

# ***2**science de la programmation*



Java™



## Table des matières

<b>1. Dessin et graphisme.....</b>	<b>6</b>
1.1. Le canevas « Graphics ».....	7
1.1.1. La géométrie du canevas.....	7
1.1.2. Les méthodes du canevas.....	8
1.1.3. La méthode repaint().....	8
1.1.4. Affichage et alignement du texte.....	9
1.2. La classe « Color ».....	10
1.2.1. Constructeur.....	10
1.2.2. Constantes.....	11
1.3. La classe « Point ».....	11
1.3.1. Constructeur.....	11
1.3.2. Attributs.....	11
1.3.3. Méthodes.....	11
1.4. Les dessins et le modèle MVC.....	12
<b>2. Le temps et la date.....</b>	<b>15</b>
2.1. Mesurer le temps écoulé.....	15
2.2. Les fonctions du calendrier.....	16
2.3. Le chronomètre.....	17
2.3.1. Fonctionnement.....	17
2.3.2. La classe « Timer ».....	17
2.4. Proposes d'implémentation.....	18
2.4.1. Méthode dite « par bouton caché ».....	18
2.4.2. Méthode dite « par classe anonyme ».....	18
2.4.3. Méthode par expression lambda.....	19
<b>3. Les événements de la souris.....</b>	<b>20</b>
3.1. Types d'événements.....	20
Pour avancés : .....	20
3.2. Méthodes de réaction.....	21
3.3. La classe « MouseEvent ».....	21
<b>4. Les quatre principes de la POO.....</b>	<b>22</b>
4.1. Abstraction.....	22
4.2. Encapsulation.....	22
4.2.1. Avantages.....	23
4.2.2. Niveaux d'accessibilité.....	23
4.3. Agrégation.....	25

4.4. Héritage.....	28
4.4.1. Le mot clé "extends".....	29
4.4.2. Exercice résolu.....	29
4.4.3. Redéfinition de méthodes.....	32
4.4.4. Accès aux éléments de la super-classe - <i>super</i> .....	33
4.4.5. Redéfinition du constructeur.....	34
4.4.6. Classes abstraites.....	35
4.5. Polymorphisme.....	35
4.5.1. Polymorphisme et redéfinition de méthodes ( <i>late binding</i> ).....	37
4.5.2. Méthodes abstraites.....	39
4.6. L'opérateur « instanceof ».....	40
<b>5. Modificateurs <i>static</i> et <i>final</i>.....</b>	<b>41</b>
5.1. Les éléments statiques - <i>static</i> .....	41
5.2. Les éléments constants - <i>final</i> .....	44
5.2.1. Définition de constantes (attributs constants).....	44
5.2.2. Pour avancés : Paramètres et variables locales constants.....	45
5.2.3. Pour avancés : Classes non dérivables.....	45
5.2.4. Pour avancés : Méthodes non redéfinissables.....	45
<b>6. Arrays - listes statiques.....</b>	<b>46</b>
<b>7. Gestion des paquets.....</b>	<b>48</b>
<b>8. Exceptions.....</b>	<b>50</b>
8.1. Générer une exception.....	50
8.2. Intercepter une exception.....	51
8.3. Plusieurs exceptions dans le même bloc.....	52
8.4. Le principe "catch-or-throw" & l'instruction "throws".....	53
8.5. Les classes RuntimeException et Error.....	54
8.6. Exceptions dans le constructeur.....	55
8.7. Java 7.....	55
8.8. Pour avancés : Le bloc finally.....	56
<b>9. Fichiers.....</b>	<b>57</b>
9.1. Les fichiers textes.....	58
9.1.1. Ecriture dans un fichier texte.....	58
9.1.2. Lecture dans un fichier texte.....	59
9.1.3. Appel des méthodes de sauvegarde et de lecture.....	60
9.2. Lecture et sauvegarde de couleurs.....	60
9.3. Ajout à la fin d'un fichier ("Append").....	60
9.4. Sauvegarde de plusieurs données par ligne.....	62

9.5. Remarques supplémentaires pour avancés.....	63
9.6. Les fichiers XML (pour avancés).....	64
9.6.1. Les analyseurs syntactiques.....	65
9.6.2. La classe XStream.....	66
9.7. Localisation des fichiers.....	68
9.7.1. Dialogues d'ouverture et de sauvegarde.....	68
9.7.2. Définition d'un filtre de fichiers :.....	69
9.8. Opérations sur fichiers et chemins d'accès.....	70
9.9. Fichiers de ressources dans un projet (pour avancés).....	71
9.10. Charger au démarrage et sauvegarder à la fin.....	72
<b>10. Annexe A - Applications Java sur Internet.....</b>	<b>73</b>
10.1. Java Applet.....	73
10.2. Java Web Start Application.....	77
<b>11. Annexe B - Impression de code NetBeans.....</b>	<b>78</b>
11.1. Impression à l'aide du logiciel « JavaSourcePrinter ».....	78
11.2. Impression en NetBeans.....	81
<b>12. Annexe C - Assistance et confort en NetBeans.....</b>	<b>82</b>
12.1. Suggestions automatiques :.....	82
12.2. Compléter le code par <Ctrl>+<Space>.....	82
12.3. Ajout de propriétés et de méthodes <Insert Code...>.....	82
12.4. Rendre publiques les méthodes d'un attribut privé.....	83
12.5. Insertion de code par raccourcis.....	84
12.6. Formatage du code.....	84
12.7. Activer/Désactiver un bloc de code.....	84
<b>13. Annexe D - Accès à JavaDoc sur le disque local.....</b>	<b>86</b>
13.1. Télécharger la documentation.....	86
13.2. Instruire NetBeans à utiliser le fichier JavaDoc local.....	86
<b>14. Annexe E - Live Debugging.....</b>	<b>87</b>

## Sources

« *Introduction à la programmation orientée objets - classes GTG* » du groupe de travail PrograTG de la Commission Nationale des Programmes d'Informatique – ESG.

« *Introduction à la programmation orientée objets - classes GIN* » par Fred Faber

Le cours est adapté et complété en permanence.

## Rédacteurs

- Fred Faber
- Robert Fisch

## Site de référence

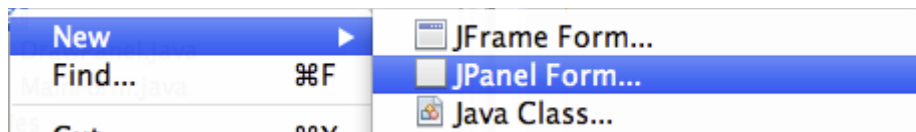
Des documents supplémentaires sont accessibles sur le site :

<http://java.cnpi.lu>

# 1. 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 :

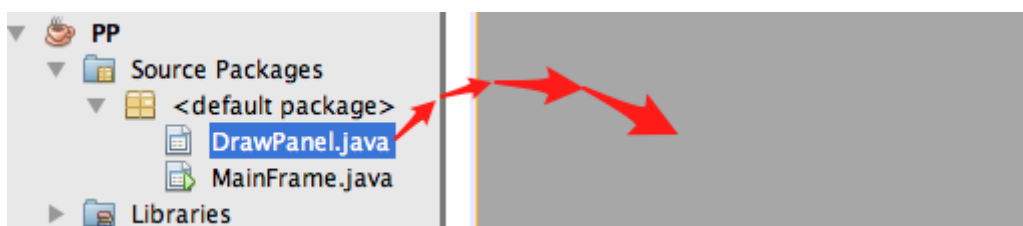
1. Ajoutez un nouveau **JPanel Form** à votre projet et nommez-le **DrawPanel**.
2. Cette nouvelle classe possède une vue « source » et une vue « design » (comme un « JFrame »). En principe, nous n'allons jamais placer d'autres composants sur un **JPanel Form**. Activez la vue « source » et ajoutez la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

## 1.1. Le canevas « Graphics »

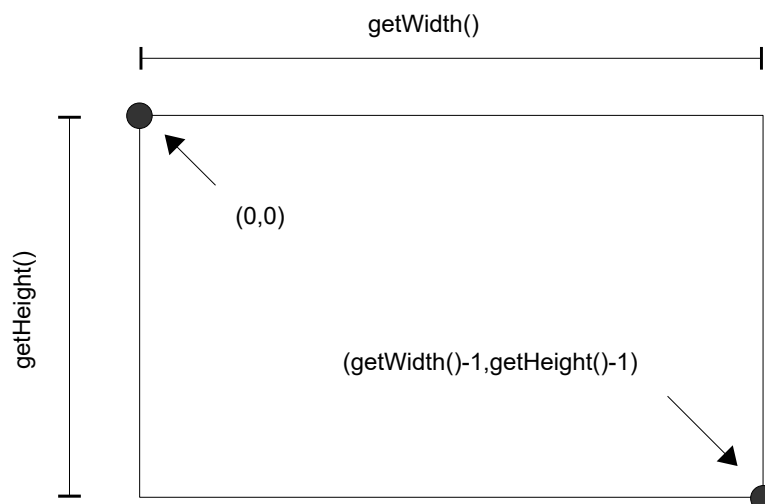
Un artiste peintre dessine sur un canevas (DE : *die Leinwand*). En informatique l'image de l'écran est aussi appelée canevas et elle consiste en une grille composée de pixels minuscules, des points qui peuvent prendre des millions de couleurs différentes.

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.

### 1.1.1. 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.



### 1.1.2. Les méthodes du canevas

Méthode	Description
<code>Color getColor()</code> <code>void setColor(Color)</code>	Permet de récupérer la couleur actuelle, et d'en choisir une nouvelle.
<code>void drawLine(int x1, int y1,                   int x2, int y2)</code>	Dessine à l'aide de la couleur actuelle une ligne droite entre les points <b>(x1,y1)</b> et <b>(x2,y2)</b> .
<code>void drawRect(int x, int y,                   int width, int height)</code>  <code>void fillRect(int x, int y,                   int width, int height)</code>	Dessine à l'aide de la couleur actuelle, un rectangle tel que <b>(x,y)</b> représente le point supérieur gauche, tandis que <b>width</b> et <b>height</b> représentent sa largeur respectivement sa hauteur.
<code>void drawOval(int x, int y,                   int width, int height)</code>  <code>void fillOval(int x, int y,                   int width, int height)</code>	Dessine à l'aide de la couleur actuelle, une ellipse telle que <b>(x,y)</b> représente le point supérieur gauche, tandis que <b>width</b> et <b>height</b> représentent sa largeur respectivement sa hauteur.
<code>void drawString(String s, int x, int y)</code>	Dessine le texte <b>s</b> à la position <b>(x,y)</b> tel que <b>(x,y)</b> 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);`

### 1.1.3. 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 :

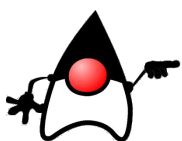
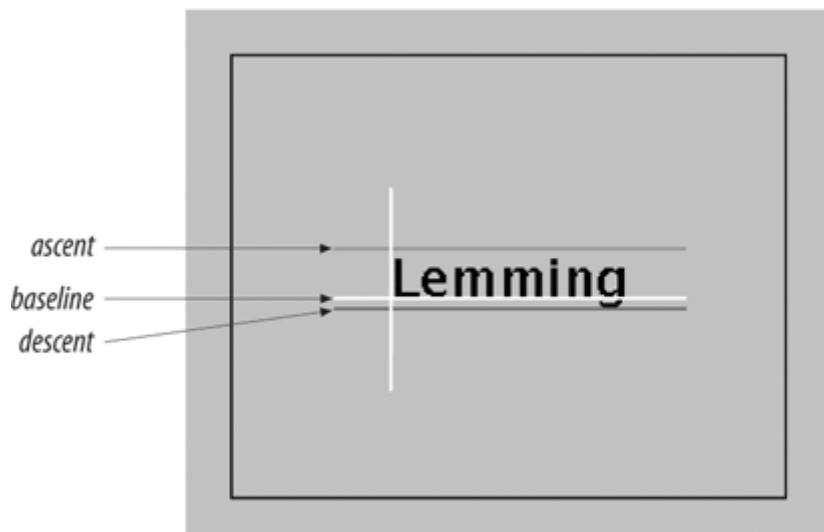
```
/**
 * Méthode qui dessine une _____
 */
public void paintComponent(Graphics g)
{
    g.setColor(Color.RED);
    g.drawLine(0,0,getWidth(),getHeight());
    g.drawLine(0,getHeight(),getWidth(),0);
}
```

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 1.2).

**1.1.4. 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>

## 1.2. 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**'.

### 1.2.1. 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
<code>Color(int red, int green, int blue)</code>	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].
<code>Color(int red, int green, int blue, int alpha)</code>	comme ci-dessus, mais le paramètre <b>alpha</b> 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 \_\_\_\_\_

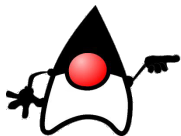
`Color color7 = new Color(0,255,0,64);`      crée une couleur \_\_\_\_\_

### 1.2.2. 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 Unimozzer, 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>

### 1.3. 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.

#### 1.3.1. Constructeur

Constructeur	Description
<b>Point(int x, int y)</b>	Crée un nouveau point avec les coordonnées (x,y).

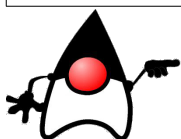
#### 1.3.2. Attributs

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

Attributs	Description
<b>int x , int y</b>	Les coordonnées (x,y) du point dans le plan.

#### 1.3.3. Méthodes

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

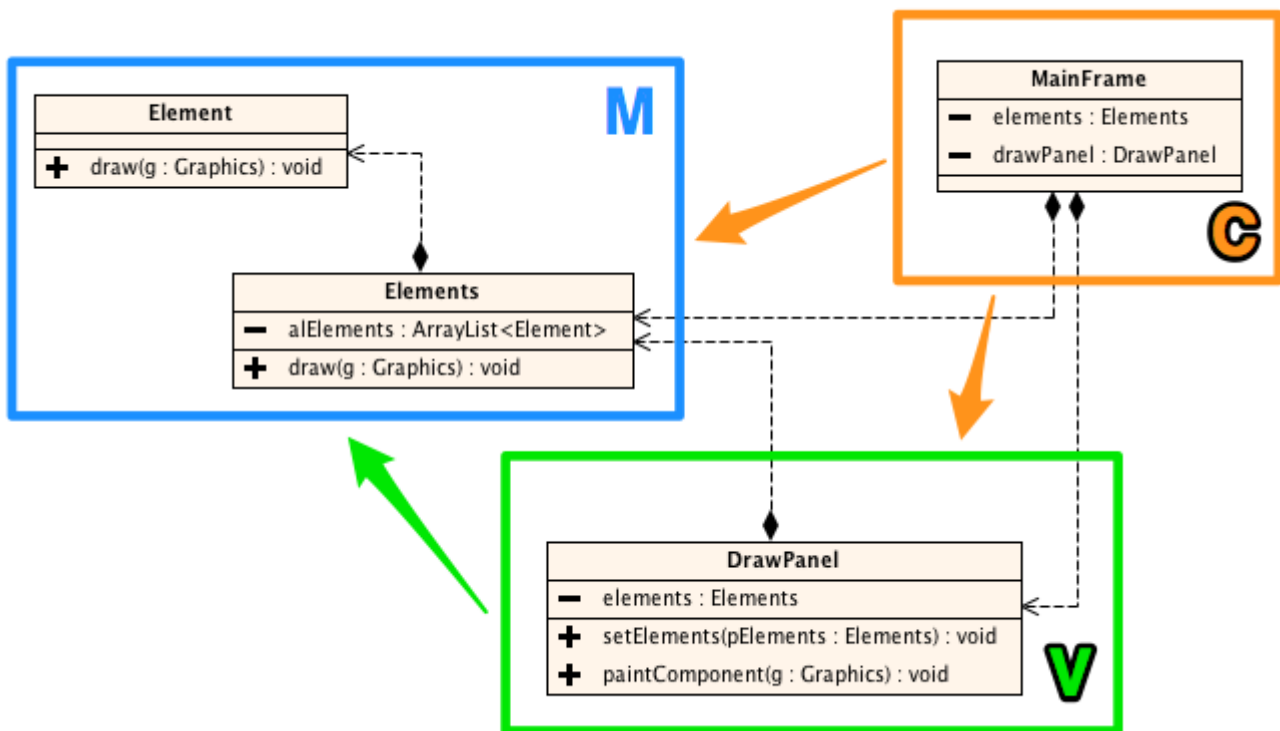


Voici le lien direct vers la page JavaDoc :

<http://download.oracle.com/javase/8/docs/api/java/awt/Point.html>

### 1.4. 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<sup>1</sup>. 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.

