

Science de la programmation



Java™



Table des matières

1. Notions importantes.....	9
1.1. Qu'est-ce qu'un « objet » ?.....	9
1.2. Exemples d'introduction.....	10
1.2.1. Exemple 1 : Voitures.....	10
1.2.2. Exemple 2 : Personnes.....	11
1.3. Qu'est-ce qu'une « classe » ?.....	12
1.4. Qu'est-ce qu'un « attribut » ?.....	14
1.5. Qu'est-ce qu'une « méthode » ?.....	15
1.5.1. L'instruction return.....	16
1.5.2. Le type void.....	16
1.6. Qu'est-ce qu'un « paramètre » ?.....	19
1.7. Qu'est-ce qu'un « type » ?.....	22
1.8. Conventions de noms pour les méthodes.....	24
1.8.1. Les accesseurs - préfixe get.....	24
1.8.2. Les manipulateurs - préfixe set.....	24
1.8.3. Les méthodes effectuant un calcul - préfixe calculate.....	24
1.8.4. Les méthodes booléennes - préfixes is/has.....	24
1.8.5. La méthode toString.....	24
1.9. Qu'est-ce qu'un « constructeur » ?.....	25
1.10. Qu'est-ce qu'une « variable » ?.....	26
1.11. Opérateurs, compatibilité et conversions.....	28
1.11.1. Opérateur d'affectation « = ».....	28
1.11.2. Opérateurs arithmétiques.....	28
1.11.3. Conversions forcées de types (explicites).....	29
1.11.4. Conversions automatiques de types (implicites).....	29
1.11.5. Opérateur de concaténation « + ».....	31
1.12. Affichage d'une ligne de texte.....	31
1.13. La classe « Math ».....	32
1.14. Générer des nombres aléatoires.....	33
2. Présentation et documentation du code.....	35
2.1. Qu'est-ce qu'un « commentaire » ?.....	35
2.1.1. Règles pour l'utilisation des commentaires « JavaDoc ».....	35
2.1.2. Précisions : les commentaires « JavaDoc » des classes.....	36
2.1.3. Précisions : les commentaires « JavaDoc » des méthodes	36
2.2. Qu'est-ce que « l'indentation » ?.....	37
3. Les structures de contrôle.....	38
3.1. La structure alternative.....	38
3.2. La structure alternative et le retour de résultats.....	40
3.3. Les opérateurs de comparaison.....	41
3.4. Les opérateurs logiques.....	42
3.5. La structure répétitive « tant que ».....	43

3.6. La structure répétitive « pour ».....	45
3.7. Les blocs et la durée de vie des variables.....	47
4. Annexe : Automatisation des tests des classes.....	48
4.1. Création d'objets avec «new».....	48
4.2. Saisie de données au clavier en mode texte.....	48
4.3. Démarrage automatique du programme.....	49
5. Quickstart avec NetBeans.....	50
6. Manipulation de NetBeans.....	53
6.1. Installation.....	53
6.2. Les projets.....	53
6.2.1. Créer un nouveau projet.....	53
6.2.2. Ajouter des classes à un projet.....	55
6.3. Importer un projet de Unimozzer dans NetBeans.....	56
6.4. Distribuer une application.....	56
7. Notions importantes.....	57
7.1. Les paquets.....	57
7.2. Le modèle « MVC ».....	58
7.2.1. Le modèle.....	58
7.2.2. La vue.....	58
7.2.3. Le contrôleur.....	59
7.2.4. Flux de traitement.....	59
7.2.5. Avantages du MVC.....	59
7.3. Création de nouveaux objets avec 'new'.....	60
7.4. La valeur 'null'.....	62
7.5. Égo-référence 'this'.....	63
8. Types primitifs et classes enveloppes.....	64
9. Les chaînes de caractères « String ».....	65
9.1.1. Déclaration et affectation.....	65
9.1.2. Conversions de types.....	65
9.1.3. Autre méthode importante.....	66
10. Comparaison d'objets.....	67
10.1. Comparaison deux objets du type String.....	67
10.2. Comparaison d'instances de classes enveloppes.....	67
10.3. Explications pour avancés.....	68
11. Interfaces graphiques simples.....	69
11.1. Les fenêtres « JFrame ».....	69
11.1.1. Attributs.....	69
11.1.2. Initialisations.....	70
11.2. Les composants standards.....	71
11.2.1. Attributs.....	71
11.2.2. Événement.....	71

11.3. Manipulation des composants visuels.....	72
11.3.1. Placer un composant visuel sur une fiche.....	72
11.3.2. Affichage et lecture de données.....	73
11.3.3. Édition des propriétés.....	73
11.3.4. Noms des composants visuels.....	73
11.3.5. Événements et méthodes de réaction.....	74
11.3.6. Attacher une méthode de réaction à un événement.....	75
11.3.7. Comment supprimer un événement.....	77
11.4. Les champs d'entrée « JTextField ».....	78
11.4.1. Attributs.....	78
11.4.2. Événements.....	78
11.5. Afficher une image.....	79
11.6. Autres composants utiles.....	79
11.6.1. Attributs.....	79
11.6.2. Événements.....	79
11.7. Les panneaux « JPanel ».....	80
11.7.1. Attributs.....	80
11.7.2. Accès au canevas.....	80
11.8. Confort et ergonomie de la saisie.....	81
12. Les listes.....	82
12.1. Les listes « ArrayList ».....	82
12.1.1. Listes de types primitifs.....	83
12.1.2. Méthodes.....	83
12.2. Les listes « JList ».....	86
12.2.1. Attributs.....	86
12.2.2. Événements.....	86
12.2.3. Préparer la liste pour accepter toute sorte d'objets.....	88
12.3. Les listes et le modèle MVC.....	90
13. Dessin et graphisme.....	91
13.1. Le canevas « Graphics ».....	92
13.1.1. La géométrie du canevas.....	92
13.1.2. Les méthodes du canevas.....	93
13.1.3. La méthode repaint().....	93
13.1.4. Affichage et alignement du texte.....	94
13.2. La classe « Color ».....	95
13.2.1. Constructeur.....	95
13.2.2. Constantes.....	96
13.3. La classe « Point ».....	96
13.3.1. Constructeur.....	96
13.3.2. Attributs.....	96
13.3.3. Méthodes.....	96
13.4. Les dessins et le modèle MVC.....	98
14. Annexe A - Applications Java sur Internet.....	100
14.1. Java Applet.....	100

14.2. Java Web Start Application.....	102
15. Annexe B - Impression de code NetBeans.....	103
15.1. Impression à l'aide du logiciel « JavaSourcePrinter ».....	103
15.2. Impression en NetBeans.....	106
16. Annexe C - Assistance et confort en NetBeans.....	107
16.1. Suggestions automatiques :.....	107
16.2. Compléter le code par <Ctrl>+<Space>.....	107
16.3. Ajout de propriétés et de méthodes <Insert Code...>.....	107
16.4. Rendre publiques les méthodes d'un attribut privé.....	108
16.5. Insertion de code par raccourcis.....	109
16.6. Formatage du code.....	109
16.7. Activer/Désactiver un bloc de code.....	109

Sources

- *Introduction à la programmation orienté objets*, 2011-2017, CNPI-Gdt-PrograTG
- *Introduction à Java*, Robert Fisch, 2009
- *Introduction à la programmation*, Robert Fisch, 11TG/T0IF, 2006
- *Introduction à la programmation*, Simone Beissel, T0IF, 2003
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006
- *Programmation en ANSI-C*, Fred Faber, T3IF, 1993-2006

Adaptations et actualisations

Robert Fisch, Fred Faber

Site de référence

<http://java.cnpi.lu>

A la base de ce cours est le document « *Introduction à la programmation orientée objets* » du groupe de travail PrograTG de la CNPI.

Le cours est adapté et complété en permanence.

Des documents supplémentaires sont accessibles sur le site : <http://java.cnpi.lu>

Introduction

Ce cours d'introduction à la programmation suit le principe "**objects first**" - "les objets d'abord" et ceci dans un double sens :

- au centre des intérêts se trouve la programmation orientée objet : tous les exemples ou exercices sont traités du point de vue orienté objet,
- le cours commence par l'introduction des principes et du vocabulaire orienté objet avant même de traiter les opérateurs et les structures de programmation.

Dans ce cours, nous allons d'abord développer des classes et manipuler des objets. Dans un premier temps sans nous occuper de l'interface ni de la saisie ou de la représentation des données.

Le logiciel Unimozzer joue le rôle de "banc d'essai" pour nos objets et nous permet de vérifier le bon fonctionnement de nos classes sans que nous n'ayons besoin de développer des applications autour de nos classes. Ainsi Unimozzer crée automatiquement des dialogues pour demander les valeurs et pour afficher des résultats.

Exemple : Le projet 'Cistern'

The screenshot shows the Unimozzer IDE. On the left, a yellow box displays the **Cistern** class with the following attributes and methods:

- Attributes:
 - maximumVolume : double
 - currentVolume : double
- Methods:
 - + Cistern(pRadius : double, pHeight : double)
 - + add(pVolume : double) : void
 - + drain(pVolume : double) : void
 - + getCurrentVolume() : double
 - + getCurrentRate() : double
 - + toString() : String

Below the class box, a red box shows the state of an object:

```
cistern0 : Cistern
maximumVolume = 6283.185307179586
currentVolume = 0.0
```

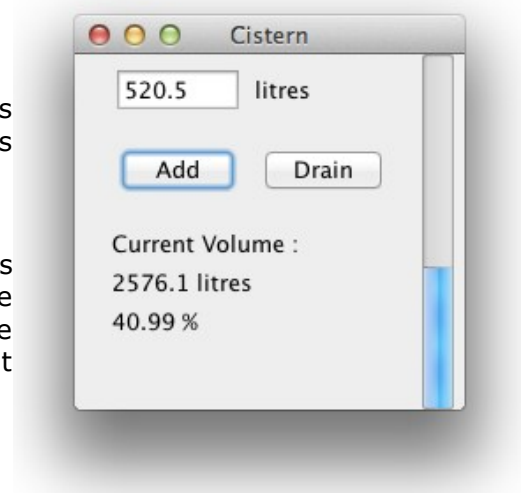
On the right, a dialog box titled "void add(double)" is open. It contains a label "pVolume (double) =" followed by a text input field containing "520.5". At the bottom are "Cancel" and "OK" buttons.

Lors du test de la méthode **add**, c'est Unimozzer qui crée automatiquement un dialogue pour nous demander le nombre de litres à ajouter à la citerne.

Ensuite, nous allons voir comment développer des applications avec des interfaces graphiques individuels pour visualiser et manipuler nos objets.

Exemple : Le projet 'Cistern'

Ce programme utilise la classe *Cistern* que nous avons développée, mais nous saurons construire une interface plus conviviale pour présenter et changer le contenu de la citerne. Ainsi, les détails de la réalisation resteront cachés à l'utilisateur.



