit(false);

Statement state = conn.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

String query = "UPDATE professeur SET prof\_prenom = 'Cyrille' "+"WHERE prof\_nom = 'MAMOU'";

ResultSet result = state.executeQuery("SELECT \* FROM professeur"+" WHERE prof\_nom = 'MAMOU'");

result.first();

System.out.println("NOM : " + result.getString("prof\_nom") + " - PRENOM : " + result.getString("prof\_prenom"));

state.executeUpdate(query);

result = state.executeQuery("SELECT \* FROM professeur WHERE prof\_nom = 'MAMOU'");

result.first();

System.out.println("NOM : " + result.getString("prof\_nom") + " - PRENOM : " + result.getString("prof\_prenom"));

result.close();

state.close();

} catch (Exception e) {

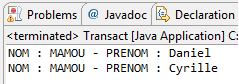
e.printStackTrace();

}

}

}

Vous pouvez exécuter ce code autant de fois que vous voulez, vous obtiendrez toujours la même chose que sur la figure suivante.



Il faudrait executer un conn.commit() à la fin du code pour valider les changements.

# Limiter le nombre de connexion

On va découvrir le pattern singleton.

## Pourquoi ne se connecter qu’une seule fois :

**Le principe est d’interdire l’instanciation d’une classe grâce à un constructeur déclaré private.**

## Le pattern singleton :

//CTRL + SHIFT + O pour générer les imports

public class SdzConnection{

//URL de connexion

private String url = "jdbc:postgresql://localhost:5432/Ecole";

//Nom du user

private String user = "postgres";

//Mot de passe de l'utilisateur

private String passwd = "postgres";

//Objet Connection

private static Connection connect;

//Constructeur privé

private SdzConnection(){

try {

connect = DriverManager.getConnection(url, user, passwd);

} catch (SQLException e) {

e.printStackTrace();

}

}

//Méthode qui va nous retourner notre instance et la créer si elle n'existe pas

public static Connection getInstance(){

if(connect == null){

new SdzConnection();

}

return connect;

}

}

Du coup on ne peut créer qu’une seule fois la connection :

Pour en être bien sûrs, nous allons faire un petit test… Voici le code un peu modifié de la méthode getInstance() :

public static Connection getInstance(){

if(connect == null){

new SdzConnection();

System.out.println("INSTANCIATION DE LA CONNEXION SQL ! ");

}

else{

System.out.println("CONNEXION SQL EXISTANTE ! ");

}

return connect;

}

Cela nous montre quand la connexion est réellement créée. Ensuite, il ne nous manque plus qu'un code de test. Oh ! Ben ça alors ! J'en ai un sous la main :

//CTRL + SHIFT + O pour générer les imports

public class Test {

public static void main(String[] args) {

try {

//Nous appelons quatre fois la méthode getInstance()

PreparedStatement prepare = SdzConnection.getInstance().prepareStatement("SELECT \* FROM classe WHERE cls\_nom = ?");

Statement state = SdzConnection.getInstance().createStatement();

SdzConnection.getInstance().setAutoCommit(false);

DatabaseMetaData meta = SdzConnection.getInstance().getMetaData();

} catch (SQLException e) {

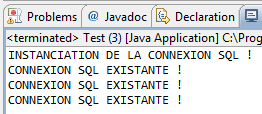
e.printStackTrace();

}

}

}

La méthode en question est appelée quatre fois. Que croyez-vous que ce code va afficher ? Quelque chose comme la figure suivante !



La classe SdzConnection peut être un peu simplifiée :

//CTRL + SHIFT + O pour générer les imports

public class SdzConnection{

private static String url = "jdbc:postgresql://localhost:5432/Ecole";

private static String user = "postgres";

private static String passwd = "postgres";

private static Connection connect;

public static Connection getInstance(){

if(connect == null){

try {

connect = DriverManager.getConnection(url, user, passwd);

} catch (SQLException e) {

e.printStackTrace();

}

}

return connect;

}

}

Attention toutefois, vous devrez rajouter la déclaration static à vos paramètres de connexion.

Vous pouvez relancer le code de test, vous verrez qu'il fonctionne toujours ! J'avais commencé par insérer un constructeur privé car vous deviez savoir que cela existait, mais remarquez que c'était superflu dans notre cas…

Par contre, dans une application *multithreads*, pour être sûrs d'éviter les conflits, il vous suffit de synchroniser la méthode getInstance() et le tour est joué. Mais - parce qu'il y a un mais - cette méthode ne règle le problème qu'*avant* l'instanciation de la connexion. Autrement dit, une fois la connexion instanciée, la synchronisation ne sert plus à rien.

Le problème du *multithreading* ne se pose pas vraiment pour une connexion à une BDD puisque ce *singleton* sert surtout de passerelle entre votre BDD et votre application. Cependant, il peut exister d'autres objets que des connexions SQL qui ne doivent être instanciés qu'une fois ; tous ne sont pas aussi laxistes concernant le multithreading.