<sup>1</sup> 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);  
    }  
}
```

## 2. Le temps et la date

### 2.1. Mesurer le temps écoulé

Parfois il est utile de mesurer le temps entre deux points précis d'un programme. Java procure un plusieurs possibilités, et nous allons en discuter les plus utiles.

#### **long System.nanoTime()**

retourne le temps actuel en nanosecondes (ns). Il est à noter que la précision n'est pas nécessairement donnée à une nanoseconde près, puisqu'elle dépend de l'horloge interne de l'ordinateur. Aucune garantie ne peut être donnée quant à la fréquence de changement de **.nanoTime()**. La valeur de **.nanoTime()** n'est pas donnée par rapport à une date ou un temps précis, elle indique le temps écoulé depuis le démarrage de la JVM. Elle peut seulement être employée pour calculer le temps écoulé (différences de deux appels de la méthode).

#### Utilisation :

```
long startTime ;
long stopTime ;
startTime = System.nanoTime() ;
//action à tester
...
stopTime = System.nanoTime() ;
timeLabel.setText((stopTime-startTime)/1000000000.0 + "s");
```

## 2.2. Les fonctions du calendrier

Les classes `LocalDate`, `LocalTime`, `LocalDateTime` du paquet `java.time` nous fournissent la date et le temps actuel. L'appel `LocalDateTime.now()` nous fournit une instance de la classe `LocalDateTime` initialisée au temps et à la date actuelle en considérant la zone temporaire de votre ordinateur.

Par la ligne suivante, nous obtenons une représentation préformatée de la date et du temps actuels :

```
System.out.println(LocalDateTime.now());
```

P.ex : **2016-10-13T21:05:54.864**

Pour pouvoir formater ces informations individuellement, nous pouvons accéder à une instance de la classe `LocalDateTime` et lui demander les informations qui nous intéressent une à une. Pour cela, nous employons ici des méthodes prédéfinies de la classe `LocalDateTime`.

Exemple :

```
LocalDateTime now = LocalDateTime.now();
int year      = now.getYear();
int month     = now.getMonthValue();
int day       = now.getDayOfMonth();
int hour      = now.getHour();
int min       = now.getMinute();
int sec       = now.getSecond();
int nano      = now.getNano();
System.out.println("Date : "+day+"-"+month+"-"+year);
System.out.println("Time : "+hour+": "+min+": "+sec+" (" + nano/1000000 + "ms)");
```

P.ex : **Date : 13-10-2016**  
**Time : 21:05:54 (864ms)**

Remarques :

- Le paquet `java.time` qui a été introduit avec Java 8 contient une foule de classes et de méthodes pour travailler avec les dates et les temps. Celles-ci remplacent les anciennes classes `Calendar`, `Date`, `TimeZone` du paquet `java.util` qui sont dorénavant déconseillées. Une présentation de toutes les fonctionnalités de `java.time` remplirait des dizaines de pages. Nous allons nous limiter ici aux fonctions de base nommées ci-dessus et énumérer les classes les plus intéressantes :
  - Il existe des méthodes et des classes pour représenter les dates sous forme préformatée selon les formats usuels de différents pays (`DateTimeFormatter`, `FormatStyle`, `Locale`).
  - Les classes `ZoneDateTime`, `ZoneId`, `ZoneOffset` permettent de travailler avec des dates et des temps de différents fuseaux horaires.



## 2.3. Le chronomètre

### 2.3.1. Fonctionnement

Un chronomètre est un objet qui exécute périodiquement une méthode donnée, c'est-à-dire qui exécute des actions à intervalles réguliers. Entre deux appels consécutifs à cette méthode s'écoule toujours un temps donné fixe.

#### **Exemple**

Voici l'axe du temps représentant l'exécution d'un chronomètre qui est déclenché chaque seconde :



Le trait orange représente le temps d'exécution de la méthode que le chronomètre exécute périodiquement.

### 2.3.2. La classe « Timer »

La classe **Timer** fait partie du paquet **javax.swing**. De ce fait, il faut indiquer en haut d'une classe utilisant un **Timer** la ligne d'importation correspondante :

```
import javax.swing.Timer;
```

La classe **Timer** dispose de plusieurs méthodes, dont voici celles qui nous intéressent le plus :

Méthodes	Description
<b>void start()</b>	Démarre le chronomètre.
<b>void stop()</b>	Arrête le chronomètre.
<b>boolean isRunning()</b>	Détermine si le chronomètre est actif ou non.
<b>void setDelay(int)</b>	Permet de modifier l'intervalle d'exécution du chronomètre. Le paramètre indique en temps en « millisecondes ».
<b>int getDelay()</b>	Permet de lire l'intervalle d'exécution du chronomètre. Le paramètre indique en temps en « millisecondes ».

## 2.4. Proposes d'implémentation

Il existe plusieurs méthodes ou « recettes », suivant lesquelles on peut mettre en œuvre un chronomètre. La méthode dite « par bouton caché » est la plus simple et la méthode préférée dans notre cours. La deuxième méthode est à considérer comme une information supplémentaire pour ceux qui désirent approfondir la matière.

### 2.4.1. Méthode dite « par bouton caché »

1. Ajoutez un bouton à votre fiche. Nous appelons ce bouton **stepButton** parce que chaque clic sur le bouton effectue un pas des opérations (EN : step). Plus tard, le chronomètre va exécuter ces pas automatiquement et périodiquement.
2. Développez la méthode de réaction du bouton **stepButton** et testez-la !

(Pour tester, on peut cliquer régulièrement sur ce bouton, pour simuler l'action du chronomètre.)

3. Créez un nouveau chronomètre **timer** (l'instance **timer** peut être un attribut ou une variable du type **Timer**).

Le premier paramètre du constructeur indique la **période** de la répétition **en millisecondes**. (Ici : périodicité 500ms → activation 2 fois par seconde).

Le second paramètre indique au chronomètre d'exécuter la 1ère méthode de réaction liée au bouton **stepButton** :

```
timer = new Timer( 500 , stepButton.getActionListeners()[0] );
```

4. Démarrer le chronomètre :

```
timer.start();
```

5. Finalement on peut rendre invisible le bouton :

```
stepButton.setVisible(false);
```

Dans l'exemple donné, on crée donc un chronomètre qui exécute toutes les demi secondes le code attaché à la méthode de réaction du bouton **stepButton**.

### 2.4.2. Méthode dite « par classe anonyme »

Pour cette méthode, il n'y a pas besoin de créer un bouton. Par contre, le code à écrire pour la création d'un chronomètre est nettement plus complexe :

```
timer = new Timer( 500, new ActionListener() {  
    public void actionPerformed(ActionEvent e)  
    {  
        // code à exécuter périodiquement  
    }  
});  
timer.start();
```

Tapez <Ctrl><Espace> après le mot **ActionListener** afin d'activer la complétion automatique de code ...

### 2.4.3. Méthode par expression lambda

Java 8 a introduit la possibilité de définir des "expressions lambda". Nous pouvons utiliser cette technique pour alléger davantage la définition d'une réaction au chronomètre :

```
timer = new Timer( 500, event -> {  
// code à exécuter périodiquement  
});  
timer.start();
```

Si vous avez défini une réaction par classe anonyme, NetBeans vous propose automatiquement de transformer la définition en une expression lambda. ]

### 3. Les événements de la souris

Par **événements souris** on entend le fait qu'un programme réagisse sur un clic de la souris respectivement sur son déplacement.

Le présent chapitre n'expliquera pas en détail le fonctionnement des événements souris mais se concentre uniquement sur leur utilisation tel que le propose l'éditeur NetBeans.

Les événements souris existent pour la plupart des composants qu'on peut ajouter sur une fenêtre, comme par exemple les boutons (**JButton**), les champs d'édition (**JTextField**) et les panneaux (**JPanel**). Tous les événements souris sont définis dans la classe **MouseEvent**.

#### 3.1. Types d'événements

Dépendant du composant sélectionné, l'éditeur de propriété affiche un certain nombre d'événement relatifs à l'utilisation de la souris. Il s'agit de tous les événements préfixés avec le mot « mouse ». Malgré le fait qu'il existe plusieurs événements, le présent cours n'en traite que trois :

- **mousePressed**

Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question et **enfonce** l'un des boutons de la souris.

- **mouseReleased**

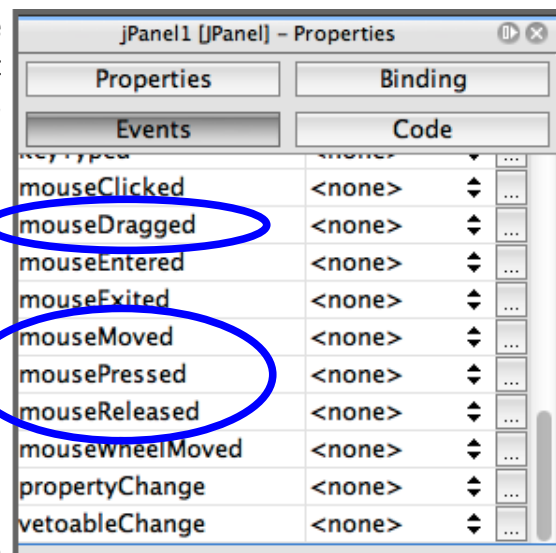
Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question et **relâche** l'un des boutons de la souris.

- **mouseDragged**

Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question, a **enfoncé** un bouton de la souris **et déplace** celle-ci.

- **mouseMoved**

Cet événement est déclenché lorsque l'utilisateur pointe avec le curseur de la souris sur le composant en question, **et déplace** la souris **sans avoir enfoncé** un bouton.



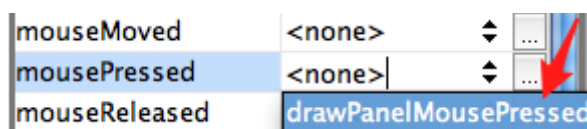
#### Pour avancés :

Pour pouvoir réagir aux événements **mousePressed** et **mouseReleased**, la classe doit implémenter l'interface **MouseListener**.

Pour pouvoir réagir aux événements **mouseMoved** et **mouseDragged**, la classe doit implémenter l'interface **MouseMotionListener**.

### 3.2. Méthodes de réaction

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 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.

Pour le composant **drawPanel** et l'événement **mousePressed**, la méthode de réaction, telle que générée par NetBeans, est la suivante :

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

Afin de pouvoir réagir à un tel événement, il est essentiel de disposer de certaines informations relatives à la souris, comme par exemple :

- la position de la souris ou
- le bouton enfoncé ou relâché.

Ces informations sont passées à la méthode de réaction via le paramètre **evt** qui est du type **MouseEvent**. Ce paramètre est présent pour tous les trois types d'événements souris traités dans ce cours.

### 3.3. La classe « MouseEvent »

Voici les méthodes les plus importantes de la classe **MouseEvent**<sup>2</sup>.

Méthodes	Description
<b>int getX()</b> <b>int getY()</b>	Retournent les coordonnées <b>x</b> ou <b>y</b> du point auquel se trouvait le curseur de la souris lorsque l'événement a été déclenché.
<b>Point getPoint()</b>	Retourne le point où se trouvait le curseur de la souris lorsque l'événement a été déclenché.
<b>int getButton()</b>	Indique quel bouton de la souris a été enfoncé ou relâché. Voici les valeurs les plus courantes : <b>MouseEvent.BUTTON1</b> , <b>MouseEvent.BUTTON2</b> et <b>MouseEvent.BUTTON3</b> .

<sup>2</sup> <http://docs.oracle.com/javase/6/docs/api/java/awt/event/MouseEvent.html>

## 4. Les quatre principes de la POO

Maintenant que nous connaissons les techniques de base de la programmation orientée objet (POO) et de la programmation graphique en Java, il est temps de nous occuper plus en détail de l'organisation et de la structuration des éléments à l'intérieur des classes et des classes entre elles. Nous allons d'abord revenir sur quelques points que vous connaissez déjà pour les approfondir. Ensuite, nous allons ajouter des notions qui auront un effet massif sur la manière dont vous pouvez organiser votre application. En fait du point de vue technique, il n'y aura pas beaucoup de nouveautés à 'apprendre' ( le plus grand bond en avant sera causé par un seul mot clé : 'extends' ), mais vous aurez la possibilité de voir la programmation à un niveau plus élevé, plus conceptionnel.

Résumons d'abord : la POO est basée sur quatre principes fondamentaux que nous allons éclairer dans ce chapitre :

1. Abstraction
2. Encapsulation
3. Héritage
4. Polymorphisme

### 4.1. Abstraction

**L'abstraction** est le fait d'ignorer les propriétés marginales et de se concentrer sur les aspects essentiels dans le cadre du projet à réaliser. C'est un principe essentiel pour pouvoir gérer la complexité du monde réel dans le domaine informatique.

En informatique, l'abstraction nous permet de ne considérer que les caractéristiques les plus importantes des objets que nous manipulons. Ce qui est important dépend du projet que nous sommes en train de réaliser. Lorsque nous devons réaliser une classe **Personne**, nous allons inclure des attributs pour le nom et le prénom des personnes, mais probablement pas le groupe sanguin, qui est alors victime de l'abstraction. Si par contre nous réalisons une application pour les patients d'une clinique, le groupe sanguin serait un attribut très important à inclure dans la classe.

### 4.2. Encapsulation

**L'encapsulation** est le fait de cacher la structure et le fonctionnement interne d'un objet et de ne rendre accessible que les éléments et les opérations absolument nécessaires à l'utilisation.

On sépare donc l'interface (d'utilisation) de la réalisation (l'implémentation). Nous avons déjà appliqué le principe de l'encapsulation lorsque nous avons protégé les attributs et donné l'accès uniquement par un accesseur. De la même façon, des méthodes et des attributs peuvent être cachés à l'intérieur de la classe sans qu'on en ait conscience à l'extérieur.

Le principe de l'encapsulation est le même dans d'autres domaines, p.ex. en électronique où on cache la complexité des appareils électroniques dans un boîtier en ne donnant accès aux fonctionnalités que par des boutons, des connecteurs normés ou des télécommandes. L'utilisateur

n'a donc pas besoin de connaître toute la complexité de la **réalisation** des circuits électroniques et des connexions entre les différentes platines, mais il lui suffit de dominer ce que les constructeurs ont prévu comme **interface utilisateur**.

En informatique comme en électronique ou en mécanique, la conception de l'encapsulation et de l'interface utilisateur définit le confort d'utilisation et la sécurité des objets.

#### 4.2.1. Avantages

L'encapsulation a plusieurs avantages :

- cacher la complexité d'un objet et faciliter son utilisation (*EN : information hiding*)
- protéger l'objet contre des opérations illicites de l'extérieur
- savoir modifier les interna d'un objet sans que l'utilisateur de l'objet ne doive changer son code

#### 4.2.2. Niveaux d'accessibilité

En Java, l'encapsulation est réalisée par les niveaux d'accessibilité. Jusqu'ici, vous connaissez deux niveaux d'accessibilité (**private** et **public**) pour les classes, attributs et méthodes. En Java, il existe 4 niveaux différents d'accessibilité :

- **Accessibilité par défaut**

Mot clé : (aucun)

Symbole UML : ~

Si le niveau d'accessibilité d'un élément n'est pas spécifié, toutes les classes du même **paquet** ont accès à cet élément.

- **Accessibilité protégée**

Mot clé : **protected**

Symbole UML : #

Si un élément est préfixé du mot clé **protected**, toutes les classes du même paquet ainsi que tous les **héritiers** directs (→ voir chapitre 4.4) ont accès à cet élément.

- **Accessibilité privée**

Mot clé : **private**

Symbole UML : -

Un élément préfixé par le mot clé **private** est uniquement accessible depuis la classe elle-même. (Les instances d'une classe ont quand même accès aux éléments privés des autres instances de la même classe.)

- **Accessibilité publique**

Mot clé : **public**

Symbole UML : +

Si un élément est marqué comme étant **public**, il n'existe aucune restriction par rapport à la visibilité de cet élément.

## Exemple

Soient les deux classes suivantes qui se trouvent dans le même paquet :

```
public class Test
{
    int defaultInt = 0;
    protected int protectedInt = 1;
    private int privateInt = 2;
    public int publicInt = 3;
}

public class Launcher
{
    public static void main(String[] args)
    {
        Test test = new Test();
        test.defaultInt = 10;           // OK, car même paquet
        test.protectedInt = 10;         // OK, car même paquet
        test.privateInt = 10;           // !! ERROR !!
        test.publicInt = 10;            // OK
    }
}
```

+	Test
~	defaultInt : int
#	protectedInt : int
-	privateInt : int
+	publicInt : int



### 4.3. Agrégation

Dans la POO, il peut exister différentes relations entre les classes. Une relation que nous connaissons déjà sans lui avoir donné un nom est **l'agrégation**.

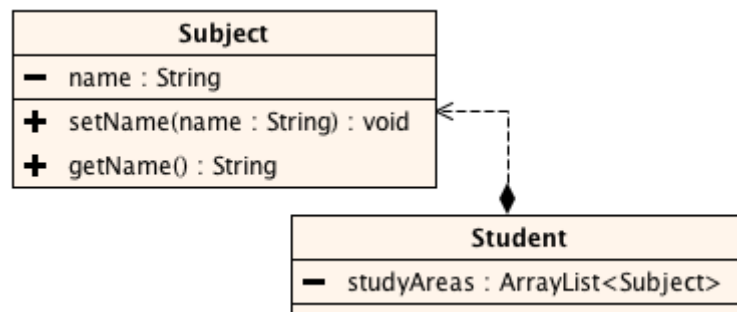
L'**agrégation** est une relation entre deux classes. Elle décrit le fait qu'une classe "**possède**" des instances d'une autre classe ou vice-versa que des instances d'une classe "**font partie**" des instances d'une autre classe (EN : **is-part-of**).

En effet, nous avons employé l'agrégation à chaque fois que nous avons intégré une instance d'un objet comme attribut dans une autre classe. On utiliserait p.ex. l'agrégation pour modéliser le fait que

- les élèves *font partie* d'une classe  $\Leftrightarrow$  une classe *possède* des élèves
- un (ou plusieurs) moteurs *font partie* d'un avion  $\Leftrightarrow$  un avion *possède* un (ou plusieurs) moteurs,
- ...

#### Exemple

Considérons les classes **Student** et **Subject** qui représentent un étudiant respectivement une branche.



On constate que :

1. Un étudiant (**Student**) **possède** une liste de domaines dans lesquels il réalise ses études.
2. La liste (**ArrayList**) de branches (**Subject**) est un attribut de la classe **Student**.
3. La classe **Subject** **est une partie** de la classe **Student** puisque la classe **Student** ne peut pas exister sans la classe **Subject**.

## 4.4. Héritage

**L'héritage** est une relation entre deux classes. **L'héritage** est la possibilité de définir une classe en ne formulant que les différences par rapport à une classe existante. Cette relation s'appelle une **relation de généralisation**, ou encore relation "**est-un**" (EN : "**is-a**" ).

Dans la POO il est possible qu'une classe **B hérite d'une classe A**, ce qui veut dire que la classe **B** possède tous les éléments la classe **A** (sans même écrire une ligne de code). Après, évidemment, on ajoute des éléments à la classe **B** ce qui rend la classe **B** plus **spécifique** que **A**. Ce concept est très fréquent dans la vie réelle.

### Exemple : Prenons les termes 'personne', 'employé' et 'client'

Un employé **est une** personne et possède donc tout ce qui définit une personne. Dire qu'une personne est un employé est cependant plus spécifique et implique qu'elle possède des caractéristiques supplémentaires (employeur, salaire, ...).

- 'Personne' est donc un terme plus général,
- 'Employé' est un terme plus spécifique et plus restrictif,
- 'Employé' possède toutes les caractéristiques de 'Personne' et
- 'Employé' possède des caractéristiques supplémentaires.

Un client **est une** personne et nous pouvons répéter les mêmes constatations que pour 'employé' et 'personne'.

Si nous devons modéliser un programme comprenant employés et clients, il est donc très utile de regrouper dans une classe **Person** tous les points communs entre clients et employés. Les classes **Client** et **Employee** seront définies sur base de **Person**. Ainsi, il suffira de définir seulement les caractéristiques spécifiques qui distinguent **Employee** et **Client** de **Person** => On évitera la duplication du code.

On dira donc que

**Employee est dérivé de Person**      ou que      **Employee hérite de Person.**

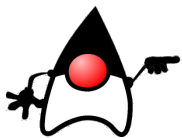
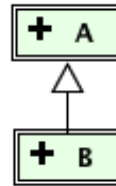
**Client est dérivé de Person**      ou que      **Client hérite de Person.**

En UML, on représente cette relation de généralisation comme suit :



Pour exprimer cette relation d'héritage entre A et B on dit que :

- A est la **classe de base**
- B est une **spécialisation** de A
- A est la **généralisation** de B
- B est une **sous-classe** de A
- A est la **super-classe** de B
- B est la **classe fille** de A
- A est la **classe mère** de B



#### Remarques

- Pour autant qu'elle ne soit pas protégée, il est même possible d'étendre une classe dont on ne possède pas le code source !
- Avec les raccourcis <Alt-F12> en Windows et <Ctrl-F12> en Mac OSX, vous pouvez faire afficher la hiérarchie d'héritage de la classe actuelle.

#### 4.4.1. Le mot clé "extends"

Pour exprimer l'héritage en Java, il suffit d'ajouter dans la définition de la sous-classe le mot clé **extends** suivi du nom de la classe de base :

