La programmation orientée objet 2

Java™

Version 2024/2025

Sources

- *Introduction à la programmation orientée objet, CNPI-Gdt-PrograTG'10, 2010*
- *Introduction à Java, Robert Fisch, 2009*
- *Introduction à la programmation,* Robert Fisch, 11TG/T0IF, 2006
- *Introduction à la programmation,* Simone Beissel, T0IF, 2003
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006
- Wikipedia (Modèle-Vue-Contrôleur)

Rédacteurs

- Fred Faber
- Robert Fisch
- Georges Kugener

Site de référence

- **http://java.cnpi.lu**

Le cours est adapté et complété en permanence d'après les contributions des enseignants. Les enseignants sont priés d'envoyer leurs remarques et propositions via mail à Fred Faber ou Robert Fisch qui s'efforceront de les intégrer dans le cours dès que possible.

La version la plus actuelle est publiée sur le site : **http://java.cnpi.lu**

Table des matières

1. Quickstart

Ce chapitre montre étape par étape comment créer une application à interface graphique en NetBeans suivant le modèle MVC sans pour autant donner des explications détaillées. Celles-ci suivront dans la suite de ce document.

Étape 1 : Créer un nouveau projet vierge dans NetBeans :

1. Ouvrez NetBeans, puis créez un nouveau projet en choisissant « Java with Ant » et le type « Java Application ».
2. Nommez votre projet « Quickstart » et décochez l'option « Create Main Class ».
3. Veillez à ce que la « Project Location » pointe vers votre dossier dans lequel vous voulez sauvegarder tous vos projets Java.

Étape 2 : Ajouter deux classes au projet :

4. Faites un clic droit sur « default package » qui se trouve dans « Source package » et choisissez dans le menu contextuel « New », puis « Java Class ».
5. Nommez cette nouvelle classe « Adder ».
6. Refaites l'opération du point 4, mais créez maintenant une nouvelle « JFrame Form ».
7. Nommez cette nouvelle classe « MainFrame ».

Étape 3 : Programmer la classe **`Adder :`**

8. Sélectionnez le fichier **`Adder.java`** afin de l'éditer.
9. Voici le code source très minimaliste de cette classe :

```
public class Adder
{
    private int sum = 0;

    public void add(int pValue)
    {
        sum = sum + pValue;
    }

    public int getSum()
    {
        return sum;
    }
}
```

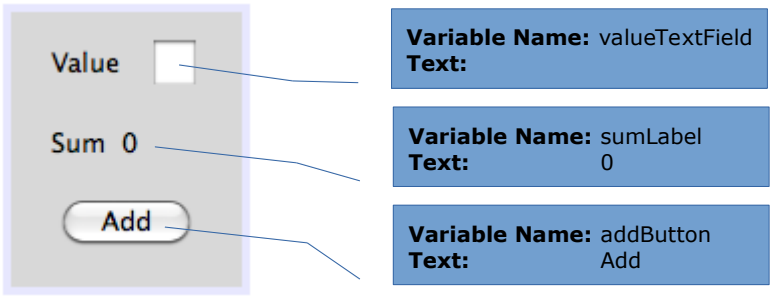
Crée l'attribut "sum" et l'initialise à zéro.

Méthode pour ajouter un nombre.

Accesseur qui retourne la somme actuelle.

Étape 4 : Programmer un interface graphique autour de cette classe :

10. Sélectionnez maintenant le fichier **`MainFrame.java`** et veillez à ce que vous soyez dans le mode « Design ».
11. Utilisez la palette « Swing Controls » pour ajouter 3 « Label », 1 « TextField » et 1 « Button » sur la fiche principale. Arrangez-les comme indiqué ci-dessous.



12. Nommez les composants comme indiqué ci-dessus et changez la propriété « text » comme indiqué. Ici, il n'est pas nécessaire de changer les noms des libellés « Value » et « Sum ». (Utilisez un clic droit sur les composants pour modifier ces propriétés.)
13. Changez dans le mode « Source » et ajoutez la déclaration de l'attribut suivant en haut de la classe **`MainFrame`** :

```
private Adder adder = new Adder();
```

14. Retournez dans le mode « Design » et faites un double clic sur le bouton « Add ». Vous allez vous retrouver dans le mode source et le curseur se trouvera dans la méthode **`addButtonActionPerformed(...)`**. Ajoutez le code suivant, puis faites Build/Run :

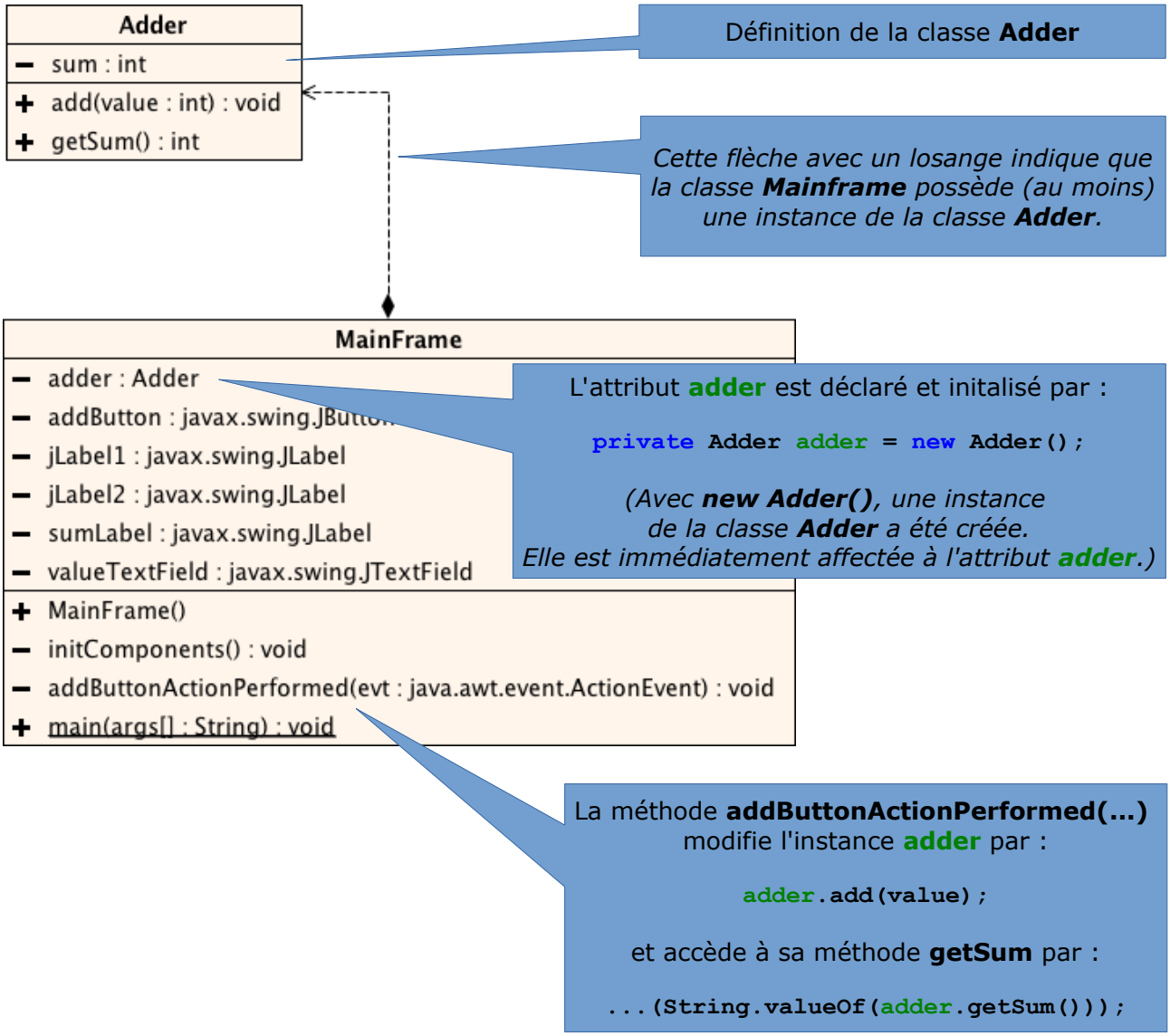
Récupère le texte du champ d'entrée, convertit ce texte en un entier et sauvegarde cette valeur entière dans la variable "value".

```
    private void addButtonActionPerformed(java.awt.event.ActionEvent evt)
    {
        int value = Integer.valueOf(valueTextField.getText());
        adder.add(value);
        sumLabel.setText(String.valueOf(adder.getSum()));
    }
```

Ajoute la valeur au calculateur.

Récupère la valeur actuelle du calculateur, la convertit en un texte et l'affiche dans le libellé

En UML (p.ex. en l'ouvrant en Unimozer), le projet se présente comme suit :



Questions :

A) Où faudrait-il ajouter du code pour que le programme compte le nombre d'additions que l'utilisateur a faites ?

B) Où faudrait-il ajouter le code pour que le programme affiche à l'utilisateur le nombre d'additions qu'il a effectuées ?

2. Manipulation de NetBeans

Pour tous les exercices et projets qui seront abordés dans la suite du cours à l'aide du logiciel NetBeans, voici quelques règles, astuces, recommandations et surtout un guide à suivre. Dans le chapitre précédent vous avez déjà été guidés à travers certaines étapes, dont voici maintenant le détails, y compris des explications supplémentaires.

2.1. Installation

Pour l'installation de NetBeans ainsi que du ***JDK (Java Development Kit)***, veuillez utiliser les liens sur le site **java.cnpi.lu**. Le téléchargement et les programmes d'installation correspondent aux conventions usuelles et sont auto-explicatifs.

Installez d'abord le JDK pour votre système avec le lien Java Development Kit [Choisir '**AdoptOpenJDK 11 (LTS)**' et '**HotSpot**'].

Installez ensuite NetBeans avec le lien que vous trouvez sur le site.

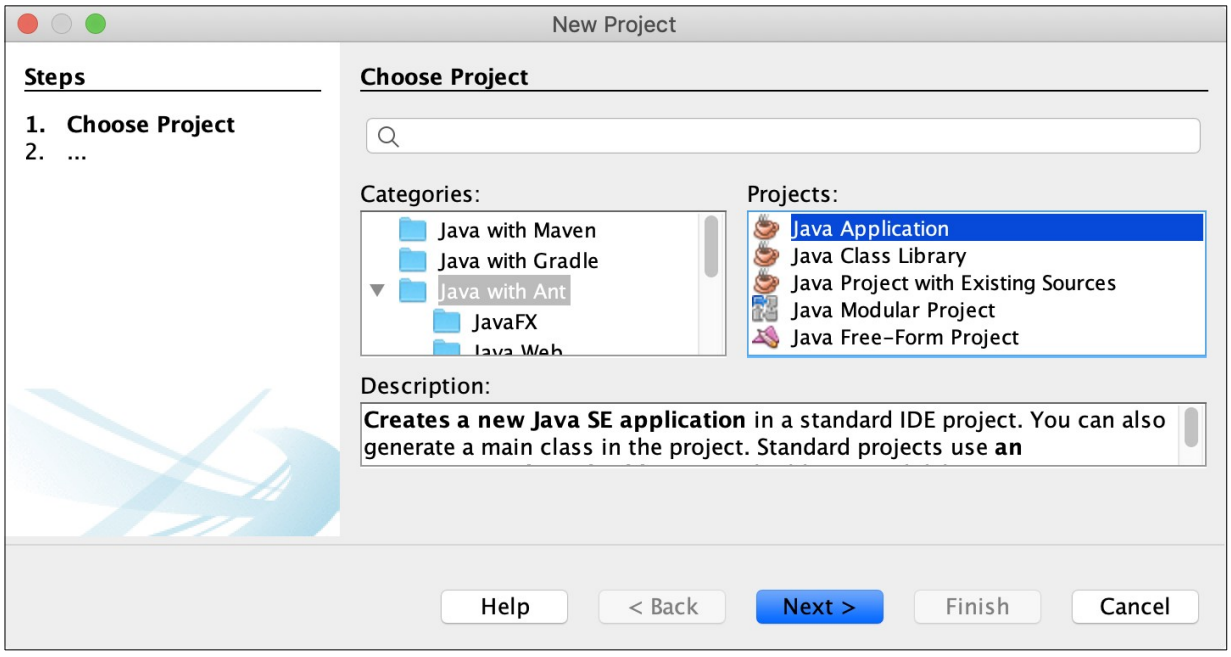
2.2. Les projets

Dans NetBeans, tous les éléments appartenant à une application sont regroupés dans un « projet ». Pour créer une nouvelle application, nous devons donc créer un nouveau projet.

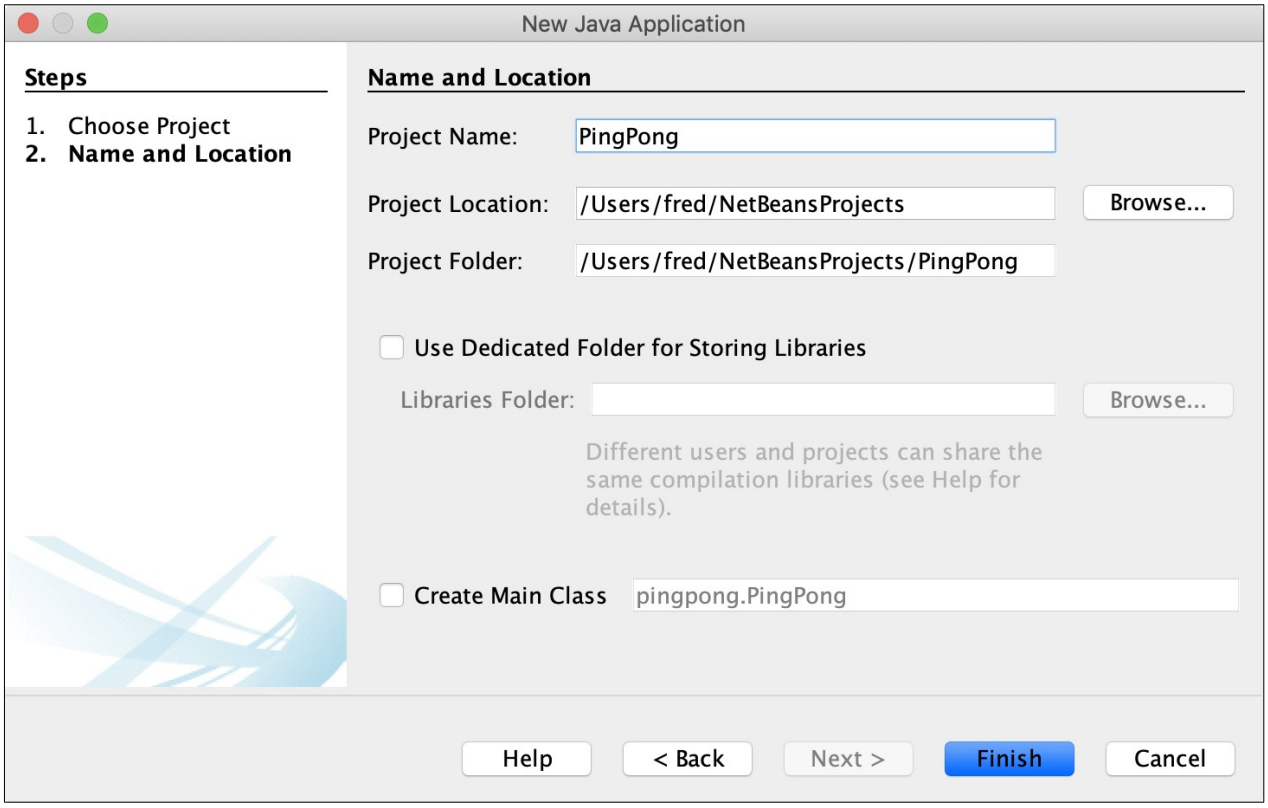
2.2.1. Créer un nouveau projet

Voici les étapes à suivre pour créer un nouveau projet vierge dans NetBeans :

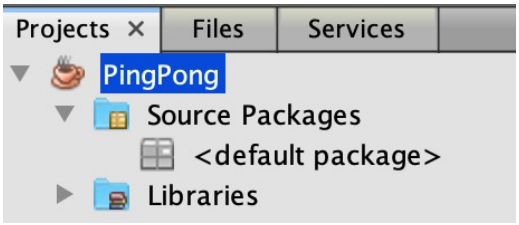
1. Choisissez dans le menu principal « File », puis « New project ... ».
2. Dans le dialogue suivant, sélectionnez dans la liste intitulée « Categories » l'élément « Java with Ant », puis dans la liste intitulée « Projects » l'élément « Java Application ».



3. Ensuite, vous devez choisir :
   - un nom pour votre projet (Project Name). Évitez les signes spéciaux, espaces et autres caractères non standards dans le nom de projet !
   - un dossier destination pour votre projet (Project Location). Évitez les signes spéciaux, espaces et autres caractères non standards aussi dans le chemin de destination de votre projet !
4. Veillez aussi à ce que l'option « **Create Main Class** » soit **désactivée** !