## Le singleton :

public class SdzSingleton {

//Le singleton

private static SdzSingleton single;

//Variable d'instance

private String name = "";

//Constructeur privé

private SdzSingleton(){

this.name = "Mon singleton";

System.out.println("\t\tCRÉATION DE L'INSTANCE ! ! !");

}

//Méthode d'accès au singleton

public static SdzSingleton getInstance(){

if(single == null)

single = new SdzSingleton();

return single;

}

//Accesseur

public String getName(){

return this.name;

}

}

Classe test :

public class TestSingleton {

public static void main(String[] args) {

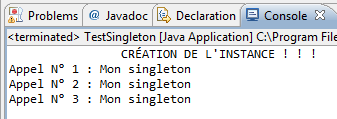
for(int i = 1; i < 4; i++)

System.out.println("Appel N° " + i + " : " + SdzSingleton.getInstance().getName());

}

}

Cela nous donne la figure suivante.



Pour limiter les ennuis, nous allons donc laisser cette lourde tâche à la JVM, dès le chargement de la classe, en instanciant notre singleton lors de sa déclaration :

public class SdzSingleton {

//Le singleton

private static SdzSingleton single = new SdzSingleton();

//Variable d'instance

private String name = "";

//Constructeur privé

private SdzSingleton(){

this.name = "Mon singleton";

System.out.println("\t\tCRÉATION DE L'INSTANCE ! ! !");

}

//Méthode d'accès au singleton

public static SdzSingleton getInstance(){

if(single == null)

single = new SdzSingleton();

return single;

}

//Accesseur

public String getName(){

return this.name;

}

}

Avec ce code, c'est la machine virtuelle qui s'occupe de charger l'instance du singleton, bien avant que n'importe quel thread vienne taquiner la méthode getInstance()…

Il existe une autre méthode permettant de faire cela, mais elle ne fonctionne parfaitement que depuis le JDK 1.5. On appelle cette méthode « le verrouillage à double vérification ». Elle consiste à utiliser le mot clé volatile combiné au mot clé synchronized.

Pour les lecteurs qui l'ignorent, déclarer une variable volatile permet d'assurer un accès ordonné des threads à une variable (plusieurs threads peuvent accéder à cette variable), marquant ainsi le premier point de verrouillage. Ensuite, la double vérification s'effectue dans la méthode getInstance() : on effectue la synchronisation **uniquement** lorsque le singleton n'est pas créé.

Voici ce que cela nous donne :

public class SdzSingleton {

private volatile static SdzSingleton single;

private String name = "";

private SdzSingleton(){

this.name = "Mon singleton";

System.out.println("\n\t\tCRÉATION DE L'INSTANCE ! ! !");

}

public static SdzSingleton getInstance(){

if(single == null){

synchronized(SdzSingleton.class){

if(single == null)

single = new SdzSingleton();

}

}

return single;

}

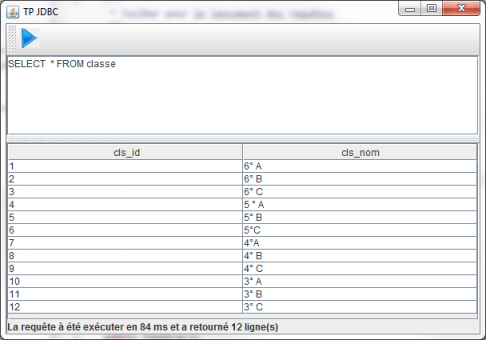
public String getName(){

return this.name;

}

}

# TP testeur de requête



##### Classe SdzConnection.java

package com.sdz.connection;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import javax.swing.JOptionPane;

public class SdzConnection{

/\*\*

\* URL de connection

\*/

private static String url = "jdbc:postgresql://localhost:5432/Ecole";

/\*\*

\* Nom du user

\*/

private static String user = "postgres";

/\*\*

\* Mot de passe du user

\*/

private static String passwd = "postgres";

/\*\*

\* Objet Connection

\*/

private static Connection connect;

/\*\*

\* Méthode qui va retourner notre instance

\* et la créer si elle n'existe pas...

\* @return

\*/

public static Connection getInstance(){

if(connect == null){

try {

connect = DriverManager.getConnection(url, user, passwd);

} catch (SQLException e) {

JOptionPane.showMessageDialog(null, e.getMessage(), "ERREUR DE CONNEXION ! ", JOptionPane.ERROR\_MESSAGE);

}

}

return connect;

}

}

##### Classe Fenetre.java

package com.sdz.tp;

import java.awt.BorderLayout;

import java.awt.Dimension;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.sql.ResultSet;

import java.sql.ResultSetMetaData;

import java.sql.SQLException;

import java.sql.Statement;

import javax.swing.ImageIcon;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JSplitPane;

import javax.swing.JTable;

import javax.swing.JTextArea;

import javax.swing.JToolBar;

import com.sdz.connection.SdzConnection;

public class Fenetre extends JFrame {

/\*\*

\* ToolBar pour le lancement des requêtes

\*/

private JToolBar tool = new JToolBar();