```
public class B extends A
{
    ...
}
```

Maintenant, B contient tous les éléments de A et nous pouvons ajouter les attributs et méthodes qui distinguent B de A.

#### 4.4.2. Exercice résolu

Imaginons que nous devons écrire un programme pour une petite entreprise. Nous savons déjà que nous devons implémenter des classes pour gérer les clients et les employés. Tous les deux possèdent des éléments communs qui nécessitent des attributs, avec des méthodes de gestion (nom, prénom, adresse, date de naissance, etc.). Au lieu de répéter les éléments dans les deux classes, nous rassemblons les éléments dans une **classe de généralisation** **Person**.

Nous créons ensuite la classe **Client** en ajoutant '**extends Person**' derrière la déclaration. (En Unimoz, vous pouvez alternativement employer l'assistant 'Add Class...' et entrer 'Person' dans le champ 'extends').

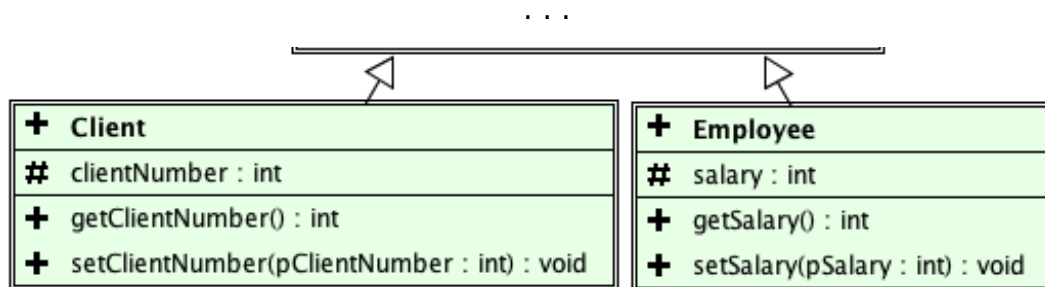
+ <b>Person</b>
# givenName : String
# surName : String
# dateOfBirth : String
# address : String
+ setGivenName(pGivenName : String) : void
+ setSurName(pSurName : String) : void
+ setDateOfBirth(pDateOfBirth : String) : void
+ setAddress(pAddress : String) : void
+ getGivenName() : String
+ getSurName() : String
+ getDateOfBirth() : String
+ getAddress() : String
+ toString() : String

La méthode `toString` est définie de façon à ce qu'elle retourne un texte de la forme :

```
<givenName> <surName> *<dateOfBirth> (<address>)
```

Si vous créez maintenant un nouveau **Client**, vous verrez qu'il dispose déjà des attributs et méthodes de **Person**. Ajoutez aussi une classe **Employee** pour les employés.

Ajoutons maintenant les éléments spécifiques des clients (no. carte client) et des employés (salaire) dans les sous-classes **Client** et **Employee**.

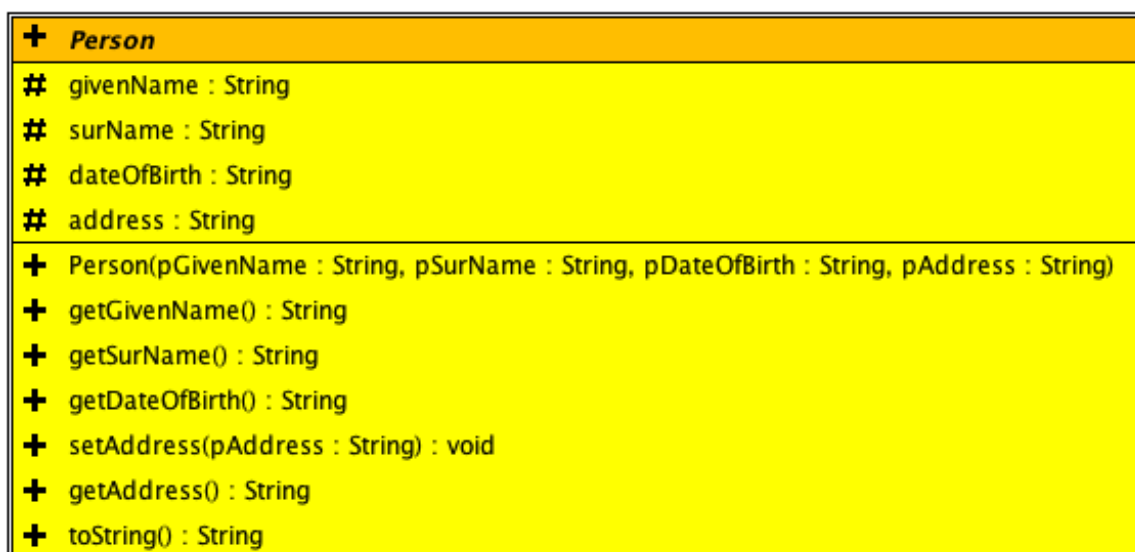


Si vous créez maintenant un nouveau **Client**, vous verrez qu'il dispose de l'attribut `clientNumber` en plus des attributs et méthodes de **Person**.

En vérifiant les trois classes, nous voyons que la réalisation est loin d'être parfaite dans l'esprit de **l'encapsulation**. En ce qui concerne l'**exactitude des données**, il est même possible de créer des clients sans nom, sans date de naissance et sans numéro de client. En plus, il est possible de modifier le nom ou la date de naissance des personnes plusieurs fois.

La meilleure solution est donc d'ajouter des constructeurs qui nous forcent d'initialiser les instances correctement, puis de supprimer la possibilité de modification des données persistantes (nom, prénom, date de naissance, numéro de client). L'adresse d'une personne ou le salaire d'un employé peuvent aussi changer plus tard.

Changeons d'abord la classe de base **Person** :





Avant de continuer, nous constatons plusieurs choses :

- La méthode `toString` héritée de `Person` n'est plus correcte (complète) pour `Client` et `Employee` (le numéro de client et le salaire manquent).
- Les constructeurs de `Client` et `Employee` différeront de celui de `Person`, mais ils auront une grande partie commune.
- En plus, le compilateur affiche deux messages d'erreur du genre '`... cannot find symbol`' aussitôt que nous avons ajouté un constructeur (avec des paramètres) à `Person`. Les messages d'erreur sont affichés au début des classes `Client` et `Employee` (même si nous n'avons rien changé dans ces classes...).

Commençons par nous occuper du premier "problème" ...

#### 4.4.3. Redéfinition de méthodes

**La redéfinition d'une méthode** est le fait de définir dans une sous-classe une méthode qui existe déjà **sous le même nom** (et avec le même nombre et type de paramètres) dans une classe de base de la classe. Dans ce cas la méthode redéfinie de la classe de base ne sera plus disponible (directement) pour les instances de la sous-classe.

##### Attention :

- Les méthodes du même nom mais avec d'autres paramètres ne sont pas touchées par la redéfinition.
- Dans le code de la sous-classe nous avons toujours accès aux méthodes redéfinies, mais de façon indirecte : c.-à-d. nous pouvons toujours les appeler par "**super**" (→ voir aussi chapitre 4.4.4), même si elles ne sont plus disponibles directement dans les instances de la sous-classe.

##### Exercice résolu (suite) :

Redéfinissons la méthode `toString` pour notre classe `Client` :

```
public String toString()
{
    return givenName + " " + surName + " *" + dateOfBirth
        + " (" + address + ") no.:" + clientNumber;
}
```

Maintenant, un appel de `toString` d'un client va employer cette nouvelle méthode. La méthode originale de `Person` ne sera plus disponible aux instances de `Client`.

Adaptez aussi la méthode `toString` de `Employee`.

#### 4.4.4. Accès aux éléments de la super-classe - *super*

Une classe peut utiliser les éléments hérités comme s'ils se trouvaient dans la classe elle-même. Dans la classe **Client**, nous pouvons donc écrire directement :

```
System.out.println(givenName + " " surName);  
setAddress("57, rue Bellevue L-3883 Manternach");
```

Si par contre :

- nous devons appeler une méthode de la super-classe qui existe **sous le même nom** dans la sous-classe, ou
  - si nous voulons appeler le **constructeur** de la super-classe,
- alors nous devons placer le mot-clé **super** devant le nom de la méthode.

Le mot clé **super** peut être considéré comme une référence à la super-classe.

#### Exercice résolu (suite) :

En redéfinissant la méthode **toString**, vous avez certainement remarqué que vous avez dû répéter du code qui se trouvait déjà dans la classe **Person**. On peut bien sûr copier le code par *copy-paste*, mais ce serait bien plus flexible de faire une référence à la méthode héritée au lieu de la copier.

Dans la classe **Client**, nous pouvons écrire très élégamment:

```
public String toString()  
{  
    return super.toString() + " no.:" + clientNumber;  
}
```

Ceci sera encore plus important dans des méthodes plus complexes, où il faudrait à chaque modification recopier le code de la super-classe dans toutes les sous-classes...

#### 4.4.5. Redéfinition du constructeur

##### ATTENTION : LES CONSTRUCTEURS NE SONT PAS HÉRITÉS !

**Mais : Lors de sa construction, la sous-classe appelle automatiquement le constructeur par défaut de sa super-classe (s'il existe) !**

**Constructeur par défaut :** Si nous ne définissons pas de constructeur pour une classe, alors la classe dispose d'un constructeur par défaut sans paramètres. Ce constructeur par défaut est hérité de la classe `java.lang.Object` qui est directement ou indirectement la super-classe de toutes les classes. Le constructeur par défaut construit une instance et initialise les attributs à leurs valeurs par défaut (0, null, false).

Dès que nous définissons notre propre constructeur, celui-ci remplace le constructeur par défaut. Si dans une super-classe nous définissons un constructeur avec des paramètres, il n'existe plus de constructeur sans paramètres (sauf, si nous nous définissons en plus un constructeur sans paramètres). **La sous-classe ne trouve donc plus de constructeur par défaut qu'elle peut appeler** et en conséquence elle produit une erreur du type `'... cannot find symbol'`.

##### Exercice résolu (suite) :

Enfin, nous pouvons nous expliquer les messages d'erreur dans notre projet et y remédier : Les messages d'erreur dans **Client** et **Employee** s'expliquent par le fait que nous avons remplacé le constructeur par défaut (sans paramètres) de la super-classe **Person**. par un constructeur avec des paramètres.

Les constructeurs de **Client** et **Employee** doivent donc appeler le constructeur de **Person** explicitement. Pour ceci, nous aurons besoin de la référence **super**. L'appel au constructeur par défaut de la super-classe se ferait tout simplement par **super()**, mais comme le constructeur que nous voulons appeler a des paramètres, nous devons les indiquer entre parenthèses derrière **super**.

##### **Le constructeur d'une sous-classe a typiquement deux charges :**

- 1. appeler le constructeur de la super-classe,**
- 2. initialiser ses propres attributs** (ici : `clientNumber`).

Dans **Client**, nous définirons le constructeur donc comme suit :

```
public Client(String pGivenName, String pSurName, String pDateOfBirth,
               String pAddress, int pClientNumber)
{
    super(pGivenName, pSurName, pDateOfBirth, pAddress);    //(1.)
    clientNumber = pClientNumber;                            //(2.)
}
```

Définissez aussi le constructeur pour **Employee** et testez les deux classes.



#### 4.4.6. Classes abstraites

Une **classe abstraite** est une classe qui n'est pas utilisée pour en créer des instances, mais uniquement comme classe de généralisation pour pouvoir en dériver des classes plus spécialisées. En UML le nom de la classe abstraite est inscrit en italique.

Une classe abstraite est le plus souvent employée pour regrouper les éléments (méthodes et attributs) communs d'un ensemble de classes qui seront spécialisées par la suite.

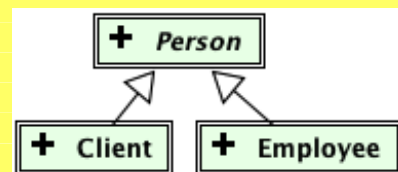
Par opposition, une **classe concrète** est une classe qui est utilisée pour en créer des instances.

En Java, les classes abstraites sont précédées du mot clé **abstract** pour que le compilateur les reconnaisse comme telles et puisse émettre une erreur de compilation lorsqu'on essaie d'en instancier un objet.

##### Exercice résolu (suite) :

Lors de la modélisation de notre petite entreprise, il est clair que la classe **Person** doit être réalisée comme classe abstraite puisqu'elle sert seulement comme 'dénominateur commun' aux classes **Client** et **Employee**, sans jamais être utilisée pour en dériver des instances.

```
public abstract class Person
{
    . . .
}
```



Dans la représentation UML, la classe **Person** apparaît alors en italique et un appel du genre **new Person(...)** va provoquer une erreur de compilation. (Évidemment les appels de **new Client(...)** ou **new Employee(...)** seront toujours possibles).

#### 4.5. Polymorphisme

Le **polymorphisme** est la possibilité de considérer une instance d'une classe comme étant aussi une instance des classes de base de cette classe.

Dans notre exemple, il est possible de traiter une instance de **Employee** comme étant aussi une instance de **Person** ou même de **Object**. Ceci est extrêmement pratique parce qu'on peut envoyer une instance de **Employee** à une méthode désirant comme paramètre une instance de **Object** ou **Person**. Ainsi on peut définir des méthodes générales qui fonctionnent pour tout un sous-arbre de la hiérarchie des classes.

De même on peut définir un attribut ou une variable d'un type de base (p.ex. **Object** ou **Person**) et on peut lui affecter des instances d'une de ces sous-classes (directes ou indirectes).