Conventions :

Dans ce cours, nous allons utiliser des désignations anglaises pour tous les éléments que nous définissons dans le code de nos classes (noms des classes, attributs, méthodes, variables, ...).

Ainsi, nous évitons :

- d'une part des aberrations linguistiques (p.ex: **getValeur**, **setMoyenne**),
- d'autre part des incompatibilités dues par exemple à des caractères accentués.

De préférence, les textes retournés par nos objets ou affichés à l'écran seront des textes anglais. Les commentaires dans notre code seront de préférence en français et serviront donc aussi à expliciter les désignations anglaises moins connues.

1. Notions importantes

OOP - *Object Oriented Programming* - *objektorientierte Programmierung*

POO - *Programmation orientée objet*

Dans la programmation Java, nous allons manipuler des « objets » dont les « attributs » et les « méthodes » sont définis dans des « classes ». Dans ce chapitre, nous allons essayer d'éclaircir ce vocabulaire qui constitue le fondement de la programmation orientée objet.

1.1. Qu'est-ce qu'un « objet » ?

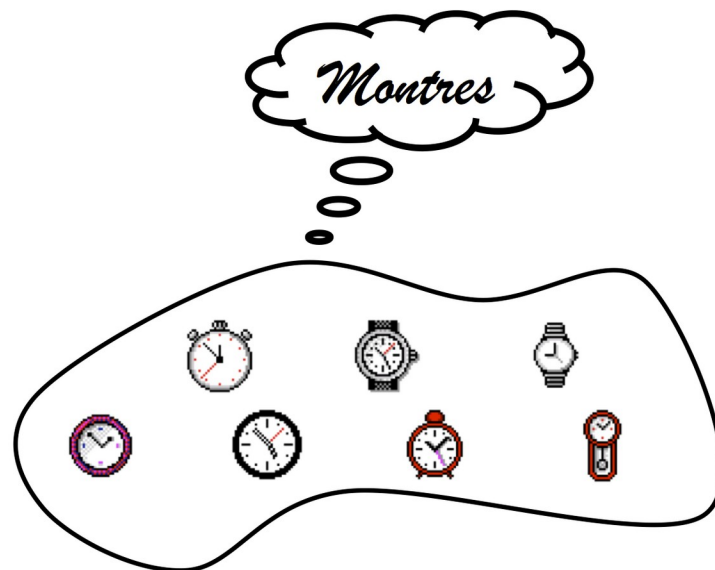
Dans la vie courante, toute chose, vivante ou non, peut être appelée *objet*. Les « objets » peuvent être une montre, un thermomètre, un client, un rendez-vous par exemple.

Nous pouvons définir un objet informatique comme suit :

Un **objet** est la représentation informatique d'un individu, d'un objet ou d'un concept de la vie réelle. Il est défini par ses caractéristiques propres (appelés « *champs* » ou « **attributs** ») et par son comportement (ses « **méthodes** »).

Cependant, chaque individu décrit un même objet différemment parce que chaque observateur a une autre personnalité, d'autres préférences, d'autres priorités, un autre point de vue.

Ainsi, chacun attribue d'autres importances aux caractéristiques d'un objet. On dit que chaque personne a une perception différente de la même chose. L'homme crée donc une image abstraite d'un objet dans sa tête. On dit qu'on a fait **abstraction**.



L'abstraction est le fait d'ignorer les propriétés marginales et de se concentrer sur les aspects essentiels. 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.

1.2. Exemples d'introduction

Avant de passer aux définitions plus formelles du vocabulaire de la POO, discutons quelques exemples pour comprendre l'approche de la modélisation d'objets dans la programmation.

1.2.1. Exemple 1 : Voitures

Trois personnes, Monsieur X, Madame Y et Monsieur Z, possèdent chacun leur propre voiture. Chacune de leurs voitures a 4 roues, un moteur, une certaine vitesse de pointe et ainsi de suite. De même, chacune de ces voitures peut accélérer, freiner, changer de direction, etc. On voit donc que chacune de ces trois voitures a le même type **d'attributs** et de **comportements** ou fonctionnalités. C'est pour cela qu'on dit que ce sont des voitures.

Et pourtant, les trois voitures ne sont pas identiques. En fait, voici une comparaison des trois voitures :

Propriétaire	Monsieur X	Madame Y	Monsieur Z
Peinture	verte	blanche	noire
Poids	1.500 kg	1.200 kg	1.700 kg
Puissance	100 kW	75 kW	170 kW
Vitesse de pointe	190 km/h	180 km/h	254 km/h
Accélérer	oui	oui	oui
Freiner	oui	oui	oui
Changer de direction	oui	oui	oui

Nous pouvons dire que les 3 véhicules partagent les mêmes types de caractéristiques d'un point de vue conceptionnel :

- chaque véhicule a un propriétaire, une certaine peinture, un certain poids, une certaine puissance et une certaine vitesse de pointe (→ **attributs**)
- chaque véhicule peut accélérer, freiner et changer de direction (→ **méthodes**).

Il y a bien sûr de nombreuses autres caractéristiques que nous allons ignorer dans cet exemple, afin de limiter la complexité (→ **abstraction**).

Même si les valeurs des attributs sont individuelles, les trois objets sont quand même des objets similaires qui ont la même identité : ce sont des voitures (et non pas des téléphones, des chaises ou des pommes).

On peut donc dire qu'il s'agit de trois **objets** différents, qui appartiennent tous à la même **classe**, appelée « Voiture ».

En d'autres mots, **la classe décrit le type d'objet**.

Un objet concret et spécifique est appelé une instance (une réalisation) d'une classe.

Ainsi l'objet « voiture de Monsieur X » est une instance de la classe « Voiture ». Il en est de même pour les deux autres voitures.

1.2.2. Exemple 2 : Personnes

Dans votre salle de classe, il y a un nombre d'individus, d'« objets vivants ».

- Chacun de ces objets a 2 bras, 2 jambes, une tête, une personnalité, un certain âge, un lieu de résidence, etc.
- De même, chacun de ces objets peut marcher, s'asseoir, courir, rire, parler, etc.

Il y a toute une série de types **d'attributs** et de comportements ou de **méthodes** que ces **objets** ont en commun. On peut donc qualifier tous les objets vivants dans cette salle d'êtres humains. On dit que ces objets sont des **instances** de la **classe** « être humain ».

On remarque à nouveau que chacun de ces objets se distingue des autres. Ainsi nous avons tous les mêmes types d'attributs et de méthodes, mais nos attributs n'ont pas toujours la même valeur. P. ex. certains ont des cheveux bruns, d'autres des cheveux noirs, d'autres des cheveux blonds. Nous avons des tailles et personnalités différentes, etc. Nous sommes donc tous des objets différents, mais nous faisons partie de la même classe.

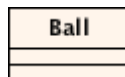
*Reprenons maintenant le vocabulaire de façon plus formelle et
intéressons-nous à la pratique de la POO...*

1.3. Qu'est-ce qu'une « classe » ?

Une **classe** est une description formelle d'une collection d'objets similaires, de même identité.

Prenons l'exemple d'une classe dénommée **Ball**.

En **UML (Unified Modeling Language)**, la **classe** est représentée par un diagramme de classe :



Le code source y relatif est le suivant :

```
public class Ball
{
}
```

Explications :

- Le mot clé « **public** » indique qu'il s'agit d'une classe publique, c'est-à-dire d'une classe qui est utilisable par d'autres programmes ou d'autres classes.
- Le mot clé « **class** » indique qu'il s'agit d'une classe.
- Le mot « **Ball** » est le nom de cette classe. Par convention la première lettre d'un nom de classe en Java est écrite en majuscules et le reste en minuscules. Des caractères spéciaux ne sont pas admis !
- Les accolades « { » et « } » indiquent le début, respectivement la fin de la classe.

Comme défini précédemment, une classe est une description formelle d'une collection d'objets de même identité. Il s'agit donc purement d'une description générale. **Un objet est une instance d'une classe, c'est-à-dire que l'objet est créé dans la mémoire de l'ordinateur en suivant exactement les définitions contenues dans la classe.** Tous les objets instanciés à partir d'une même classe possèdent le même fonctionnement.

Autre analogie pour clarifier ces deux vocables :

classe correspond au **plan** d'une maison

objet correspond à la **maison** construite selon le plan

En POO, nous allons donc définir d'abord une ou plusieurs classes. Lorsque nous voulons tester ou faire tourner un programme, nous devons créer une ou plusieurs instances de ces classes. Ce sont ces instances qui 'vivent' dans la mémoire de l'ordinateur et avec lesquelles nous pouvons interagir.

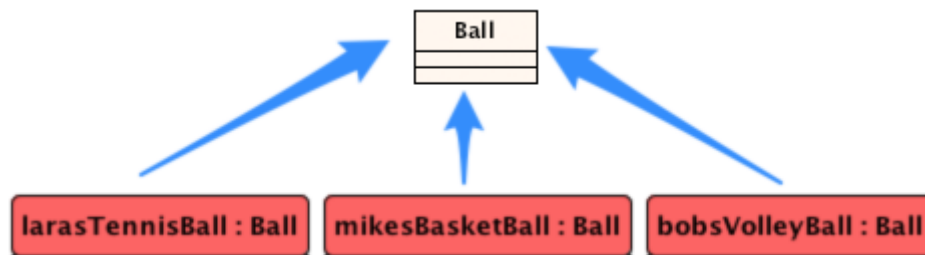
Comme désignations pour nos classes et objets, nous allons utiliser des noms ou des noms composés, p.ex. :

classes :	Person,	Client,	TennisBall,	SavingsAccount
objets :	person1,	client,	laurasTennisBall,	savingsAccount

En programmation, on dit qu'un **objet** est une « **instance** » d'une classe.

Le fait de créer un objet à partir d'une classe donnée s'appelle « **instanciation** ».

Reprenons l'exemple de la classe **Ball** :



Ball est la **classe** et contient la description abstraite (le plan) d'une balle dans le contexte de notre programme.

larasTennisBall, **mikesBasketBall** et **bobsVolleyBall** sont des **instances** de la classe **Ball**, donc des objets, c.-à-d. des balles concrètes avec leurs caractéristiques spécifiques (avec des valeurs différentes pour leur taille, leur couleur, leur pression).

L'environnement de développement (*Unimozer*)

Dans ce cours, nous utilisons un environnement de développement très simple (Unimozer) qui nous permet de construire des classes, de créer des instances, de voir leur contenu et de les tester.

Unimozer par exemple, affiche automatiquement la représentation UML correcte de la classe dans la partie gauche de l'écran. Le code Java se trouve dans la fenêtre à droite. L'outil actualise les données dans deux directions, c.-à-d. lorsque nous modifions le code Java, la représentation UML est actualisée automatiquement et vice-versa.