5. Après son initialisation, votre nouveau projet apparaît à gauche dans l'onglet « Projects » de la façon suivante :

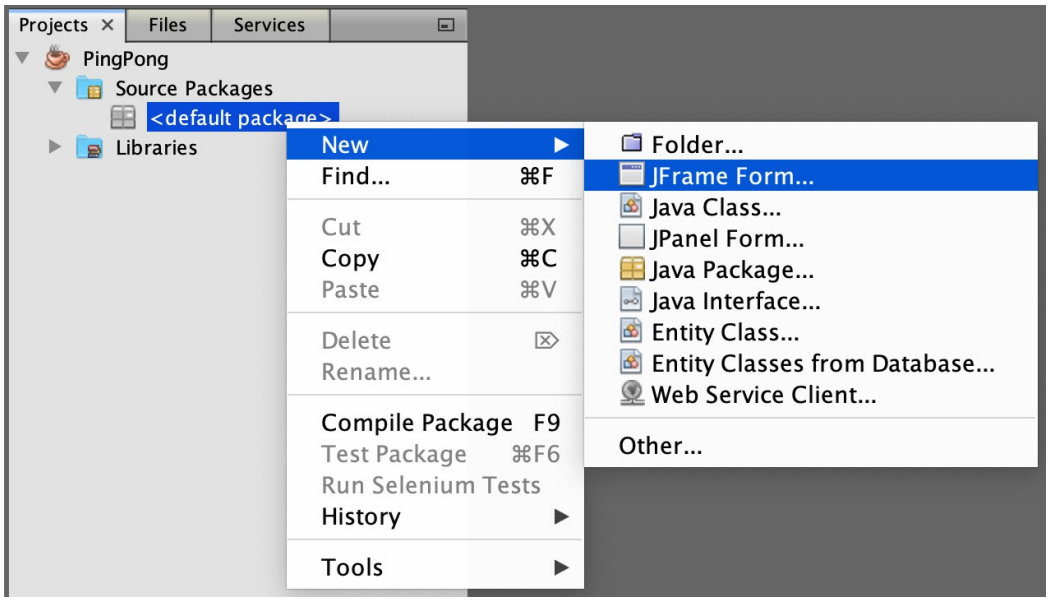


6. Vous venez de mettre en place un projet vide, qui ne contient encore aucune classe. Le chapitre suivant montre comment ajouter des classes.

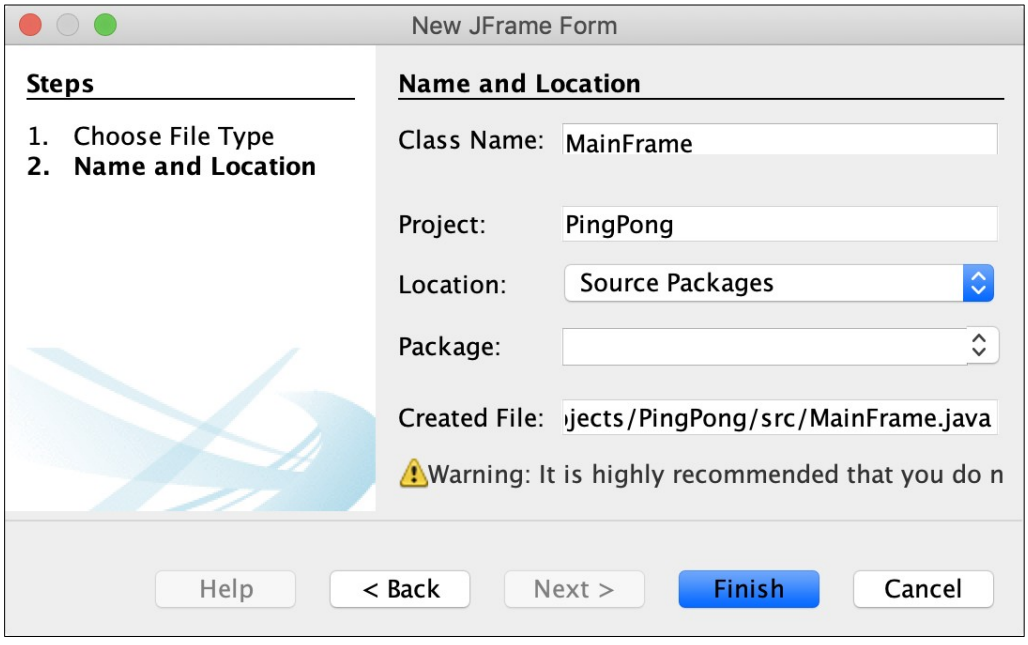
2.2.2. Ajouter des classes à un projet

En se basant sur le chapitre précédent, votre projet contient dans l'élément « Source Packages » un autre élément intitulé « <default package> ». Afin d'ajouter une classe à votre projet, vous devez :

1. faire un clic droit sur « <default package> » et choisir « New » dans le menu contextuel qui s'ouvre,



2. dépendant de ce que vous voulez ajouter, choisir :
   - « **JFrame Form...** » pour l'ajout d'une classe à interface graphique,
   - « **Java Class...** » pour l'ajout d'une simple classe (sans interface graphique).
3. Peu importe le type de classe que vous ajoutez à votre projet, vous devez ensuite indiquer le nom de la nouvelle classe avant de finaliser l'action :



2.3. Importer un projet de Unimozer dans NetBeans

Vu que Unimozer génère aussi des fichiers de configuration NetBeans lors de la sauvegarde d'une classe, il n'est pas nécessaire « d'importer » son projet.

En fait, il suffit de démarrer NetBeans et d'ouvrir le projet Unimozer de manière usuelle via le menu : **`File > Open Project...`**

2.4. [2GIN] Distribuer une application

Les applications Java ont le grand avantage d'être portables, c.-à-d. qu'elles tournent sur toutes les machines où on a installé l'environnement d'exécution de Java *(JRE - Java Runtime Environment)* et ceci sur n'importe quel système d'exploitation (Windows , Mac OS, Linux, ...). Le JRE est compris dans le *JDK (Java Development Kit)* que vous avez installé avec NetBeans.

Pour créer une application exécutable, faites un clic droit sur votre projet et cliquez sur **Clean and Build**. À la fin de l'opération le dossier **`dist`** de votre projet va contenir :

- Un sous-dossier **`lib`** contenant les bibliothèques nécessaires à l'exécution du projet
- Le fichier **`.jar`** qui contient l'application exécutable de votre projet
- Un fichier **`README.TXT`** décrivant les procédures pour exécuter le **`.jar`** et comment préparer le projet pour le distribuer et le déploiement sur un autre système

Pour préparer le projet pour une distribution, il suffit de zipper le contenu du dossier **`dist`** y compris le dossier **`lib`**.

Sur la machine cible il faut décomprimer le dossier et lancer le fichier **`.jar`**. Sur la plupart des systèmes, un double clic sur le fichier suffit, sinon il faut entrer dans la ligne de commande, localiser le dossier de l'application et taper : **`java -jar <nom de l'application>.jar`**

3. Notions importantes

Ce chapitre explique différentes notions importantes relatives à la création d'applications à interface graphique dans NetBeans.

3.1. Les paquets

En Java, un ensemble de classes peut être regroupé dans ce qu'on désigne par **package**, en français « paquet ». La spécification de Java propose un système similaire aux noms de domaine Internet (p.ex unimozer.fisch.lu) pour assurer l'unicité des noms de ces paquets.

Voici quelques exemples de programmes avec le nom du paquet de base respectif. Bien sûr, les classes sont réparties dans des sous-paquets se trouvant à l'intérieur de ce paquet de base.

Programme	Paquet de base
Unimozer	lu.fisch.unimozer
NetBeans	org.netbeans
Java « Swing »	javax.swing

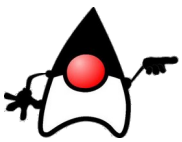
Dans le système fichier, les paquets se traduisent par une arborescence de répertoires. Voici l'exemple de l'arborescence dans le système fichier des paquets du programme « Unimozer » :

Lorsqu'une classe Java utilise des classes issues d'autres paquets, il faut indiquer ceci dans le code source devant la définition de la classe elle-même en faisant appel à l'instruction « **`import`** ».

Par exemple, lorsqu'on veut utiliser la classe « ArrayList » ainsi que la classe « JButton », il faut insérer les lignes de code suivantes :

```
import java.util.ArrayList;
import javax.swing.JButton;
```

Si vous ne savez pas dans quel paquet une classe se trouve, alors vous pouvez soit lancer une recherche sur Internet avec le mot clé « java » suivi du nom de la classe, soit laisser NetBeans ou Unimozer trouver lui-même la classe



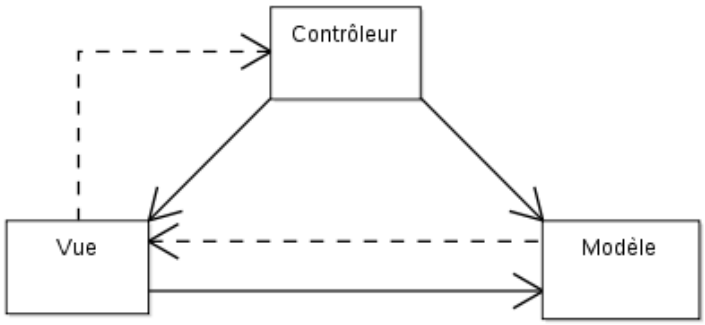
Il faut utiliser la touche **`<Ctrl><Space>`** pour activer la **complétion de code** lors de l'écriture du code, puis valider l'entrée par **`<Enter>`**. Ensuite, le programme placera automatiquement l'instruction « **`import`** » correspondante au début du code.

3.2. Le modèle « MVC »

Le **Modèle-Vue-Contrôleur** (en abrégé **MVC**, de l'anglais **Model-View-Controller**) est une architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application. Ce paradigme divise l'IHM en un **modèle** (modèle de données), une **vue** (présentation, interface utilisateur) et un **contrôleur** (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.

L'organisation globale d'une interface graphique est souvent délicate. L'architecture MVC ne résout pas tous les problèmes. Par contre, elle fournit dans la majorité des cas une bonne approche permettant de bien structurer une application.

Ce modèle d'architecture impose la séparation entre les données, la présentation et les traitements, ce qui donne trois parties fondamentales dans l'application finale : le modèle, la vue et le contrôleur.



Le schéma de cette figure résume les différentes interactions entre le modèle, la vue et le contrôleur. Les lignes pleines indiquent une association directe tandis que les pointillés sont une association indirecte.

3.2.1. Le modèle

Le modèle représente le comportement de l'application : traitements des données, interactions avec d'autres classes, etc. Il décrit ou contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité. Le modèle offre des méthodes pour mettre à jour ces données (insertion, suppression, changement de valeur). Il offre aussi des méthodes pour récupérer ces données. Les résultats renvoyés par le modèle sont dénués de toute présentation.

La majorité des classes réalisées en classe de 3GIG représentent en effet des modèles. En les réutilisant cette année ci, il ne reste qu'à les équiper d'une vue ainsi que d'un contrôleur afin de disposer d'application graphique toute faite.

3.2.2. La vue

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, activation de boutons, etc). Ces différents événements sont envoyés au contrôleur. La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

3.2.3. Le contrôleur

Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle, ce dernier avertit la vue que les données ont changé pour qu'elle se mette à jour. Certains événements de l'utilisateur ne concernent pas les données, mais la vue (p.ex. effacer des champs texte, activer/désactiver des boutons, ...). Dans ce cas, le contrôleur demande à la vue de se modifier. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée. Il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

Dans le cas du présent cours, la classe hébergeant la vue et celle hébergeant le contrôleur seront souvent la même, c'est-à-dire que le contrôleur et la vue se trouvent dans une même classe Java.

3.2.4. Flux de traitement

En résumé, lorsqu'un utilisateur déclenche un événement (par exemple en cliquant sur un bouton ou en sélectionnant un élément d'une liste) :

- l'événement envoyé depuis la vue est analysé par le contrôleur (par exemple un clic de souris sur un bouton),
- le contrôleur demande au modèle d'effectuer les traitements appropriés et notifie la vue que la requête est traitée (par exemple via une valeur de retour),
- la vue notifiée fait une requête au modèle pour se mettre à jour (par exemple afficher le résultat du traitement via le modèle).

3.2.5. Avantages du MVC

Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose. Cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. En effet, la modification des traitements ne change en rien la vue. D'un autre côté, on peut développer différentes vues et contrôleurs pour un même modèle.

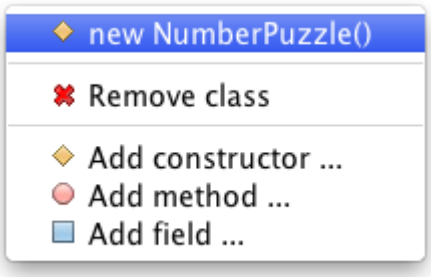
Exemples :

- On peut développer un interface texte et un interface graphique pour la classe **`EquationSolver`**.
- On peut améliorer un calcul ou un algorithme en changeant simplement le code source du modèle. La vue et le contrôleur ne s'en trouvent pas affectés.

3.3. *Création de nouveaux objets avec* 'new'

Dans le passé, vous avez créé des objets à l'aide de Unimozer en :

1. compilant votre code,
2. en faisant un clic droit sur la classe correspondante et
3. en choisissant « New ... » dans le menu contextuel:



La création de nouveaux objets peut aussi se faire dans le code à l'aide du mot-clé **new**. En général la syntaxe pour déclarer, créer et initialiser un objet de la classe est la suivante :

```
<type> <nom> = new <constructeur>;
```

Par exemple : création d'un objet **myClock** de la classe **Clock** avec le constructeur par défaut

```
Clock myClock = new Clock();
```

Remarques:

- Les instances ainsi créées ne seront pas visibles dans l'interface d'Unimozer. Elles existent dans la mémoire, mais nous ne pouvons pas les analyser directement.
- Si une classe possède plusieurs constructeurs, il existe aussi plusieurs possibilités pour créer un objet de cette classe. Dans le menu contextuel d'Unimozer se trouvent alors plusieurs lignes « New ... »

Il est aussi possible (mais plus rare) de séparer la déclaration d'un objet de sa création :

```
<type> <nom>;                 //déclaration de l'objet (encore inutilisable)
...
<nom> = new <constructeur>;   //création de l'objet (et initialisation)
```

Par exemple :

```
Clock myClock;          //l'objet myClock est déclaré mais n'existe pas encore

...

myClock = new Clock();  //un objet du type Clock est créé et affecté à myClock
```

Ici, **myClock** est **null** (→ voir plus bas), donc inutilisable. On obtient une erreur si on essaie d'accéder aux éléments de l'objet **myClock** (qui n'existe pas encore)

Exemple :

Soit la classe suivante :

```
public class Point
{
    private double x;
    private double y;

    public Point(double pX, double pY)
    {
          x = pX;
          y = pY;
    }

    public String toString()
    {
           return "(" + x + "," + y + ")";
    }
}
```

Cette classe ne possède pas de constructeur par défaut (= sans paramètres). Dans ce cas-ci, la ligne de code suivante peut être utilisée pour créer un nouveau point :

```
  Point myPoint = new Point(12,50);
```

Dans le cas présent, le point **myPoint** sera initialisée avec les coordonnées (12,50).

On peut utiliser l'instance **myPoint** dans la suite du code, p.ex :

```
  System.out.println(myPoint) ;
```

affiche dans ce cas **(12.0,50.0)**

Rappel :

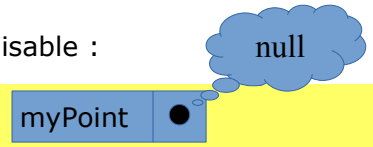
Ici il est possible d'écrire simplement **myPoint** au lieu de **myPoint.toString()**, parce que la méthode **println** fait automatiquement appel à **toString** (qui est définie pour tout objet).

3.4. *La valeur* 'null'

Une variable (attribut, paramètre ou variable locale) représentant un objet peut avoir la valeur spéciale **null**. Cette valeur indique que la variable objet a été déclarée, mais qu'elle est vide, c.-à-d. qu'il n'y a pas (encore) d'objet associé à la variable.

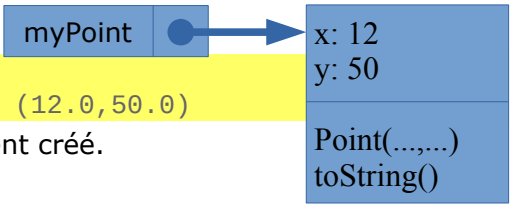
Exemples :

Sans initialisation, une variable objet est **null** (vide) et encore inutilisable :




```
Point myPoint;
System.out.println(myPoint);  // produit une erreur !!
```

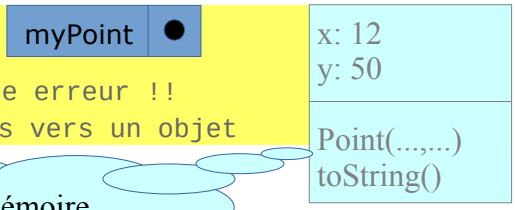
Après initialisation, elle fait référence à l'objet créé :



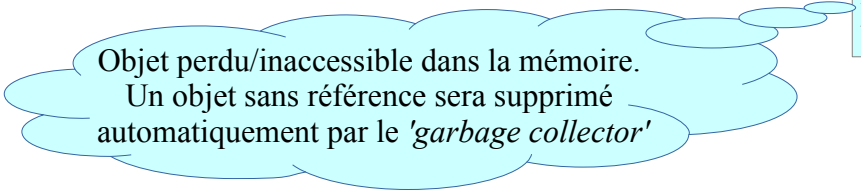

```
Point myPoint = new Point(12,50);
System.out.println(myPoint);        // affichage: (12.0,50.0)
```

On peut dire que **myPoint** ***pointe*** sur l'objet nouvellement créé.