/\*\*

\* Le bouton

\*/

private JButton load = new JButton(new ImageIcon("img/load.png"));

/\*\*

\* Le délimiteur

\*/

private JSplitPane split;

/\*\*

\* Le conteneur de résultat

\*/

private JPanel result = new JPanel();

/\*\*

\* Requête par défaut pour le démarrage

\*/

private String requete = "SELECT \* FROM classe";

/\*\*

\* Le composant dans lequel taper la requête

\*/

private JTextArea text = new JTextArea(requete);

/\*\*

\* Constructeur

\*/

public Fenetre(){

setSize(900, 600);

setTitle("TP JDBC");

setLocationRelativeTo(null);

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

initToolbar();

initContent();

initTable(requete);

}

/\*\*

\* Initialise la toolbar

\*/

private void initToolbar(){

load.setPreferredSize(new Dimension(30, 35));

load.setBorder(null);

load.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent event){

initTable(text.getText());

}

});

tool.add(load);

getContentPane().add(tool, BorderLayout.NORTH);

}

/\*\*

\* Initialise le contenu de la fenêtre

\*/

public void initContent(){

//Vous connaissez ça...

result.setLayout(new BorderLayout());

split = new JSplitPane(JSplitPane.VERTICAL\_SPLIT, new JScrollPane(text), result);

split.setDividerLocation(100);

getContentPane().add(split, BorderLayout.CENTER);

}

/\*\*

\* Initialise le visuel avec la requête saisie dans l'éditeur

\* @param query

\*/

public void initTable(String query){

try {

//On crée un statement

long start = System.currentTimeMillis();

Statement state = SdzConnection.getInstance().createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_READ\_ONLY);

//On exécute la requête

ResultSet res = state.executeQuery(query);

//Temps d'exécution

//On récupère les meta afin de récupérer le nom des colonnes

ResultSetMetaData meta = res.getMetaData();

//On initialise un tableau d'Object pour les en-têtes du tableau

Object[] column = new Object[meta.getColumnCount()];

for(int i = 1 ; i <= meta.getColumnCount(); i++)

column[i-1] = meta.getColumnName(i);

//Petite manipulation pour obtenir le nombre de lignes

res.last();

int rowCount = res.getRow();

Object[][] data = new Object[res.getRow()][meta.getColumnCount()];

//On revient au départ

res.beforeFirst();

int j = 1;

//On remplit le tableau d'Object[][]

while(res.next()){

for(int i = 1 ; i <= meta.getColumnCount(); i++)

data[j-1][i-1] = res.getObject(i);

j++;

}

//On ferme le tout

res.close();

state.close();

long totalTime = System.currentTimeMillis() - start;

//On enlève le contenu de notre conteneur

result.removeAll();

//On y ajoute un JTable

result.add(new JScrollPane(new JTable(data, column)), BorderLayout.CENTER);

result.add(new JLabel("La requête à été exécuter en " + totalTime + " ms et a retourné " + rowCount + " ligne(s)"), BorderLayout.SOUTH);

//On force la mise à jour de l'affichage

result.revalidate();

} catch (SQLException e) {

//Dans le cas d'une exception, on affiche une pop-up et on efface le contenu

result.removeAll();

result.add(new JScrollPane(new JTable()), BorderLayout.CENTER);

result.revalidate();

JOptionPane.showMessageDialog(null, e.getMessage(), "ERREUR ! ", JOptionPane.ERROR\_MESSAGE);

}

}

/\*\*

\* Point de départ du programme

\* @param args

\*/

public static void main(String[] args){

Fenetre fen = new Fenetre();

fen.setVisible(true);

}

}

# Lier ses tables avec des objets Java

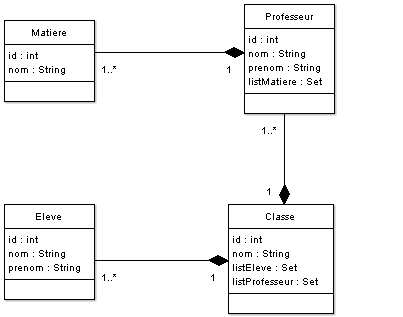
Le pattern DAO : Data Access Object.

C’est utiliser les données d’une BDD dans des objets java.

## Avant toute chose :

En tout premier lieu, il faut créer une classe par entité (les tables, exceptées celles de jointure), ce qui nous donnerait les classes suivantes :

* Eleve ;
* Matiere ;
* Professeur ;
* Classe.



Grâce à ce diagramme, nous voyons les liens entre les objets : une classe est composée de plusieurs élèves et de plusieurs professeurs, et un professeur peut exercer plusieurs matières. Les tables de jointures de la base sont symbolisées par la composition dans nos objets.