```
Person p = new Employee("James", "Black", "12/3/1960", "2323 Downtown NY", 5800);
Object o = new Employee("James", "Black", "12/3/1960", "2323 Downtown NY", 5800);
```

Le polymorphisme fonctionne uniquement dans une seule direction, c.-à-d. une instance d'une classe ne peut pas être considérée comme instance d'une de ses sous-classes. L'affectation suivante est donc incorrecte :

```
Employee e = new Person("James", "Black", "12/3/1960", "2323 Downtown NY"); //FAUX !!
```

### Compatibilité des affectations :

Une classe est compatible avec ses classes ancêtres.

### Remarque : Polymorphisme et la 'vie réelle'

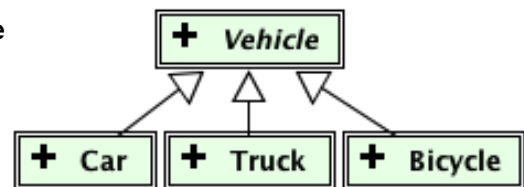
Cette façon de traiter l'héritage correspond bien à notre conception des choses de la vie réelle : Effectivement, un employé est une personne et peut être traité comme telle, mais inversement on ne peut pas dire que toute personne est un employé, et la traiter comme tel ...

### Exercice :

Soient les classes **Vehicle**, **Car**, **Truck**, **Bicycle** définies comme suit :

Que pouvez-vous dire de la classe **Vehicle** ?

Soient les déclarations suivantes :



```
Car      myCar;
Truck    myTruck;
Bicycle  myBicycle;
Vehicle  myVehicle;
```

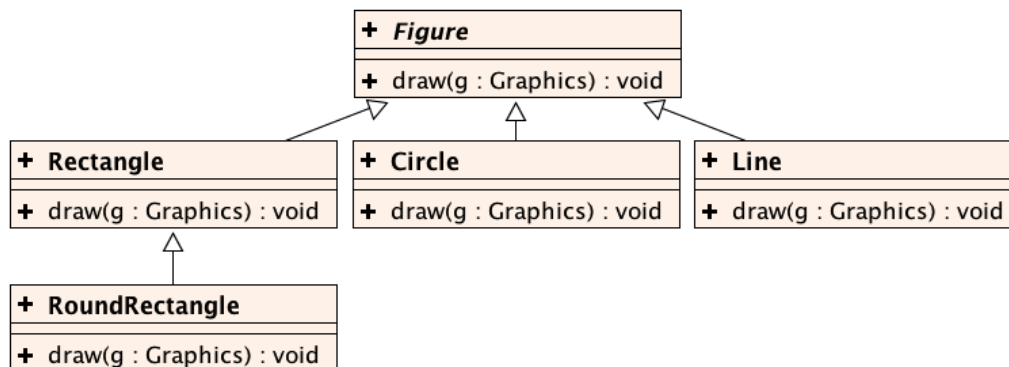
Lesquelles des affectations suivantes sont compatibles ?

Lesquelles sont incompatibles, ou incorrectes pour d'autres raisons ?

1. Truck t = myVehicle;
2. myVehicle = myBicycle;
3. Vehicle v = myTruck;
4. myCar = new Vehicle();
5. Car t = new Truck();
6. Vehicle v = new Car();

#### 4.5.1. Polymorphisme et redéfinition de méthodes (*late binding*)

Le concept du polymorphisme devient particulièrement intéressant si on considère en plus qu'il est possible de redéfinir des méthodes qui ont déjà été définies dans des classes de base (→ voir chapitre 4.4.3).



Pour illustrer ceci, considérons l'exemple suivant qui est issu d'un programme de dessin :

```

public abstract class Figure
{
    //définition des méthodes et attributs communs à toutes les figures
    . . .
    public void draw(Graphics g)
    {
        // ? ? ?
    }
}

```

```

public class Rectangle extends Figure
{
    public void draw(Graphics g)
    {
        //instructions pour dessiner un rectangle
        . . .
    }
}

```

```

public class Circle extends Figure
{
    public void draw(Graphics g)
    {
        //instructions pour dessiner un cercle
        . . .
    }
}

```

```

public class Line extends Figure
{
    public void draw(Graphics g)
    {
        //instructions pour dessiner une ligne
        . . .
    }
}

```

```

public class RoundRectangle extends Rectangle
{
    public void draw(Graphics g)
    {
        //instructions pour dessiner un rectangle aux coins arrondis
        . . .
    }
}

```

Nous voyons que chaque classe dérivée de **Figure** a redéfini la méthode héritée **draw** pour que les objets soient dessinés de façon individuelle et correcte. Il faut cependant se poser la question ce qui se passe si, en appliquant le polymorphisme, on procède comme suit :

```
Figure f1 = new Rectangle();  
Figure f2 = new RoundRectangle();  
f1.draw(g);  
f2.draw(g);
```

**f1** et **f2** sont du type **Figure**, mais ils font référence à des instances du type **Rectangle** et **RoundRectangle**.

Quelle méthode **draw** sera alors appelée, celle de **Figure**, celle de **Rectangle** ou celle de **RoundRectangle** ?

Java suit le principe le plus confortable (pour nous) en appelant automatiquement la méthode qui correspond à la nature de l'instance (et non celle de la référence).

Donc notre appel de **f1.draw(g)** produira effectivement un rectangle à l'écran et l'appel de **f2.draw(g)** produira un rectangle aux coins arrondis.

Si maintenant au lieu de deux figures **f1** et **f2**, vous vous imaginez toute une liste (*array*, **Vector** ou **ArrayList**) de figures, qui est remplie en désordre de rectangles, cercles, lignes etc, vous saurez apprécier l'utilité de cette façon de procéder... !

### Remarque pour avancés : '*late binding*' vs. '*early binding*'

Cette façon de lier automatiquement une méthode redéfinie au type réel de l'instance, nécessite une technique connue sous le nom de **late binding** ou aussi **dynamic binding**.

Expliquons ceci en reprenant l'exemple d'une liste **ArrayList<Figures> figureList** remplie de toutes sortes de figures : En effet, lors de la compilation, on ne peut pas encore savoir quelle méthode doit être liée à l'appel de **figureList.get(i).draw()** (celle de **Rectangle**, celle de **RoundRectangle**, celle de **Circle**, ...). Cette liaison de l'appel de **draw** au code de la 'bonne' méthode (celle correspondant au type réel de l'instance) a lieu seulement lors de l'exécution du programme – donc très tard et de façon dynamique. En principe, Java emploie le principe du *late binding* pour tout appel d'une méthode d'instance.

Lorsque le compilateur sait résoudre les adresses lors de la compilation (p.ex : les adresses des attributs ou des méthodes statiques), on appelle cela **early binding** ou aussi **static binding**.

### 4.5.2. Méthodes abstraites

En regardant de près l'exemple du chapitre 4.5.1, nous remarquons une incohérence dans la définition de la méthode **Figure.draw**. En effet, quelle est l'utilité du bloc d'instructions de **Figure.draw** ? Cette méthode est redéfinie complètement dans chaque sous-classe et il n'y a pas une ligne de code utile que l'on pourrait écrire dans son bloc d'instructions. Cependant on ne peut pas la supprimer, sinon le mécanisme de la redéfinition polymorphe de la méthode (décrit dans le chapitre précédent) ne fonctionnerait pas.

En Java, une telle méthode est définie comme méthode abstraite.

Une **méthode abstraite** est une méthode qui n'est pas implémentée, mais qui est déclarée uniquement pour pouvoir être surchargée dans les classes dérivées.

**abstract** : mot clé marquant une méthode dont l'implémentation est reportée à ses classes descendantes.

#### Attention :

- Une méthode abstraite n'a pas de bloc d'instructions, mais sa déclaration est suivie d'un point-virgule.
- Une méthode abstraite peut seulement se trouver dans une classe abstraite.
- Chaque classe descendante doit nécessairement redéfinir la méthode abstraite.

Dans notre exemple :

```
public abstract class Figure
{
    //définition des méthodes et attributs communs à toutes les figures
    . . .

    public abstract void draw(Graphics g);
}
```

## 4.6. L'opérateur « instanceof »

Cet opérateur permet de tester si un objet est une instance d'une classe donnée.

### Exemple

Prenons le code suivant :

```
!! Animal myAnimal = new Cat();

if(myAnimal instanceof Animal)
{
    System.out.println("It is an animal ...");
}
if(myAnimal instanceof Cat)
{
    System.out.println("It is a cat ...");
}
if(myAnimal instanceof Dog)
{
    System.out.println("It is a dog ...");
}
```

Après exécution, le code affiche que `myAnimal` est un animal et que `myAnimal` est un chat.

## 5. Modificateurs *static* et *final*

### 5.1. Les éléments statiques - *static*

En Java, on dit qu'un élément (attribut ou méthode) est **statique**, s'il est partagé par toutes les instances d'une même classe. Cet élément existe alors au niveau de la classe et il est utilisable même sans qu'il n'y ait des instances de la classe. On n'a donc pas besoin de créer des objets avant de pouvoir accéder à des méthodes ou attributs statiques.

En conséquence, un élément statique (niveau de la classe) ne peut pas accéder directement à des éléments non statiques (niveau des instances) de sa propre classe.

Cependant, chaque instance peut accéder aux attributs et aux méthodes statiques.

En UML, les éléments statiques sont soulignés.

On dit aussi **variables de classe** ou **méthodes de classe** pour les **attributs statiques** ou les **méthodes statiques**.

#### Exemple 1 : attribut statique

Nous voulons compter le nombre d'instances qui ont été créées d'une classe. Il est évident qu'un tel compteur ne peut pas faire partie de chaque instance de la classe, mais qu'il doit se situer au niveau de la classe elle-même. Nous créons donc un variable statique :

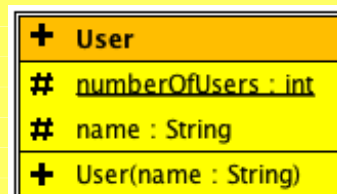
```
protected static int numberOfUsers = 0;
```

Cette variable existe une seule fois pour toute la classe (même s'il n'existe pas encore d'instances) et elle aura la même valeur pour toutes les instances de la classe.

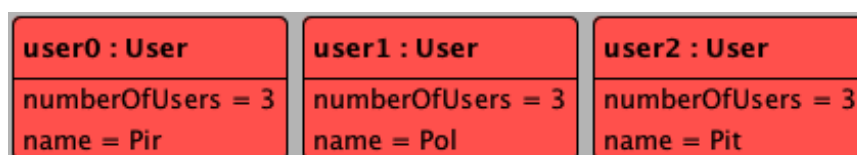
Chaque instance de la classe peut changer la valeur de **numberOfUsers**. Pour compter les instances, on peut ajouter une ligne au constructeur de la classe. Le code se présente alors comme suit :

```
public class User
{
    protected static int numberOfUsers = 0; //existe 1 fois pour la classe
    protected String name; //non statique: existe 1 fois dans chaque objet

    public User(String name)
    {
        this.name = name;
        numberOfUsers = numberOfUsers + 1;
    }
}
```



Après avoir créé trois instances avec les noms Pir, Pol, Pit, les instances peuvent se présenter comme suit :



En ajoutant un utilisateur, **numberOfUsers** est incrémenté pour tous les objets à la fois.

### Exemple 2 : méthode statique

Pour accéder à `numberOfUsers`, il faut écrire un accesseur. En principe, cet accesseur peut être statique ou non statique. Pour l'utilisateur de la classe `User`, il est quand même très utile de pouvoir demander à la classe combien d'utilisateurs existent. Si l'accesseur n'était pas statique, il faudrait connaître ou créer une instance pour pouvoir demander le nombre d'utilisateurs...

On décide donc de déclarer une méthode `getNumberOfUsers` comme étant statique (ce qui est d'ailleurs préférable pour tous les accesseurs d'attributs statiques) :

```
public class User
{
    protected static int numberOfUsers = 0; //existe 1 fois pour la classe
    protected String name; //non statique: existe 1 fois dans chaque objet

    public User(String name)
    {
        this.name = name;
        numberOfUsers = numberOfUsers + 1;
    }

    public static int getNumberOfUsers()
    {
        return numberOfUsers;
    }

    public String getName()
    {
        return name;
    }
}
```

+ User
# numberOfUsers : int
# name : String
+ User(name : String)
+ getNumberOfUsers() : int
+ getName() : String

On peut donc faire ceci:

```
public class Main
{
    public static void main(String[] args)
    {
        System.out.println( User.getNumberOfUsers() ); //affiche 0
        User user0 = new User("Pir");
        User user1 = new User("Pol");
        User user2 = new User("Pit");
        System.out.println( User.getNumberOfUsers() ); //affiche 3
    }
}
```

Notez que la **méthode** `getNumberOfUsers()` est directement appelée sur la **classe** `User` et non sur l'une des instances !



### La méthode **main()** :

En regardant le dernier exemple de plus près, vous comprendrez enfin, pourquoi la méthode **main()** d'une application peut être démarrée sans avoir besoin de créer une instance de la classe ...

Remarquez aussi que la méthode statique **getNumberOfUsers** n'a pas d'accès direct à l'attribut **name** et ne peut pas appeler **getName()** sauf pour une instance dont elle reçoit éventuellement la référence. A l'intérieur de **getNumberOfUsers** les instructions suivantes provoquent donc une erreur avec les messages :

*"non-static variable name cannot be referenced from a static context"* et

*"non-static method getName() cannot be referenced from a static context"*

```
public static int getNumberOfUsers()
{
    name = "Hallo";                // impossible !
    System.out.println(getName()); // impossible !
    return numberOfUsers;
}
```

### Autres méthodes statiques connues :

- Toutes les méthodes de la classe **Math** (**Math.sqrt(...)**, **Math.sin(...)**, ...) sont des méthodes statiques. En effet, nous n'avons jamais défini d'instance de cette classe et nous avons appelées ses méthodes en commençant par le nom de la **classe**. De telles classes sont rares puisqu'elles contiennent des méthodes d'utilité générale, non liées à un type d'objet spécifique.
- La méthode **Calendar.getInstance()** fournissant une instance du calendrier contenant la date actuelle est une méthode statique.
- La méthode **JColorChooser.showDialog(...)** ouvrant un dialogue de sélection de couleurs est une méthode statique (→ voir exercice F.2).

## 5.2. Les éléments constants - final

Un élément peut être préfixé du mot clé **final**, ce qui a comme effet qu'il ne peut plus être modifié par la suite. Il s'agit donc de quelque chose de constant.

### 5.2.1. Définition de constantes (attributs constants)

En Java, il n'existe pas de mot clé pour définir une constante, mais on peut quand même définir des constantes (en fait des attributs de classe constants) en combinant les modificateurs **static** et **final** :

**final** => la valeur ne peut plus être modifiée après l'initialisation

**static** => la constante n'existe qu'une seule fois et ceci au niveau de la classe

Les constantes de classe sont écrites entièrement en majuscules.

Les constantes de classe doivent être initialisées tout de suite lors de la déclaration.

Exemple :

```
public class Circle
{
    public static final double PI = 3.141592;
    protected static final double RADIUS = 100.4;
    static final double MAX_RADIUS = 500;
    //...
}
```

Attention:

Le modificateur **final** appliqué à un objet fixe la référence (l'adresse) de l'objet. En conséquence, on ne peut pas attribuer un nouvel objet à l'attribut, mais on peut changer le contenu de l'objet.

Exemple :

```
final Point p = new Point(10, 20);
p.x = 20; //Permis, le contenu de l'objet p est changé
p = new Point(20, 20); //ERREUR! essay de changer la référence de l'objet p
```

Evidemment, si un objet ne permet pas la modification de ses attributs (c.-à-d. si la classe est '**immutable**', comme p.ex. **String**), alors aucune modification de l'objet n'est possible.

Exemple :

```
final String name = "John Doe"; //L'attribut name ne peut plus être changé
```

### 5.2.2. Pour avancés : Paramètres et variables locales constants

Plus rarement, le mot clé **final** est appliqué aux paramètres et aux variables locales pour éviter qu'ils soient changés après leur initialisation. Dans ce cas, il suffit d'indiquer le modificateur **final** devant la déclaration. Il n'est pas nécessaire d'initialiser ces éléments immédiatement lors de la déclaration, mais il ne peut y avoir qu'une seule affectation au cours de leur vie.

Pour les paramètres objets, **final** peut être utile pour éviter que le paramètre pointe sur un autre objet que celui reçu initialement.

### 5.2.3. Pour avancés : Classes non dérivables

Si le modificateur **final** est appliqué à une classe, alors on ne peut pas hériter de cette classe pour en dériver des sous-classes. Ce blocage de l'héritage peut avoir différentes raisons :

- la conception de la classe est telle qu'on n'a jamais besoin de la modifier,
- on veut garantir la sécurité de la classe,
- on veut optimiser la performance : on veut éviter que les méthodes soient redéfinies, ce qui évite aussi la recherche dynamique des méthodes (*late binding*).
- on veut travailler avec des objets 'immuables' pour éviter des précautions supplémentaires dans des applications multi-tâches.

Exemple : la classe **String** est définie comme **final** pour des raisons de performance.

### 5.2.4. Pour avancés : Méthodes non redéfinissables

Si le modificateur **final** est appliqué à une méthode, alors on ne peut pas redéfinir cette méthode dans les sous-classes. Ce blocage de l'héritage peut avoir différentes raisons :

- on veut garantir que la méthode fasse exactement ce qui est décrit dans cette méthode,
- on veut optimiser la performance : le compilateur sait produire un code plus efficace s'il n'y a pas la possibilité d'une redéfinition de la méthode. Il n'y aura pas de recherche dynamique pour la méthode (*late binding*).

## 6. Arrays - listes statiques

Les **tableaux**, généralement appelés « **array** », sont des ensembles de données du même type. Les éléments peuvent être des **objets ou des valeurs des types primitifs** (int, double, ...).

Les *arrays* sont une structure de base plus primitive, mais plus rapide que *ArrayList*. En fait, la classe *ArrayList* est réalisée sur base de *arrays*, mais elle permet de les utiliser de façon plus confortable. Parfois, il est quand-même utile ou inévitable de travailler avec des *arrays*, p.ex. si des méthodes prédéfinies ont besoin d'un *array* comme paramètre ou retournent un *array* comme résultat (--> donnez un exemple...).

En Java, les tableaux sont déclarés de la manière suivante :

```
<type>[] <name> = { <value_1>, <value_2>, ... , <value_n> };  
<type>[] <name> = new <type>[<length>];  
<type>[] <name>;
```

- La première ligne déclare un tableau et l'initialise directement avec un certain nombre d'éléments. La taille du tableau est donnée par le nombre d'éléments y inscrits.
- La deuxième ligne déclare un tableau et l'initialise avec autant d'éléments que spécifiés par « **<length>** ». Les éléments contiennent la valeur **null** pour les objets, **0** ou **false** pour les types primitifs.
- La troisième ligne déclare un tableau sans l'initialiser.
- **En Java, la taille d'un tableau est fixe. Une fois initialisée, la taille du tableau ne peut plus être changée. C'est pourquoi les tableaux sont encore appelés listes statiques.**
- Le premier élément d'un tableau se trouve toujours à la position « 0 ».
- Tous les tableaux en Java possèdent un attribut « **length** » qui contient le nombre d'éléments du tableau.
- L'accès aux éléments d'un tableau est donné en ajoutant [*<position>*] derrière le nom du tableau, où *<position>* est la position à laquelle on veut accéder.
- Comme la taille d'un tableau est fixe, on essaie en général d'estimer le nombre maximal d'éléments utiles à l'application et on réserve l'espace nécessaire lors de l'initialisation. Peu à peu on remplit le tableau du début vers la fin en gardant toujours un bloc continu d'éléments au début du tableau. Ainsi, les **n** premières positions contiennent des éléments, le reste du tableau est vide. Le nombre maximal d'éléments utilisables (c.-à-d. la taille utilisée lors de l'initialisation) s'appelle alors la **dimension maximale**. Le nombre **n** d'éléments qui se trouve actuellement dans un tableau s'appelle la **dimension effective**.

**Exemples :**

```
int[] fibonacci = {1,1,2,3,5,8,13,21,34}
for(int i=0; i<fibonacci.length; i++)
    System.out.println( fibonacci[i] );
```

Supposons l'existence d'une classe « **Voiture** ».