Exercice résolu :

- Si vous ne l'avez pas encore fait, alors créez un dossier pour vos exercices Java. Ouvrez votre interface de développement et ajoutez la classe publique **Ball**. Sauvez le projet sous '**00_Exercice_resolu**' dans votre dossier Java.
- Compilez la classe.
- Créez une nouvelle instance du nom de **larasTennisBall** de la classe **Ball**.
- Créez ensuite des nouvelles instances du nom de **mikesBasketBall** et **bobsVolleyBall**.

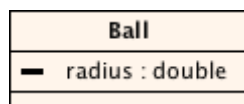
1.4. Qu'est-ce qu'un « **attribut** » ?

Un « **champ** » (angl. : *field*) ou « **attribut** » d'une classe est une propriété qu'elle possède. Voici quelques exemples courants :

- La couleur d'une voiture est une propriété.
- La puissance dissipée d'une ampoule électrique est une propriété.
- La nationalité, âge et la taille sont des propriétés d'une personne.
- Le diamètre, le poids, la pression, la couleur sont les propriétés d'une balle.

Chaque attribut possède un certain **type** (→ voir chapitre 1.7) et un certain **nom**.

En UML, un attribut est inscrit dans la deuxième case d'un diagramme de classe. Voici un exemple de la classe **Ball** qui possède un attribut **radius** du type **double** (nombre réel) :



Le code source y relatif est le suivant :

```
public class Ball
{
    private double radius = 0;
}
```

Explications :

- Les attributs d'une classe doivent être protégés afin qu'uniquement les méthodes de la classe elle-même y aient accès. C'est pour cette raison que toutes les déclarations d'attributs que nous utilisons commencent par le mot clé « **private** ».
- L'attribut représente ici un nombre réel, donc du type **double** (→ voir chapitre 1.7).
- **radius** est le nom de l'attribut.
Par convention les noms des attributs en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot (p.ex. **numberOfSides**).
- L'attribut **radius** est initialisé dès sa déclaration par la valeur zéro.
Une déclaration d'un attribut sans initialisation se lirait comme suit :

```
private double radius;
```

Exercice résolu :

Ajoutez l'attribut **pressure** du type **double** à la classe **Ball**. Cet attribut contient la pression de la balle. Initialisez-le par la valeur 2,50. Compilez la classe et créez quelques objets. Que remarquez-vous lorsque vous inspectez les objets ?

- Étapes à suivre dans Unimozer :
 - clic droit sur l'objet
 - Add field ...
 - puis entrez le nom et le type de l'attribut

1.5. Qu'est-ce qu'une « méthode » ?

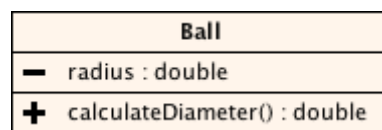
Une **méthode** est un sous-programme qui décrit une action que les instances de la classe savent effectuer. En termes informatiques, une méthode n'est rien d'autre qu'une suite d'instructions qui sont exécutées lorsqu'on fait appel à la méthode.

Une méthode peut retourner une valeur comme résultat ou ne rien retourner du tout (→ voir chapitre 1.5.2).

Le plus souvent, les méthodes changent le contenu des attributs de la classe ou utilisent les attributs de la classe pour effectuer une opération.

Exemple :

Donnons aux balles la capacité de nous fournir la valeur de leur diamètre :



En UML, une méthode est inscrite dans la troisième case d'un diagramme de classe :

En Java le code source y relatif est le suivant :

```
public class Ball
```

```
{
    private double radius = 11.95;

    public double calculateDiameter()
    {
        return 2 * radius;
    }
}
```

calculateDiameter est une méthode publique qui retourne une valeur décimale (du type double)

Explications :

- Le code source d'une méthode est indenté d'une tabulation par rapport au corps de la classe (→ voir chapitre 2.2).
- Le mot clé « **public** » indique qu'il s'agit d'une méthode publique, c'est-à-dire d'une méthode qui est utilisable par d'autres sous-programmes ou d'autres classes.
- Le mot clé « **double** » indique que la méthode retourne un nombre décimal (→ détails voir chapitre 1.7). En UML, le type du résultat est noté derrière le nom de la méthode (séparé par un « : »). Dans notre exemple: *calculateDiameter() : double*
- Le mot « **calculateDiameter** » est le nom de cette méthode.
Les noms des méthodes en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot. Les noms des méthodes contiennent toujours un verbe.
- Les accolades « { » et « } » indiquent le début, respectivement la fin de la méthode.
- Une méthode peut avoir besoin de paramètres (→ voir chapitre 1.6) qui sont alors indiqués entre parenthèses derrière le nom de la méthode. Dans notre cas, la méthode ne possède pas de paramètres. Nous devons indiquer ce fait par une paire de parenthèses vides **()** derrière le nom.

1.5.1. L'instruction **return**

Une méthode qui ne travaille pas seulement avec les attributs, mais qui retourne en plus un résultat doit contenir l'instruction **return**. Le type du résultat à fournir doit être noté devant le nom de la méthode. L'instruction **return** se trouve en général à la fin de la méthode.

L'instruction **return** sert à :

1. **retourner comme résultat** de la méthode la valeur (ou le résultat de l'expression) qui se trouve derrière le mot clé **return**. Cette valeur ou expression doit nécessairement avoir le même type que la méthode.
2. **quitter** immédiatement la méthode.

1.5.2. Le type **void**

Le type **void** (DE: *leer* / FR: *vide*) est un type spécial qui est uniquement utilisé lors de la déclaration de méthodes. Une méthode retourne le type **void** si elle effectue un travail sans pour autant retourner un résultat. Evidemment, une telle méthode ne possède pas d'instruction **return**.

Remarque :

Une méthode qui n'est pas du type **void** doit **dans tous les cas** se terminer par une instruction **return**. Si on oublie une instruction **return**, le compilateur s'arrête avec le message d'erreur '*missing return statement*'. Ceci jouera un rôle plus important lors du traitement des structures alternatives et répétitives (→ voir chapitre 3.1).

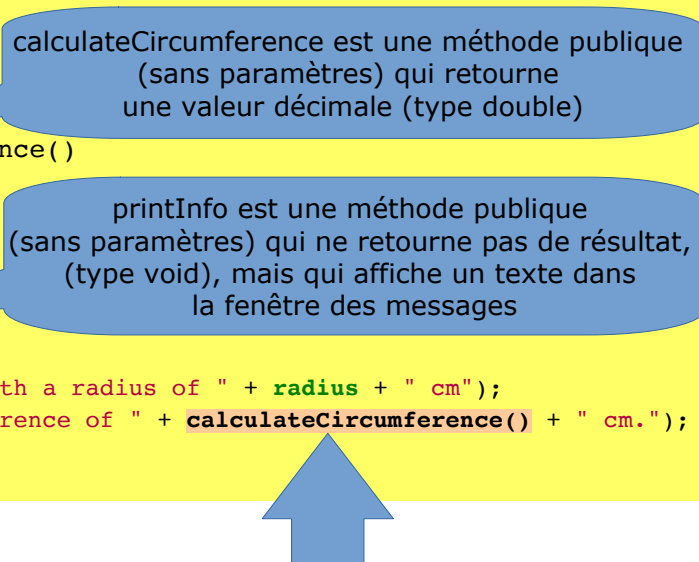
Exemple :

La classe suivante permet de calculer la circonférence de la balle et d'afficher ses informations:

```
public class Ball
{
    private double radius = 23.9;

    public double calculateCircumference()
    {
        return 3.1415926*2*radius;
    }

    public void printInfo()
    {
        System.out.println("I am a ball with a radius of " + radius + " cm");
        System.out.println("and a circumference of " + calculateCircumference() + " cm.");
    }
}
```



calculateCircumference est une méthode publique (sans paramètres) qui retourne une valeur décimale (type double)

printInfo est une méthode publique (sans paramètres) qui ne retourne pas de résultat, (type void), mais qui affiche un texte dans la fenêtre des messages

Observation importante :

Une méthode peut être appelée par d'autres méthodes. Ainsi on n'a pas besoin de programmer ou de recopier plusieurs fois les mêmes opérations. En fait, c'est l'utilité principale d'une méthode : le code devient plus compact, plus structuré et plus lisible.

Ici, la méthode **printInfo** fait appel à la méthode **calculateCircumference** pour obtenir la valeur de la circonférence de la balle. Le résultat retourné par la méthode **calculateCircumference** est intégré directement dans l'instruction d'affichage.

Discussion d'un exemple :

Créez la classe suivante:

```
public class Ball
{
    private double radius = 23.9;

    public double calculateCircumference()
    {
        return 3.1415926*2*radius;
        System.out.println("My circumference is " + 3.1415926*2*radius);
    }
}
```

Essayez de compiler la classe **Ball** et d'en créer une instance.

Notez vos observations et vos explications :

.....

.....

.....

.....

.....

Pour avancés :

Quelle est la différence fondamentale entre l'instruction **System.out.println** et l'instruction **return** ? Que peut-on dire de l'utilité de l'instruction **System.out.println** à l'intérieur de la méthode **calculateCircumference**?

.....

.....

.....

.....

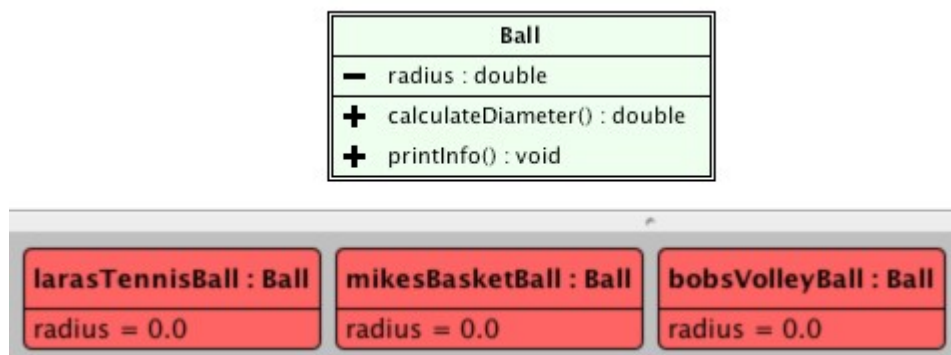
.....

Exercice résolu :

- Ouvrez votre projet '**00_Exercice_resolu**' dans votre dossier Java.
- Ajoutez la méthode publique **calculateDiameter** comme définie ci-dessus.
- Ajoutez la méthode publique **calculateCircumference** sans l'instruction *System.out.println()*.
- Ajoutez la méthode publique **printInfo** comme définie ci-dessus.
- Compilez et créez une nouvelle instance du nom de **mikesBasketBall** de la classe **Ball**.
- Appelez les méthodes **calculateDiameter**, **calculateCircumference** et **printInfo** de l'objet **mikesBasketBall**.
- Répétez les deux points ci-dessus pour **larasTennisBall** et **bobsVolleyBall**. Que remarquez-vous?