Une fois que cela est fait, nous devons coder ces objets avec les accesseurs et les mutateurs adéquats :

* getters et setters pour tous les attributs de toutes les classes ;
* méthodes d'ajout et de suppression pour les objets constitués de listes d'objets

On appelle ce genre d'objet des « POJO », pour Plain Old Java Object ! Ce qui nous donne ces codes source :

##### Classe Eleve.java

package com.sdz.bean;

public class Eleve {

//ID

private int id = 0;

//Nom de l'élève

private String nom = "";

//Prénom de l'élève

private String prenom = "";

public Eleve(int id, String nom, String prenom) {

this.id = id;

this.nom = nom;

this.prenom = prenom;

}

public Eleve(){};

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNom() {

return nom;

}

public void setNom(String nom) {

this.nom = nom;

}

public String getPrenom() {

return prenom;

}

public void setPrenom(String prenom) {

this.prenom = prenom;

}

}

##### Classe Matiere.java

package com.sdz.bean;

public class Matiere {

//ID

private int id = 0;

//Nom du professeur

private String nom = "";

public Matiere(int id, String nom) {

this.id = id;

this.nom = nom;

}

public Matiere(){}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNom() {

return nom;

}

public void setNom(String nom) {

this.nom = nom;

}

}

##### Classe Professeur.java

package com.sdz.bean;

import java.util.HashSet;

import java.util.Set;

public class Professeur {

//ID

private int id = 0;

//Nom du professeur

private String nom = "";

//Prénom du professeur

private String prenom = "";

//Liste des matières dispensées

private Set<Matiere> listMatiere = new HashSet<Matiere>();

public Professeur(int id, String nom, String prenom) {

this.id = id;

this.nom = nom;

this.prenom = prenom;

}

public Professeur(){}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNom() {

return nom;

}

public void setNom(String nom) {

this.nom = nom;

}

public String getPrenom() {

return prenom;

}

public void setPrenom(String prenom) {

this.prenom = prenom;

}

public Set<Matiere> getListMatiere() {

return listMatiere;

}

public void setListMatiere(Set<Matiere> listMatiere) {

this.listMatiere = listMatiere;

}

//Ajoute une matière à un professeur

public void addMatiere(Matiere matiere){

this.listMatiere.add(matiere);

}

//Retire une matière à un professeur

public void removeMatiere(Matiere matiere){

this.listMatiere.remove(matiere);

}

}

##### Classe Classe.java

package com.sdz.bean;

//CTRL + SHIFT + O pour générer les imports

public class Classe {

//ID

private int id = 0;

//Nom du professeur

private String nom = "";

//Liste des professeurs

private Set<Professeur> listProfesseur = new HashSet<Professeur>();

//Liste des élèves

private Set<Eleve> listEleve = new HashSet<Eleve>();

public Classe(int id, String nom) {

this.id = id;

this.nom = nom;

}

public Classe(){}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getNom() {

return nom;

}

public void setNom(String nom) {

this.nom = nom;

}

public Set<Professeur> getListProfesseur() {

return listProfesseur;

}

public void setListProfesseur(Set<Professeur> listProfesseur) {

this.listProfesseur = listProfesseur;

}

public void addProfesseur(Professeur prof) {

if(!listProfesseur.contains(prof))

listProfesseur.add(prof);

}

public void removeProfesseur(Professeur prof ) {

this.listProfesseur.remove(prof);

}

public Set<Eleve> getListEleve() {

return listEleve;

}

public void setListEleve(Set<Eleve> listEleve) {

this.listEleve = listEleve;

}

//Ajoute un élève à la classe

public void addEleve(Eleve eleve){

if(!this.listEleve.contains(eleve))

this.listEleve.add(eleve);

}

//Retire un élève de la classe

public void removeEleve(Eleve eleve){

this.listEleve.remove(eleve);

}

public boolean equals(Classe cls){

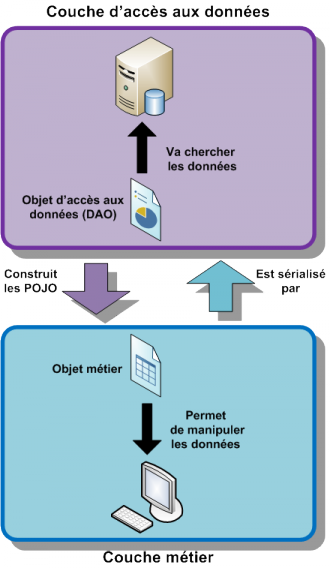
return this.getId() == cls.getId();

}

}

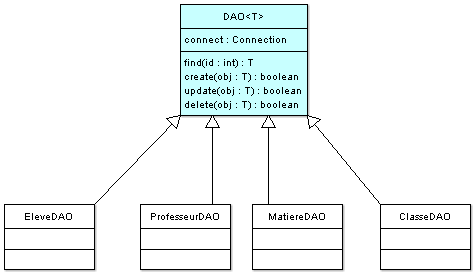
## Le pattern DAO :

Il permet de resister aux changements de structure. Fait le lien entre la couche d’accès aux données et la couche métier (nos classes). Les objets qu’on a créé plus haut sont nos POJO (*Plain Old Java Object*).