```
Voiture[] voitures = new Voiture[100]; // dimension maximale = 100
int n=0; // dimension effective = 0
voitures[n] = new Voiture(); // création du premier élément
n++; // dimension effective = 1
```

## 7. Gestion des paquets

Au début du cours de NetBeans, vous avez vu que les classes prédéfinies sont organisées dans une hiérarchie de paquets. Vous savez aussi comment importer les classes d'un paquet existant dans votre projet (à l'aide de l'instruction **import**).

Dans vos projets, vous avez certainement remarqué que le nombre de vos classes augmente rapidement avec la taille du projet. Jusqu'ici, nous avons défini toutes nos classes dans le paquet par défaut (sans nom) du projet, c.-à-d. les fichiers **.java** étaient sauvés directement dans le dossier **src** du projet. Cette façon de procéder est déconseillée pour des projets plus complexes. Il devient donc nécessaire d'organiser vos propres classes dans des paquets (p.ex. pour séparer les classes du modèle des classes de la vue et du contrôleur).

- Pour créer un nouveau **paquet**, il suffit de créer un nouveau dossier de ce nom dans l'arbre des paquets du dossier **src** du projet et d'y placer les fichiers contenant l'instruction **package** pour ce paquet.
- Pour placer une classe dans un **paquet**, il faut faire deux choses :
  - Placer le fichier « **.java** » dans le répertoire en question (c.-à-d. dans le dossier qui porte le nom du paquet) et
  - indiquer dans la classe le nom du paquet à l'aide du mot clé « **package** ». Cette instruction doit être la première instruction dans le fichier, elle doit précéder les instructions **import**.

**Convention :** Les noms des paquets sont écrits entièrement en minuscules.

### Exemple

```
package lu.ecole.exemple;  
  
public class Voiture  
{  
    // ...  
}
```

En pratique, un grand nombre des opérations sont effectuées automatiquement :

En Unimozzer, il suffit d'écrire l'instruction **package** avec le nom du (nouveau) paquet au début du fichier. Lors de la sauvegarde, les dossiers pour les paquets sont automatiquement créés s'ils n'existent pas encore et les fichiers **.java** y sont copiés automatiquement.

En NetBeans, on peut créer un nouveau paquet par un clic droit sur un paquet, puis copier les fichiers **.java** par 'drag & drop' d'un paquet vers un autre. NetBeans demande s'il doit adapter toutes les références.

**Remarques :**

- Il n'existe pas d'endroit central où on peut placer les paquets qu'on veut réutiliser dans plusieurs projets. Il faut recopier les paquets vers les dossiers **src** de chaque projet en question.
- On peut regrouper les fichiers compilés de paquets dans un fichier **.jar** (non exécutable). Ce paquet archivé peut alors être redistribué avec l'application dans le dossier **dist/lib**.
- Le compilateur Java n'ajoute pas automatiquement les paquets **.jar** précompilés aux dossiers **dist/lib** (à l'exception des librairies du JDK et des gestionnaires 'LayoutManagers' du système NetBeans). Ceci doit être configuré p.ex. en NetBeans avec '**Libraries → Add Library...**'.

(→ *Organiser le projet MiniDraw en paquets*)

## 8. Exceptions

Analysons l'exemple suivant d'une méthode qui calcule le quotient de deux nombres :

```
public class Fraction
{
    ...

    public double getDecimalValue()
    {
        if (denominator!=0)
            return (double)numerator/denominator;
        else
            <??>
    }
}
```

Que faire dans le cas où le dénominateur est égal à zéro? La fonction doit retourner un « **double** » mais pour le cas où « **b==0** », on ne peut pas calculer le quotient « **a/b** »!

### 8.1. Générer une exception

Dans le cas décrit précédemment, la solution consiste à lancer ce qu'on appelle en Java une **exception**. En fait, ce n'est rien d'autre que la production contrôlée d'une erreur.

Solution

```
public double getDecimalValue()
{
    if (denominator!=0)
        return (double)numerator/denominator;
    else
        throw new ArithmeticException("Division by zero");
}
```

Lors du lancement d'une exception, il faut créer une nouvelle instance de ce type d'exception en faisant appel à son constructeur. Normalement, les exceptions possèdent (en plus du constructeur par défaut) un constructeur avec un paramètre du type **String**. Dans ce paramètre, on peut indiquer le message à afficher lors de la génération de l'exception.

Bien qu'il existe un certain nombre d'**exceptions prédéfinies**<sup>3</sup> en Java, il est aussi possible de définir ses propres **exceptions** (→ voir chapitre Error: Reference source not found ).

<sup>3</sup> <http://java.sun.com/javase/6/docs/api/java/lang/Exception.html>



## 8.2. Intercepter une exception

Lorsqu'une **exception** est lancée par une méthode, le programme appelant est généralement interrompu tout de suite. Or ceci n'est pas toujours souhaitable. En effet, Java permet d'intercepter les **exceptions** et de les traiter convenablement pour pouvoir continuer ensuite l'exécution de l'application.

Prenons l'exemple du programme suivant qui utilise la **classe** « **Fraction** » :

```
import java.util.Scanner;
public class Launcher
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the numerator: ");
        int n = input.nextInt();
        System.out.print("Enter the denominator: ");
        int d = input.nextInt();
        Fraction myFraction = new Fraction(n,d);

        try
        {
            double result = myFraction.getDecimalValue();
            System.out.println(n+" / "+d+" = "+result);
        }
        catch (ArithmeticException ariExc)
        {
            System.err.println("Error! Division by zero...");
            //System.err est la console d'erreurs (-> messages en rouge)
        }
    }
}
```

La **syntaxe générale** pour intercepter une exception est la suivante :

```
try
{
    <instructions>
}
catch (<exception> <nom>)
{
    <instructions_traitement_d'erreurs>
}
```

S'il se produit une erreur lors de l'exécution des **<instructions>**, alors le programme sort du bloc **try** et regarde si l'exception générée doit être traitée ou non. En effet, il peut y avoir un ou plusieurs blocs **catch** avec des exceptions différentes. **<exception>** est le type de l'exception<sup>4</sup> à traiter et **<nom>** est le nom donné à cette exception (uniquement valable dans le bloc respectif).

Toutes les exceptions et erreurs possèdent les méthodes suivantes qui peuvent être employées pour afficher des détails sur l'exception actuelle :

**getMessage()**                      retourne un message détaillé décrivant l'exception

4 <http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html>

<code>toString()</code>	retourne un court message décrivant l'exception
<code>printStackTrace()</code>	affiche sur la console d'erreur un message décrivant l'exception actuelle ainsi que tous ses antécédents (→ " <b>stack trace</b> ")

### 8.3. Plusieurs exceptions dans le même bloc

S'il y a plusieurs blocs **catch**, alors Java exécute le premier bloc qui correspond à l'exception. Il est donc nécessaire de placer les blocs avec des exceptions plus spécifiques DEVANT les blocs plus généraux.

Exemple :

```
try {  
    ...  
} catch (FileNotFoundException e) {  
    System.err.println("FileNotFoundException: " + e.getMessage());  
    throw new SampleException(e);  
  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

L'exception **FileNotFoundException** est dérivée directement de **IOException**. Si une exception du type **FileNotFoundException** a eu lieu, alors le premier bloc est exécuté. Toutes les autres erreurs du type **IOException** sont traitées par le deuxième bloc. (Remarquez dans l'exemple que la première instruction **catch** n'affiche pas seulement un message, mais lance aussi une autre exception).

**Mais attention :** Le bloc **catch (FileNotFoundException e)** doit précéder le bloc **catch (IOException e)**, sinon le bloc **FileNotFoundException** ne sera pris en compte.

## 8.4. Le principe "catch-or-throw" & l'instruction "throws"

En Java une exception, une fois lancée, doit être traitée sur place (par un bloc **try-catch**) ou son traitement doit être relégué (lancé, *thrown*) à la méthode appelante. Ce procédé est connu sous le nom "**catch-or-throw**".

Concrètement, si on ne veut pas traiter une exception sur place on peut laisser de côté le bloc **try-catch**. Dans ce cas, il faut ajouter l'instruction **throws** derrière l'en-tête de la méthode et énumérer toutes les exceptions qui ne sont pas traitées dans cette méthode.

La méthode appelante (qui reçoit l'exception via '**throws**') doit elle aussi suivre le principe "catch-or-throw", c.-à-d. elle a aussi la possibilité d'employer **throws** au lieu de **try-catch**. Ainsi une exception peut être passée plusieurs fois à des niveaux d'appels supérieurs, mais au plus tard dans le programme principal, elle doit être traitée par un **try-catch**.

Le compilateur Java veille à ce que le principe "catch-or-throw" soit toujours vérifié.

### Exemple :

```
public void printAllDecimalValues(Fraction[] fractions)
    throws ArithmeticException
{
    for(int i=0 ; i<fractions.length ; i++)
        System.out.println(fractions[i].getDecimalValue() );
}
```

Instruction  
pouvant produire  
une "Arithmetic  
Exception"

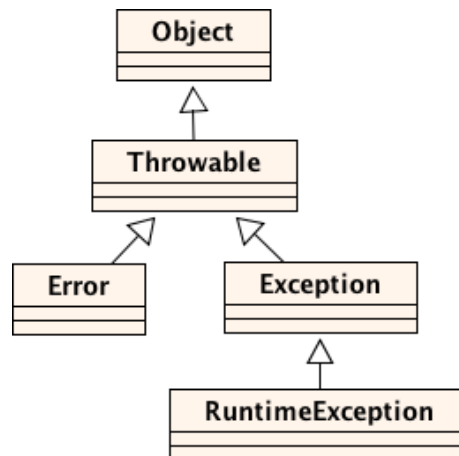
La méthode qui appelle **printDecimalValues** doit aussi suivre le principe "catch-or-throw", c.-à-d. elle doit entourer l'appel d'un bloc **try-catch** (ou bien passer l'exception de son côté à la méthode appelante à l'aide d'une instruction **throws** ).

```
...
try
{
    printAllDecimalValues(fractions);
}
catch(ArithmeticException e)
{
    ... ;
}
...
```

Traitement dans la  
méthode appelante de  
l'exception reléguée par  
'throws'

## 8.5. Les classes `RuntimeException` et `Error`

Voyons d'abord la hiérarchie des classes liées aux erreurs et exceptions (il existe encore quelques douzaines de classes dérivées de **Error**, **Exception** et **RunTimeException** qui ne sont pas représentées ici → voir *JavaDoc* si nécessaire) :



Dans le chapitre précédent, nous avons dit que les exceptions (classe **Exception**) doivent suivre le principe "**catch-or-throw**". Il y a deux exceptions au principe "**catch-or-throw**" : les classes **RuntimeException** et **Error** ainsi que leurs héritiers.

La classe **RuntimeException** est la classe ancêtre de toutes les exceptions d'exécution. Les exceptions de ce type peuvent être traitées, mais pas obligatoirement (→ "**unchecked exception**"). De cette façon, le programmeur peut limiter les efforts du traitement d'erreurs et décider lui même si une telle exception doit être traitée ou non. Le plus souvent ces exceptions qui sont dues à des erreurs de programmation et il vaut mieux éliminer l'erreur que de les traiter par un bloc **try...**

Toutes les exceptions qui ne sont pas dérivées de **RuntimeException** doivent nécessairement être traitées dans la méthode appelante (→ "**checked exception**").

La classe **Error** est la classe 'ancêtre' de toutes les erreurs d'exécution. Celles-ci sont dues en principe à des causes externes que le programme ne peut pas prévoir ni y remédier (p.ex. un fichier non accessible à cause d'un problème de hardware → **java.io.IOException**). Une application peut intercepter l'erreur et en informer l'utilisateur, mais il peut aussi être nécessaire d'afficher la pile des erreurs (**e.printStackTrace();**) et de terminer l'application.

## 8.6. Exceptions dans le constructeur

**Les exceptions peuvent être utiles dans les constructeurs pour éviter la création d'instances avec des valeurs incorrectes pour les attributs.** Jusqu'ici nous n'avons pas encore vu de méthode propre et élégante pour réagir à des paramètres avec des valeurs incorrectes ou inconsistantes lors de la construction d'un objet. Pour cette raison nous n'avons pas défini de test de validité dans les constructeurs ou les manipulateurs. Un utilisateur qui ne respectait pas le domaine de définition d'une classe risquait donc d'obtenir des résultats erronés, sans en être informé.

Une exception permet de sortir du constructeur **sans créer d'instance**. Il est cependant conseillé de tenir compte des réflexions suivantes :

- Éviter dans le constructeur d'affecter l'instance à une variable finale ou à intégrer l'instance dans une collection (p.ex. **ArrayList**) avant le test de validité. Sinon, le ramasse-miettes (= *garbage collector*) ne va pas éliminer l'instance incorrecte.
- Produire de préférence une exception traitée (*checked exception*), pour que la méthode appelante puisse réagir à la non-construction de l'objet. Sinon dans la classe appelante, la référence reste **null** ou attachée à un autre objet sans que l'utilisateur ne le sache.

## 8.7. Java 7

- Depuis Java 7 il existe la possibilité de traiter plusieurs exceptions dans un même bloc. Ainsi nous pouvons éviter de devoir définir plusieurs fois le même code dans différents blocs **catch**. Pour ce faire, il faut simplement ajouter le symbole **|** (**or**) entre les types des exceptions à traiter.

Exemple :

```
try
    ...
catch (IOException | SQLException ex) {
    ...
}
```

- Depuis Java 7 il existe la possibilité de définir un bloc nommé "**try with resources**" pour la création de ressources (fichiers, imprimantes, etc). De cette façon, les ressources ouvertes sont fermées automatiquement (sans devoir définir explicitement un bloc **finally** - voir ci-dessous).

Nous allons utiliser *try with resources* partout où c'est possible.

## 8.8. ***Pour avancés : Le bloc finally***

A la fin d'un bloc **try** se trouve très souvent un bloc **finally** qui sera exécuté dans tous les cas, donc même après le traitement d'un bloc **catch**. De cette façon on peut garantir que certaines opérations sont effectuées, même si une erreur imprévue se produit (p.ex. pour fermer un fichier, libérer une imprimante ou une autre ressource, ...). Ainsi la structure générale s'élargit comme suit :

```
try
{
    <instructions>
}
catch (<exception1> <nom_1>)
{
    <instructions_traitement_d'erreurs_1>
}
catch (<exception2> <nom_2>)
{
    <instructions_traitement_d'erreurs_2>
}
catch
...
finally
{
    <instructions_terminales>
}
```

- Un bloc **try** peut avoir un bloc **finally** même s'il n'y a pas de bloc **catch**.
- Si un bloc **catch** contient une instruction **return**, le bloc **finally** n'est pas exécuté.

En général c'est une bonne idée de placer le code terminal d'une opération ("**cleanup code**") dans un bloc **finally**.

## 9. Fichiers

Pour pouvoir mémoriser les données de nos programmes à long terme, il nous manque encore la possibilité de lire et d'écrire dans des fichiers. Il existe deux principes pour la sauvegarde de fichiers :

1. mémorisation des données sous forme de **texte** (p.ex. ligne par ligne ; sous forme de code XML).  
**Avantage** : nous pouvons visualiser facilement le contenu des fichiers ; les fichiers restent compatibles avec de futures versions du programme ;  
**Désavantage** : lecture et écriture plus complexe ( $\Leftrightarrow$  conversion des données, extraction des données d'une ligne de texte)
2. mémorisation des données sous forme **binaire**, c.-à-d. les données sont sauvegardées exactement comme elles se trouvent dans la mémoire.  
**Avantage** : lecture et écriture facile des données ;  
**Désavantage** : nous ne pouvons pas visualiser le contenu du fichier ; chaque modification de la structure des données entraîne une incompatibilité des fichiers avec les versions antérieures. Il faut donc penser à programmer un convertisseur de fichiers pour chaque changement des données.

Dans ce cours, nous traiterons uniquement les fichiers textes. Commençons cependant avec les aspects communs aux fichiers binaires et textes :

- Les classes pour le traitement de fichiers se trouvent dans le paquet **java.io**.
- Un fichier est toujours considéré comme un **flux** de données (**Stream**). Un flux peut être un flux d'octets, un flux de caractères, un flux de données (primitives), ou même un flux d'objets. Pour les fichiers texte, il s'agit de flux de caractères.<sup>5</sup>
- Le traitement d'un fichier se passe en trois étapes :
  1. **Ouverture du fichier,**
  2. **Opérations de lecture ou opérations d'écriture,**
  3. **Fermeture du fichier.**
- Les instructions de traitement de fichiers se trouvent en général dans un bloc **try**.
- Dans ce cours, nous allons traiter uniquement des fichiers à lecture/écriture **séquentielle**. C.-à-d. nous traitons les fichiers à partir du début et à chaque opération de lecture/écriture, on avance automatiquement à la donnée/ligne suivante.
- Lors de l'écriture d'un fichier, le fichier existant du même nom est remplacé par le nouveau fichier. L'ancien fichier du même nom est donc perdu.
- Le chemin (EN:*path*) par défaut est le dossier du projet Java.
- Lors de l'indication d'un nom de fichier, on peut indiquer un chemin complet. Sur tous les systèmes (même en Windows) il faut utiliser le symbole '/' lors de l'indication du chemin.