Conclusions :

- Si la classe **Ball** possède un **attribut** **radius**, alors chaque instance de la classe (**larasTennisBall**, **mikesBasketBall**, **bobsVolleyBall**, ...) possède cet attribut, mais la valeur de cet attribut peut être différente pour chaque instance :



(Nous allons voir dans les chapitres suivants comment donner des valeurs différentes aux attributs des différentes instances.)

- Si la classe **Ball** possède une **méthode** **calculateCircumference**, alors chaque instance de la classe (**larasTennisBall**, **mikesBasketBall**, **bobsVolleyBall**, ...) possède cette même méthode. Ainsi la circonférence de toutes les balles est calculée d'après la même formule.

L'environnement de développement (Unimozzer)

Lorsque nous appelons une méthode (qui n'est pas du type **void**) dans notre environnement de développement, alors le programme nous présente confortablement le résultat dans une fenêtre de dialogue. Voilà un luxe qui est spécifique à *Unimozzer* et que nous ne trouvons pas dans d'autres environnements de développement. En général, le contenu des attributs se trouve caché dans la mémoire de l'ordinateur et les résultats fournis par une méthode passent de façon invisible d'une méthode à une autre. (Pour visualiser des valeurs, nous devrions employer l'instruction **System.out.println** ou programmer nous même une fenêtre de dialogue).

Ainsi nous pouvons considérer notre interface de développement comme une sorte de banc d'essai et d'analyse confortable qui nous permet de visualiser et de tester nos classes avant de les intégrer dans un programme fini.

1.6. Qu'est-ce qu'un « paramètre » ?

Un **paramètre** est une donnée à l'aide de laquelle on peut passer une information à une méthode. Une méthode peut avoir aucun, un ou plusieurs **paramètres**.

Voici les caractéristiques d'un paramètre :

- il est d'un certain type (→ voir chapitre 1.7) et
- il possède un nom.

Donnons aux balles la possibilité de changer la valeur du radius :

Ball	
—	radius : double
+	getRadius() : double
+	setRadius(pRadius : double) : void

Le schéma UML de la classe **Ball** peut être étendu de la manière suivante :

Voici le code source :

```
public class Ball
{
    private double radius = 0;

    public double getRadius()
    {
        return radius;
    }

    public void setRadius (double pRadius)
    {
        radius = pRadius;
    }
}
```

Annotations du code :

- `getRadius()` ne possède pas de paramètres
- `setRadius` possède un paramètre du type double
- `void` : la méthode ne retourne pas de résultat

Explications :

- Le mot clé «**public**» indique qu'il s'agit d'une méthode publique, c'est-à-dire d'une méthode qui est utilisable par d'autres sous-programmes ou d'autres classes.
- Le mot clé «**void**» indique que la méthode **setRadius** ne retourne pas de résultat.
- Le mot «**setRadius**» est le nom de cette méthode.
- «**double pRadius**» est la déclaration du **paramètre**. Dans notre cas il contiendra la nouvelle valeur pour le rayon:
 - « **double** » indique que le **type du paramètre** est un nombre décimal.
 - « **pRadius** » est le **nom du paramètre**. Comme les noms des méthodes et des attributs, les noms des paramètres en Java sont écrits en lettres minuscules. Si le nom se compose de plusieurs mots, la première lettre de chaque mot est écrite en majuscules, sauf pour le premier mot.
 - Pour mieux distinguer les paramètres des attributs, nous allons dans ce cours commencer les noms des paramètres par un 'p'.

Autre exemple :

Prenons la classe **Point** et ajoutons la possibilité de modifier sa position :

Point
- x : double
- y : double
+ setCoordinates(pX : double, pY : double) : void

```
public class Point
{
    private double x = 0;
    private double y = 0;

    public void setCoordinates(double pX, double pY)
    {
        x = pX;
        y = pY;
    }
}
```

Comme déjà indiqué, une méthode peut avoir plusieurs paramètres. Dans ce cas, les paramètres sont séparés par une virgule. Il faut indiquer pour chaque paramètre de quel type il s'agit. Dans le cas de la méthode **setCoordinates**, les deux paramètres sont des nombres réels (type **double**).

Exercice résolu :

- Ouvrez votre projet **00_Exercice_resolu** que vous avez créé auparavant.
- Ajoutez à la classe **Ball** la méthode **calculateVolume** qui calcule et retourne le volume (en cm³) de la balle.
- Ajoutez un attribut **pressure** (nombre décimal, valeur initiale 0) à la classe **Ball**.
- Ajoutez une méthode **setPressure** à la classe, qui sert à modifier l'attribut **pressure**.
- Complétez la méthode **printInfo**.
- Créez une instance de la classe **Ball** et testez les nouvelles méthodes. (Consultez Internet pour trouver des valeurs réalistes pour les pressions des balles de tennis, volleyball, basketball,...)

Remarque avancée : Objets comme paramètres

Un paramètre peut avoir un type simple (**int**, **double**, **boolean**, ...), mais un paramètre peut aussi être un objet (p.ex. du type **Rectangle**, **Point**, ...).

Exemple :

On pourrait par exemple imaginer une méthode **calculateDistanceTo** qui calcule la distance entre deux instances du type **Point** :

```
public class Point
{
    private double x = 0;
    private double y = 0;

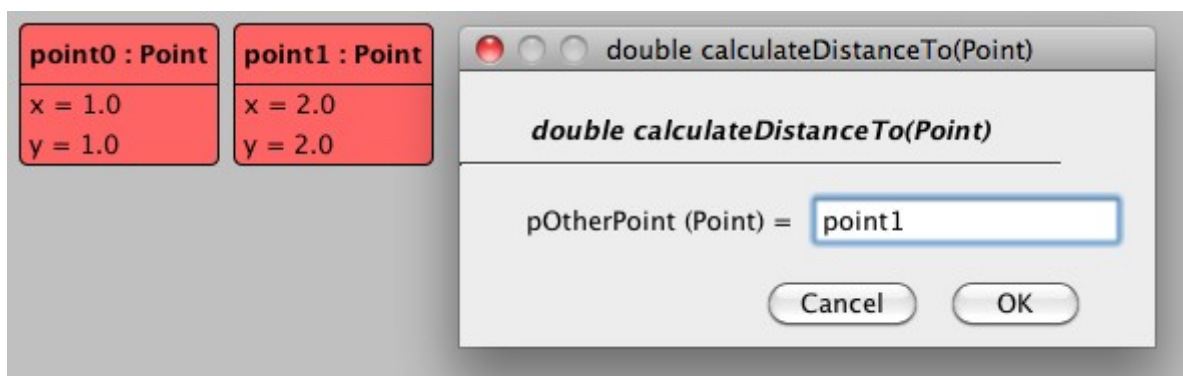
    public void setCoordinates(double pX, double pY)
    {
        x = pX;
        y = pY;
    }

    public double calculateDistanceTo(Point pOtherPoint)
    {
        return Math.sqrt(
            Math.pow(pOtherPoint.x-x, 2) +
            Math.pow(pOtherPoint.y-y, 2));
    }
}
```

D'après le cours de mathématiques, la distance entre les points $A(x_A, y_A)$ et $B(x_B, y_B)$ se calcule suivant la formule : $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

Pour les détails concernant la classe **Math**, veuillez vous référer au chapitre 1.13

Pour tester la méthode, il faut créer deux objets **point0** et **point1** du type **Point**. Ensuite, on peut appeler la méthode **calculateDistanceTo** de **point0** et fournir comme paramètre le nom de la deuxième instance **point1**.



Dans cet exemple, la méthode va retourner la valeur 1.4142...

1.7. Qu'est-ce qu'un « type » ?

Un ordinateur sait traiter différentes sortes de données. En travaillant avec un tableur (Excel/Calc), vous avez certainement déjà remarqué qu'il faut faire la différence entre des données numériques et du texte. En plus, vous avez vu que l'ordinateur sait faire des opérations différentes avec du texte qu'avec des nombres et que le programme est très sensible si on essaie de mélanger différentes sortes de données.

En général, on parle de « type » ou de « type de donnée ». En effet, dans un langage de programmation évolué comme Java, toutes les données sont typisées, c.-à-d. le type de chaque donnée doit être fixe et connu à l'avance.

Le **type d'une donnée** définit :

- de quelle **sorte** de donnée il s'agit (nombre entier, nombre décimal, texte, ...),
- dans quel **domaine** la valeur de la donnée peut varier,
- combien de **place** la donnée occupe dans la mémoire de l'ordinateur,
- quelles **opérations** peuvent être effectuées sur cette donnée.

En Java, on distingue les **types de base** suivants :

Les nombres entiers :

- **byte** 8 bits [-128 ... 127]
- **short** 16 bits [-32'768 ... 32'767]
- **int** 32 bits [-2'147'483'648 ... 2'147'483'647]
- **long** 64 bits [-9'223'372'036'854'775'808 ... 9'223'372'036'854'775'807]

Les nombres décimaux (valeurs négatives et positives) :

- **float** 32 bits [$1,4 \cdot 10^{-45}$... $3,4 \cdot 10^{38}$]
- **double** 64 bits [$4,9 \cdot 10^{-324}$... $1,79 \cdot 10^{308}$]

Les booléens :

- **boolean** 1 bit {true, false}

Remarques :

- La plupart du temps, nous allons employer le type **int** pour des entiers et le type **double** pour des nombres décimaux.
- Pour traiter des textes, on emploie le type **String** qui n'est pas un type de base et qui sera traité plus tard dans un chapitre à part. On peut cependant déjà noter que dans un programme Java les textes sont à noter entre guillemets : **"..."** (p.ex. : **streetName = "rue Bellevue";**)
On peut concaténer (coller ensemble) des textes avec le symbole d'addition «+» .

Exemple : la classe Rectangle

```
public class Rectangle
```

```
{
```

```
    private int width = 0;
```

```
    private int height = 0;
```

```
    public int getCircumference()
```

```
    {
```

```
        return 2 * (width+height);
```

```
    }
```

```
    public double calculateLengthOfDiagonal()
```

```
    {
```

```
        return Math.sqrt(width*width + height*height);
```

```
    }
```

```
    public String toString()
```

```
    {
```

```
        return "Rectangle, width:" + width + ", height:" + height;
```

```
    }
```

```
    public boolean isSquare()
```

```
    {
```

```
        return (width==height);
```

```
    }
```

```
}
```

Rectangle	
–	width : int
–	height : int
+	getCircumference() : int
+	calculateLengthOfDiagonal() : double
+	toString() : String
+	isSquare() : boolean

Explications :

- La méthode **getCircumference** retournera un entier (type **int**).
- La méthode **calculateLengthOfDiagonal** utilise le théorème de Pythagore pour calculer la longueur de la diagonale. La racine carrée est calculée à l'aide de la méthode prédéfinie **Math.sqrt** (→ voir chapitre 1.13) qui peut prendre des entiers ou des décimaux comme paramètres, mais qui retourne toujours un nombre décimal comme résultat. Ainsi la valeur retournée par **calculateLengthOfDiagonal** doit être du type **double**.
- La méthode **toString** retourne une description textuelle de l'objet et aura donc un résultat du type **String**.