*Comment faire pour demander à nos objets DAO de récupérer tel type d'objet ou de sérialiser tel autre ? Avec des cast ?*

Soit avec des cast, soit en créant une classe générique (figure suivante) !



Afin de ne pas surcharger les codes d'exemples, j'ai volontairement utilisé des objets Statement mais il aurait mieux valu utiliser des requêtes préparées (PreparedStatement) pour des questions de performances et de sécurité.

##### Classe DAO.java

package com.sdz.dao;

import java.sql.Connection;

import com.sdz.connection.SdzConnection;

public abstract class DAO<T> {

protected Connection connect = null;

public DAO(Connection conn){

this.connect = conn;

}

/\*\*

\* Méthode de création

\* @param obj

\* @return boolean

\*/

public abstract boolean create(T obj);

/\*\*

\* Méthode pour effacer

\* @param obj

\* @return boolean

\*/

public abstract boolean delete(T obj);

/\*\*

\* Méthode de mise à jour

\* @param obj

\* @return boolean

\*/

public abstract boolean update(T obj);

/\*\*

\* Méthode de recherche des informations

\* @param id

\* @return T

\*/

public abstract T find(int id);

}

##### Classe EleveDAO.java

package com.sdz.dao.implement;

//CTRL + SHIFT + O pour générer les imports

public class EleveDAO extends DAO<Eleve> {

public EleveDAO(Connection conn) {

super(conn);

}

public boolean create(Eleve obj) {

return false;

}

public boolean delete(Eleve obj) {

return false;

}

public boolean update(Eleve obj) {

return false;

}

public Eleve find(int id) {

Eleve eleve = new Eleve();

try {

ResultSet result = this.connect.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY).executeQuery("SELECT \* FROM eleve WHERE elv\_id = " + id);

if(result.first())

eleve = new Eleve(

id,

result.getString("elv\_nom"),

result.getString("elv\_prenom"

));

} catch (SQLException e) {

e.printStackTrace();

}

return eleve;

}

}

##### Classe MatiereDAO.java

package com.sdz.dao.implement;

//CTRL + SHIFT + O pour générer les imports

public class MatiereDAO extends DAO<Matiere> {

public MatiereDAO(Connection conn) {

super(conn);

}

public boolean create(Matiere obj) {

return false;

}

public boolean delete(Matiere obj) {

return false;

}

public boolean update(Matiere obj) {

return false;

}

public Matiere find(int id) {

Matiere matiere = new Matiere();

try {

ResultSet result = this.connect.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY

).executeQuery("SELECT \* FROM matiere WHERE mat\_id = " + id);

if(result.first())

matiere = new Matiere(id, result.getString("mat\_nom"));

} catch (SQLException e) {

e.printStackTrace();

}

return matiere;

}

}

##### Classe ProfesseurDAO.java

package com.sdz.dao.implement;

//CTRL + SHIFT + O pour générer les imports

public class ProfesseurDAO extends DAO<Professeur> {

public ProfesseurDAO(Connection conn) {

super(conn);

}

public boolean create(Professeur obj) {

return false;

}

public boolean delete(Professeur obj) {

return false;

}

public boolean update(Professeur obj) {

return false;

}

public Professeur find(int id) {

Professeur professeur = new Professeur();

try {

ResultSet result = this.connect.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY

).executeQuery(

"SELECT \* FROM professeur "+

"LEFT JOIN j\_mat\_prof ON jmp\_prof\_k = prof\_id " +

"AND prof\_id = "+ id +

" INNER JOIN matiere ON jmp\_mat\_k = mat\_id"

);

if(result.first()){

professeur = new Professeur(id, result.getString("prof\_nom"), result.getString("prof\_prenom"));

result.beforeFirst();

MatiereDAO matDao = new MatiereDAO(this.connect);

while(result.next())

professeur.addMatiere(matDao.find(result.getInt("mat\_id")));

}

} catch (SQLException e) {

e.printStackTrace();

}

return professeur;

}

public boolean update(Professeur obj) {

return false;

}

}

##### Classe ClasseDAO.java

package com.sdz.dao.implement;

//CTRL + SHIFT + O pour générer les imports

public class ClasseDAO extends DAO<Classe> {

public ClasseDAO(Connection conn) {

super(conn);

}

public boolean create(Classe obj) {

return false;

}

public boolean delete(Classe obj) {

return false;

}

public boolean update(Classe obj) {

return false;

}

public Classe find(int id) {

Classe classe = new Classe();

try {

ResultSet result = this.connect.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY

).executeQuery("SELECT \* FROM classe WHERE cls\_id = " + id);

if(result.first()){

classe = new Classe(id, result.getString("cls\_nom"));

result = this.connect.createStatement().executeQuery(

"SELECT prof\_id, prof\_nom, prof\_prenom from professeur " +

"INNER JOIN j\_mat\_prof ON prof\_id = jmp\_prof\_k " +

"INNER JOIN j\_cls\_jmp ON jmp\_id = jcm\_jmp\_k AND jcm\_cls\_k = " + id

);