En ajoutant maintenant une ligne fatale ...




```
Point myPoint = new Point(12,50);
myPoint = null;
System.out.println(myPoint);        // produit une erreur !!
                    // car myPoint ne pointe plus vers un objet
```



Autre exemple :

```
Point a = new Point(10,00);
Point b = a;                        // a et b pointent vers le même point!!
a = null;                           // a ne pointe plus vers l'objet
System.out.println(b);              // affichage: (10.0,0.0)
```

On peut bien entendu aussi tester si une variable pointe vers **null** ou non :

```
if (myPoint == null)
{
    System.out.println("myPoint est vide!");
}
else
{
    System.out.println(myPoint);
}
```

3.5. *[Pour avancés / 2GIN] Égo-référence* 'this'

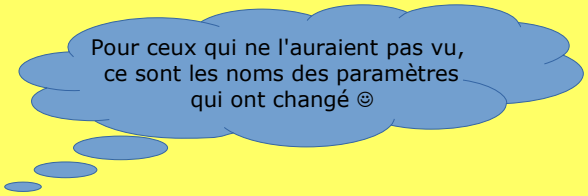
Parfois lors qu'on est en train d'écrire le code pour une classe, il faut faire référence à l'instance actuelle qui existera en mémoire lors de l'exécution. Au moment de la rédaction du code, on ne connaît cependant pas le nom de cette instance, et comme il peut exister beaucoup d'instances avec des noms différents il est même impossible d'en connaître le nom. C'est pour cette raison qu'existe la variable spéciale **this**. Cette variable pointe toujours sur l'instance actuelle, c.-à-d. l'objet lui-même.[1]

Exemple :

Reprenant le code de la classe **Point** déjà connue, mais avec une légère modification :

```
public class Point
{
    private double x;
    private double y;

    public Point(double x, double y)
    {
        x = x;
        y = y;
    }
}
```



Problème !!

Les deux affectations n'ont pas d'effet , car elles essaient d'affecter aux paramètres leurs propres valeurs...

Le problème consiste dans le fait que les noms des paramètres du constructeur sont identiques aux noms des attributs, les noms de paramètres « cachent » alors les attributs, c.-à-d : à l'intérieur du constructeur, les attributs ne sont plus accessibles par leur nom direct ![2]

C'est à cet instant qu'on a besoin de l'égo-référence **this** :

```
public class Point
{
    private double x;
    private double y;

    public Point(double x, double y)
    {
          this.x = x;
          this.y = y;
    }
}
```

Par le biais de **this**, on dit au compilateur d'utiliser non pas le paramètre, mais l'attribut de l'objet actuel. Le mot « this » traduit en français donne d'ailleurs « ceci » et veut dire dans notre cas « cet objet ». D'ailleurs, les programmeurs Java professionnels utilisent couramment cette méthode pour définir les constructeurs ou manipulateurs.

1 **this** n'existe pas dans les méthodes statiques

2 C'est d'ailleurs pour cette raison que nous avons ajouté le préfixe 'p' au paramètres dans le cours de 11e.

4. Types primitifs et classes enveloppes

En Java, les types **`int`**, **`float`**, **`double`**, **`boolean`** sont appelés types "primitifs" ou aussi types "simples". Les attributs ou variables de ces types ne sont pas des objets et ils sont utilisables dès leur déclaration, c.-à-d. on n'a pas besoin d'en créer des instances à l'aide de **`new`**.
(Tout le cours de 11e a été basé sur des attributs et variables de ces types primitifs, c'est pourquoi dans nos programmes, nous n'avons jamais eu besoin de créer explicitement des instances à l'aide de **`new`**).

Comme toutes les autres données en Java sont des objets, et la majorité des mécanismes prédéfinis en Java sont prévus pour des objets, il est parfois nécessaire de travailler avec des valeurs numériques qui sont des objets. De cette façon, on peut par exemple profiter de structures de données composées pour créer des listes de nombres (voir : listes de données → chapitre 8).

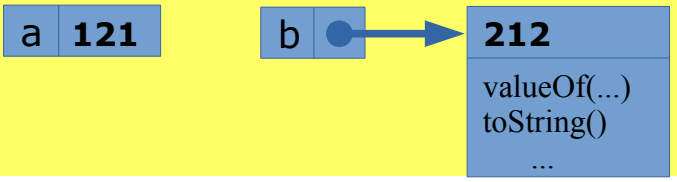
En Java, il existe pour chaque type primitif une classe **enveloppe** *(angl : **wrapper class**)* qui fournit une version objet des données :[3]

`Integer`	est la classe enveloppe pour le type	**`int`**
`Double`	est la classe enveloppe pour le type	**`double`**
`Byte`	est la classe enveloppe pour le type	**`byte`**
`Long`	est la classe enveloppe pour le type	**`long`**
`Float`	est la classe enveloppe pour le type	**`float`**
`Boolean`	est la classe enveloppe pour le type	**`boolean`**
`Character`	est la classe enveloppe pour le type	**`char`**

Exemples :

```
int a = 121 ;

Integer b = new Integer(212) ;
```



Un automatisme de Java nommé "***autoboxing***" effectue automatiquement les conversions entre les types primitifs et les objets des classes enveloppes. En général nous n'aurons donc pas besoin de créer des instances des classes enveloppes à l'aide de **`new`**.

Un autre avantage des classes enveloppes est qu'elles possèdent des méthodes utiles qui ne sont pas disponibles pour les types primitifs. La plus utile de ces méthodes est la méthode **`valueOf`** qui nous permet de convertir des nombres en textes et vice-versa (→ voir chapitre 5.1.2)

3 Programme 2GIG : utilisation des classes enveloppes limitée à **`Integer`** et **`Double`**.

5. Les chaînes de caractères « String »

Le présent chapitre est dédié à la manipulation simple des chaînes de caractères en Java. Les chaînes de caractères en Java (par exemple **"abc"**), sont représentées comme des instances de la classe **String**, qui est une classe spéciale.

5.1.1. Déclaration et affectation

Voici des exemples de déclaration d'une chaîne de caractères :

```
private String name;                        // déclaration d'un attribut
private String name = new String("René");   // ... avec initialisation
private String name = "René";               // ... et init. avec une constante

String filename;               // déclaration d'une variable
String filename = "test.doc";  // ... et initialisation avec une constante
```

Comme on peut le voir dans les exemples, les chaînes de caractères en Java sont comprises entre des guillemets (doubles).

5.1.2. Conversions de types

Souvent on a besoin de convertir des chaînes de caractères en des nombres, respectivement des nombres en des chaînes de caractères. La méthode la plus simple est d'utiliser la méthode « **valueOf(...)** » des classes **Integer**, **Double** et **String**.

Syntaxe	Explications
String.valueOf(int) **String.valueOf(double)**	Convertit un nombre entier ou un nombre décimal en une chaîne de caractères.
Integer.valueOf(String)	Convertit une chaîne de caractères en un nombre entier.
Double.valueOf(String)	Convertit une chaîne de caractères en un nombre décimal.

Si la conversion d'un texte en un nombre ne réussit pas, une exception du type **NumberFormatException** est levée et l'exécution de la méthode concernée se termine.

Exemples

```
String integerRepresentation = "3";
String doubleRepresentation  = "3.141592";

// faire la conversion d'une chaîne de caractères en un nombre entier
int myInt = Integer.valueOf(integerRepresentation);

// faire la conversion d'une chaîne de caractères en un nombre décimal
double myDouble = Double.valueOf(doubleRepresentation);

// faire la conversion d'un nombre entier en une chaîne de caractères
String intText = String.valueOf(myInt);

// faire la conversion d'un nombre décimal en une chaîne de caractères
String doubleText = String.valueOf(myDouble);
```

5.1.3. Autre méthode importante

Syntaxe	Explications
`boolean contains(String)`	Retourne **`true`** ssi la chaîne fournie comme paramètre fait partie de la chaîne actuelle (celle qui se trouve devant **contains**).

Exemples

```
String someText = "Hello world";
System.out.println(someText.contains("Hello"));       // affiche "true"
System.out.println(someText.contains("hello"));       // affiche "false"
System.out.println(someText.contains("lo wo"));       // affiche "true"
```

6. Comparaison d'objets

Pour comparer des objets, on **ne** peut **pas** se servir des opérateurs de comparaison standard (==, >, <, >=, <=) puisque ces derniers s'appliquent uniquement aux types primitifs (**`double`**, **`int`**, …). Pour les objets, il faut se servir des méthodes suivantes :

Syntaxe	Explications
`boolean equals(Object)`	Retourne « **`true`** » si les valeurs des deux objets sont égales, « **`false`** » sinon.
`int compareTo(Object)`	Retourne un nombre positif si la valeur de l'objet pour lequel on a appelé **compareTo** est plus grande que celle de l'objet passé en tant que paramètre, un nombre négatif sinon. La valeur 0 est retournée si les valeurs des deux objets sont égales.

6.1. Comparaison deux objets du type String

equals et **compareTo** font une différence entre majuscules et minuscules.
compareTo retourne la précédence alphabétique (lexicographique) de deux chaînes.

Exemples

```
String name1 = "abba";
String name2 = "queen";
String name3 = "Queen";
String name4 = "que" + "en";
name1.equals(name2)           =>    false
name1.equals(name3)           =>    false
name1.compareTo(name2)        =>    -16   (nombre négatif car "abba" < "queen")
name2.compareTo(name4)        =>    0     (zéro car "queen" est égal à "queen")
```

6.2. Comparaison d'instances de classes enveloppes

Pour comparer deux instances de classes enveloppes, il faut aussi employer **equals** et **compareTo**.

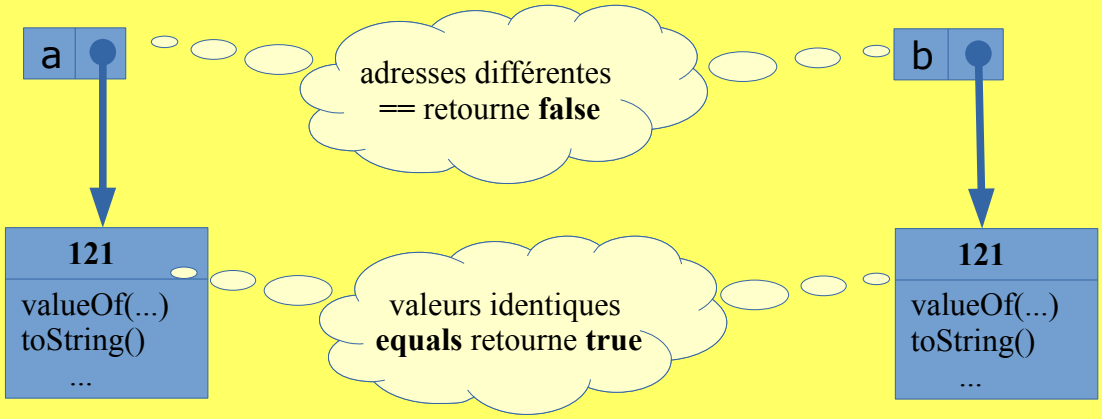
Exemples

```
Double n1 = 34.7;
Double n2 = 12.0;
n1.equals(n2)           =>    false
n1.compareTo(n2)        =>    1     (nombre positif car 34.5 > 12.0)
```

6.3. Explications pour avancés / 2GIN :

- Si on compare des objets avec les opérateurs de comparaison (==, >, <, >=, <=) alors Java compare les adresses en mémoire des deux objets et non leurs valeurs.

```
Integer a = 121;
Integer b = new Integer(121);
```

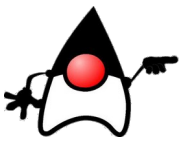


- Pour chaînes de caractères et classes enveloppes, Java essaie si possible d'économiser de la mémoire en réemployant des objets ayant la même valeur, ainsi il se peut que les résultats des opérateurs de comparaison soient identiques à ceux de **equals** et **compareTo**, mais ce n'est absolument pas garanti.
- Si nous voulons que **equals** et **compareTo** fonctionnent correctement pour les classes que nous définissons, nous devons redéfinir ces méthodes par nos soins.

7. Interfaces graphiques simples

En ouvrant NetBeans pour la première fois, vous avez remarqué qu'il existe toute une palette de composants visuels que nous pouvons intégrer dans nos programmes. Chacun de ces composants possède à son tour une multitude d'attributs, d'événements et de méthodes. Évidemment, il n'est pas possible (ni utile) de traiter tous ces éléments, voilà pourquoi nous nous limitons dans ce cours à traiter les attributs et événements incontournables des composants les plus simples.

Avant de commencer, veuillez noter que les noms de diverses classes standards du moteur principal graphique de Java (qui porte le nom « Swing »), commencent par la lettre « J ».



NetBeans possède deux modes d'édition : 

Source : pour entrer et éditer le code (le texte du programme)
Design : pour créer et modifier l'interface graphique de votre programme

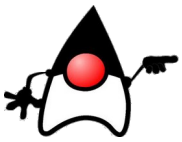
7.1. Les fenêtres « JFrame »

Le mot anglais « frame » désigne en français un « cadre », respectivement une « fenêtre » ou une « fiche ». La classe **`JFrame`** ne représente donc rien d'autre qu'une fiche vierge sur laquelle on peut poser d'autres composants visuels.

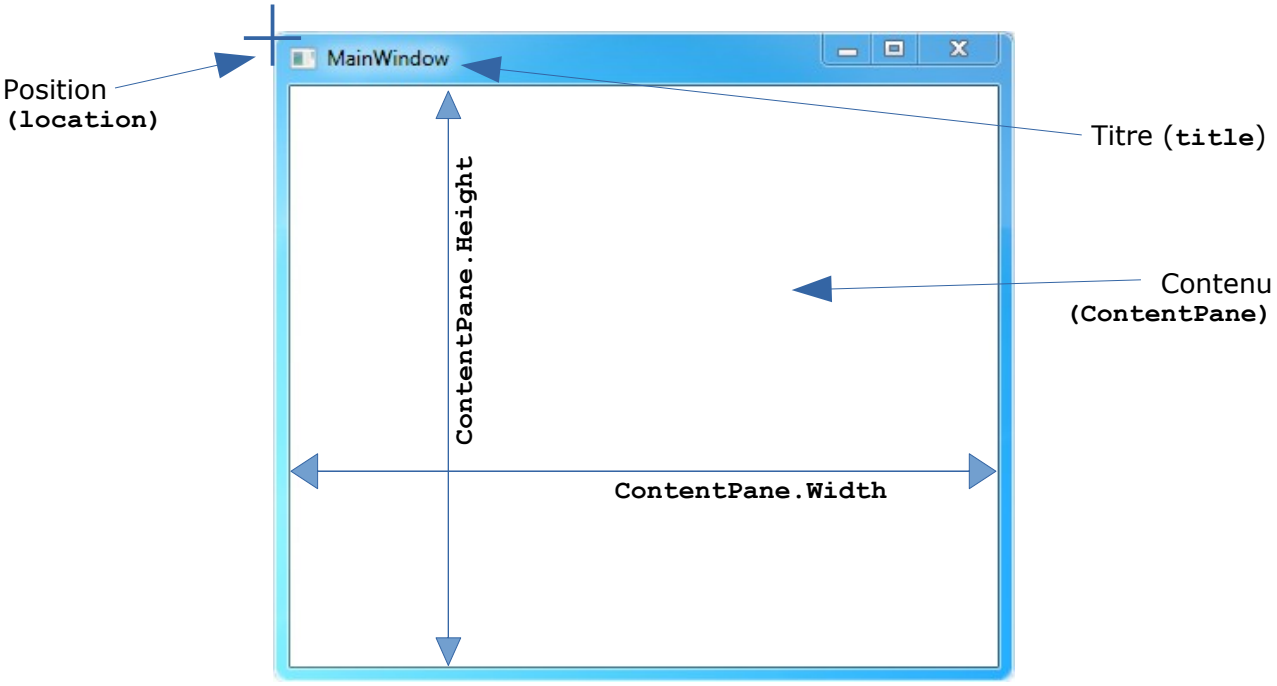
7.1.1. Attributs

Voici les éléments les plus importants d'une fenêtre :

Désignation	Propriété dans NetBeans	Accesseurs
le titre de la fenêtre	**`title`**	**`void setTitle(String)`** **`String getTitle()`**
les dimensions intérieures de la fenêtre	**`ContentPane.Width`** **`ContentPane.Height`**	**`int getContentPane().getWidth()`** **`int getContentPane().getHeight()`**



Suivez le guide du chapitre précédent pour ajouter une fenêtre à votre projet. En fait, une fenêtre n'est rien d'autre qu'une classe à interface graphique.