Remarque :

Habituellement toutes les classes en Java contiennent une méthode **toString** qui donne une description textuelle des objets.

- La méthode **isSquare** teste si le rectangle est carré ou non. La réponse à la question est soit *vrai* (**true**), soit *faux* (**false**), ce qui correspond au type de retour **boolean**. Le test¹ (**width==height**) fournit une valeur booléenne qui est directement retournée comme résultat.

¹ Voir aussi chapitre 3.2 sur les opérateurs de comparaison

1.8. Conventions de noms pour les méthodes

Dans les exemples précédents, nous avons employé différents préfixes pour relever le rôle de différentes méthodes. Avec ces noms, nous avons suivi les conventions de noms qui existent pour les identifiants des méthodes en Java :

1.8.1. Les accesseurs - préfixe *get*

Par le préfixe **get**, on désigne une méthode qui retourne directement la valeur d'un attribut. Une telle méthode est encore appelée un **accesseur**. Un accesseur a toujours un type retour différent de **void**. Le plus souvent, les accesseurs n'ont pas de paramètres. (Le préfixe **get** peut aussi être employé pour des méthodes qui retournent une valeur dérivée des attributs par un calcul très simple, même s'il ne s'agit pas accesseur dans le sens pur du terme²).

Exemples : `getRadius`, `getDiameter`, `getAge`, `getName`, `getX`, `getY`

1.8.2. Les manipulateurs - préfixe *set*

Par le préfixe **set**, on désigne une méthode qui permet de modifier directement la valeur d'un ou de plusieurs attributs. Une telle méthode est encore appelée un **manipulateur**. Un manipulateur est le plus souvent du type **void** et il possède un ou plusieurs paramètres. Dans ce cours, nous supposons (sans vérification) que les données fournies correspondent toujours au domaine de définition de la classe. Si l'utilisateur fournit des données incorrectes, la classe se trouve dans un état inconsistant et les résultats sont imprévisibles.³

Exemples : `setDiameter`, `setTime`

1.8.3. Les méthodes effectuant un calcul - préfixe *calculate*

Par le préfixe **calculate**, on désigne une méthode qui réalise un calcul plus complexe et retourne le résultat de ce calcul.

Exemples : `calculateSumOfDivisors`, `calculateGreatestCommonDivisor`

1.8.4. Les méthodes booléennes - préfixes *is/has*

Pour rendre le code plus lisible, les méthodes et accesseurs qui retournent une valeur booléenne (et les variables de type booléen) portent de préférence les préfixes **is** ou **has**. Plus rarement on peut même trouver des préfixes comme **can**, **was**, **allows**, ...

Exemple : `isSquare`, `isEven`, `isRunning`, `hasWheels`, `hasHitAWall`, `canFly`

1.8.5. La méthode *toString*

En Java, en principe tous les objets peuvent être 'affichés' à l'aide de **System.out.println** ou concaténés avec un texte. Dans ces cas, Java appelle automatiquement la méthode **toString** de l'objet. Ainsi, tous les objets possèdent directement ou indirectement une méthode **toString** qui retourne les informations de base de l'objet sous forme d'un texte. Ces informations sont en général les valeurs actuelles des attributs principaux. La méthode **toString** n'a jamais de paramètres et elle retourne toujours une valeur du type **String**.

Exemple : `Rectangle.toString` (→ voir chapitre 1.7)

² En effet, pour l'utilisateur des méthodes d'une classe il n'est pas nécessaire de savoir si c'est le rayon qui est mémorisé (et que le diamètre est dérivé du rayon) ou si c'est l'inverse...

³ Dans des programmes professionnels de telles erreurs sont évitées par le lancement et le traitement **d'exceptions**. Ce concept est illustré plus tard.

1.9. Qu'est-ce qu'un « constructeur » ?

Le « **constructeur** » est la méthode qui est appelée lors de la construction d'un objet. Il s'agit d'une méthode spéciale dont le rôle est de **créer une nouvelle instance** et d'**initialiser les attributs** et sous-objets de la classe afin de mettre l'objet dans son état initial.

Ce qu'il faut savoir à propos du constructeur :

- Tout constructeur porte toujours le **nom de la classe**.
- Un constructeur n'a **pas de type de retour**.
- Un constructeur peut avoir des **paramètres**.
- Un constructeur doit être **public** (**public**).
- La définition d'un constructeur n'est **pas obligatoire**.
- Une même classe peut posséder **plusieurs constructeurs**.

Attention : Les différents constructeurs d'une classe ne peuvent pas avoir la même liste de paramètres : Ou bien le nombre de paramètres doit être différent ou bien au moins l'un des paramètres doit avoir un type différent. (Les noms des paramètres ne jouent pas de rôle pour la différenciation.)

Voici l'exemple de la classe **Point** définie précédemment :

Point
- x : double
- y : double
+ Point(pX : double, pY : double)
+ setCoordinates(pX : double, pY : double) : void

Le code source du **constructeur** est le suivant :

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

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

Le compilateur Java reconnaît le constructeur au fait qu'il a le même nom que la classe et qu'il n'a pas de type retour.

Le **constructeur** `Point(double pX, double pY)` initialise le point lors de la création d'un objet du type **Point** et assure de ce fait que ses coordonnées soient toujours initialisées. L'initialisation des attributs se fait souvent dans le **constructeur**.

Remarque : Comme pour les manipulateurs (→ voir chap. 1.8.2), nous supposons dans ce cours (sans vérification) que les données fournies correspondent toujours au domaine de définition de la classe. Si l'utilisateur fournit des données incorrectes, l'objet se trouve dans un état inconsistant et les résultats sont imprévisibles.

1.10. Qu'est-ce qu'une « variable » ?

Une variable nous aide en général à mémoriser temporairement une donnée lors d'un calcul ou d'une autre opération à l'intérieur d'une méthode. Une variable est déclarée dans le corps d'une méthode et elle n'est accessible que dans ce bloc de cette méthode et à partir de la ligne dans laquelle elle est déclarée. On regroupe souvent toutes les déclarations de variables tout au début d'une méthode.

Résumons les **caractéristiques d'une variable** :

- une variable est d'un certain type,
- elle possède un nom,
- elle peut être initialisée lors de sa déclaration (comme un attribut),
- elle est connue uniquement dans le bloc de sa déclaration à l'intérieur d'une méthode.

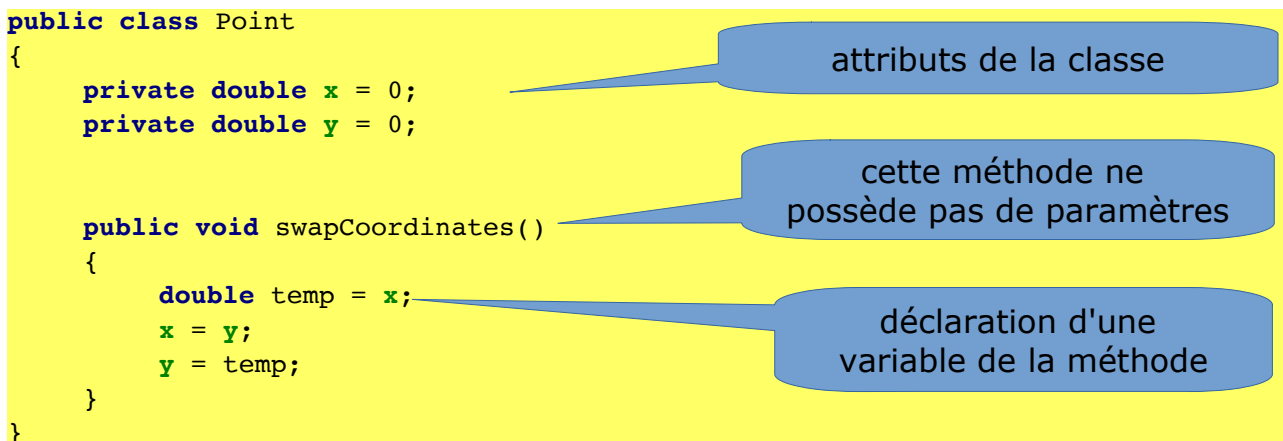
Comparaison : Une variable est assez similaire à un paramètre ou un attribut, mais

- un **attribut** est une propriété fondamentale de la classe. Il est déclaré au début de la classe et il est accessible partout dans toutes les méthodes de la classe,
- un **paramètre** sert à fournir des données à une méthode lors de son appel. Il est déclaré dans l'en-tête de la méthode et n'est connu que dans cette méthode.
- une **variable** sert à mémoriser temporairement une valeur. Elle n'est connue que dans un domaine restreint de la méthode et son rôle dans le programme est très limité.

Voici l'exemple de la classe **Point** à laquelle on a rajouté la méthode **swapCoordinates**. Cette méthode échange les valeurs des coordonnées x et y :

```
public class Point
{
    private double x = 0;
    private double y = 0;

    public void swapCoordinates()
    {
        double temp = x;
        x = y;
        y = temp;
    }
}
```



Explications et remarques :

- La variable **temp** est du type **double**, donc un nombre réel. Dans cet exemple, elle est initialisée lors de sa déclaration.
- Les trois lignes de code servent à échanger les valeurs des coordonnées à l'aide d'une variable temporaire.
- Comme pour les attributs, il est possible de déclarer une variable **sans** l'initialiser :

```
double temp;
```

Reprenons la classe **Point** et ajoutons une méthode ayant besoin de paramètres :

Point
- x : double
- y : double
+ Point(pX : double, pY : double)
+ setCoordinates(pX : double, pY : double) : void
+ swapCoordinates() : void
+ move(pDeltaX : double, pDeltaY : double) : void

```

public class Point
{
    private double x = 0;
    private double y = 0;

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

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

    public void swapCoordinates()
    {
        double temp = x;
        x = y;
        y = temp;
    }

    public void move(double pDeltaX, double pDeltaY)
    {
        x = x + pDeltaX;
        y = y + pDeltaY;
    }
}

```

attributs de la classe

à partir d'ici, cette méthode possède une variable "temp"

à partir d'ici, la variable "temp" n'existe plus

cette méthode possède les deux paramètres "pDeltaX" et "pDeltaY" et change les attributs "x" et "y"

Explications :

- La méthode **move** ajoute des valeurs aux attributs afin de déplacer le point.
- Veuillez noter que sans explications, le code n'est pas très compréhensible. Afin d'éviter une telle situation, on préfère ajouter des **commentaires** au code source (→ voir chapitre 2.1).