ProfesseurDAO profDao = new ProfesseurDAO(this.connect);

while(result.next())

classe.addProfesseur(profDao.find(result.getInt("prof\_id")));

EleveDAO eleveDao = new EleveDAO(this.connect);

result = this.connect.createStatement().executeQuery(

"SELECT elv\_id, elv\_nom, elv\_prenom FROM eleve " +

"INNER JOIN classe ON elv\_cls\_k = cls\_id AND cls\_id = " + id

);

while(result.next())

classe.addEleve(eleveDao.find(result.getInt("elv\_id")));

}

} catch (SQLException e) {

e.printStackTrace();

}

return classe;

}

}

### Tests :

Voici un code de test :

import com.sdz.bean.Classe;

//CTRL + SHIFT + O pour générer les imports

public class FirstTest {

public static void main(String[] args) {

//Testons des élèves

DAO<Eleve> eleveDao = new EleveDAO(SdzConnection.getInstance());

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

Eleve eleve = eleveDao.find(i);

System.out.println("Elève N°" + eleve.getId() + " - " + eleve.getNom() + " " + eleve.getPrenom());

}

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

//Voyons voir les professeurs

DAO<Professeur> profDao = new ProfesseurDAO(SdzConnection.getInstance());

for(int i = 4; i < 8; i++){

Professeur prof = profDao.find(i);

System.out.println(prof.getNom() + " " + prof.getPrenom() + " enseigne : ");

for(Matiere mat : prof.getListMatiere())

System.out.println("\t \* " + mat.getNom());

}

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

//Et là, c'est la classe

DAO<Classe> classeDao = new ClasseDAO(SdzConnection.getInstance());

Classe classe = classeDao.find(11);

System.out.println("Classe de " + classe.getNom());

System.out.println("\nListe des élèves :");

for(Eleve eleve : classe.getListEleve())

System.out.println(" - " + eleve.getNom() + " " + eleve.getPrenom());

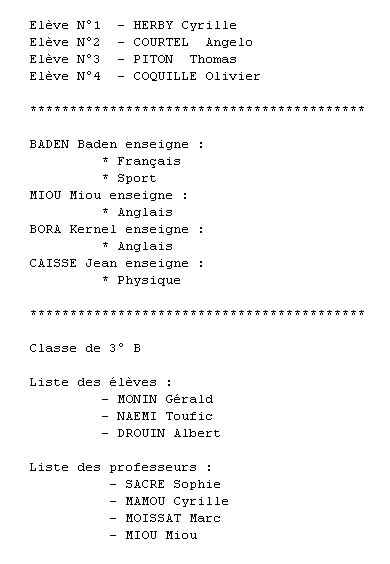
System.out.println("\nListe des professeurs :");

for(Professeur prof : classe.getListProfesseur())

System.out.println(" - " + prof.getNom() + " " + prof.getPrenom());

}

}



### Pattern factory :

Le pattern DAO implémente le pattern factory. Cela consiste à déléguer l’instanciation d’objet d’une classe.

En fait, une fabrique ne fait que ça. En général, lorsque vous voyez ce genre de code dans une classe :

class A{

public Object getData(int type){

Object obj;

//----------------------

if(type == 0)

obj = new B();

else if(type == 1)

obj = new C();

else

obj = new D();

//----------------------

obj.doSomething();

obj.doSomethingElse();

}

La création d’objet est conditionnée par une variable.

Du coup, lorsque nous voudrons instancier les objets de la fabrique, nous l'utiliserons à présent comme ceci :

B b = Factory.getData(0);

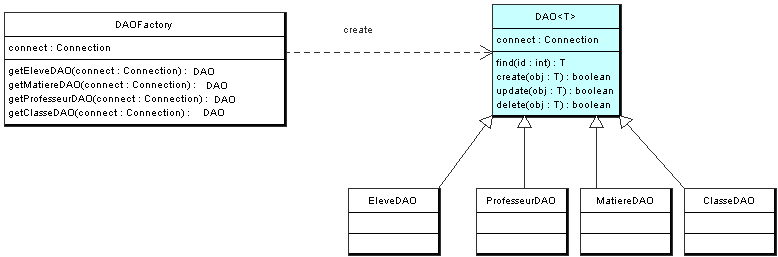
C c = Factory.getData(1);

//…

Pourquoi faire tout ça ? En temps normal, nous travaillons avec des objets concrets, non soumis au changement. Cependant, dans le cas qui nous intéresse, nos objets peuvent être amenés à changer. Et j'irai même plus loin : le type d'objet utilisé peut changer !