<sup>5</sup> Les flux interviennent aussi dans d'autres opérations en Java, p.ex. dans des opérations sur le réseau, sur les données audio, etc.

## 9.1. Les fichiers textes

A la base des opérations de fichiers textes en Java se trouvent la lecture et l'écriture de **flux de caractères** (individuels). Ces opérations sont réalisées dans les classes `java.io.FileReader` et `java.io.FileWriter`. En pratique, on utilise rarement ces classes de façon directe, car elles ne réalisent que des opérations très élémentaires.

Pour accéder à un fichier texte, on emploie des flux de caractères qui sont passés par un flux à mémoire tampon : **BufferedReader** ou **PrintWriter**<sup>6</sup>. Voici les instructions qu'il faut utiliser pour ouvrir un fichier texte avec mémoire tampon :

```
BufferedReader in = new BufferedReader( new FileReader("monfichier.txt") );  
PrintWriter out = new PrintWriter ( new FileWriter("monfichier.txt") );
```

La méthode pour fermer un fichier texte s'appelle `close()`, mais avec la méthode que nous employons, elle est appelée automatiquement, sans que nous n'ayons besoin de nous en occuper.

### 9.1.1. Ecriture dans un fichier texte

La classe **PrintWriter** possède les méthodes `print(...)` et `println(...)` que vous connaissez de **System.out**. En fait, **System.out** n'est rien d'autre qu'un **PrintWriter**. De cette façon, il est possible d'écrire toutes sortes de données dans un fichier texte. La manière la plus simple pour mémoriser des données dans un fichier texte est de mémoriser chaque donnée dans une nouvelle ligne.

**Exemple :**

```
import java.io.FileWriter;  
import java.io.IOException;  
import java.io.PrintWriter;  
  
public void saveToTextFile(String fileName)  
    throws FileNotFoundException, IOException {  
    try (PrintWriter out = new PrintWriter(new FileWriter(fileName,  
        StandardCharsets.UTF_8))) {  
        for (int i = 0; i < alParticipants.size(); i++) {  
            out.println(alParticipants.get(i).getFirstname()); //String  
            out.println(alParticipants.get(i).getName()); //String  
            out.println(alParticipants.get(i).getBirthYear()); //int  
        }  
    }  
}
```

Vous pouvez faire ajouter les instructions **import** et les instructions **throws** par NetBeans en cliquant sur les indicateurs d'erreur dans la marge gauche.

<sup>6</sup> Il existe aussi une classe **BufferedWriter**, mais les méthodes offertes par **BufferedWriter** sont tellement limitées, qu'il vaut mieux employer **PrintWriter**. **PrintWriter** passe aussi par une mémoire tampon.



### 9.1.2. Lecture dans un fichier texte

La lecture dans un fichier texte passé par un **BufferedReader** peut s'effectuer le plus simplement avec les méthodes suivantes :

**int read()** lecture d'un seul caractère. Résultat **-1** si le fichier est arrivé à sa fin.

et plus confortablement :

**String readLine()** lecture d'une ligne de texte. (le ou les symboles de fin de ligne ne sont pas retournés dans le résultat). Résultat **null** si le fichier est arrivé à sa fin.

Nous pouvons donc arrêter la lecture dès que la méthode **readLine** retourne **null**. Pour pouvoir lire les données avec succès, il faut évidemment connaître la structure du fichier.

**Exemple :**

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public void loadFromTextFile(String fileName)
    throws FileNotFoundException, IOException {
    try (BufferedReader in = new BufferedReader(new FileReader(fileName,
        StandardCharsets.UTF_8))) {
        alParticipants = new ArrayList<>(); //clear current list
        String line;
        //Comme nous avons mémorisés 3 lignes par participant, il faut utiliser
        //3 fois l'instruction readLine pour chaque participant.
        //Chaque passage par la boucle lit le 3 données d'un participant.
        //La condition ci-dessous lit une ligne, la mémorise dans line et vérifie
        //si on est arrivé à la fin du fichier (test sur null)
        while ((line = in.readLine()) != null) { //lire la 1ère ligne (le prénom)
            String firstName = line;
            String name      = in.readLine(); //lire la deuxième ligne (le nom)
            int    birthYear = Integer.valueOf(in.readLine()); //lire la 3e ligne
            //créer et ajouter le participant
            alParticipants.add(new Participant(firstName, name, birthYear));
        }
    }
}
```

**Remarque :**

Comme alternative à un **BufferedReader**, il peut être intéressant d'employer un **Scanner** pour la lecture. Ainsi, vous pouvez employer les méthodes pratiques définies dans la classe **java.util.Scanner** : **hasNext()**, **next()**, **nextInt()**, **nextDouble()**, etc. (→ voir *JavaDoc* ou *Annexe* du cours de 3e).

### 9.1.3. Appel des méthodes de sauvegarde et de lecture

Lors de l'appel des méthodes, il faut traiter les erreurs éventuelles qui peuvent être lancées lors des opérations avec les fichiers. Vous pouvez cependant faire générer les blocs **try ... catch** entièrement par NetBeans.

**Exemple :**

```
private void saveButtonActionPerformed(java.awt.event.ActionEvent evt) {  
    try {  
        iList.saveToTextFile("Participants.txt");  
    } catch (FileNotFoundException ex) {  
        Logger.getLogger(MainFrame.class.getName()).log(Level.SEVERE, null, ex);  
    } catch (IOException ex) {  
        Logger.getLogger(MainFrame.class.getName()).log(Level.SEVERE, null, ex);  
    }  
}
```

```
private void loadButtonActionPerformed(java.awt.event.ActionEvent evt) {  
    try {  
        iList.loadFromTextFile("Participants.txt");  
    } catch (FileNotFoundException ex) {  
        Logger.getLogger(MainFrame.class.getName()).log(Level.SEVERE, null, ex);  
    } catch (IOException ex) {  
        Logger.getLogger(MainFrame.class.getName()).log(Level.SEVERE, null, ex);  
    }  
    updateView();  
}
```

## 9.2. Lecture et sauvegarde de couleurs

Une couleur peut être sauvegardée et lue dans un fichier texte en la convertissant dans un entier, puis dans un texte avant de la sauvegarder :

**sauvegarde :** conversion en un entier :

```
out.print( color.getRGB() );
```

**lecture :** reconversion en une couleur :

```
color = Color.decode(texteLu);
```

ou aussi :

```
color = new Color(Integer.valueOf(texteLu));
```

## 9.3. Ajout à la fin d'un fichier ("Append")

Pour ajouter des données à la fin d'un fichier texte existant, il suffit d'ouvrir le fichier en envoyant **true** comme deuxième paramètre (le paramètre 'append'). Si le fichier n'existe pas encore, il est créé automatiquement.

Exemple :

```
PrintWriter out = new PrintWriter(new FileWriter("monfichier.txt", true));
```

## 9.4. Sauvegarde de plusieurs données par ligne

En pratique, il est souvent préférable de sauvegarder plusieurs données par ligne. P.ex. on peut sauvegarder dans chaque ligne les valeurs des attributs d'un objet en les séparant par des points-virgules « ; ». Ainsi dans le fichier texte chaque objet est décrit par une seule ligne.

Ce format de fichier est encore appelé CSV (**C**omma **S**eparated **V**alues) et peut simplement être importé et exporté dans d'autres programmes (p.ex Excel).

La sauvegarde de notre exemple se présente alors comme suit :

### Exemple :

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public void saveToTextFile(String fileName)
    throws FileNotFoundException, IOException {
    try (PrintWriter out = new PrintWriter(new FileWriter(fileName,
        StandardCharsets.UTF_8))) {
        for (int i = 0; i < alParticipants.size(); i++) {
            out.println(alParticipants.get(i).getFirstname() + ";" +
                alParticipants.get(i).getName() + ";" +
                alParticipants.get(i).getBirthYear());
        }
    }
}
```

Deux lignes du fichier peuvent se présenter comme suit :

```
Jean-Claude;Hammes;1991
Mara;Kingston;1999
```

Pour lire un tel fichier, il faut extraire les données. Il est très pratique d'employer la méthode **split** (définie dans la classe **String**) qui retourne les éléments dans un tableau du type **String**.

### Exemple :

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public void loadFromTextFile(String fileName)
    throws FileNotFoundException, IOException {
    try (BufferedReader in = new BufferedReader(new FileReader(fileName,
        StandardCharsets.UTF_8))) {
        alParticipants = new ArrayList<>(); //clear current list
        String line;
        //Une instruction readLine lit toutes les données d'un participant
        //Attention à la triple finalité de cette instruction :
        while ((line = in.readLine()) != null) { //lire une ligne -> line (ou arrêt)
            String items[] = line.split(";"); //séparer les éléments de la ligne
            alParticipants.add( //créer et ajouter le participant
                new Participant( items[0], items[1], Integer.valueOf(items[2]) );
            )
        }
    }
}
```

}

## 9.5. Remarques supplémentaires *pour avancés*

Un avantage de l'utilisation des méthodes `BufferedReader.readLine` et `PrintWriter.println` est que ces méthodes reconnaissent automatiquement les différentes séquences de fin de ligne possibles (`'\n'` line-feed ; `'\r'` carriage-return ; `"\r\n"` carriage-return & line-feed).

`println` insère automatiquement le symbole de fin de ligne du système d'exploitation actuel.

Voici une classe qui sait copier un fichier ligne par ligne en remplaçant les symboles de fin de ligne par les symboles du système d'exploitation actuel. On peut donc utiliser ce programme pour convertir un fichier texte avec des retours à la ligne d'un autre SE (p.ex. MacOS ou Linux) dans un fichier texte avec des retours à la ligne du système actuel (p.ex. Windows).

Nous voyons ici qu'il est possible d'intégrer l'ouverture et la fermeture de plusieurs fichiers en même temps :

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        try (
            BufferedReader in = new BufferedReader(new FileReader("mytext.txt"));
            PrintWriter out = new PrintWriter(new FileWriter("mytext_copy.txt",
                                                            StandardCharsets.UTF_8))
        ) {
            String line;
            while ((line = in.readLine()) != null) {
                out.println(line);
            }
        }
    }
}
```

## 9.6. Les fichiers XML (pour avancés)

Un fichier XML (*Extensible Markup Language*) n'est rien d'autre qu'un simple fichier texte qui suit des règles bien définies<sup>7</sup>. Il s'en suit que l'écriture d'un fichier XML n'est rien d'autre que l'écriture d'une chaîne de caractères suivant ces règles, qui sont – en résumé – les suivantes :

- La première ligne indique la version XML utilisée ainsi que l'encodage du fichier.
- Chaque fichier XML possède un élément de base unique. En anglais on parle du « root element ».
- Les noms des balises (EN: *Tags*) peuvent être librement choisis mais doivent être conformes aux normes communément connues. Les noms des balises et attributs sont généralement écrits en minuscules, respectivement suivent la notation Camel<sup>8</sup> telle qu'elle est aussi connue en Java.
- Toute balise qui est ouverte doit obligatoirement être fermée !
- Tout comme en HTML, certains caractères tels que le « & » ou « < », doivent être encodés :
  - & → &amp;
  - < → &lt;
  - ...
- Un fichier XML peut suivre une définition de document type<sup>9</sup>. On parle de « DTD » (*Document Type Definition*). Or ceci n'est pas nécessaire ni traité dans le présent cours.
- On peut aussi rattacher une feuille de style XSL (*Extensible Stylesheet Language*)<sup>10</sup> à un fichier XML. Ceci n'est pas non plus traité dans le présent cours, mais il est bon de savoir que cela existe !

### **Exemple :**

```
<?xml version="1.0" encoding="UTF-8" ?>
<dots>
  <alDots>
    <dot>
      <x>1</x>
      <y>1</y>
      <color>
        <red>255</red>
        <green>255</green>
        <blue>0</blue>
        <alpha>255</alpha>
      </color>
    </dot>
    <dot>
      <x>2</x>
      <y>1</y>
      <color>
```

7 <http://en.wikipedia.org/wiki/XML>

8 <http://en.wikipedia.org/wiki/CamelCase>

9 [http://en.wikipedia.org/wiki/Document\\_Type\\_Definition](http://en.wikipedia.org/wiki/Document_Type_Definition)

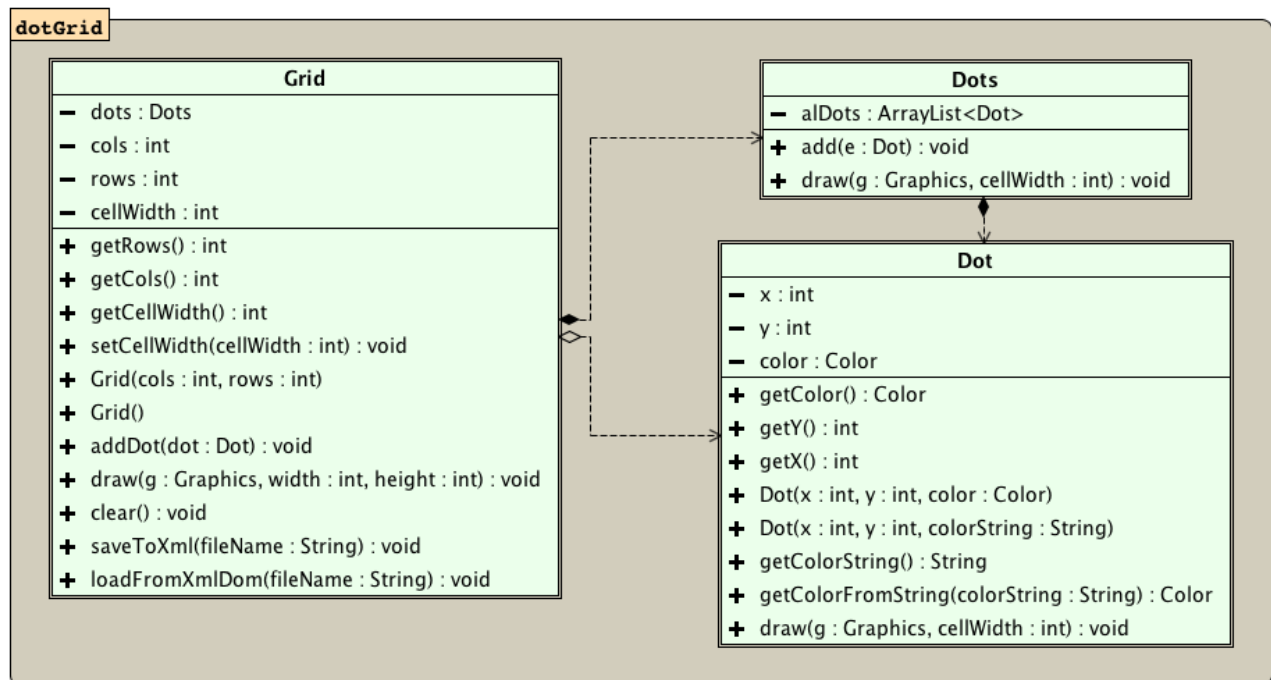
10 <http://en.wikipedia.org/wiki/XSL>

```

        <red>255</red>
        <green>0</green>
        <blue>0</blue>
        <alpha>255</alpha>
    </color>
</dot>
...
</alDots>
</dots>

```

Le fichier XML ci-dessus représente une grille avec des points colorés à certaines positions. Le diagramme de classe UML correspondant pourrait être le suivant :



### 9.6.1. Les analyseurs syntactiques

Afin de lire un fichier XML, il existe deux méthodes :

- L'analyseur syntaxique (EN : *parser*) du type « DOM » (*Document Object Model*) copie la structure entière d'un fichier XML dans la mémoire sous forme d'un grand arbre. L'avantage est qu'on aura à tout moment accès à toutes les informations. Elle convient pour de petits fichiers, tandis que la consommation de mémoire sera trop importante pour des fichiers très grands.
- L'analyseur syntaxique « SAX » (*Simple API for XML*) n'a pas besoin de copier en mémoire le fichier entier. Il parcourt le fichier séquentiellement et jette des événements lors de la rencontre des différents éléments du fichier. Le programme peut réagir ou non aux différents événements lancés. Ceci a l'avantage d'être utilisable même pour de gigantesques fichiers. Le désavantage est que l'entièreté des informations n'est pas accessible à tout moment.

Pour simplifier, nous allons seulement employer l'analyseur **Dom** dans ce cours.

### 9.6.2. La classe XStream

Pour simplifier la lecture et l'écriture de fichiers, nous allons faire recours à la bibliothèque **XStream** qui a l'avantage d'effectuer automatiquement l'analyse des objets à sauvegarder et elle sait rétablir les objets à partir du fichier et de la définition des classes. Ceci est possible grâce au '**reflection engine**' (→ **java.lang.reflect**) de Java, qui permet à un programme Java d'accéder à la définition des classes et de tous leurs composants.