1.11. Opérateurs, compatibilité et conversions

Dans les exemples précédents, vous avez déjà retrouvé les opérateurs standards qui vous sont familiers (+ , - , * , /). Complétons la liste et donnons quelques précisions.

1.11.1. Opérateur d'affectation « = »

Exemple :

```
seconds = minutes*60 + hours*3600;
```

Effet : La partie qui se trouve à droite du signe d'égalité est évaluée et le résultat est affecté (méorisé) dans la variable (ou l'attribut) qui se trouve à gauche du signe d'égalité.

Il faut que les types des deux parties soient **compatibles**, c.-à-d que leurs types doivent être égaux ou que le type du résultat à droite doit être un type 'plus petit' que le type à gauche.

P. ex. on peut affecter simplement une valeur du type **int** à une variable du type **long** ou **double**, mais pas inversement. Sinon il faut forcer la conversion (→ voir chapitre 1.11.3) vers un type plus petit.

Exemples :

```
double x = 123.3;
int i = 102;
i = x;           // impossible !!
i = (int)x;      // conversion forcée avec perte de précision : i sera 123
x = i;           // pas de problèmes !
```

Remarque pour avancés :

En Java, il existe des opérateurs d'affectation qui effectuent en même temps un calcul (++ , += , *= , ...). Il n'y a aucune nécessité de les utiliser dans ce cours, mais si cela vous intéresse, vous trouvez des précisions dans tous les livres ou sites sur Java.

1.11.2. Opérateurs arithmétiques

+	addition
-	soustraction
*	multiplication
/	division (entière et rationnelle !)
%	modulo (reste d'une division entière)

Exemples:

2.5 / 2.0	→	1.25	division de réels	→ résultat réel
20 / 7	→	2	division d'entiers	→ résultat entier
20 % 7	→	6	le reste de la division de 20 par 7 donne 6	

En effet : $20 = 2 \cdot 7 + 6$

partie entière
reste

1.11.3. Conversions forcées de types (explicites)

En faisant précéder une donnée par le nom d'un autre type entre parenthèses, la donnée est convertie dans le type entre parenthèses. On appelle ceci **conversion forcée** ou aussi **conversion explicite**.

Ceci peut être utile par exemple si on veut tronquer la partie décimale d'un nombre réel :

(int) 3.75 → 3

A cause de la compatibilité des types (→ voir chapitre 1.11.1), il n'est en général pas nécessaire de convertir un type plus petit (p.ex. **int**) en un type plus large (p.ex. **double**). Une telle conversion fait du sens dans le cas, où on veut forcer une division réelle avec des données entières (voir dernier exemple de cette page).

1.11.4. Conversions automatiques de types (implicites)

Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun. On appelle ceci **conversion automatique** ou aussi **conversion implicite**.

En principe des types plus 'petits' sont convertis en des types plus 'larges'; de cette façon on ne perd pas en précision.

Exemples :

2 + 2.5	→	4.5	un entier et un réel	→	résultat réel
3 / 2	→	1	deux entiers	→	division entière avec résultat entier
3 / 2.0	→	1.5	un entier et un réel	→	division réelle avec résultat réel

Ce dernier exemple montre bien comment on peut obtenir une division réelle pour des données du type entier.

Autre exemple:

```
double middle;
int x1 = 20;
int x2 = 15;
middle = (x1+x2) / 2.0;           // résultat réel : 17.5
```

Attention :

Dans des calculs plus complexes, il faut tenir compte de l'ordre dans lequel les opérations sont évaluées (→ **priorité des opérateurs** mathématiques) :

5 + 10 / 4	→	7	division entière puis addition d'entiers
5.0 + 10 / 4	→	7.00	division entière puis addition de réels
5 + 10.0 / 4	→	7.50	division réelle puis addition de réels
5.0 + 10.0 / 4	→	7.50	division réelle puis addition de réels

```
double x;
int i = 101;
x = i / 4;           // pas de conversion, x aura la valeur 25.00
x = i / 4.0;        // conversion implicite, x aura la valeur 25.25
x = (double)i / 4;  // conversion explicite, x aura la valeur 25.25
```

Exercice de récapitulation :

Soient les déclarations :

```
int    n = 300;  
int    m = 1000;  
double r = 400.0;  
double q = 5.35;
```

Indiquez le type et le résultat des expressions suivantes :

Expression	Type	Résultat
<code>m / 300.0</code>		
<code>r + m/n</code>		
<code>(r+n) / m</code>		
<code>r + n / m</code>		
<code>m % n</code>		
<code>(int)(q * 4)</code>		
<code>(int)q * 4.0</code>		
<code>r + m/(double)n</code>		

1.11.5. Opérateur de concaténation « + »

Deux textes (type **String**) peuvent être 'collés ensemble' à l'aide de l'opérateur de concaténation « + ». Lorsque Java détecte que le résultat est du type **String**, les données d'autres types (**int**, **double**, ...) qui sont concaténées avec '+' sont automatiquement converties en texte (→ voir exemples pour **System.out.println** ci-dessous).

Exemple :

```
String noun = "Paul";
String verb = "aime";
String sentence = noun + " " + verb + " les bananes.";
```

1.12. Affichage d'une ligne de texte

L'instruction

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

permet d'afficher un texte dans la fenêtre des messages et de passer à la ligne après l'affichage. L'utilité de la fenêtre des messages est assez restreinte, mais parfois elle nous permet de visualiser plus de détails que l'environnement de développement. Ainsi, nous pouvons par exemple utiliser cette instruction pour faire afficher des résultats intermédiaires lorsque nous sommes à la recherche d'une erreur dans notre programme.

Evidemment, on peut aussi afficher le **contenu de variables texte** (type **String**) avec une instruction d'affichage.

Comme Java exécute des conversions automatiques, on peut aussi afficher le **contenu de variables numériques** à l'aide de cette instruction. En combinaison avec l'opérateur de concaténation, cette instruction peut donc servir à afficher des messages assez complexes.

L'instruction

```
System.out.print( ... );
```

permet d'afficher un texte *sans* passer à la ligne après l'affichage. Ainsi le *prochain* affichage continuera dans la même ligne.

Exemples :

```
int sideLength = 220;
System.out.println( "Hello" );      // affiche le texte Hello
System.out.println( sideLength );   // affiche le nombre 220
System.out.println( "Un côté du carré mesure " + sideLength + " cm" );
                                   // affiche le texte : Un côté du carré mesure 220 cm

int surface = sideLength*sideLength;
System.out.println( "Surface du carré : " + surface + " cm2" );
                                   // affiche le texte : Surface du carré : 48400 cm2
```

On peut intégrer des **appels de méthodes** dans une instruction d'affichage:

```
System.out.println("Temps actuel en secondes : " + getTimeInSeconds());
```

On peut intégrer des **calculs** dans une instruction d'affichage, mais alors il est recommandé de comprendre chaque calcul entre **parenthèses** (sinon Java risque de mal interpréter certains opérateurs) :

```
System.out.println("Surface du carré : "+(sideLength*sideLength)+" cm2");
System.out.println("Périmètre du rectangle : "+(2*(x2-x1)+2*(y2-y1))+"cm");
```

1.13. La classe « *Math* »

Dans la classe **Math** qui est contenue en Java, sont définies des constantes et des méthodes très utiles que nous pouvons employer dans nos programmes :

Math.PI	approximation très précise de Pi (double) <u>Remarque :</u> PI est une constante et n'est donc pas suivi de parenthèses. Les noms des constantes en JAVA sont écrites en majuscules.
Math.abs(...)	retourne la valeur absolue du nombre fourni comme paramètre
Math.max(... , ...)	retourne la plus grande des valeurs fournies comme paramètres
Math.min(... , ...)	retourne la plus petite des valeurs fournies comme paramètres
Math.round(...)	retourne la valeur entière la plus proche du nombre décimal fourni comme paramètre (double → long ; float → int)
Math.sqrt(...)	retourne la racine carrée du nombre fourni comme paramètre (double)
Math.pow(... , ...)	retourne X^Y pour deux décimaux fournis comme paramètres
Math.random()	retourne un nombre aléatoire positif dans le domaine [0.0 ... 1.0[

```
surface = Math.PI * Math.pow(radius,2);
```

Autres méthodes intéressantes de la classe **Math** :

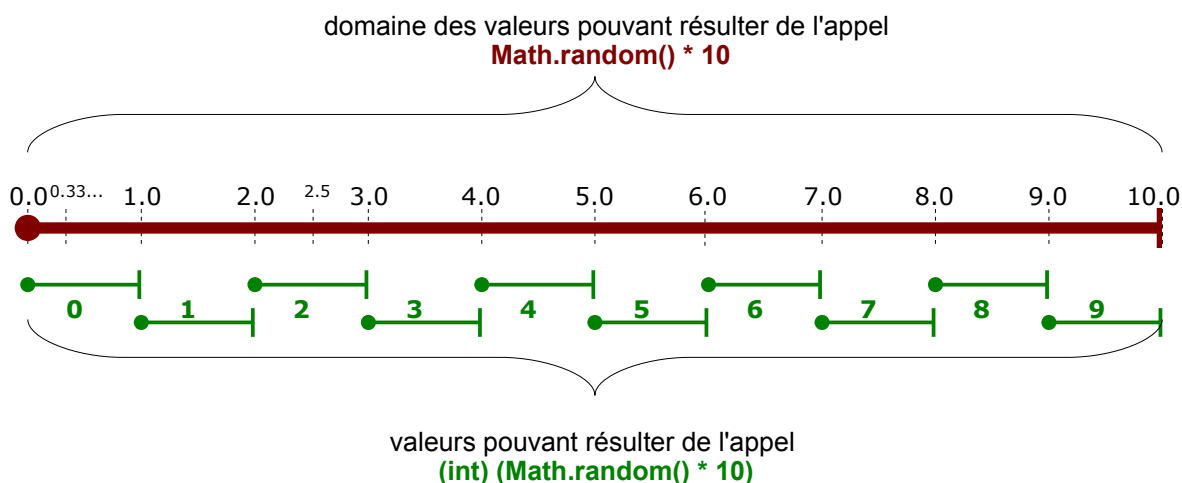
floor, sin, cos, tan, acos, asin, atan, exp, log, log10, ...

1.14. Générer des nombres aléatoires

Chaque appel de la méthode `Math.random()` retourne un nombre réel aléatoire appartenant à l'intervalle `[0.0 ... 1.0[`. Cependant nous avons souvent besoin de nombres aléatoires dans d'autres domaines ou nous voulons travailler avec des nombres entiers aléatoires.