L'avantage d'utiliser une fabrique, c'est que les instances concrètes (utilisation du mot clé new) se font à un seul endroit ! Donc, si nous devons faire des changements, il ne se feront qu'à un seul endroit. Si nous ajoutons un paramètre dans le constructeur, par exemple…

### Fabriquer nos DAO :



Voici le code de notre fabrique :

package com.sdz.dao;

//CTRL + SHIFT + O pour générer les imports

public class DAOFactory {

protected static final Connection conn = SdzConnection.getInstance();

/\*\*

\* Retourne un objet Classe interagissant avec la BDD

\* @return DAO

\*/

public static DAO getClasseDAO(){

return new ClasseDAO(conn);

}

/\*\*

\* Retourne un objet Professeur interagissant avec la BDD

\* @return DAO

\*/

public static DAO getProfesseurDAO(){

return new ProfesseurDAO(conn);

}

/\*\*

\* Retourne un objet Eleve interagissant avec la BDD

\* @return DAO

\*/

public static DAO getEleveDAO(){

return new EleveDAO(conn);

}

/\*\*

\* Retourne un objet Matiere interagissant avec la BDD

\* @return DAO

\*/

public static DAO getMatiereDAO(){

return new MatiereDAO(conn);

}

}

Et voici un code qui devrait vous plaire :

//CTRL + SHIFT + O pour générer les imports

public class TestDAO {

public static void main(String[] args) {

System.out.println("");

//On récupère un objet faisant le lien entre la base et nos objets

DAO<Eleve> eleveDao = DAOFactory.getEleveDAO();

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

//On fait notre recherche

Eleve eleve = eleveDao.find(i);

System.out.println("\tELEVE N°" + eleve.getId() + " - NOM : " + eleve.getNom() + " - PRENOM : " + eleve.getPrenom());

}

System.out.println("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

//On agit de même pour une classe

DAO<Classe> classeDao = DAOFactory.getClasseDAO();

//On cherche la classe ayant pour ID 10

Classe classe = classeDao.find(10);

System.out.println("\tCLASSE DE " + classe.getNom());

//On récupère la liste des élèves

System.out.println("\n\tCelle-ci contient " + classe.getListEleve().size() + " élève(s)");

for(Eleve eleve : classe.getListEleve())

System.out.println("\t\t - " + eleve.getNom() + " " + eleve.getPrenom());

//De même pour la liste des professeurs

System.out.println("\n\tCelle-ci a " + classe.getListProfesseur().size() + " professeur(s)");

for(Professeur prof : classe.getListProfesseur()){

System.out.println("\t\t - Mr " + prof.getNom() + " " + prof.getPrenom() + " professeur de :");

//Tant qu'à faire, on prend aussi les matières

for(Matiere mat : prof.getListMatiere())

System.out.println("\t\t\t \* " + mat.getNom());

}

System.out.println("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

//Un petit essai sur les matières

DAO<Matiere> matiereDao = DAOFactory.getMatiereDAO();

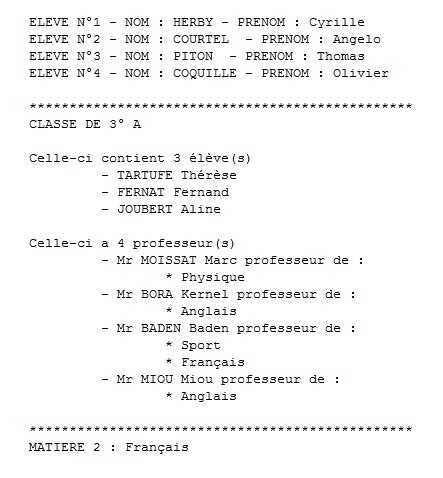
Matiere mat = matiereDao.find(2);

System.out.println("\tMATIERE " + mat.getId() + " : " + mat.getNom());

}

}

Le résultat que nous donne ce code se trouve à la figure suivante.

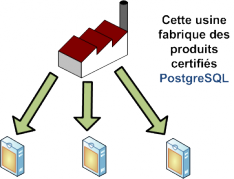


### De l’usine à la multinationale :

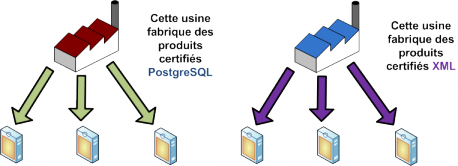
Notre patron veut utiliser Postgres et du xml.

On va créer des fabriques de fabrique.

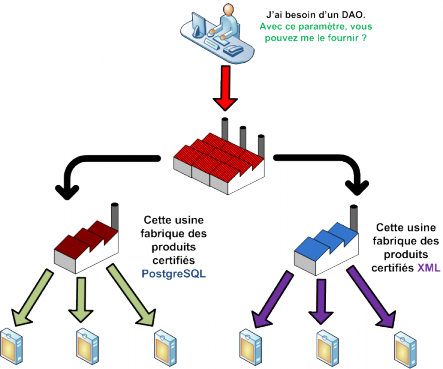
Pour le moment on a ça :



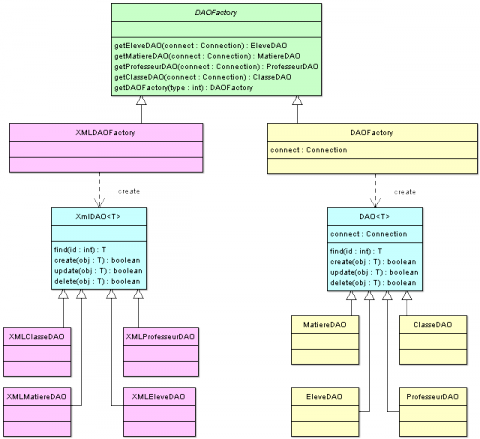
On cherche à avoir :



On va donc faire ainsi :



Du coup : *Héritage – polyomorphisme*.