7.1.2. Initialisations

Toute classe visuelle du type **JFrame** ou **JPanel** a bien entendu la forme standard d'une classe Java, avec ses champs, ses méthodes et ses constructeurs.

Souvent on a besoin de déclarer des attributs ou d'exécuter des instructions lors de la création de la fenêtre principale. Pour ceci, il suffit d'ajouter les déclarations des champs au début de la classe, respectivement le code des initialisations dans le constructeur par défaut, mais derrière l'appel à la méthode **initComponents()** :

Par exemple si la classe s'appelle **MainFrame** :

```
public class MainFrame extends javax.swing.JFrame {
    // ajoutez vos déclarations de champs ici
     ...

     /** Creates new form MainFrame */
    public MainFrame()
    {
        initComponents();
        // ajoutez vos initialisations ici
          ...
    }
   ...
}
```

Sur une fenêtre, on peut placer d'autres composants, dont les plus connus sont les boutons, les libellés et les champs d'éditions. Étant que ceux-ci fonctionnent tous de la même manière, ils sont aussi traités dans un même chapitre.

7.2. Les composants standards

Par composant standard on entend :

- les boutons (**`JButton`**),
- les libellés (**`JLabel`**) et
- les champs d'entrée (**`JTextField`**).

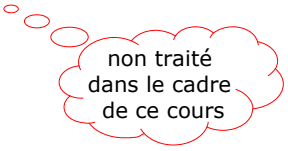
Tous ces composants possèdent chacun un nom *(Variable name)* et une inscription *(Text)*.

Voici les éléments les plus importants d'un tel composant:

7.2.1. Attributs

Désignation	Propriété dans NetBeans	Accesseurs
le texte affiché	**`text`**	**`void setText(String)`** **`String getText()`**
son état (activé ou non)	**`enabled`**	**`void setEnabled(boolean)`** **`boolean isEnabled()`**
sa visibilité		**`void setVisible(boolean)`** **`boolean isVisible()`**
l'image	**`icon`**	~~**`void setIcon(Icon)`**~~ ~~**`Icon getIcon()`**~~

Notons qu'un champ d'édition ne possède pas d'image. Pour cet attribut, on peut aussi choisir l'un des types d'images que voici : **`JPG`**, **`PNG`** ou **`GIF`**.



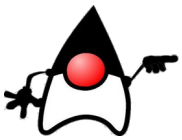
7.2.2. Événement

Nom dans NetBeans	Description
`actionPerformed`	Cet événement est déclenché si : • l'utilisateur clique sur le bouton ou • appuie <Enter> dans un champ d'édition. Un libellé ne possède pas cet événement.

7.3. Manipulation des composants visuels

Ce sous-chapitre contient des informations concernant la plupart ou même tous les composants visuels simples. Nous allons les traiter sur l'exemple du composant **`JButton`**.

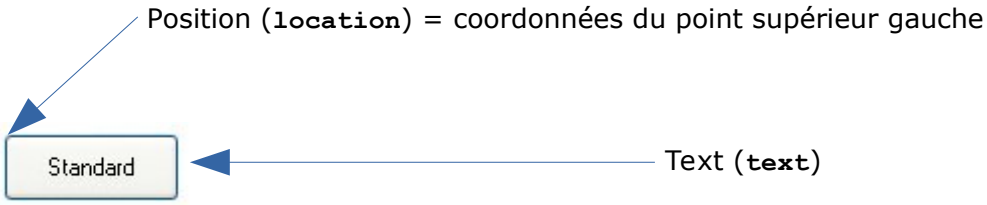
7.3.1. Placer un composant visuel sur une fiche



Afin de placer un composant sur votre fiche principale, rendez vous dans le mode « Design », cliquez dans la palette sur le composant (p.ex « Button »), puis cliquez à l'endroit auquel vous désirez le placer.

Les dimensions d'un composant (largeur et hauteur) sont en général adaptées automatiquement en fonction du texte y inscrit.

NetBeans essaie de placer les composants de façon optimale et flexible les uns par rapport aux autres et les replace automatiquement lorsque leurs dimensions changent. Ceci est parfois une aide, mais parfois il faut essayer de trouver la position optimale après des essais successifs.



Dépendant du système d'exploitation ou même de la version du système d'exploitation utilisé, les boutons peuvent avoir des apparences différentes.

7.3.2. Affichage et lecture de données

On peut afficher un texte, respectivement changer l'intitulé d'un composant à l'aide de la méthode **`setText(...)`**.

Exemple :

```
myLabel.setText("Hello world");
```

Des valeurs numériques, peuvent être affichées en employant **`String.valueOf(...)`.**

Exemple :

```
int myNumber = 35;
numLabel.setText(String.valueOf( myNumber ));
```

Pour lire un texte, par exemple à partir d'un champ d'édition, il faut utiliser la méthode **`getText()`**.

Exemple :

```
String name = nameTextField.getText();
```

Des valeurs numériques, peuvent être lues en employant **`Double.valueOf(...)`** ou **`Integer.valueOf(...)`**. Si la conversion du texte en un nombre ne réussit pas, une exception du type **`NumberFormatException`** est levée et l'exécution de la méthode concernée se termine.

Exemples :

```
int    n = Integer.valueOf(intTextField.getText());
double r = Double.valueOf(doubleTextField.getText());
```

7.3.3. Édition des propriétés

Afin de manipuler un bouton dans l'éditeur NetBeans, il suffit de le sélectionner, puis de se rendre dans l'éditeur des propriétés :

Properties	**Events**	**Code**
Ici vous pouvez changer les différentes propriétés visuelles du bouton.	Dans cet onglet, vous pouvez programmer les méthodes de réaction à des événements prédéfinis du bouton.	C'est l'endroit où vous devrez donner un nom correct au bouton.

Pour modifier la propriété **`Text`** d'un certain nombre de composants (**`JButton`**, **`JLabel`**, **`JTextField`**), on peut simplement appliquer un clic droit de la souris sur le composant et choisir *« Edit Text »*. Les dimensions du composant sont en général réadaptées automatiquement.

7.3.4. Noms des composants visuels

Pour modifier le **nom d'un composant**, il suffit d'appliquer un clic droit de la souris sur le composant et de choisir « *Edit Variable Name...* ». Dans ce cours nous suivons les conventions usuelles :

- Les noms de tous les composants visuels se terminent par leur type (en omettant le préfixe 'J'). Par exemple les noms des boutons (**`JButton`**) se terminent par **`Button`**, les noms des libellés (**`JLabel`**) se terminent par **`Label`**, les noms des champs texte (**`JTextField`**) se terminent par **`TextField`** etc.
- La première partie du nom indique leur fonctionnalité.
- La première lettre du nom est une minuscule.

Exemples : **`addButton`**, **`resultLabel`**, **`userNameTextField`**

Si vous changez le nom d'un composant pour lequel vous avez déjà écrit du code, NetBeans changera le nom aussi dans les lignes de code que vous avez déjà écrites.

Remarque : **<Ctrl>+<Space>**

Il peut sembler fastidieux de devoir entrer des noms aussi 'longs', mais NetBeans (tout comme Unimozer) vous assiste en complétant automatiquement un nom que vous avez commencé à entrer si vous tapez **<Ctrl>+<Space>**. → voir 11. Annexe B - Assistance et confort en NetBeans.

7.3.5. Événements et méthodes de réaction

Dans un environnement graphique, les actions d'un programme sont initiées par des événements. C'est pourquoi la programmation pour un environnement graphique est aussi qualifiée de **programmation événementielle** (EN: event-driven programming, DE: ereignisgesteuerte Programmierung).

Ainsi, tous les composants visuels peuvent réagir à un certain nombre d'événements (p.ex. un clic de la souris, l'activation d'une touche du clavier, le gain ou la perte de focus, l'écoulement d'un délai d'un chronomètre, l'arrivée de données sur le réseau, …). C'est le rôle du programmeur (c'est nous ☺) de définir les actions à effectuer pour les différents événements. Ces actions sont définies dans une méthode qui s'appelle alors la « **méthode de réaction** » à l'événement. Les noms des méthodes de réaction sont générés automatiquement par NetBeans en prenant le nom du composant comme préfixe et le nom de l'événement comme suffixe (p.ex. **`demoButtonActionPerformed`**).

Chaque méthode de réaction possède généralement un ou plusieurs paramètres qui renseignent sur la nature de l'événement, son origine et bien d'autres propriétés y relatives.

7.3.6. Attacher une méthode de réaction à un événement

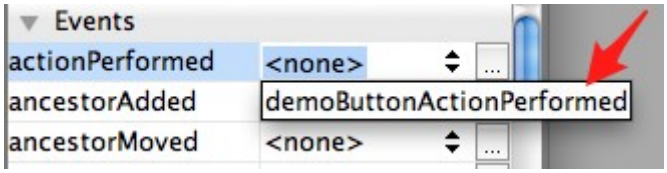
Afin d'attacher une méthode de réaction à un composant, sélectionnez le composant à l'aide de la souris, puis rendez vous dans l'onglet « Events » de l'éditeur des propriétés. Cliquez sur les petites flèches derrière l'événement, puis cliquez sur l'entrée présentée dans le menu contextuel se composant du nom du composant suivi du nom de l'événement.

L'éditeur saute ensuite automatiquement dans le mode « Source » de NetBeans et place le curseur dans la nouvelle méthode de réaction qu'il vient de créer.

Exemple :

Définition d'une méthode de réaction à l'événement **actionPerformed** d'un bouton :

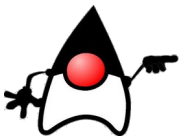
L'événement **actionPerformed** est celui qui est déclenché lorsque l'utilisateur clique sur le bouton (ou l'active le bouton par le clavier). C'est la méthode par défaut du bouton et c'est la seule qui nous intéresse pour les boutons. Dans l'exemple le bouton s'appelle « demoButton ».



L'éditeur saute ensuite automatiquement dans le mode « Source » et place le curseur dans la méthode de réaction respective qu'il a créée automatiquement auparavant :

```
private void demoButtonActionPerformed(java.awt.event.ActionEvent evt)
{
     // TODO add your handling code here:
}
```

C'est dans le corps de cette méthode que vous pouvez écrire le code qui doit être exécuté lorsque le bouton est activé (par la souris ou le clavier).



Pour certains composants (p.ex. les boutons, et les champs **JTextField**), il suffit d'appliquer un double clic sur le composant pour créer automatiquement leur méthode de réaction par défaut **actionPerformed**.

Exemple 1 :

```
private void demoButtonActionPerformed(java.awt.event.ActionEvent evt)
{
     demoButton.setText("Hello World!");
}
```

Affiche le texte "Hello World !" sur le libellé du bouton lorsqu'on clique dessus.

Pour avancés / 2GIN :

Essayez l'exemple ci-dessus, puis remplacez '**demoButton.setText**' par '**this.setTitle**'. Que constatez-vous ?

Pourquoi est-ce qu'on ne peut pas écrire '**MainFrame.setTitle**' (en supposant que la fiche s'appelle **MainFrame**) ? Quel est le rôle du mot clé '**this**' (→ voir chapitre 3.5) ?

Le paramètre **evt** du type **ActionEvent** donne des informations supplémentaires relatives à l'événement. Pour en savoir plus, veuillez consulter les pages JavaDoc y relatives !

2GIN :

En principe les détails de **evt** nous intéressent peu, mais par pure curiosité, on peut faire afficher par un simple **toString()** les informations de l'événement. Vous pouvez placer un composant du type **JTextArea** sur votre fiche et lui donner le nom **eventTextArea**. Dans la méthode de réaction au bouton placez la ligne :

```
eventTextArea.setText(evt.toString());
```

Pour mieux voir le texte, vous pouvez activer la propriété **lineWrap** de **eventTextArea**.

Que reconnaissez-vous dans ces informations ?

Poussez la touche *Ctrl* ou *Shift* en même temps que le bouton de la souris, que remarquez-vous ?

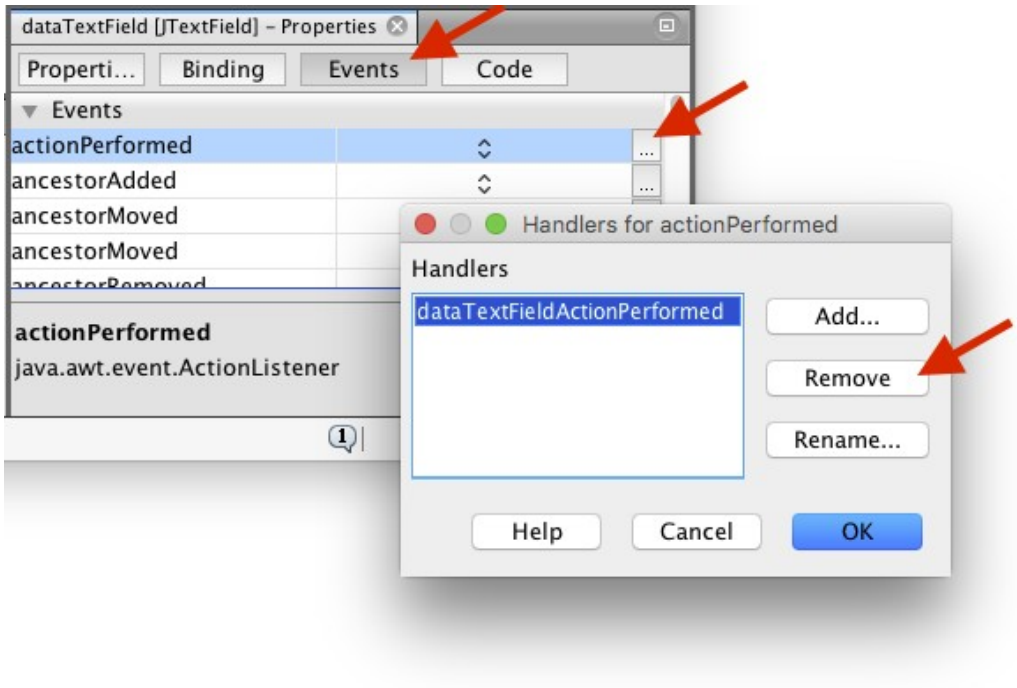
Exemple 2 :

Si vous avez intégré la classe **Point** dans votre projet :

```
private void demoButtonActionPerformed(java.awt.event.ActionEvent evt)
{
     Point myPoint = new Point(12.0 , 50.0);    // crée un nouveau point
     demoButton.setText(myPoint.toString());    // affiche les coordonnées du
                                                // point sur le bouton
}
```

7.3.7. Comment supprimer un événement

Afin de supprimer un événement, sélectionnez le composant pour lequel l'événement est défini, puis trouvez l'événement dans le gestionnaire des événements **[Events]**. Ouvrez le dialogue de l'événement en cause par le bouton **[...]** à droite, puis cliquez sur **Remove**.



ATTENTION: La suppression d'un événement supprime aussi tout de suite et sans demander de confirmation le code source complet de la méthode de réaction !

7.4. Les champs d'entrée « JTextField »

JTextField est un composant d'entrée, c'est-à-dire qu'on l'utilise essentiellement pour entrer des données dans un programme. Son contenu est toujours un texte, donc du type « String ».

Voici les éléments les plus importants d'un champ d'entrée :

7.4.1. Attributs

Désignation	Propriété dans NetBeans	Accesseurs
le texte contenu dans le champ d'entrée	**text**	**void setText(String)** **String getText()**

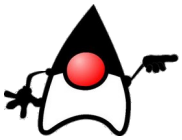
Des valeurs numériques, peuvent être lues en employant **Double.valueOf(...)** ou **Integer.valueOf(...)**. Si la conversion du texte en un nombre ne réussit pas, une exception du type **NumberFormatException** est levée et l'exécution de la méthode concernée se termine.

Exemples :

```
int    n = Integer.valueOf(intTextField.getText());
double r = Double.valueOf(doubleTextField.getText());
```

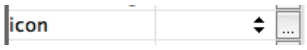
7.4.2. Événements

Nom dans NetBeans	Description
actionPerformed	Cet événement est déclenché si le curseur est placé dans le champ d'entrée et que l'utilisateur appuie sur la touche <Return> ou <Enter>.



Afin de placer un champ d'entrée sur votre fiche principale, rendez vous dans le mode « Design », cliquez dans la palette sur « Text Field », puis cliquez à l'endroit auquel vous désirez placer le champ d'entrée.

7.5. Afficher une image

Les libellés (**JLabel**) peuvent être utilisés pour afficher une image sur une fiche. On peut choisir l'un des types d'images que voici : **JPG**, **PNG** ou **GIF**. Modifiez pour ceci la propriété **icon** du libellé : 

en cliquant sur le trois points. Ensuite, vous pouvez choisir une image qui se trouve déjà dans le projet (**Image Within Project**) ou choisir une image sur le disque que vous importez dans le projet (**External Image → Import to Project ...**).