Comment pouvons-nous générer des nombres réels aléatoires dans d'autres intervalles ?

Génération de nombres aléatoires réels	
Expression	Résultat un nombre réel aléatoire appartenant au domaine :
<code>Math.random()</code>	<code>[0.0 ... 1.0[</code>
<code>Math.random() * 100</code>	<code>[0.0 ... 100.0[</code>
<code>Math.random() * 100 - 50</code>	<code>[-50.0 ... 50.0[</code>
Formule générale à retenir : <code>Math.random() * (max - min) + min</code>	<code>[min ... max[</code>

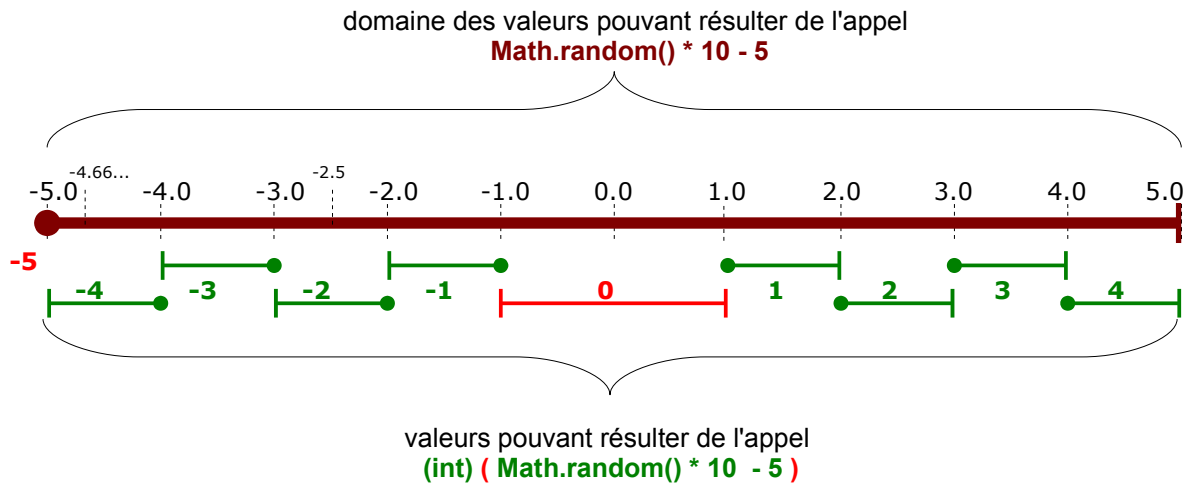


Génération de nombres aléatoires entiers	
Expression	Résultat un nombre entier aléatoire appartenant au domaine :
<code>(int) (Math.random() * 10)</code>	<code>[0 ... 9]</code>
<code>(int) (Math.random() * 10) + 1</code>	<code>[1 ... 10]</code>
<code>(int) (Math.random() * 10) - 5</code>	<code>[-5 ... 4]</code>
<code>(int) (Math.random() * 11) - 5</code>	<code>[-5 ... 5]</code>
Formule générale à retenir : <code>(int) (Math.random() * (max-min+1)) + min</code>	<code>[min ... max]</code>

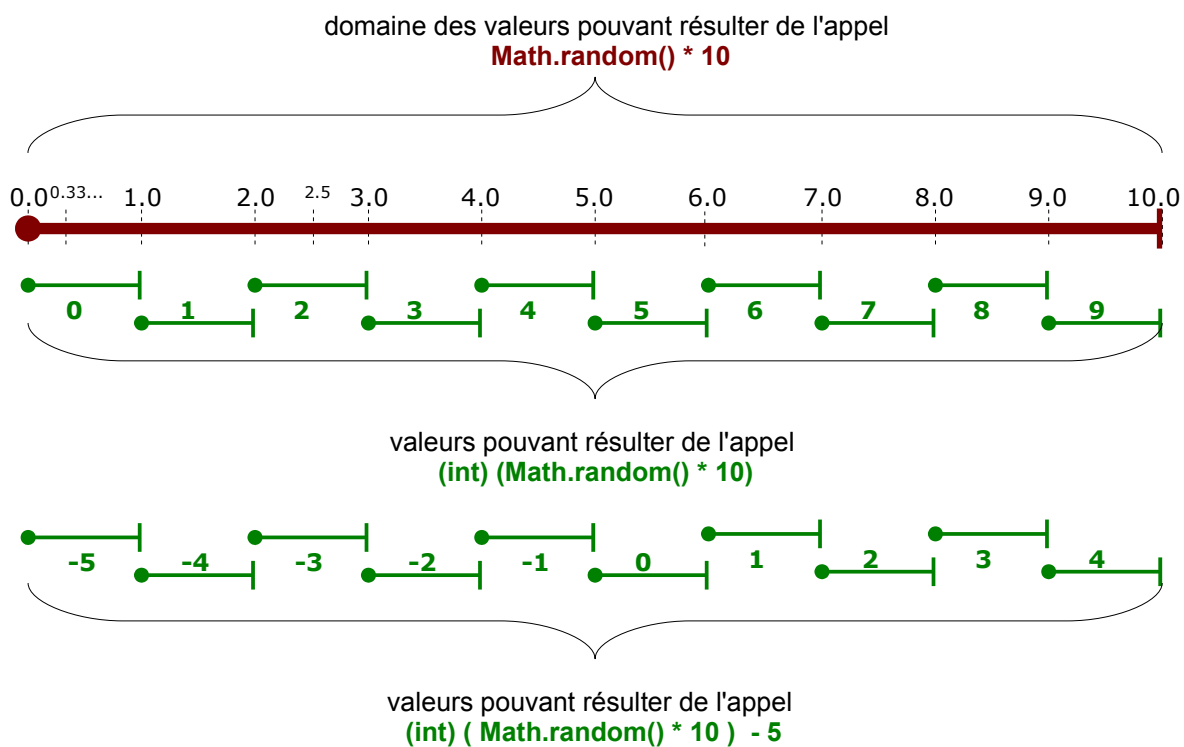
Attention :

Pour garantir que les valeurs entières soient réparties de façon régulière sur tout l'intervalle, il faut veiller à bien placer les parenthèses lors du casting. Considérez l'exemple suivant :

Parenthèses mal placées : `(int) (Math.random() * 10 - 5)` **VERSION INCORRECTE!**



Parenthèses bien placées : `(int) (Math.random() * 10) - 5` **VERSION CORRECTE!**



Comme la soustraction est effectuée APRÈS le casting, les valeurs sont réparties de façon régulière sur l'intervalle!

2. Présentation et documentation du code

2.1. Qu'est-ce qu'un « commentaire » ?

Écrire une bonne documentation constitue un complément important à un code source de bonne qualité. La documentation permet de communiquer ses intentions aux lecteurs, d'obtenir une vue d'ensemble en langage naturel, plutôt que de les obliger à décrypter un code ardu. De plus les commentaires nous aideront à mieux nous retrouver dans notre code lorsque nous y retravaillerons plus tard. Un commentaire dans un code source est un texte qui n'est pas considéré par le compilateur. Il n'influence donc pas le déroulement du programme. Il existe trois types de commentaires :

- Un **commentaire uni-ligne** commence par les deux symboles consécutifs « `//` ». Tout ce qui suit ces deux symboles dans une ligne est ignoré par le compilateur (voir exemples précédents dans ce cours). Ce type de commentaires est en général utilisé pour expliquer l'utilité ou le fonctionnement d'un bloc d'instructions dans une méthode.
- Un **commentaire multi-ligne** commence par les deux symboles « `/*` » et se termine par les deux symboles « `*/` ». Tout ce qui se trouve entre le début et la fin est ignoré par le compilateur.
- Un **commentaire du type « JavaDoc »** commence par les trois symboles « `/**` » et se termine par les deux symboles « `*/` ». Tout ce qui se trouve entre le début et la fin est ignoré par le compilateur mais pris en charge par le **générateur de documentation automatique** :
Les commentaires « JavaDoc » ont une double finalité, parce qu'ils servent aussi à générer automatiquement un fichier de documentation pour la classe et pour tous les éléments qui y sont définis. Nous allons profiter de cette fonctionnalité dans Unimozzer (et NetBeans).

2.1.1. Règles pour l'utilisation des commentaires «JavaDoc»

La description principale d'une **classe** doit indiquer son objectif en termes généraux.

```
/**
 * Cette classe calcule le poids normal d'après Broca et l'indice de masse
 * corporelle (BMI) pour un homme ou une femme
 */
public class BodyStatistics
```

Cette description doit rester assez générale, sans donner trop de détails sur son implantation. Bien souvent, une seule phrase suffira.

La description principale d'un **attribut** donne des détails concernant le but ou l'interprétation du contenu de l'attribut.

```
/**
 * Le sexe de la personne ("m" -> masculin, "f" -> féminin)
 */
private String gender = "m";
```

La description principale d'une **méthode** explique le but de la méthode.

```
/**
 * Méthode qui permet de définir l'âge de la personne
 */
public void setAge(int pAge)
```

2.1.2. Précisions : les commentaires « JavaDoc » des classes

Dans les commentaires des classes, les balises **@author** et **@version** sont prises en compte par le générateur de documentation automatique. Leur signification est la suivante :

@author le nom de l'auteur de la classe

@version la version de la classe (contient souvent la date de la dernière modification)

```
/**
 * Cette classe calcule le poids normal d'après Broca et l'indice de masse
 * corporelle (BMI) pour un homme ou une femme
 *
 * @author      J. Valjean
 * @version     21/01/2011
 */
```

2.1.3. Précisions : les commentaires « JavaDoc » des méthodes

Dans les commentaires des méthodes, les balises **@param** et **@return** sont prises en compte par le générateur de documentation automatique. Leur signification est la suivante :

@param le nom et la description qui suivent sont ceux d'un paramètre

@return description du résultat fourni par la méthode

D'après les conventions de Java il est obligatoire d'indiquer les marques **@param** et **@return** pour les méthodes. Les paramètres doivent être décrits dans le même ordre que dans l'en-tête de la méthode. **@return** est omis s'il s'agit d'une méthode du type **void**.

La balise **@param** est utilisée avec des méthodes et des constructeurs :

```
/**
 * Méthode qui permet de définir l'âge
 * @param pAge      l'âge en années
 */
public void setAge(int pAge)
```

La balise **@return** ne sert que pour les méthodes :

```
/**
 * Méthode pour calculer le poids normal suivant la
 * formule de Broca
 * @return      poids normal en "kg"
 */
public double getNormalWeight()
```

Pour de plus amples informations concernant « **JavaDoc** », veuillez consulter la page Internet suivante :

<http://www.oracle.com/technetwork/java/javase/documentation/writingdoccomments-137785.html>

2.2. Qu'est-ce que « l'indentation » ?

Par **indentation**, en allemand « Einrückung », on entend le fait de déplacer certaines parties du code source plus vers la droite que d'autres. Généralement les instructions contenues dans un bloc d'instructions sont **indentées d'une tabulation**, c'est-à-dire qu'il y a une tabulation en plus devant chaque ligne par rapport à celles au niveau du bloc d'instructions correspondant ('bloc d'instructions' → voir aussi chapitre 3.7).