##### Classe AbstractDAOFactory.java

package com.sdz.dao;

public abstract class AbstractDAOFactory {

public static final int DAO\_FACTORY = 0;

public static final int XML\_DAO\_FACTORY = 1;

//Retourne un objet Classe interagissant avec la BDD

public abstract DAO getClasseDAO();

//Retourne un objet Professeur interagissant avec la BDD

public abstract DAO getProfesseurDAO();

//Retourne un objet Eleve interagissant avec la BDD

public abstract DAO getEleveDAO();

//Retourne un objet Matiere interagissant avec la BDD

public abstract DAO getMatiereDAO();

//Méthode permettant de récupérer les Factory

public static AbstractDAOFactory getFactory(int type){

switch(type){

case DAO\_FACTORY:

return new DAOFactory();

case XML\_DAO\_FACTORY:

return new XMLDAOFactory();

default:

return null;

}

}

}

##### Classe DAOFactory.java

package com.sdz.dao;

//CTRL + SHIFT + O pour générer les imports

public class DAOFactory extends AbstractDAOFactory{

protected static final Connection conn = SdzConnection.getInstance();

public DAO getClasseDAO(){

return new ClasseDAO(conn);

}

public DAO getProfesseurDAO(){

return new ProfesseurDAO(conn);

}

public DAO getEleveDAO(){

return new EleveDAO(conn);

}

public DAO getMatiereDAO(){

return new MatiereDAO(conn);

}

}

##### Classe XMLDAOFactory.java

package com.sdz.dao;

public class XMLDAOFactory extends AbstractDAOFactory {

public DAO getClasseDAO() {

return null;

}

public DAO getEleveDAO() {

return null;

}

public DAO getMatiereDAO() {

return null;

}

public DAO getProfesseurDAO() {

return null;

}

}

Vous devez y voir plus clair : même si la classe XMLDAOFactory ne fait rien du tout, vous voyez le principe de base et c'est l'important ! Nous avons maintenant une hiérarchie de classes capables de travailler ensemble.

Je reprends le dernier exemple que nous avions réalisé, avec quelques modifications…

//CTRL + SHIFT + O pour générer les imports

public class TestDAO {

public static void main(String[] args) {

System.out.println("");

AbstractDAOFactory adf = AbstractDAOFactory.getFactory(AbstractDAOFactory.DAO\_FACTORY);

//On récupère un objet faisant le lien entre la base et nos objets

DAO<Eleve> eleveDao = adf.getEleveDAO();

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

//On fait notre recherche

Eleve eleve = eleveDao.find(i);

System.out.println("\tELEVE N°" + eleve.getId() + " - NOM : " + eleve.getNom() + " - PRENOM : " + eleve.getPrenom());

}

System.out.println("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

//On fait de même pour une classe

DAO<Classe> classeDao = adf.getClasseDAO();

//On cherche la classe ayant pour ID 10

Classe classe = classeDao.find(10);

System.out.println("\tCLASSE DE " + classe.getNom());

//On récupère la liste des élèves

System.out.println("\n\tCelle-ci contient " + classe.getListEleve().size() + " élève(s)");

for(Eleve eleve : classe.getListEleve())

System.out.println("\t\t - " + eleve.getNom() + " " + eleve.getPrenom());

//De même pour la liste des professeurs

System.out.println("\n\tCelle-ci a " + classe.getListProfesseur().size() + " professeur(s)");

for(Professeur prof : classe.getListProfesseur()){

System.out.println("\t\t - Mr " + prof.getNom() + " " + prof.getPrenom() + " professeur de :");

//Tant qu'à faire, on prend aussi les matières

for(Matiere mat : prof.getListMatiere())

System.out.println("\t\t\t \* " + mat.getNom());

}

System.out.println("\n\t\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

//Un petit essai sur les matières

DAO<Matiere> matiereDao = adf.getMatiereDAO();

Matiere mat = matiereDao.find(2);

System.out.println("\tMATIERE " + mat.getId() + " : " + mat.getNom());

}

}

Et le résultat est le même qu'avant ! Tout fonctionne à merveille ! Si vous utilisez un jour l'usine de fabrication XML, vous n'aurez qu'une seule ligne de code à changer :

AbstractDAOFactory adf = AbstractDAOFactory.getFactory(AbstractDAOFactory.XML\_DAO\_FACTORY);