7.6. Autres composants utiles

À part les composants standards, il existe encore deux autres composants très utiles :

- les glissières (**`JSlider`**) et 
- les barres de progression (**`JProgressBar`**). 

Une glissière, en anglais « slider », est un composant d'entrée permettant au programme de sélectionner une valeur numérique contenue entre deux limites numériques.

Une barre de progression est un composant de sortie permettant par exemple de visualiser le taux de remplissage d'un récipient. Vous avec certainement déjà vu de telles barres de progression lors de l'installation d'un logiciel.

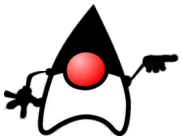
Voici les éléments les plus importants de ces composants :

7.6.1. Attributs

Désignation	Propriété dans NetBeans	Accesseurs
le minimum possible	**`minimum`**	**`void setMinimum(int)`** **`int getMinimum()`**
le maximum possible	**`maximum`**	**`void setMaximum(int)`** **`int getMaximum()`**
la valeur actuelle	**`value`**	**`void setValue(int)`** **`int getValue()`**

7.6.2. Événements

Nom dans NetBeans	Description
`stateChanged`	Cet événement est déclenché dès que la valeur actuelle de la glissière ou de la barre de progression change. Pour la glissière, cela se passe dès que le curseur de la glissière est déplacé.



Afin de placer une glissière ou une barre de progression sur votre fiche principale, rendez vous dans le mode « Design », cliquez dans la palette sur « Slider » ou « Progression Bar », puis cliquez à l'endroit auquel vous désirez placer la glissière.

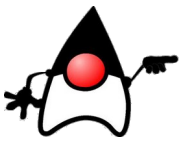
7.7. Les panneaux « JPanel »

En Java, un panneau peut être utilisé pour deux raisons majeures :

1. regrouper d'autres composants visuels et
2. faire des dessins (cf. chapitre 9).

7.7.1. Attributs

Désignation	Propriété dans NetBeans	Accesseurs
la couleur du panneau	**background**	**void setBackground(Color)** **Color getBackground()**
la largeur du panneau		**int getWidth()**
la hauteur du panneau		**int getHeight()**



Afin de placer un panneau sur votre fiche principale, rendez vous dans le mode « Design », cliquez dans la palette sur « Panel », puis cliquez à l'endroit auquel vous désirez placer le panneau.

D'autres éléments peuvent être placés sur le panneau comme sur une fiche. L'avantage est qu'on peut redimensionner le panneau ou déplacer l'ensemble des éléments qui se trouvent sur le panneau.

7.7.2. [2GIN] Accès au canevas

Parfois on a besoin d'accéder au canevas d'un panneau de l'extérieur. Comme tous les composants dérivés de **JComponent**, les éléments du type **JPanel** possèdent un méthode **getGraphics()** qui donne accès au canevas actuel du panneau.

Attention : Il faut quand même savoir que la meilleure méthode pour dessiner sur le canevas reste l'implémentation de la méthode **paintComponent.** En plus, le canevas d'un objet peut être remplacé par une nouvelle instance pendant que l'objet est redessiné.

7.8. [p. avancés/2GIN] Confort et ergonomie de la saisie

Si un utilisateur veut effectuer plusieurs traitements consécutivement dans un même interface, il doit procéder comme suit :

Après avoir entré des données dans un ou plusieurs champs texte, il doit cliquer sur un bouton pour activer le traitement, puis cliquer sur le premier champ texte, sélectionner les données qu'il a entrées auparavant, les supprimer, puis entrer de nouvelles données, et ainsi de suite.

Pour simplifier ce procédé. nous pouvons profiter des méthodes suivantes dans notre programme :

Composant(s)	méthode	Description
`JTextField`	**`selectAll()`**	Sélectionner automatiquement tout le texte d'un champ d'entrée. Utilisation : Au premier caractère que l'utilisateur entrera dans le champ texte, toute la sélection est remplacée par cette nouvelle entrée. L'utilisateur n'aura donc plus besoin de sélectionner et de supprimer le texte qui s'y trouve déjà.
`JComponent`	**`requestFocus()`**	Le *'focus'* détermine le composant actif sur le formulaire. A chaque instant, un seul composant possède le focus. Par **requestFocus**, nous pouvons déterminer par programme, quel composant possède le focus. Utilisation : A la fin de la méthode de réaction au bouton, nous faisons passer le focus au premier champ texte. Ainsi l'utilisateur n'aura pas besoin d'employer la souris pour passer dans le champ texte. De plus, en combinaison avec **ActionPerformed**, on peut faire passer le focus d'un champ texte au prochain lorsque l'utilisateur tape sur la touche 'Enter'.
`JButton`	**`doClick()`**	Cette méthode exécute la méthode de réaction liée à un bouton comme si on avait cliqué sur le bouton. Utilisation : Pour le dernier champ texte à remplir, nous créons une méthode de réaction à l'événement **ActionPerformed**. Dans la méthode de réaction, nous faisons un appel à la méthode **doClick** du bouton d'exécution.

En conséquence, l'utilisateur n'aura plus besoin de passer à la souris lors de la saisie consécutive et répétitive de données. Par ces petites améliorations de l'ergonomie[4], il économisera un temps considérable et son travail sera bien plus confortable.
Pour déterminer les possibilités d'améliorer l'ergonomie, le programmeur doit (faire) effectuer un certain nombre de tests pratiques avec son produit.

4 L'**ergonomie** est « l'étude scientifique de la relation entre l'homme et ses moyens, méthodes et milieux de travail » et l'application de ces connaissances à la conception de systèmes « qui puissent être utilisés avec le maximum de confort, de sécurité et d'efficacité par le plus grand nombre. » (source : Wikipedia.fr)

8. Les listes

8.1. Les listes « ArrayList »

La classe **ArrayList** permet de stocker une liste d'un nombre non défini d'objets quelconques. Elle permet cependant de spécifier quel type d'objet on aime y placer.

La classe **ArrayList** est contenue dans le paquet **java.util**. Il faut donc l'importer avant de pouvoir l'utiliser.

Exemple

```
// avant de pouvoir l'utiliser, il faut importer la classe ArrayList
import java.util.ArrayList;

...

// déclaration d'une liste de personnes (public class Person)
ArrayList<Person> alList = new ArrayList<>();
```

La syntaxe générique pour la déclaration d'une liste est la suivante :

ArrayList<classe_des_éléments> nom = new ArrayList<>();

C'est ici qu'on définit le type des éléments de la liste.

Nous pouvons donc nous-mêmes fixer le type des éléments de la liste. La classe **ArrayList** est donc une sorte de **modèle** (DE: **Vorlage** / EN: **template**) qui peut être appliqué à une autre classe. Pour des raisons de lisibilité, nous allons préfixer nos listes par le préfixe « **al** ».

Depuis Java 7, il n'est plus nécessaire de spécifier le type des éléments contenus dans la liste lors de l'appel au constructeur. Il suffit donc d'indiquer **<>** lors de l'initialisation.[5]

La taille d'une telle liste est dynamique, c'est-à-dire qu'elle s'adapte automatiquement en fonction du nombre d'éléments qu'on y place. Lors de la création de la liste, celle-ci est bien entendu vide.

Exemples :

```
ArrayList<String> alDemoList = new ArrayList<>();
```

alDemoList =

```
alDemoList.add("Hello");
alDemoList.add("World!");
```

alDemoList =

Hello	World!

```
alDemoList.add("Some");
alDemoList.add("1 2 3");
```

alDemoList =

Hello	World!	Some	1 2 3

5 En Java 6 et dans les versions antérieures, il faut répéter le type d'éléments lors de l'initialisation. P.ex. : `ArrayList<Person> alMyList = new ArrayList<Person>();`

8.1.1. Listes de types primitifs

Si on désire créer des listes contenant des éléments de types primitifs, donc par exemple **int** ou **double**, il faut passer obligatoirement par les classes enveloppes ! (cf. chapitre 4)

Déclaration d'une liste de nombres entiers :

```
ArrayList<Integer> alNomDeLaListe = new ArrayList<>();
```

Déclaration d'une liste de nombres décimaux :

```
ArrayList<Double> alNomDeLaListe = new ArrayList<>();
```

8.1.2. Méthodes

La classe **ArrayList** possède un bon nombre de méthodes qui nous permettent de manipuler la liste. Voici celles que nous utiliserons dans ce cours :

Méthode	Description
`boolean` **`add(Object)`**	Permet d'ajouter un élément à la liste. (Retourne toujours **true** – le résultat est en général ignoré.)
`void clear()`	Vide la liste en supprimant tous les éléments.
`boolean contains(Object)`	Teste si la liste contient un élément donné.
`Object get(int)`	Retourne l'élément à la position indiquée dans le paramètre.
`int indexOf(Object)`	Retourne la position de l'élément indiqué dans le paramètre ou -1 si l'élément ne se trouve pas dans la liste.
`Object` **`remove(int)`**	Supprime l'élément à la position indiquée dans le paramètre. Les éléments qui suivent l'élément supprimé avancent automatiquement d'une position. (Retourne l'élément supprimé comme résultat – le résultat est en général ignoré.)
`Object` **`set(int, Object)`**	Remplace l'élément à la position spécifiée par celui passé en tant que paramètre. (Retourne l'élément supprimé comme résultat – le résultat est en général ignoré.)
`int size()`	Retourne la taille de la liste, donc le nombre d'éléments y contenus.
`boolean isEmpty()`	Teste si la liste est vide (c.-à-d. test identique à **`size()==0`**)
`Object[] toArray()`	Transforme la liste en un « **`Array`** ». Cette méthode est uniquement nécessaire pour la visualisation des éléments à l'aide d'un composant « **`JList`** » (voir chapitre suivant).

Remarques :

- Les éléments d'une **`ArrayList`** sont indexés à partir de **0**.
- P.ex : Si une liste contient 10 éléments (**`size()==10`**) alors ces éléments ont les indices (positions) : 0, 1, 2, … , 9.
- Convention : Le nom d'une liste commence par le préfixe « **al** ».

Exemples :

Soit la liste suivante :

```
ArrayList<String> alList = new ArrayList<>();
```

Effet	Code

Ajouter les noms "Jean", "Anna" et "Marc" à la liste.	`alList.add("Jean");`
	`alList.add("Anna");`
	`alList.add("Marc");`

Supprimer le premier nombre de la liste (celui à la position 0).	`alList.remove(0);`
Sauvegarder le nombre d'éléments dans la variable **count**.	`int count = alList.size();`
Sauvegarder la position de l'élément "Marc" dans la variable **pos**.	`int pos = alList.indexOf("Marc");`
Remplacer l'élément à la position **pos** (ici le nom "Marc") avec le nom "Michelle".	`alList.set(pos,"Michelle");`
Sauvegarder le dernier élément de la liste dans la variable **last**.	`String last = alList.get(alList.size()-1);`

Tester si la liste contient la valeur "Marc".	`if (alList.contains("Marc"))`
	`  System.out.println("contained");`
	`else`
	`  System.out.println("not contained");`

Vider la liste.	`alList.clear();`

Tester si la liste est vide.	`if (alList.isEmpty()) …`
	ou bien :
	`if (alList.size()==0) …`

Conseil :

Consultez l'annexe C : 11.3 Ajout de propriétés et de méthodes <Insert Code...> pour simplifier la rédaction d'une classe contenant une **ArrayList**.

8.2. Les listes « JList »

La **JList** est assez similaire à la **ArrayList** avec la grande différence qu'il s'agit ici d'un composant visuel. Dans le cadre de ce cours, la liste visuelle **JList** est utilisée uniquement pour visualiser le contenu d'une **ArrayList**. En reprenant la logique du MVC, **JList** joue donc le rôle de la vue et du contrôleur pour le modèle **ArrayList**.

8.2.1. Attributs

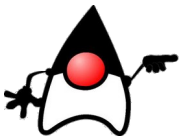
Désignation	Propriété dans NetBeans	Accesseurs
les éléments contenus dans la liste	**model**	**void setListData(Object[])**
le mode de sélection de la liste	**SelectionMode** Remarque: Veuillez toujours choisir la valeur « **SINGLE** »[6].	
la position de l'élément sélectionné de la liste. (-1 si aucun élément n'est sélectionné)	**selectedIndex**	**void setSelectedIndex(int)** **int getSelectedIndex()** **void clearSelection()**

Remarque :

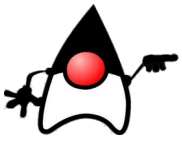
- **clearSelection()** est utilisé pour enlever toute sélection de la liste. Cette méthode modifie uniquement la sélection, le contenu de la liste reste inchangé.

8.2.2. Événements

Nom dans NetBeans	Description
valueChanged	Cet événement est déclenché dès que l'utilisateur clique sur un élément de la liste et change donc l'élément sélectionné.



Afin de placer une liste sur votre fiche principale, rendez vous dans le mode « Design », cliquez dans la palette sur « List », puis cliquez à l'endroit auquel vous désirez placer la liste.



Lorsque vous placez une **JList** sur la fiche principale, NetBeans l'incorpore automatiquement dans un **JScrollPane**. En effet, ce dernier est nécessaire pour le défilement du contenu de la liste.

6 **SelectionMode** a par défaut la valeur **MULTIPLE_INTERVAL** qui permet de sélectionner plusieurs groupes de données dans une **JList**. Dans notre cours, nous n'aurons besoin que d'une seule sélection par liste. Ainsi, nous allons toujours choisir l'option **SINGLE**, ce qui nous évitera aussi le traitement beaucoup plus complexe de sélections multiples.

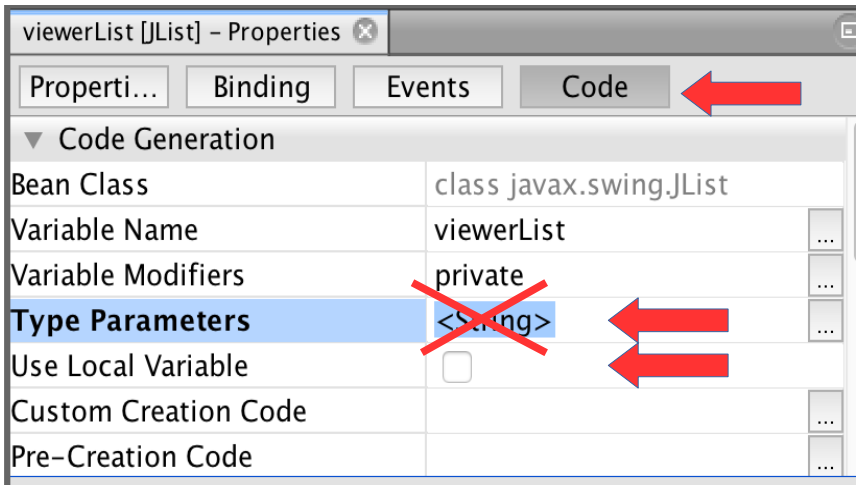
8.2.3. Préparer la liste pour accepter toute sorte d'objets

Dans les nouvelles versions de NetBeans, une **JList** accepte par défaut des listes de chaînes de caractères uniquement. Après avoir placé une **JList** sur votre fiche, vous pouvez vérifier cela dans la liste des composants à la fin du code dans **MainFrame**. Vous y trouvez une déclaration comme p.ex. :

```
private javax.swing.JList<String> viewerList;
```

Pour préparer la liste à accepter toute sorte d'objets et à afficher leur description textuelle, vous devez changer la définition du contenu de la **JList** comme suit :

- Dans le mode *Design*, sélectionnez la **JList** en question et choisir '**Code**' dans l'éditeur de propriétés.
- Supprimez ensuite l'indication **<String>** de '***Type Parameters***' et confirmez par 'Enter'.
- Veillez également à ce que l'option '***Use Local Variable***' reste décochée.

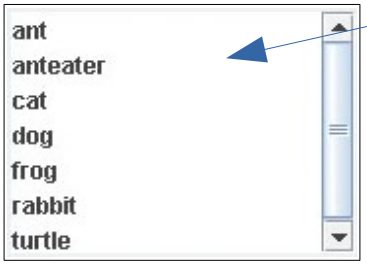


Maintenant, la déclaration dans la liste des composants à la fin du code dans **MainFrame** devrait se présenter comme suit :

```
private javax.swing.JList viewerList;
```

Dans la suite, nous allons utiliser la **JList** uniquement pour les deux cas de figure suivants :

- Vue : visualiser le contenu d'une liste du type **ArrayList**,
- Contrôleur : permettre à l'utilisateur de choisir un élément d'une liste.



Liste avec différents éléments.

Le texte affiché est celui retourné par la méthode **toString()** des objets contenus dans la liste.

Exemple :

Soit la structure de donnée suivante :

```
private ArrayList<Double> alMyList = new ArrayList<>();
```

On initialise la liste de la manière suivante :

```
alMyList.add(0.0);
alMyList.add(3.141592);
alMyList.add(1033.0);
```

En supposant l'existence d'une **JList** nommée **myNumberList**, le code suivant est nécessaire pour afficher le contenu de **alMyList** dans l'interface graphique :

```
  myNumberList.setListData( alMyList.toArray() );
```

En effet toute liste possède une méthode « **toArray()** » et une **JList** en a besoin afin de pouvoir visualiser le contenu d'une liste du type **ArrayList**.