Utilisation de la bibliothèque :

- Téléchargez d'abord *XStream* du site <http://x-stream.github.io>
- Créez un dossier **lib** dans le dossier du projet où vous voulez utiliser des fichiers XML. Copiez-y le fichier **jar** principal de *XStream* (p.ex. : **xstream-1.4.8.jar** , le nom change avec la version)
- Ouvrez le projet en NetBeans et ajoutez le fichier **jar** à votre projet : clic droit sur '**Libraries**', puis sélectionnez 'Add JAR/folder...'. Veillez à faire une référence relative au fichier.
- Dans la classe qui contiendra les opérations de lecture et d'écriture, importez la classe *XStream* et celle du gestionnaire *Dom* : *com.thoughtworks.xstream.XStream*, *com.thoughtworks.xstream.io.xml.DomDriver*

#### Écriture du fichier :

Une version de base pour écrire le fichier avec les points colorés peut se présenter comme suit :

```
public void saveToXml(String fileName) throws IOException {
    XStream xstream = new XStream(new DomDriver("UTF-8"));

    String xml = xstream.toXML(dots);

    try (PrintWriter out = new PrintWriter(new FileWriter(fileName))) {
        out.println("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
        out.println(xml);
    }
}
```

Réalisez ce programme et essayez de rapprocher la représentation xml de celle montrée plus haut en employant les instructions **xstream.setMode(...)** et **xstream.alias(...)**

#### Lecture du fichier :

Une version de base pour écrire le fichier avec les points colorés peut se présenter comme suit :

```
public void loadFromXmlDom(String fileName) throws IOException {
    XStream xstream = new XStream(new DomDriver("UTF-8"));
    try (BufferedReader in = new BufferedReader(new FileReader(fileName))) {

        dots = (Dots)xstream.fromXML(in);

    } catch (com.thoughtworks.xstream.mapper.CannotResolveClassException e) {
        JOptionPane.showMessageDialog(null,
            "Cannot resolve XML Tags in file '"+fileName+"'",
            "XML read Error", JOptionPane.ERROR_MESSAGE);
    }
}
```



*Rétablissez les objets à partir du fichier xml !*

## 9.7. Localisation des fichiers

### 9.7.1. Dialogues d'ouverture et de sauvegarde

Pour ouvrir ou sauvegarder des fichiers, il est pratique d'employer des dialogues prédéfinis de Java qui permettent de choisir des fichiers sur un support électronique (disque, USB, réseau, etc.). Ces dialogues n'effectuent pas d'action de sauvegarde ou de lecture : elles nous permettent uniquement de choisir un chemin et un nom de fichier de façon confortable.

Pour cela, nous pouvons utiliser la classe **JFileChooser** et ses méthodes :

Constructeur	<b>JFileChooser</b> ( <i>&lt;fichier initial&gt;</i> )
int <b>showSaveDialog</b> ( <i>&lt;parent&gt;</i> )	ouvre un dialogue de sauvegarde. <i>&lt;parent&gt;</i> définit la fiche qui possède le dialogue, ce paramètre influence la présentation et la position du dialogue. Le plus souvent, nous utiliserons <b>this</b> ou <b>null</b> pour ce paramètre. La valeur entière retournée par la méthode peut avoir l'une des valeurs suivantes : <b>JFileChooser.CANCEL_OPTION</b> : le dialogue a été fermé avec le bouton 'Cancel' ou la touche 'escape'. L'action doit donc être annulée. <b>JFileChooser.APPROVE_OPTION</b> : le dialogue a été fermé avec le bouton 'Ok' ou la touche 'enter'. L'action doit donc être exécutée. <b>JFileChooser.ERROR_OPTION</b> : le dialogue a été fermé avec une erreur. L'action doit donc être annulée.
int <b>showOpenDialog</b> ( <i>&lt;parent&gt;</i> )	ouvre un dialogue d'ouverture de fichier. - voir <b>showSaveDialog</b> -
File <b>getSelectedFile</b> ()	retourne le fichier sélectionné dans le dialogue
String <b>getName</b> ( <i>&lt;File&gt;</i> )	retourne le nom (avec le chemin) du fichier donné comme paramètre
<b>addChoosableFileFilter</b> ( <i>&lt;FileFilter&gt;</i> )	Réduit la liste des fichiers affichés en affichant uniquement les fichiers donnés par le ou les filtres. <b>- Définir des filtres : voir ci-dessous -</b>
<b>setAcceptAllFileFilterUsed</b> (false)	supprimer le filtre "All Files" qui permet d'afficher tous les fichiers

### 9.7.2. Définition d'un filtre de fichiers :

Il est possible de définir des filtres de fichiers personnels en surchargeant les méthodes **getDescription** et **accept** de **FileFilter**. Depuis Java 6, il existe cependant une méthode beaucoup plus confortable : la classe **FileNameExtensionFilter** permet de définir des filtres de fichiers personnalisés en une seule instruction :

```
FileFilter ffilter = new FileNameExtensionFilter("JPEG file", "jpg", "jpeg");
```

crée un filtre décrit comme "JPEG File" qui limite l'affichage aux fichiers portant les extensions *jpg* ou *jpeg* (en majuscules ou en minuscules).

Autres exemples :

```
FileFilter ff = new FileNameExtensionFilter("Images", "jpeg", "jpg", "gif", "png");  
FileFilter ff = new FileNameExtensionFilter("Text files (*.txt)", "txt");
```

#### Exemple pour l'ouverture d'un fichier :

```
JFileChooser fc = new JFileChooser(currentFile); //currentFile du type File  
fc.setAcceptAllFileFilterUsed(false);  
fc.addChoosableFileFilter(new FileNameExtensionFilter("Draw files", "drw"));  
  
if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {  
    currentFile = fc.getSelectedFile();  
    drawPanel.loadFromFile(fc.getName(currentFile));  
}
```

Pour obtenir le dossier de travail du projet actuel (ou du programme actuel), vous pouvez vous référer aux informations du système :

```
System.getProperty("user.dir")
```

Pour obtenir le dossier des documents de l'utilisateur, vous pouvez employer :

```
System.getProperty("user.home")
```

Dans la plupart des projets, c'est une bonne idée d'employer un attribut '**currentFile**' et de l'initialiser avec l'un des deux chemins décrits ci-dessus. Ainsi, l'utilisateur n'a pas besoin de parcourir chaque fois tout le chemin jusqu'aux fichiers qui appartiennent à votre projet.

## 9.8. Opérations sur fichiers et chemins d'accès

Jusqu'ici, nous avons travaillé avec des fichiers, sans nous occuper de leur existence ou conditions d'accès. La classe **java.io.File** permet de travailler simplement avec des fichiers et des chemins d'accès.

Voici un bref aperçu sur les méthodes les plus usuelles disponibles pour un objet du type **File** :

- Un objet du type **File** peut être construit à partir d'un nom de chemin (avec des séparateurs '\') ou d'un objet **URI** (descripteur universel de ressources).
- Pour obtenir le chemin absolu d'un fichier, on peut employer la méthode **getAbsolutePath** . P.ex :

```
String path = openFileDialog.getSelectedFile().getAbsolutePath()
```

- La méthode **exists** permet de tester si un tel fichier existe déjà.
- Les méthodes **canExecute**, **canRead**, **canWrite** permettent de détecter les droits d'accès de l'utilisateur sur un fichier.
- La méthode **delete** permet d'effacer un fichier.
- Les méthodes **isDirectory** et **isFile** permettent de détecter si on a à faire à un fichier ou à un dossier.
- La méthode **length** retourne la taille du fichier en octets ou zéro si le fichier n'existe pas.
- La méthode **mkdir** et **mkdirs** permettent de créer un dossier en créant ou non les sous-dossiers qui manquent.
- Les méthodes **getTotalSpace** et **getFreeSpace** retournent en octets l'espace total et l'espace encore disponible sur le support du fichier.
- La méthode **renameTo** permet de renommer un fichier.

### Exemple :

Voici une méthode statique qui retourne **true** si un fichier donné existe :

```
public static boolean fileExists(String fileName) {  
    java.io.File f = new java.io.File(fileName);  
    return f.exists();  
}
```

## 9.9. Fichiers de ressources dans un projet (pour avancés)

Souvent, on a besoin de ressources supplémentaires (fichiers son, texte ou image) pour faire fonctionner un projet. Ces fichiers doivent être disponibles dès que le programme démarre et ceci aussi bien si le projet est en développement que plus tard dans la version compilée (fichier **.jar**).

Exemple pour ajouter un dossier contenant des images à un projet :

Ajouter un dossier **pics** au dossier **src** de votre projet. Ainsi, lors de la compilation, ce dossier sera ajouté automatiquement au dossier '**build/classes**' et dans le dossier racine du fichier '**jar**'. L'accès aux fichiers (images) se fait alors par :

```
try {
    URL myurl = this.getClass().getResource("/pics/Picture1.png");
    image = ImageIO.read(myurl);
} catch (IOException ex) {
    ...
}
```

Exemple pour un fichier texte qui se trouve sous '**.../src/resources/text.csv**' :

```
InputStream inStream = getClass().getResourceAsStream("/resources/text.csv");
if(inStream == null)
    throw new FileNotFoundException();
else
    try (BufferedReader in=new BufferedReader(new InputStreamReader(inStream)))
        {
            ...
        }
```

### 9.10. Charger au démarrage et sauvegarder à la fin

Parfois il est utile de charger un fichier dès le démarrage d'un programme et de le sauvegarder dès que le programme est fermé.

#### Exemples :

- Actualiser les meilleurs scores d'un jeu,
- mémoriser et rétablir l'état des options (préférences) de l'utilisateur.

Pour charger des fichiers au démarrage, il suffit d'ajouter les opérations nécessaires au constructeur de la classe principale.

Pour sauvegarder des fichiers lors de la fermeture, le cas est plus difficile, puisqu'il faut détecter l'action de fermeture et la retarder jusqu'à ce que le fichier soit écrit.

Une solution serait de définir l'événement **WindowClosing** de la fiche principale du programme et d'y effectuer toutes les opérations nécessaires.

**Attention** : L'événement **WindowClosing** est lié à la fiche et n'est pas évalué lorsqu'on ferme la fiche de façon forcée (p.ex. par *System.exit(0)*, *Task Manager*, *Force quit*, ...).

Une autre possibilité plus sûre, p.ex. pour sauvegarder les meilleurs scores, est de sauvegarder le fichier à chaque changement.

#### Remarque pour avancés :

Pour sauvegarder les préférences de l'utilisateur, il existe un paquet spécial **java.util.prefs** qui permet de sauvegarder les préférences à l'endroit prévu par la plateforme sur laquelle tourne le programme. Le programmeur n'a pas besoin de s'occuper des détails de l'implémentation. Il existe deux arborescences pour les préférences : préférences 'utilisateur' et 'système'. La majorité de nos informations devra être sauvegardée dans l'arborescence des préférences 'utilisateur'. Pour les détails, recherchez l'aide et le tutoriel sur '*Preferences API*'.

## 10. Annexe A - Applications Java sur Internet

Le présent chapitre reprend deux procédés, expliqués étape par étape, pour publier une application Java sur Internet. Son contenu ne fait pas partie du programme officiel, mais devrait être vu plutôt comme une source de motivation ou d'inspiration (p.ex. pour un projet).

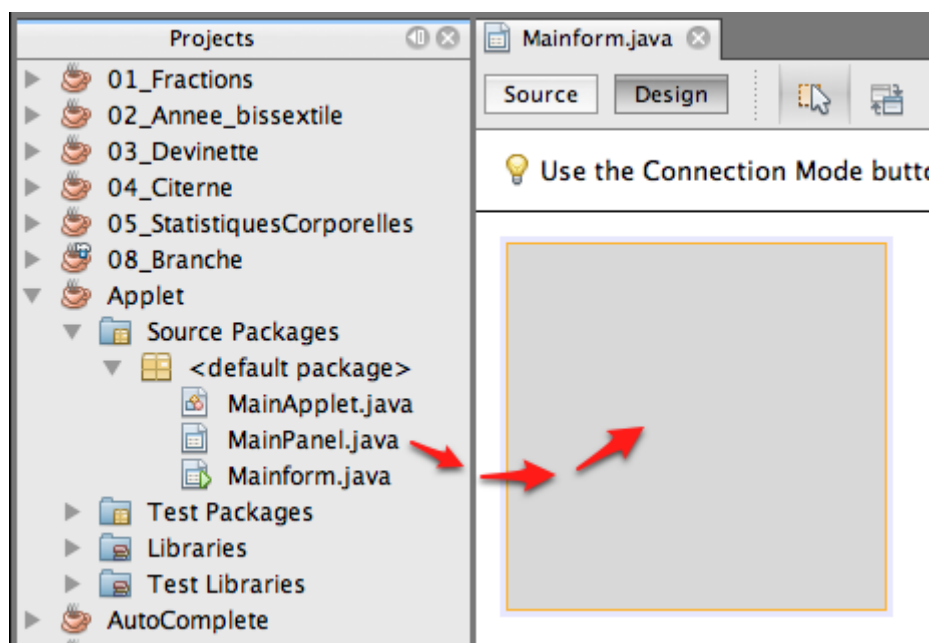
### 10.1. Java Applet

Une « applet » est une application Java intégrée dans une page web. Afin de créer une telle page web, il faut suivre les étapes suivantes :

1. Créer une application Java vierge.
2. Ajouter une fenêtre « **JFrame Form...** » vierge au projet en l'appelant **MainFrame**.
3. Ajouter un panneau « **JPanel Form...** » au projet en l'appelant **MainPanel**.
4. Ajouter ses composants au formulaire **MainPanel** et programmer les méthodes de réactions dans **MainPanel**.

Ceci est en fait l'étape dans laquelle vous écrivez votre application. La seule différence avec les exercices du cours est celle, que vous écrivez votre application dans une classe du type « panneau » et non dans une classe du type « fenêtre ». Ceci est nécessaire afin de pouvoir intégrer le panneau dans l'applet.

5. Compiler le projet à l'aide du menu : **Run > Clean and Build Main Project**
6. Afficher **MainFrame** dans la vu « **Design** ».
7. Tirer à l'aide de la souris **MainPanel** à partir de l'arbre de la partie gauche de la fenêtre sur la **MainFrame**.



8. Ajouter une « **Java Class...** » au projet en l'appelant **MainApplet**.
9. Copier et coller le code suivant dans la classe **MainApplet** :

```
public class MainApplet extends JApplet
{
    @Override
    public void init()
    {
        try
        {
            UIManager.setLookAndFeel (
                UIManager.getCrossPlatformLookAndFeelClassName());
            this.getContentPane().add(new MainPanel());
        } catch (Exception ex) {}
    }
}
```

10. Compiler le projet à l'aide du menu : **Run > Clean and Build Main Project**
11. Copier le contenu du répertoire « **dist** » de votre projet sur le serveur web.
12. Ajouter au même endroit le fichier « **applet.html** » avec le contenu que voici :

```
<html>
  <head>
    <title>Titre de votre application</title>
  </head>
  <body>
    <applet archive="Applet.jar,lib/swing-layout-1.0.4.jar"
            code="MainApplet"
            width="680" height="500"></applet>
  </body>
</html>
```

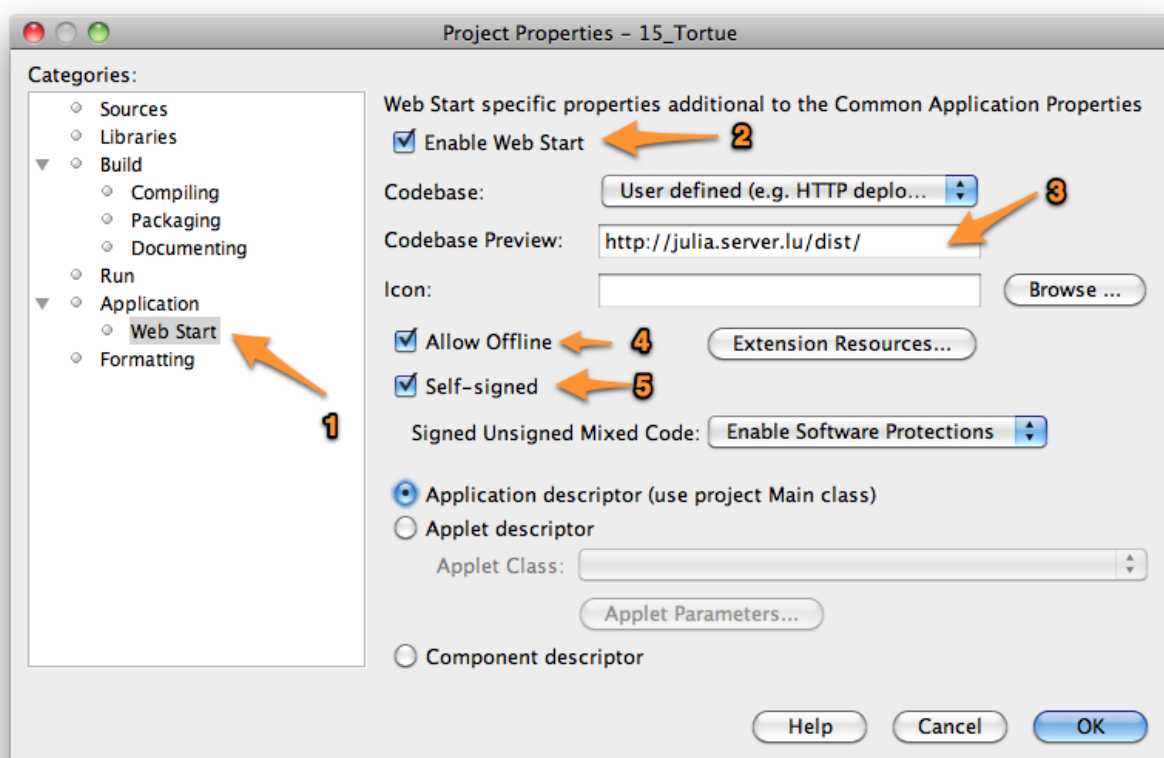
Si vous voulez, vous pouvez encore adapter le nom de la page en remplaçant « **Titre de votre application** » par un nom de votre choix. De même, vous pouvez adapter la taille de la fenêtre applet en modifiant les paramètres **width** et **height** y relatifs.



## 10.2. Java Web Start Application

Une application Java du type « **Java Web Start** » (JWS) est une application Java tout à fait standard mais qui a été compilée spécialement pour être lancée via Internet. Afin de créer une telle application JWS, il faut suivre les étapes suivantes :

1. Créer une nouvelle application ou en ouvrir une qui existe.
2. Faire un clic droit sur le projet en question et choisir la dernière entrée du menu contextuel dénommée « **Properties** ».
3. Dans la fenêtre qui s'ouvre ensuite :



1. Choisir dans la partie droite « **Web Start** », puis
2. configurer toutes les options telles qu'indiquées sur la capture d'écran **mais**
3. remplacer l'URL entrée sous (3) par celle du serveur sur lequel vous voulez publier votre application.
4. Compiler le projet à l'aide du menu : **Run > Clean and Build Main Project**
5. Transférer tout le contenu du répertoire **dist** de votre projet à l'emplacement de votre serveur que vous avez indiqué sous (3) dans les propriétés « **Web Start** ».

**Attention :** Si la case « **Enable Web Start** » est cochée, votre application ne démarre plus à l'aide du bouton « **Run** » ! Avant de pouvoir lancer votre application à nouveau en local, il faut désactiver JWS.

## 11. Annexe B - Impression de code NetBeans

En Unimozzer, 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.

### 11.1. Impression à l'aide du logiciel « JavaSourcePrinter »

Le logiciel **JavaSourcePrinter** peut être lancé par Webstart sur le site [java.cnpi.lu](http://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...**).

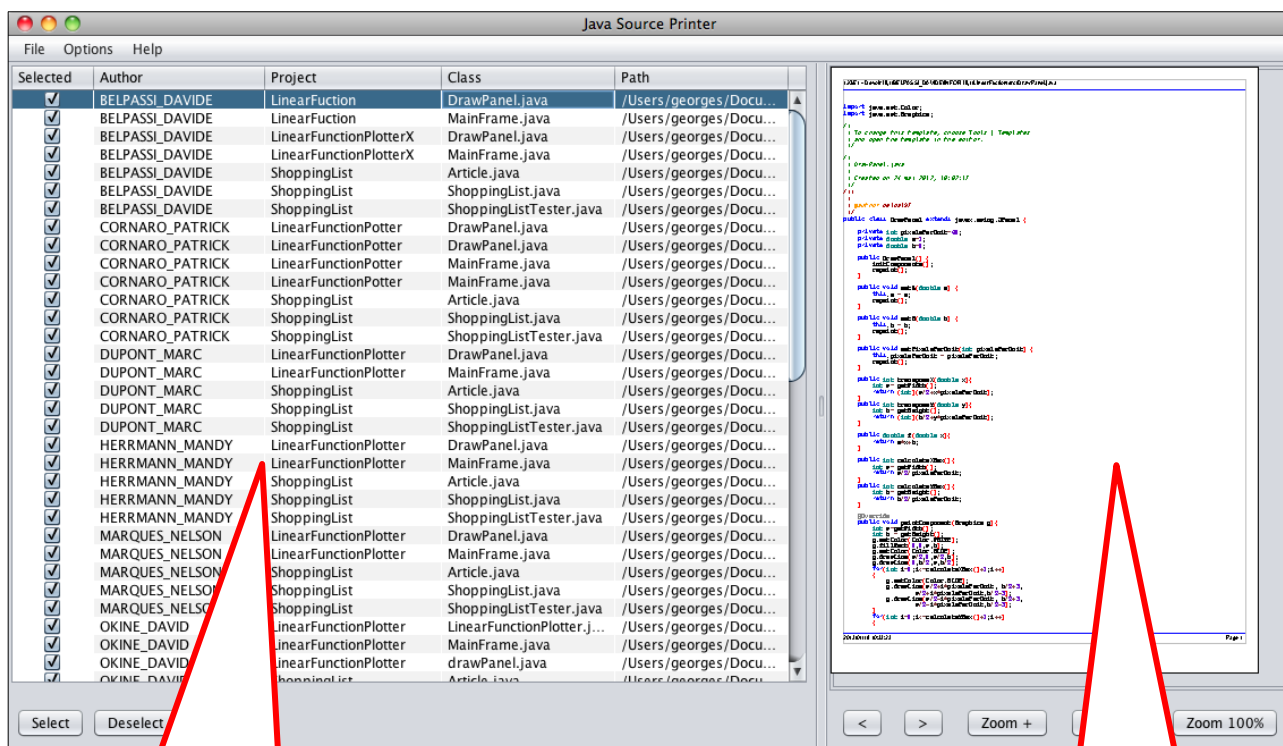
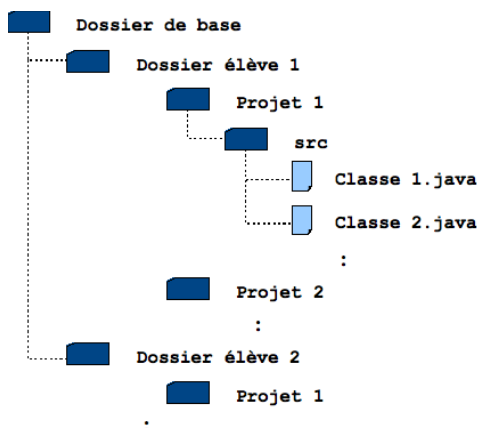


Table de sélection pour impression

Panneau de prévisualisation


## 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 désélectionner 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.



Selected	Author	Project	Class	Path
<input checked="" type="checkbox"/>	BELPASSI_DAVIDE	LinearFunctionPlotterX	DrawPanel.java	/Users/georges/
<input checked="" type="checkbox"/>	BELPASSI_DAVIDE	LinearFunctionPlotterX	MainFrame.java	/Users/georges/
<input checked="" type="checkbox"/>	BELPASSI_DAVIDE	ShoppingList	Article.java	/Users/georges/
<input checked="" type="checkbox"/>	BELPASSI_DAVIDE	ShoppingList	ShoppingList.java	/Users/georges/
<input checked="" type="checkbox"/>	BELPASSI_DAVIDE	ShoppingList	ShoppingListTester.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	LinearFunctionPotter	DrawPanel.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	LinearFunctionPotter	MainFrame.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	ShoppingList	Article.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	ShoppingList	ShoppingList.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	ShoppingList	ShoppingListTester.java	/Users/georges/
<input checked="" type="checkbox"/>	DUPONT_MARC	LinearFunctionPlotterX	DrawPanel.java	/Users/georges/

Selected	Class	Author	Project	Path
<input checked="" type="checkbox"/>	Article.java	BELPASSI_DAVIDE	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	CORNARO_PATRICK	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	DUPONT_MARC	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	HERRMANN_MANDY	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	MARQUES_NELSON	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	OKINE_DAVID	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	POVEROMO_CEDRIC	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	REUTER_JOANNA	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	SABOTIC_EDIS	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	SABOVIC_MUHAMED	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	SINNER_PIT	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	Article.java	SPINELLI_VANESSA	ShoppingList	/Users/georges/
<input checked="" type="checkbox"/>	DrawPanel.java	BELPASSI_DAVIDE	LinearFunctionPlotterX	/Users/georges/

## 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**

## 11.2. 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.

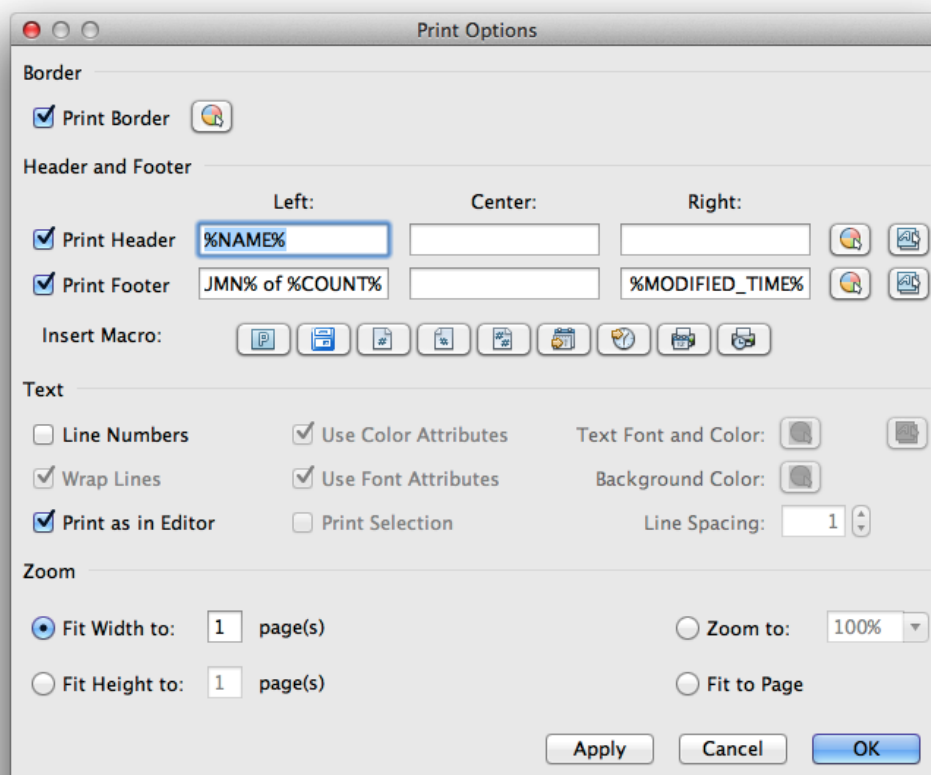
Cliquez d'abord sur **File** → **Print** et vous vous retrouvez dans le mode '**Print Preview**' qui vous permet de vérifier la disposition des pages.

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.

## 12. Annexe C - 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 <sup>11</sup>:

### 12.1. 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.

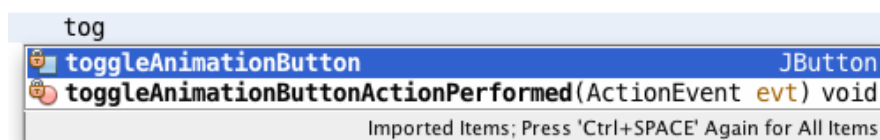
Mais attention : ces suggestions ne correspondent pas toujours à nos intentions ou besoins. Ceci compte surtout pour les valeurs des paramètres que NetBeans propose...

### 12.2. 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 Unimozzer) 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>.



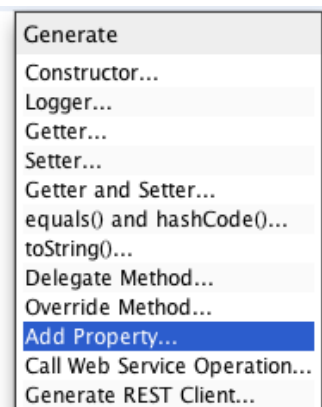
### 12.3. Ajout de propriétés et de méthodes <Insert Code...>

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>



<sup>11</sup> 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**.

## 12.4. 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 :

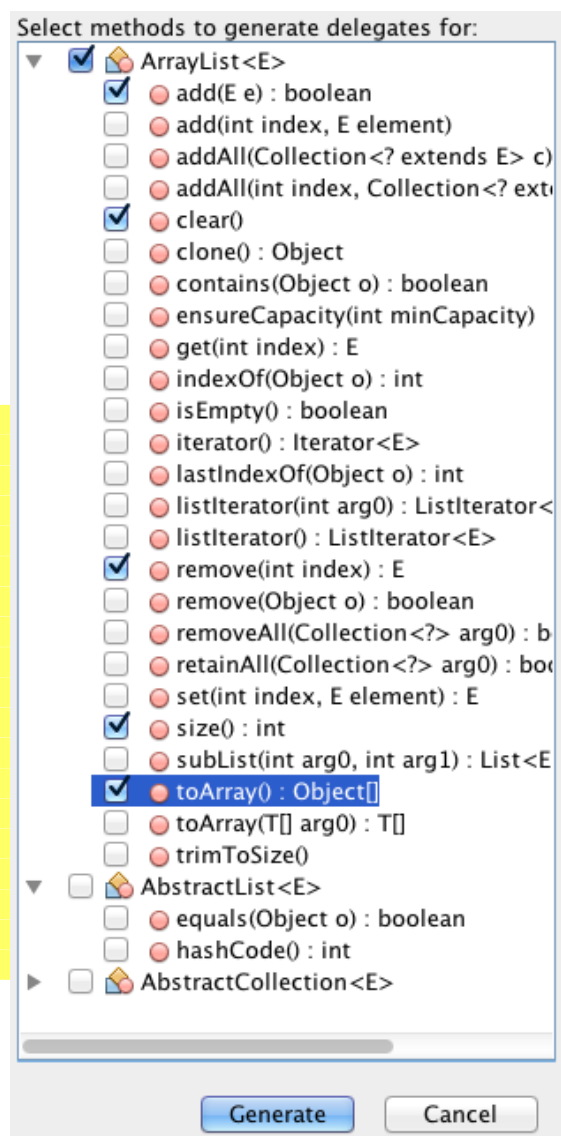
```
public int size() {
    return allLines.size();
}

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();
}
```





## 12.5. 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.

## 12.6. 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.

## 12.7. 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 :







## 13. Annexe D - Accès à JavaDoc sur le disque local

Par défaut, NetBeans référence JavaDoc, le système d'aide de Java, directement en-ligne, sur le site de Oracle. Pour les cas où vous n'avez pas d'accès à Internet il est donc pratique d'utiliser JavaDoc sur votre disque local. Pour ceci, il faut :

- Télécharger et sauvegarder le fichier comprimé contenant les fichiers de JavaDoc.
- Instruire NetBeans d'utiliser le fichier local au lieu de se référer à la documentation en-ligne.

### 13.1. Télécharger la documentation

C'est effectivement plus dur de retrouver la documentation sur internet que de l'installer sous NetBeans. Actuellement, la méthode la plus fiable est de commencer sur la page des téléchargements Java de Oracle :

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Sous '**Additional resources**' on peut télécharger la version actuelle de JavaDoc.

Par exemple : la description pour la version Java 8 (update 92) est

#### ***Java SE Development Kit 8u92 Documentation***

et le fichier à télécharger s'appelle alors :

***jdk-8u92-docs-all.zip***

Sauvegardez le fichier **zip** sur votre disque **sans** le décompresser à un endroit où NetBeans pourra l'accéder.

(Evidemment on peut aussi faire une recherche sur internet pour '**java documentation download**' et avec un peu de chance on trouve rapidement la page de téléchargement.)


### 13.2. Instruire NetBeans à utiliser le fichier JavaDoc local

- En NetBeans, saisir le point '**Java Platforms**' dans le menu '**Tools**',
- choisir l'onglet **JavaDoc**,
- choisir '**Add Zip/Folder**',
- retrouver sur votre disque le fichier **zip** que vous avez sauvegardé,
- cliquer sur le bouton '**Add Zip/Folder**',
- avec '**Move up**' déplacer au début de la liste les liens que vous avez ajoutés (les liens sur internet peuvent rester en bas de la liste).

## 14. Annexe E - Live Debugging

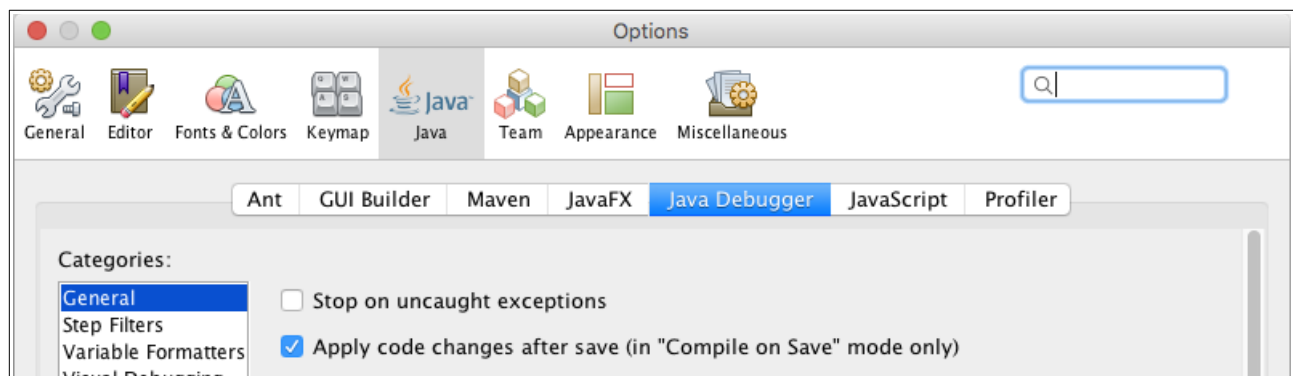
NetBeans offre la possibilité de modifier le code pendant une session de débogage et ainsi de tester des modifications sans devoir recompiler ou redémarrer le programme. Ceci peut être pratique pour tester de petites modifications dans un projet plus volumineux.

Pour ce faire, on peut procéder comme suit :

- démarrer le programme en mode de débogage ,
- effectuer les modifications (il n'est pas possible d'ajouter ou de créer des méthodes),
- sauvegarder,
- cliquer '**Apply code changes**' dans le menu de débogage,
- si nécessaire, activer *repaint* (p.ex. en redimensionnant le formulaire).

On peut ajouter un peu de confort en automatisant l'étape '*Apply code changes*' comme suit :

- dans les préférences de NetBeans choisir **Java**,
- choisir l'onglet **Java Debugger**,
- activer : "**Apply code changes after save (in "Compile on Save" mode only)**".



Maintenant, il suffit de modifier le code et de pousser sur 'save' pour que les modifications soient appliquées.

[Par défaut dans tous les projets NetBeans, l'option '**Compile on Save**' est déjà activée, mais si vous voulez vérifier, elle se trouve dans les propriétés du projet sous **Build-Compiling**.]