En implémentant maintenant l'événement **valueChanged** de la façon suivante :

```
private void myNumberListValueChanged(javax.swing.event.ListSelectionEvent evt)
{
    System.out.println( myNumberList.getSelectedIndex() );
}
```

On peut observer dans la console de NetBeans (c'est la partie juste en dessous de la partie dans laquelle on écrit le code source) qu'à chaque changement de l'élément sélectionné de la liste, la position de ce dernier y est affiché.

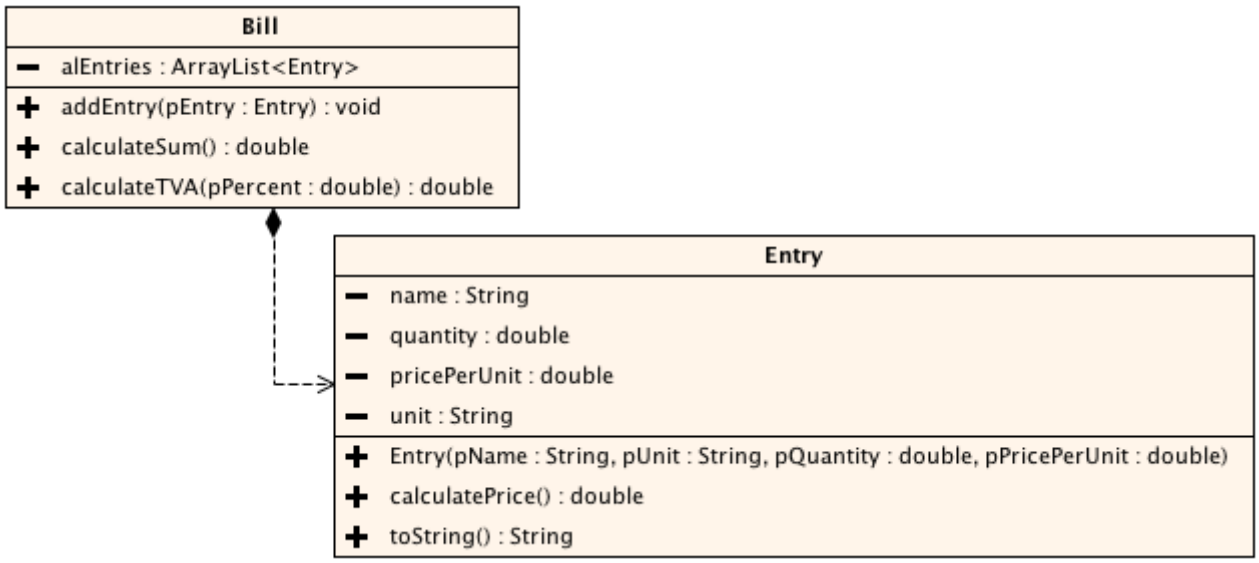
En modifiant le code de la manière suivante, ce n'est plus la position qui est affichée, mais l'élément lui-même :

```
private void myNumberListValueChanged(javax.swing.event.ListSelectionEvent evt)
{
    System.out.println( alMyList.get( myNumberList.getSelectedIndex() ) );
}
```

8.3. Les listes et le modèle MVC

Dans la plupart de nos projets, la liste (**ArrayList**) se trouve encapsulée à l'intérieur d'une autre classe (→ voir exercices). Nous ne pouvons donc pas appeler directement la méthode **toArray()** de la liste et il faudra passer par une étape intermédiaire.

Supposons par exemple que nous ayons une classe **Bill**, représentant une facture, ainsi que la classe **Entry**, représentant une entrée d'une facture. Le schéma UML est le suivant :



La liste **alEntries** est privée, donc inaccessible de l'extérieur. Afin de pouvoir afficher les éléments d'une facture dans une **JList**, il nous faut cependant un accès à la méthode **toArray** de **alEntries**. Comme nous ne voulons pas donner un accès complet à la liste **alEntries** de l'extérieur, nous allons seulement rendre accessible le résultat de **toArray()**. Nous créons donc à l'intérieur de **Bill** une méthode publique qui ne fait rien d'autre que retourner le résultat de **alEntries.toArray()**.

La méthode **toArray()** qu'on va ajouter à la classe **Bill** sera la suivante :

```
public Object[] toArray()
{
    return alEntries.toArray();
}
```

Remarquez que le résultat de **toArray()** est **Object[]**. En effet, la méthode **setListData(Object[])** de **JList** nécessite un paramètre du type **Object[]** et c'est précisément ce qu'on va lui offrir.[7]

Si **myList** est le nom d'une **JList** et **myBill** une instance de la classe **Bill**, la liste pourra être mise à jour de la manière suivante :

```
myList.setListData( myBill.toArray() );
```

7 Object[] est un tableau d'objets. La structure 'tableau' (array) est une structure de base de Java. Son utilité correspond à celle de ArrayList, mais elle est beaucoup moins confortable que ArrayList. Dans ce cours, nous ne traitons pas ce type de tableaux. Il suffit que vous reteniez qu'il faut noter des crochets [] à la fin.

9. Dessin et graphisme

Dans un environnement graphique, tout ce que nous voyons à l'écran (fenêtres, boutons, images, …) doit être dessiné point par point à l'écran. Les objets prédéfinis (JLabel, JButton, ...) possèdent des méthodes internes qui les dessinent à l'écran. Nous pouvons cependant réaliser nos propres dessins à l'aide de quelques instructions.

Le présent chapitre décrit une technique assez simple qui permet de réaliser des dessins en Java à l'aide d'un panneau du formulaire du type **`JPanel Form`**.

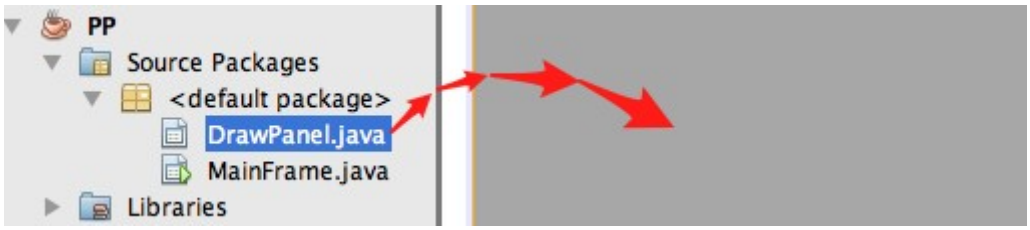
Voici le procédé à suivre, étape par étape, afin de créer un nouveau dessin :

1. Ajoutez un nouveau **`JPanel Form`** à votre projet et nommez-le **`DrawPanel`**.
2. Cette nouvelle classe possède une vue « source » et une vue « design » (comme un « JFrame »). En principe, nous n'allons jamais placer d'autres composants sur un **`JPanel Form`**. Activez la vue « source » et ajoutez la méthode suivante :

```
public void paintComponent(Graphics g)
{
    // ici sera programmé le dessin
}
```

3. Le compilateur va indiquer qu'il ne connaît pas la classe **`Graphics`**. Tout comme on l'a fait pour la classe **`ArrayList`**, il faut importer la classe **`Graphics`** afin de pouvoir employer ses méthodes, attributs et constantes. Il faut donc ajouter au début du fichier Java l'instruction : **`import java.awt.Graphics;`**

   Si vous l'oubliez, NetBeans vous affichera une petite ampoule d'erreur rouge dans la marge à côté du mot 'inconnu'. Si vous cliquez sur cette ampoule, NetBeans vous propose d'importer la classe pour vous : Choisissez simplement '**Add import for java.awt.Graphics**' et NetBeans va ajouter l'instruction.
4. Compilez votre projet en cliquant par exemple sur le bouton : 
5. Après avoir compilé, vous pouvez tirer avec la souris le **`DrawPanel`** à partir de l'arbre de la partie gauche de la fenêtre sur la **`MainFrame`**. Placez-le où vous voulez. Donnez-lui toujours le nom **`drawPanel`**.



Maintenant nous pouvons commencer à programmer le dessin proprement dit, en remplissant la méthode **`paintComponent`** de la classe **`DrawPanel`** avec du code. Mais avant ceci, il faut apprendre à connaître la classe **`Graphics`**. C'est elle qui représente un canevas. Elle nous permet de réaliser des dessins.

9.1. Le canevas « Graphics »

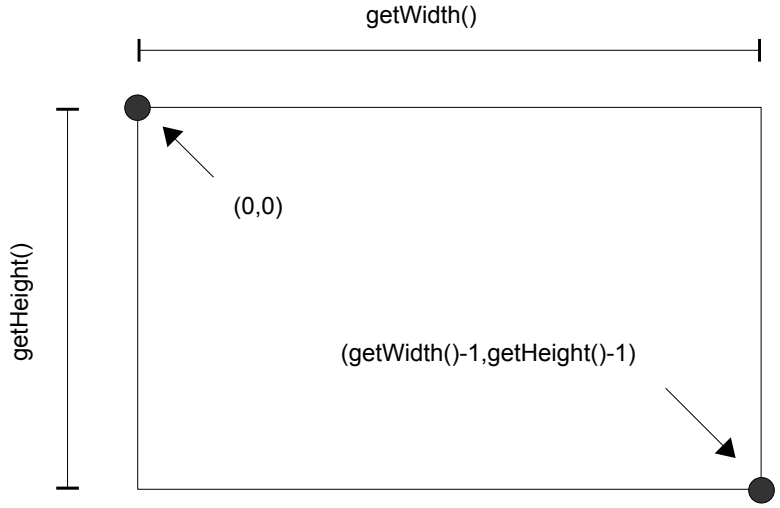
Un artiste peintre dessine sur un canevas (DE : *die Leinwand*). En informatique l'image de l'écran est aussi appelée canevas et elle consiste en une grille composée de pixels minuscules, des points qui peuvent prendre des millions de couleurs différentes.

En Java, le canevas n'est rien d'autre qu'une instance de la classe **`Graphics`**. Celle-ci possède un grand nombre de méthodes relatives aux dessins. Tous les composants graphiques possèdent un tel objet.

9.1.1. La géométrie du canevas

Quant à la géométrie d'un canevas, il faut savoir que l'axe des Y est inversé par rapport au plan mathématique usuel. L'origine, c'est-à-dire le point (**`0`**,**`0`**) se trouve en haut à gauche. Le point en bas à droite possède les coordonnées (**`getWidth()-1`**, **`getHeight()-1`**).

Attention : Le canevas lui-même n'a pas d'attribut indiquant sa largeur ou sa hauteur. Voilà pourquoi ces données doivent être prises du panneau **`JPanel`** sur lequel on dessine.



9.1.2. Les méthodes du canevas

Méthode	Description
`Color getColor()` `void setColor(Color)`	Permet de récupérer la couleur actuelle, et d'en choisir une nouvelle.
`void drawLine(int x1, int y1,` `int x2, int y2)`	Dessine à l'aide de la couleur actuelle une ligne droite entre les points **(x1,y1)** et **(x2,y2)**.

`void drawRect(int x, int y,`
`int width, int height)`
 	
`void fillRect(int x, int y,`
`int width, int height)`
	Dessine à l'aide de la couleur actuelle, un rectangle tel que **(x,y)** représente le point supérieur gauche, tandis que **width** et **height** représentent sa largeur respectivement sa hauteur.

`void drawOval(int x, int y,`
`int width, int height)`
 	
`void fillOval(int x, int y,`
`int width, int height)`
	Dessine à l'aide de la couleur actuelle, une ellipse telle que **(x,y)** représente le point supérieur gauche, tandis que **width** et **height** représentent sa largeur respectivement sa hauteur.

`void drawString(String s, int x, int y)`	Dessine le texte **s** à la position **(x,y)** tel que **(x,y)** représente le point inférieur de l'alignement de base du texte.

Remarque :

On peut colorer un seul point (pixel) du canevas en traçant une 'ligne' d'un point vers le même point. P.ex. **`g.drawLine(50,100,50,100);`**

9.1.3. La méthode repaint()

Le panneau (et tous les composants visibles) possèdent une méthode **`repaint()`** qui redessine le composant et tout ce qui se trouve dessus. Lors d'un appel de **`repaint()`** d'un **`DrawPanel`** la méthode **`paintComponent(...)`** est appelée automatiquement. C.-à-d. si vous voulez faire redessiner le panneau, il est pratique d'appeler simplement **`repaint()`**.

Exemple pour l'emploi du canevas :

Supposons l'existence d'un panneau **drawPanel** placé de telle manière sur la fiche principale, qu'il colle à tous les côtés. La méthode **paintComponent** de la classe **DrawPanel** contient le code suivant :

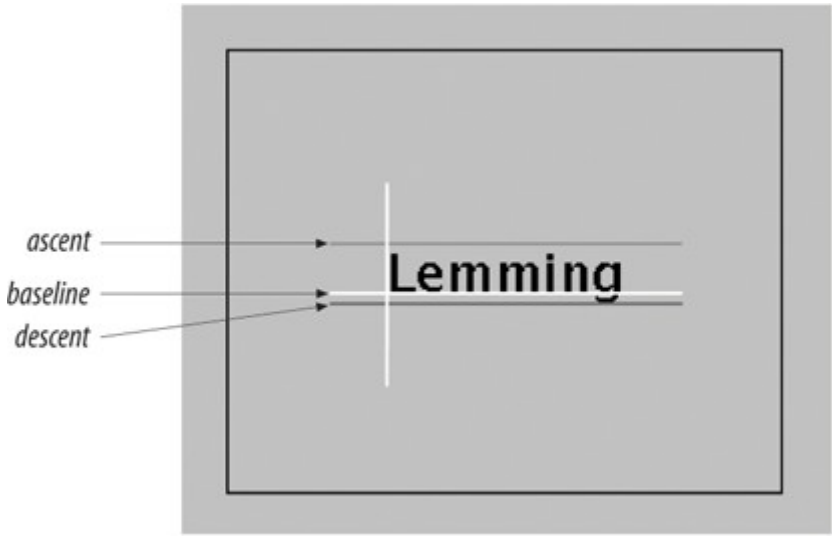
```
/**
 * Méthode qui dessine une _____________________
 */
public void paintComponent(Graphics g)
{
    g.setColor(Color.RED);
    g.drawLine(0,0,getWidth(),getHeight());
    g.drawLine(0,getHeight(),getWidth(),0);
}
```

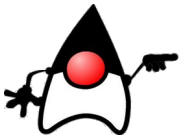
Conclusions :

- Que dessine cette méthode ?
- Que se passe-t-il lorsqu'on agrandit la fenêtre principale ?
- Ajoutez, respectivement complétez les commentaires !
- Comme contrôle, essayez cet exemple en pratique (après avoir lu le chapitre 9.2).

9.1.4. Affichage et alignement du texte

Tout texte dessiné à l'aide de la méthode **drawString(...)** est dessiné de telle manière à ce que les coordonnées passées comme paramètres représentent le point inférieur gauche de l'alignement de base (« baseline ») du texte.





Voici le lien direct vers la page JavaDoc :
http://download.oracle.com/javase/8/docs/api/java/awt/Graphics.html

9.2. La classe « Color »

La classe **Color** modélise une couleur. Elle prédéfinit un grand nombre de couleurs mais elle permet aussi de spécifier une nouvelle couleur selon le schéma **RGB** (red/green/blue).

Pour pouvoir employer les méthodes, attributs et constantes de la classe **Color**, il faut ajouter au début du fichier Java l'instruction : **`import java.awt.Color;`**

Si vous l'oubliez, NetBeans ne connaîtra pas les instructions de la classe **Color**, et il vous affichera une petite ampoule d'erreur rouge dans la marge à côté du mot 'inconnu'. Si vous cliquez sur cette ampoule, NetBeans vous propose d'importer la classe pour vous : Choisissez pour cela '**Add import for java.awt.Color**'.

9.2.1. Constructeur

Une couleur est une instance de la classe **Color** et chaque nouvelle couleur doit en principe être créée, comme tout autre objet. La classe **Color** possède plusieurs constructeurs, dont voici le plus important :

Constructeur	Description
`Color(int red, int green, int blue)`	Crée une nouvelle couleur avec les différentes quantités de rouge, vert et bleu. Les valeurs des paramètres doivent être comprises dans l'intervalle [0,255].
`Color(int red, int green, int blue, int alpha)`	comme ci-dessus, mais le paramètre **alpha** permet de fixer le degré de transparence pour la couleur. Exemples : alpha=0 => la couleur est complètement transparente alpha=127 => la couleur est transparente à 50% alpha=255 => la couleur est opaque (non transparente)

Exemples :

`Color color1 = new Color(0,0,0);` crée une couleur _____

`Color color2 = new Color(0,0,128);` crée une couleur _____

`Color color3 = new Color(200,200,0);` crée une couleur _____

`Color color4 = new Color(255,0,255);` crée une couleur _____

`Color color5 = new Color(64,64,64);` crée une couleur _____

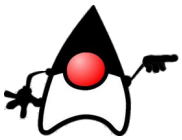
`Color color6 = new Color(0,0,0,127);` crée une couleur _____

`Color color7 = new Color(0,255,0,64);` crée une couleur _____

9.2.2. Constantes

La classe **`Color`** possède un certain nombre de couleurs prédéfinies. Ce sont des constantes qui n'ont pas besoin d'être créées avant d'être employées. En voici quelques exemples :

`Color.RED` **`Color.BLUE`** **`Color.YELLOW`** **`Color.GREEN`** **`Color.GRAY`**



- Dans NetBeans ou Unimozer, tapez « **`Color.`** », puis utilisez les touches <Ctrl>+<Space> pour activer le complément automatique de code (EN: *code completion*) qui vous proposera toutes les constantes importantes.
- Voici le lien direct vers la page JavaDoc :
  http://download.oracle.com/javase/8/docs/api/java/awt/Color.html

9.3. *La classe « Point »*

Un grand nombre de méthodes en Java fonctionnent sur base de coordonnées dans un plan. Souvent, des coordonnées sont indiquées sous forme de points définis à l'aide d'une classe **`Point`**. Toute instance de cette classe **`Point`** représente donc un point dans un plan bi-dimensionnel.

9.3.1. Constructeur

Constructeur	Description
`Point(int x, int y)`	Crée un nouveau point avec les coordonnées **`(x,y)`**.

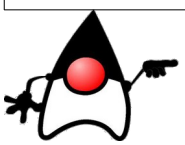
9.3.2. Attributs

La classe **`Point`** possède deux attributs publics, et donc accessibles directement, qui représentent les coordonnées du point.

Attributs	Description
`int x , int y`	Les coordonnées **`(x,y)`** du point dans le plan.

9.3.3. Méthodes

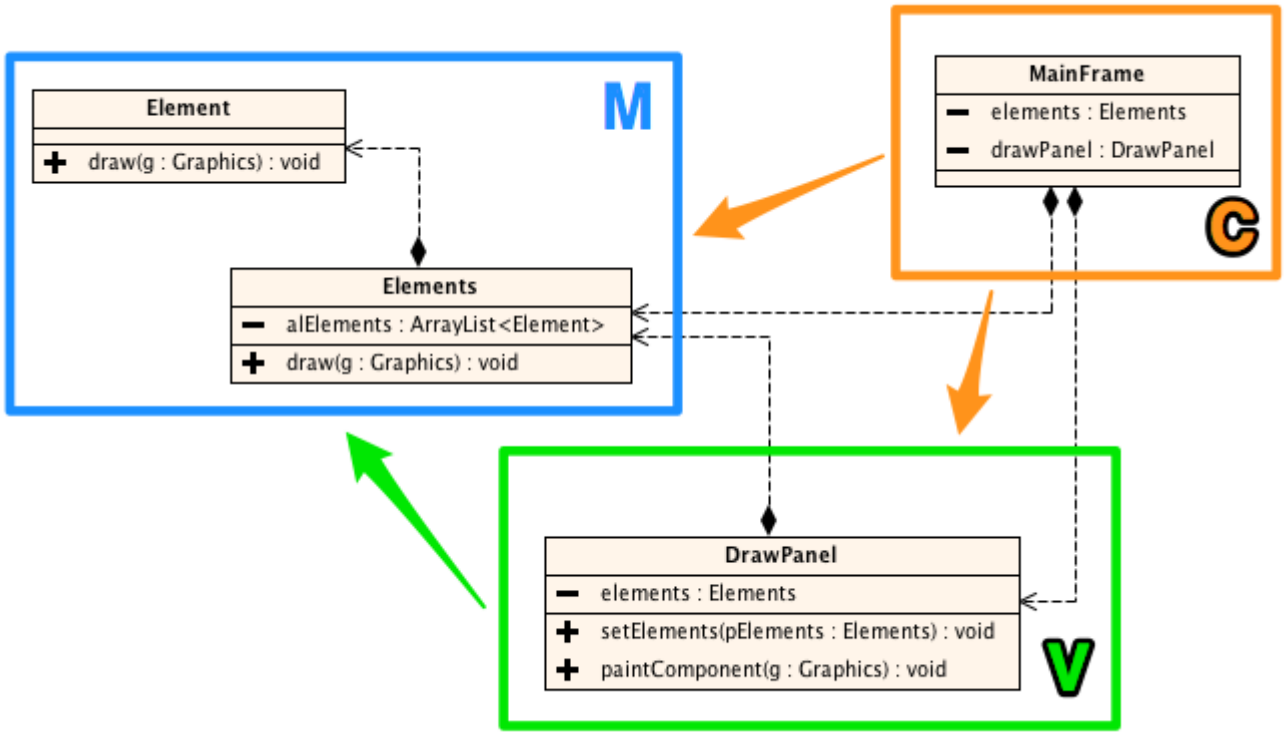
Méthodes	Description
`double getX()` **`double getY()`**	Retournent la coordonnée **`x`** ou **`y`** du point en question.
`void setLocation(int x, int y)`	Méthodes qui permettent de repositionner un point dans l'espace.
`void setLocation(double x,` **`double y)`**	Cette version de la méthode arrondit les valeurs réelles automatiquement vers des valeurs entières.
`Point getLocation()`	Retourne le point lui-même.



Voici le lien direct vers la page JavaDoc :
http://download.oracle.com/javase/8/docs/api/java/awt/Point.html

9.4. Les dessins et le modèle MVC

Souvent, nous allons avoir besoin de dessiner plusieurs éléments se trouvant dans une liste. Pour ce faire, nous allons suivre une logique qui repose le plus possible sur le schéma MVC, mais en évitant de trop compliquer la structure des classes[8]. Afin de mieux illustrer ceci, prenons le schéma UML que voici :



Explications :

- Afin de simplifier la lecture du schéma, un grand nombre de propriétés et méthodes dont on a généralement besoin dans un tel programme ont été omis !
- La classe **`Element`** représente un des éléments à dessiner. On remarque que chaque élément possède une méthode **`draw`** permettant à l'élément de se dessiner soi-même sur un canevas donné.
- La classe **`Elements`** représente une liste d'éléments (**`alElements`**). Sa tâche est de gérer cette liste. On remarque que cette classe possède aussi une méthode **`draw`** permettant de dessiner tous les éléments contenus dans la liste sur un canevas donné.
- Les deux classes **`Element`** et **`Elements`** représentent notre modèle.
- La classe **`DrawPanel`** est bien évidemment la vue. Cette classe a comme tâche unique de réaliser le dessin. Pour ceci, elle a besoin d'un lien vers le modèle. Pour cette raison elle possède une méthode **`set`**... (ici : **`setElements`**) qui est appelée par le contrôleur.

8 En suivant le modèle MVC dans sa dernière conséquence, à chaque classe du modèle devrait être associée une classe correspondante pour la vue. Comme nos projets sont de taille assez limitée, nous allons utiliser une solution de compromis où les classes du modèle peuvent posséder une méthode **draw** pour dessiner leurs instances.

- La classe **`MainFrame`** est le contrôleur. Elle gère donc toute l'application. Ses tâches principales sont les suivantes :
    1. Créer et manipuler les instances du modèle,
    2. garantir que la vue et le contrôleur travaillent avec la même instance du modèle. A cet effet, il faut appeler la méthode **`setElements`** de **`drawPanel`** à chaque fois qu'une nouvelle instance de **`Elements`** est créée,
    3. gérer et traiter les entrées de l'utilisateur (clic sur bouton, souris, …),
    4. mettre à jour la vue (p.ex. en appelant **`repaint()`**).

Exemple :

Voici comme illustration des extraits de code pour une application typique :

MainFrame.java :

```
public class MainFrame extends javax.swing.JFrame {

    private Elements elements = new Elements(); //initialiser les éléments

    public MainFrame() {
        initComponents();
        drawPanel.setElements(elements);        //synchronisation initiale
    }
     . . .

    private void newButtonActionPerformed(java.awt.event.ActionEvent evt) {
        elements = new Elements();              //réinitialiser les éléments
        drawPanel.setElements(elements);        //ré-synchronisation
        repaint();
    }
     . . .
}
```

DrawPanel.java :

```
public class DrawPanel extends javax.swing.JPanel {

     private Elements elements = null;

     public void setElements(Elements pElements) {
          elements = pElements;
     }

     public void paintComponent(Graphics g) {
          g.setColor(Color.WHITE);
          g.fillRect(0, 0, getWidth(), getHeight());

          if (elements != null)       //pour éviter une NullPointerException
               elements.draw(g);
     }
}
```

9.5. *Pour avancés (2GIN) : Graphics2D, Stroke, Point2D*

9.5.1. Graphics2D

Le canevas **g** est en vérité du type **Graphics2D**. Pour des raisons de compatibilité, les paramètres sont resté du type **Graphics**. La classe **Graphics2D** offre un certain nombre de fonctions avancées, dont nous pouvons profiter, si nous effectuons un cast de **Graphics g** vers **Graphics2D**. Par exemple dans **paintComponent**, nous pouvons faire une conversion et ensuite passer le canevas en tant que type **Graphics2D** à la méthode **draw** :

```
public void paintComponent(Graphics g) {
     g.setColor(Color.WHITE);
     g.fillRect(0, 0, getWidth(), getHeight());

     Graphics2D g2D = (Graphics2D)g;  //conversion (cast) vers Graphics2D
     if (elements != null)
          elements.draw(g2D);
}
```

Dans ce cas, la méthode **draw** est alors définie comme suit :

```
public void draw(Graphics2D g) {     //g est maintenant du type Graphics2D
     g.setColor(Color.WHITE);

     //setStroke est une méthode de Graphics2D
     g.setStroke(new BasicStroke(3f));
}
```

9.5.2. *Stroke*, la largeur du pinceau

La méthode **setStroke** de **Graphics2D** nous permet de définir une largeur pour le pinceau lors du dessin. Les méthodes **drawRect**, **drawOval** et **drawLine** utilisent alors cette largeur pour dessiner les lignes et les contours des figures. Modifier la propriété **Stroke**, nous créons un nouvel objet du type **BasicStroke** en utilisant l'un de ses constructeurs. Par exemple :

```
     //faire des traits avec 3 points de largeur
     //le paramètre width doit être du type float.
     //3.0f définit une constante du type float (au lieu de double)
     g.setStroke(new BasicStroke(3.0f));
```

```
     //faire des traits de 10 points de largeur
     //avec des coins et des bouts arrondis
     g.setStroke(new BasicStroke(10.0f, BasicStroke.CAP_ROUND,
                                        BasicStroke.JOIN_ROUND));
```

9.5.3. Le type Point2D

La classe **Point** permet de mémoriser des points avec des coordonnées entières (type **int**). Il existe un type **Point2D**, qui permet de mémoriser des coordonnées plus précises du type **float** ou **double**. Pour créer une instance de ce type, nous devons spécifier la précision désirée lors de l'initialisation :

```
Point2D p1 = new Point2D.Float(12.4f, 33.59f);   //coord. du type float

Point2D p2 = new Point2D.Double(0.25, 1.0/3);    //coord. du type double
```

Les méthodes de **Point2D** sont similaires à celles **Point**, mais elles travaillent avec des valeurs du type **double**.

Les classes mentionnées dans ce chapitre vous offrent une panoplie de méthodes supplémentaires. C'est une bonne idée de consulter JavaDoc à ce sujet.

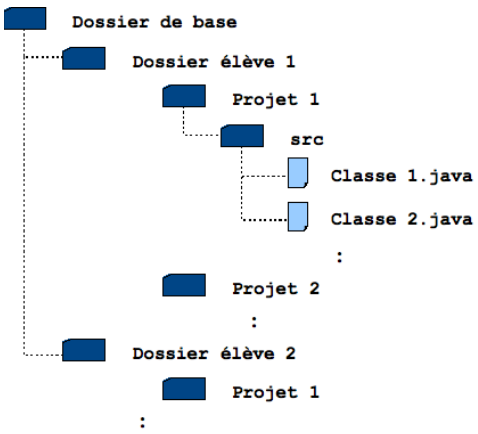
10. Annexe A - Impression de code NetBeans

En Unimozer, il est facile d'imprimer le code des classes modèles. Pour imprimer le code des classes Vue/Contrôleur (***JFrame***) développées en NetBeans, il est cependant important de bien choisir les options d'impression pour ne pas imprimer trop de pages superflues (contenant p.ex. le code généré ou des pages vides). L'impression en NetBeans est possible, mais nous conseillons le programme « JavaSourcePrinter » qui a été développé spécialement pour nos besoins.

10.1. Impression à l'aide du logiciel « JavaSourcePrinter »

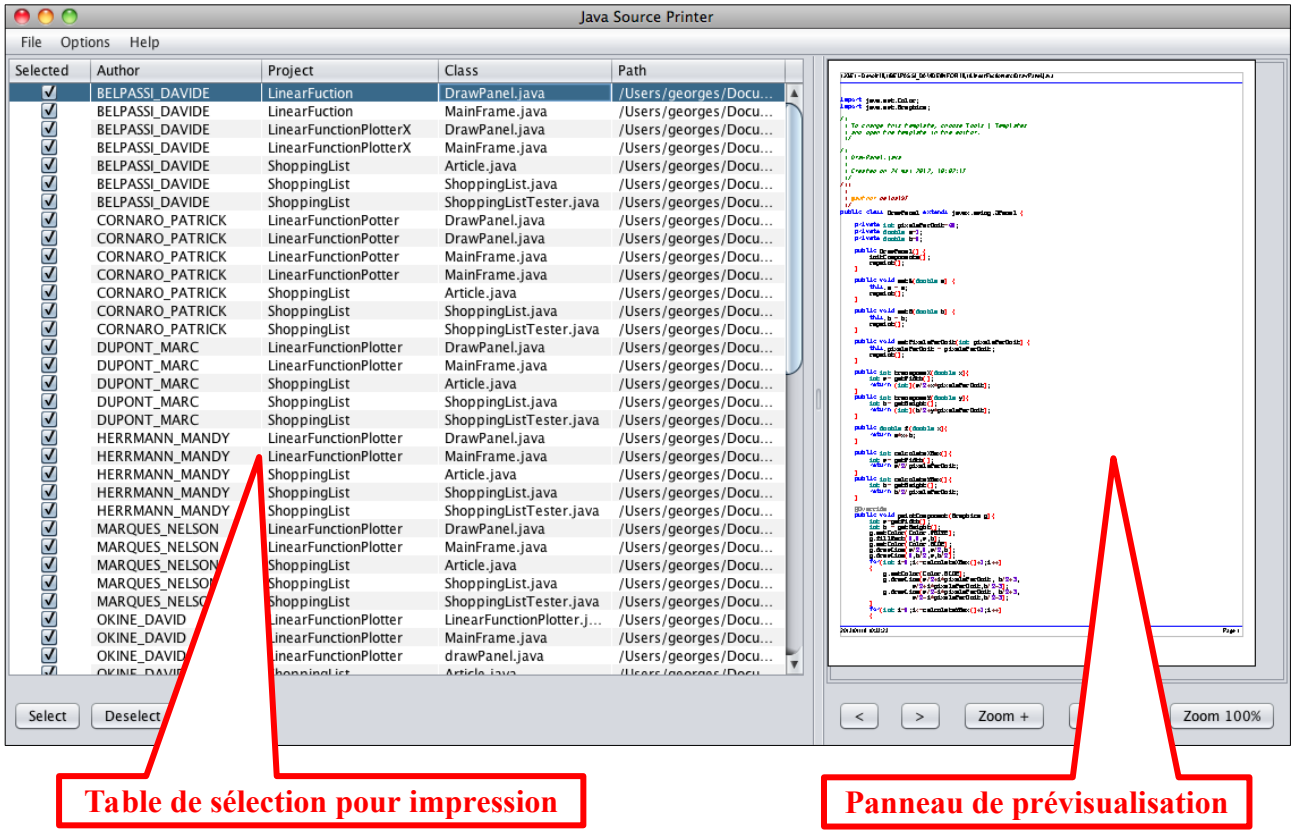
Le logiciel **JavaSourcePrinter** peut être lancé par Webstart sur le site **java.cnpi.lu**.

Le logiciel a été conçu pour imprimer en bloc le code source Java de l'ensemble des projets contenus dans un dossier, p.ex. un enseignant qui veut imprimer tous les projets de ses élèves en une fois. Il est aussi très pratique pour un élève qui veut imprimer l'un ou plusieurs de ses projets en évitant des pages superflues. Le logiciel fonctionne de manière optimale si le dossier est structuré de la manière suivante: → →



Sélection du dossier de base

Après le démarrage du programme, sélectionnez le dossier de base (menu: ***File*** ➔ ***Select Directory...***).



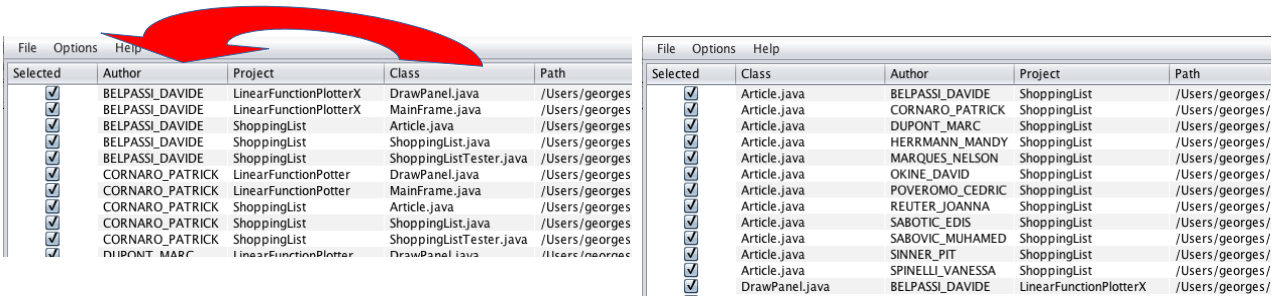
Sélection des classes pour impression

Dans la table de sélection du panneau gauche se trouvent toutes les classes Java du dossier de base. Les champs affichés dans la table sont:

- le champ de sélection (***Selected***)
- le nom du dossier de l'élève (***Author***)
- le nom du projet (***Project***)
- le nom de la classe (***Class***)
- le chemin complet de la classe (***Path***)

Vous pouvez y sélectionner ou désélectionner les classes à imprimer en cliquant directement dans le champs de sélection ou en sélectionnant avec la souris les lignes concernées et en cliquant sur les boutons ***Select*** ou ***Deselect*** (pour sélectionner toutes les lignes de la table, utilisez le raccourci clavier **CTRL-A**).

Pour faciliter la sélection ou désélection de certaines classes il est souvent commode de changer l'ordre de tri (par exemple: si vous voulez déselectionner une certaine classe il est utile de trier par nom de classe). Vous pouvez changer l'ordre de tri en changeant l'ordre des colonnes dans la table en les glissant à l'aide de la souris. L'ordre des champs dans la table correspond à l'ordre de tri.



Prévisualisation de l'impression d'une classe

L'impression d'une classe peut être prévisualisée dans le panneau de prévisualisation en sélectionnant la ligne correspondante dans la table.

Impression des classes

Le point menu ***File → Print all selected...*** démarre l'impression de toutes les classes sélectionnées dans la table. Elles sont imprimées dans le même ordre qu'elles figurent dans la table.

Options de filtrage

Les options de filtrage permettent de spécifier les lignes du code à ne pas imprimer (menu: ***Options → Filter Settings...***) :

- ***JavaDoc***
  Les commentaires JavaDoc ne sont pas imprimés
- ***Comments***
  Les commentaires ne sont pas imprimés.
- ***Double blank line***
  Dans le cas de plusieurs lignes vides consécutives, une seule est imprimée.

- ***Netbeans generated code - Attribute declarations of components***
  Les déclarations des attributs correspondant à des composants créés en mode « Design » (par exemple: les labels, boutons, … sur les fiches de type JFrame et Jpanel), qui ont été générés automatiquement par Netbeans, ne sont pas imprimés.
- ***Netbeans generated code – Other***
  Tout autre code généré par Netbeans n'est pas imprimé.

Options d'impression

Menu: ***Options*** → ***Print Settings...*** :

- ***Font size***
  Taille de la police.
- ***Tab size***
  Quantité d'espaces blancs correspondant à une tabulation.
- ***Monochrome print***
  Impression en noir et blanc (si vous n'aimez pas les nuances de gris dans le cas d'une impression avec une imprimante noir et blanc).
- ***Relative path to base directory in header***
  Affiche seulement le chemin de l'emplacement des classes à partir du dossier de base (sinon le chemin complet) dans le haut de page.
- ***Print (original) line numbers***
  Impression des numéros de lignes originales (avant filtrage de code).
- ***Show date in footer*** et ***Date format***
  Impression de la date dans le bas de page et format de la date.
- ***Page break must be a multiple of … at level***
  Cette option est utile dans le cas où vous voulez imprimer en recto-verso et/ou imprimer un multiple de pages sur une seule page de papier. Cette option permet alors de gérer les sauts de page (physiques).

  Exemple:
  Vous voulez imprimer un multiple de 2 pages en recto-verso (donc 4 pages par feuille de papier). En outre, vous voulez éviter que le code d'élèves différents soit imprimé sur une même feuille de papier. Les classes sont triées d'abord par élève (niveau 1), ensuite par projet (niveau 2) et finalement par classe (niveau 3).
  Dans cette situation vous choisissez l'option:
  Page break must be a multiple of 4 at level 1

10.2. Impression en NetBeans

Bien que nous conseillions l'emploi de« JavaSourcePrinter », voici une description de l'impression avec NetBeans (même si le résultat n'est pas toujours satisfaisant). Les réglages suivants sont mémorisés en NetBeans, il suffit donc de les effectuer une seule fois pour chaque machine où vous utilisez NetBeans.

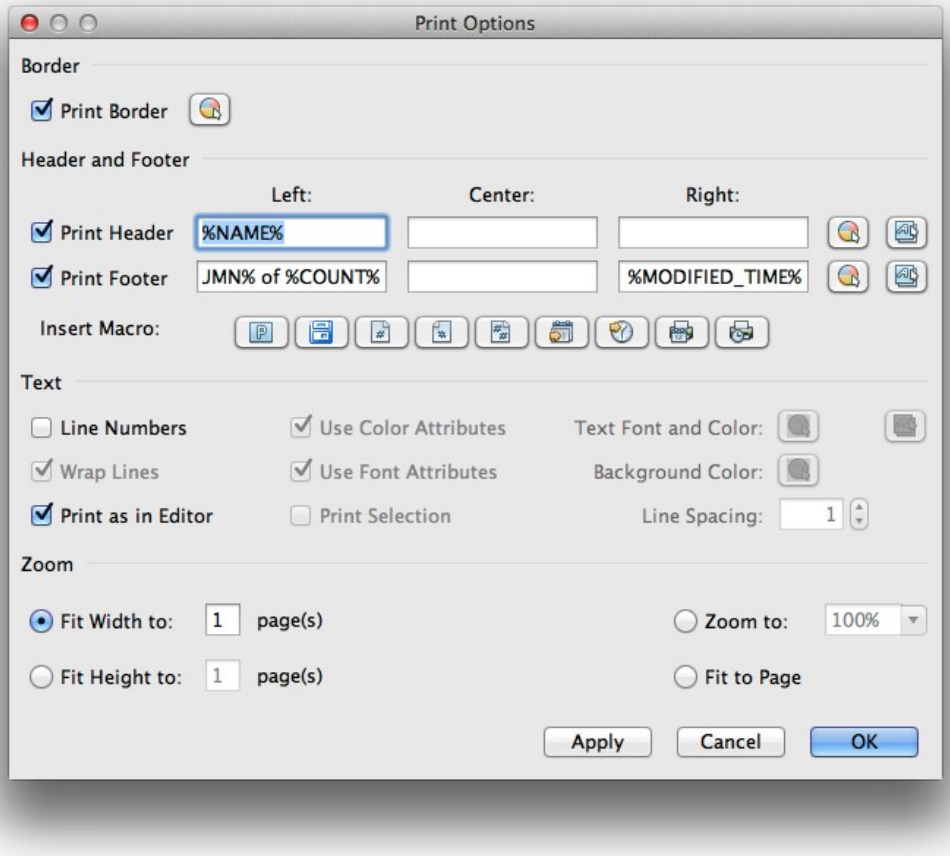
Cliquez d'abord sur **File → Print** et vous vous retrouvez dans le mode '**Print Preview**' qui vous permet de vérifier la disposition des pages.

Dans '**Page Setup**', il peut être utile choisir le mode **'Landscape'/'Paysage'** de votre imprimante.

Dans **'Print Options'** activez les options suivantes :

- ☑ **Fit Width to 1 page(s)** dans la rubrique 'Zoom' en bas du dialogue,
- ☑ **Wrap Lines** dans la rubrique 'Text',
- ☑ **Print as in Editor** dans la rubrique 'Text', pour éviter l'impression du code généré.

Le dialogue devrait se présenter à peu près comme suit :



Cliquez ensuite sur **Apply** pour vérifier dans **Print Preview**.

Confirmez par **OK** puis cliquez sur **Print** et vous vous retrouvez dans votre dialogue d'impression usuel.

11. Annexe B - Assistance et confort en NetBeans

NetBeans propose un grand nombre d'automatismes qui nous aident à gérer notre code et à l'entrer de façon plus confortable. En voici les plus utiles [9]:

11.1. Suggestions automatiques :

Même sans action supplémentaire de notre part, NetBeans nous suggère automatiquement des méthodes, attributs, paramètres,... Nous n'avons qu'à les accepter en poussant sur <Enter>. NetBeans choisit ces propositions d'après certains critères (syntaxe, types, objets définis...) qui correspondent à la situation actuelle.

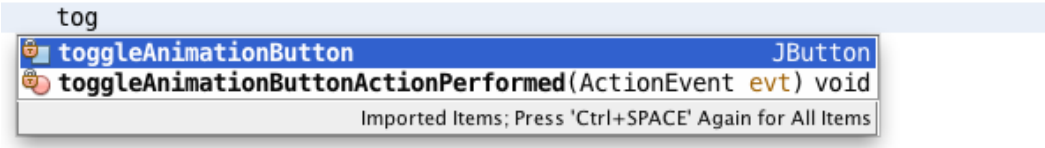
Mais attention : ces suggestions ne correspondent pas toujours à nos intentions ou besoins. Ceci compte surtout pour les valeurs des paramètres que NetBeans propose...

11.2. Compléter le code par <Ctrl>+<Space>

Il peut sembler fastidieux de devoir entrer des noms longs comme p.ex. les noms des composants avec suffixes, mais NetBeans (tout comme Unimozer) vous assiste en complétant automatiquement un nom que vous avez commencé à entrer si vous tapez **<Ctrl>+<Space>**.

En général il suffit d'entrer les deux ou 3 premières lettres d'un composant, d'une variable, d'une méthode ou d'une propriété et de taper sur **<Ctrl>+<Space>** pour que NetBeans complète le nom ou vous propose dans une liste tous les noms de votre programme qui correspondent.

Exemple : Dans ce programme qui contient un bouton ***toggleAnimationButton*** NetBeans propose ce nom après avoir entré '*tog*' suivi de **<Ctrl>+<Space>**.



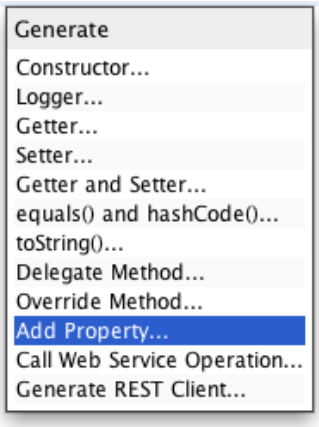
11.3. Ajout de propriétés et de méthodes <Insert Code...>

La définition d'une classe commence en général par un certain nombre de tâches monotones et répétitives (définition de propriétés, accesseurs, modificateurs, constructeurs, JavaDoc, méthode toString(), etc.).

Ces actions sont assistées par NetBeans, en faisant un clic droit de la souris à l'endroit où on veut insérer le code puis en saisissant le menu <**Insert Code...**>

raccourci en Windows : **<Alt> + <Insert>**

raccourci sur MacOSX: **<Ctrl>+<I>**



9 Voir aussi: *https://netbeans.org/kb/docs/java/editor-codereference.html*

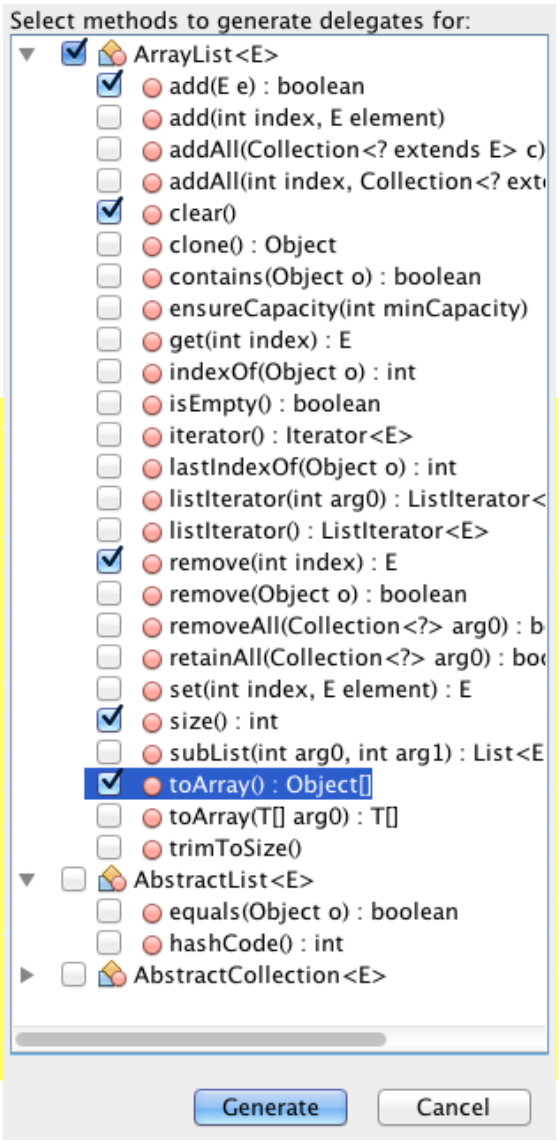
Le nombre d'options dans le menu <**Insert Code...**> dépend de l'état actuel de votre classe. Il est recommandé de commencer par définir tous les attributs (**<Add Properties...>**) avec leurs accesseurs et modificateurs (si nécessaire) et de définir ensuite le constructeur et les méthodes **toString.**

11.4. Rendre publiques les méthodes d'un attribut privé

Si votre classe contient un attribut privé dont vous voulez rendre accessible quelques méthodes à d'autres objets, il est très pratique d'employer l'option **<Delegate Method...>** du menu <**Insert Code...**>.

Exemple typique :

Imaginez que votre classe contienne une **ArrayList allLines** comme attribut et que vous vouliez rendre publiques les méthodes **add**, **clear**, **remove**, **size** et **toArray.** L'option **<Delegate Method...>** vous propose alors d'insérer toutes ces méthodes en les cochant simplement dans la liste.



En cliquant sur **<Generate>** vous obtiendrez alors le code suivant :

```
    public int size() {
        return allLines.size();
    }

    public Object[] toArray() {
        return allLines.toArray();
    }

    public boolean add(Line e) {
        return allLines.add(e);
    }

    public Line remove(int index) {
        return allLines.remove(index);
    }

    public void clear() {
        allLines.clear();
    }
```

11.5. Insertion de code par raccourcis

Pour des blocs d'instruction utilisés fréquemment il est possible de définir des raccourcis. NetBeans se charge d'étendre ces raccourcis dans les blocs définis dès que nous entrons le raccourci suivi de <**Tab**>.

Il n'est même pas nécessaire de définir ces blocs nous mêmes, puisque NetBeans connaît déjà la plupart des blocs les plus usuels.

Vous pouvez consulter la liste des raccourcis prédéfinis dans le menu des préférences sous **<Editor>** → **<Code Templates>**.

Essayez par exemple les codes suivants (vous allez être surpris :-) :

sout<Tab>	**ife**<Tab>	**wh**<Tab>
for<Tab>	**forl**<Tab>	

Remarque : L'extension des raccourcis fonctionne uniquement si vous avez entré le code en une fois sans le corriger.

11.6. Formatage du code

Entretemps, vous voyez certainement l'intérêt à endenter votre code (c.-à-d. écrire les blocs de code en retrait) pour qu'il soit bien structuré et bien lisible. Si quand même vous devez changer l'indentation de code déjà existant, il est recommandé, de le sélectionner, puis d'utiliser **<Tab>** ou **<Shift>+<Tab>** pour l'endenter ou le désendenter en une seule fois.

Vous pouvez aussi employer les boutons : 

Vous pouvez faire formater un bloc de code simplement en employant le **formatage automatique** :

Sélectionner le bloc à formater, puis :	<*Clic droit*>+Option **Format**	
ou bien enfoncer les touches :	**<Alt>+<Shift>+F**	*(sous Windows)*
	<Ctrl>+<Shift>+F	*(sous Mac-OSX)*

Conseil : Appliquez ce formatage seulement, si la structure de votre classe est correcte et le code se laisse compiler sans erreurs.

11.7. Activer/Désactiver un bloc de code

Il est parfois utile de mettre tout un bloc d'instructions entre commentaires pour le désactiver (p.ex. parce qu'il contient une erreur dont on ne trouve pas la cause ; ou parce qu'on a trouvé une meilleure solution, mais on veut garder le code de l'ancienne version pour le cas de besoin). En principe, il est recommandé de mettre le bloc entre commentaires **/* ... */**. mais il est souvent plus rapide de sélectionner le bloc et d'utiliser les boutons d'activation et de désactivation de NetBeans, qui placent ou suppriment des commentaires devant toutes les lignes sélectionnées :