Voici une partie de la classe **Point** qui nous sert d'exemple :

```
public class Point
{
    private double x = 0;
    private double y = 0;

    /**
     * Déplace le point dans le plan mathématique
     * @param pDeltaX      le déplacement de l'abscisse
     * @param pDeltaY      le déplacement de l'ordonnée
     */
    public void move(double pDeltaX, double pDeltaY)
    {
        /**
         * Déplace le point dans le plan
         */
        x = x + pDeltaX;
        y = y + pDeltaY;
    }
}
```

Niveau d'indentation 2 : Le code source est indenté d'une tabulation par rapport au niveau 1 et de deux tabulations par rapport au niveau 0.

Niveau d'indentation 1 : Le code source est indenté d'une tabulation par rapport à la ligne précédente.

Niveau d'indentation 0 : Le code source colle à la bordure gauche.

3. Les structures de contrôle

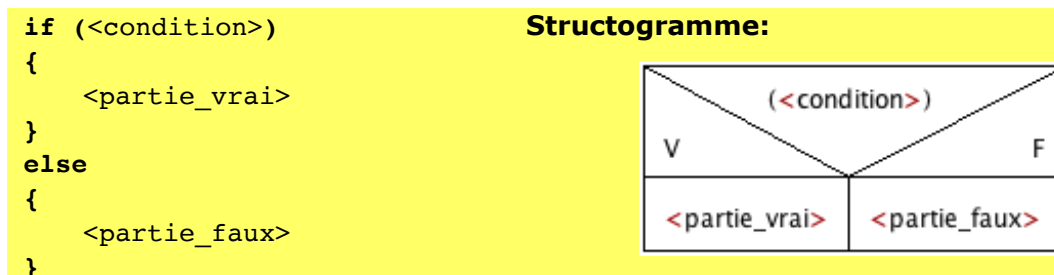
Les programmes développés jusqu'à présent se composent d'instructions qui sont exécutées de manière **séquentielle** : les instructions sont exécutées du haut vers le bas. Cependant, afin de pouvoir écrire des programmes plus utiles, il est indispensable de disposer de structures de contrôle qui peuvent **changer la suite de l'exécution** des instructions.

Ces structures permettent d'une part d'influencer le déroulement du programme et d'autre part de rendre plus simple certaines parties du code source. A partir de ce chapitre, nous allons employer de façon régulière des **structogrammes** (angl. : *NSD - Nassi-Shneiderman Diagrams*) pour représenter la structure de nos méthodes de façon graphique.

3.1. La structure alternative

La **structure alternative**, encore appelée structure « **if** », permet de faire un **choix** entre deux actions suivant qu'une certaine condition soit remplie ou non.

Voici la syntaxe de la **structure alternative** en Java et sous forme de structogramme :



Les différents blocs d'instructions sont délimités par des accolades (comme le début et la fin d'une méthode). Pour les détails sur les blocs d'instructions → voir chapitre 3.7.

Si l'évaluation de la **<condition>** donne le résultat **vrai (true)**, le bloc **<partie_vrai>** est exécuté et le bloc **<partie_faux>** est ignoré.

Si l'évaluation de la **<condition>** donne le résultat **faux (false)**, le bloc **<partie_faux>** est exécuté et le bloc **<partie_vrai>** est ignoré.

S'il n'existe pas de **<partie_faux>**, le bloc « **else** » peut simplement être omis :



Si **<partie_vrai>** ou **<partie_faux>** ne contiennent qu'une seule instruction, il n'est pas nécessaire de noter les accolades :

```

if (<condition>) <partie_vrai>;
else <partie_faux>;
    
```

respectivement :

```

if (<condition>) <partie_vrai>;
    
```

Exemples :

Calculer le maximum de deux valeurs

```

if (a>b)                ou aussi :      if (a>b) max = a;
{                        else           max = b;
    max = a;
}
else
{
    max = b;
}

```

Garantir que **a** contienne la plus grande de deux valeurs réelles et **b** la plus petite (→ permutation)

```

if (a<b)
{
    double help = a;
    a = b;
    b = help;
}

```

Remarques:

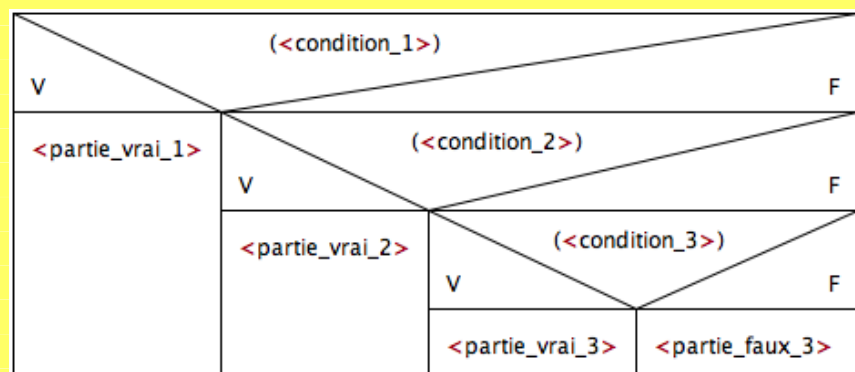
- Puisqu'une condition peut avoir la valeur **true** ou **false**, on peut parler **d'expression logique ou expression booléenne** au lieu de condition. La <condition> peut donc être une simple variable booléenne, un appel d'une méthode retournant le type **boolean**, ou toute autre expression retournant le type **boolean**.
- Il n'y a jamais de point-virgule ';' derrière une accolade fermante '}' d'un bloc d'instructions.
- Il n'y a pas de point-virgule ';' derrière la parenthèse fermante ')' de la condition. En effet, dans une construction de la forme **if(...);{...}** le point-virgule est interprété comme une instruction vide et la structure **if** se rapporterait uniquement à cette instruction vide. Le bloc d'instruction qui suit ne serait plus dépendant de la condition et serait TOUJOURS exécuté...
- Pour combiner plusieurs conditions, on peut **imbriquer** (all: *verschachteln*) les conditions:

```

if (<condition_1>)
{
    <partie_vrai_1>
}
else if (<condition_2>)
{
    <partie_vrai_2>
}
else if (<condition_3>)
{
    <partie_vrai_3>
}
else
{
    <partie_faux_3>
}

```

Structogramme:



3.2. La structure alternative et le retour de résultats

Dans une méthode qui retourne un résultat (autre que du type `void`), il faut veiller à **retourner un résultat dans TOUS les cas de la méthode**, sinon le compilateur produit une erreur du type *'missing return statement'*.

Exemples :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
}                                     <== missing return statement
```

Cette méthode ne se laisse pas compiler et elle produit une erreur *'missing return statement'*, car elle ne retourne pas de valeur si la condition n'est pas remplie.

Même l'exemple suivant produit une erreur :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
    if (pA<=pB)
        return pB - pA;
}                                     <== missing return statement
```

En analysant le code, nous voyons bien que les deux conditions traitent tous les cas possibles, mais le compilateur traite les deux structures `if` séparément et ne fait pas ce rapprochement.

Solutions :

1. Dans des cas simples (comme dans cet exemple), nous pouvons employer une seule instruction `if` avec un bloc `else` :

```
public int test(int pA, int pB)
{
    if (pA>pB)
        return pA - pB;
    else
        return pB - pA;
}
```

2. Dans des cas plus complexes, il est recommandable d'employer **une seule instruction return à la fin de la méthode** et une **variable intermédiaire** pour la valeur du résultat :

```
public int test(int pA, int pB)
{
    int result = 0;
    if (pA>pB)
        result = pA - pB;
    else
        result = pB - pA;
    return result;                // <== une seule instruction return qui est
                                // toujours exécutée à la fin de la méthode
}
```

3.3. Les opérateurs de comparaison

Pour pouvoir formuler des conditions simples, on peut se servir des opérateurs de

comparaisons suivants :

<code>==</code>	est égal à
<code>!=</code>	est différent de
<code>></code>	est strictement supérieur à
<code>>=</code>	est supérieur ou égal à
<code><</code>	est strictement inférieur à
<code><=</code>	est inférieur ou égal à

Le résultat d'une comparaison est toujours une valeur du type booléen.

Exemples :

Si nous partons des déclarations suivantes,

```
double price;  
int a,b,c,divisor;  
boolean found;
```

nous pouvons formuler les conditions suivantes :

<code>price>1000</code>	test, si la valeur de la variable <code>price</code> est supérieure à 1000
<code>a==b</code>	test, si la valeur de <code>a</code> est égale à la valeur de <code>b</code>
<code>divisor!=0</code>	test, si la valeur de <code>divisor</code> est différente de 0
<code>found</code>	test, si la variable <code>found</code> a la valeur <code>true</code>

Attention :

Les opérateurs de comparaison fonctionnent uniquement pour des types de base (`int`, `float`, `double`, `char`, ...). Pour des objets et des variables du type `String`, il faut utiliser des méthodes spéciales.

3.4. Les opérateurs logiques

Les conditions peuvent être combinées à l'aide de la **conjonction** (**ET** logique) et de la **disjonction** (**OU** logique). On peut inverser logiquement le résultat d'une condition à l'aide de la **négation** (**NON** logique).

opérateur	opération logique	désignation	retourne <i>true</i> , si et seulement si ...
&&	ET	conjonction	... les deux conditions sont remplies
	OU	disjonction	... au moins l'une des deux conditions est remplie
!	NON	négation	... la condition derrière ! retourne false

Soient C1 et C2 deux conditions (ou deux expressions booléennes) :

C1	C2	C1 && C2	C1 C2	!C1
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Exemples de conditions composées :

`(x>0) && (x<=10)` test, si `x` appartient à l'intervalle `]0,10]`, donc si `x` est strictement plus grand que 0 **et** `x` est inférieur ou égal à 10

`!(a==b)` test, si `a` n'est pas égal à `b` (même effet que `a!=b`)

`(a==0) || (b==0)` test, si au moins l'une des deux variables `a` ou `b` est 0, donc si `a` est 0 **ou** `b` est 0 **ou** `a` et `b` sont 0

`found && !list.isEmpty()` test, si la variable `found` est **true** **et** la méthode `list.isEmpty()` retourne **false**

Attention - La loi de De Morgan -

Soient C1 et C2 deux conditions (ou deux expressions booléennes), alors :

`!(C1 && C2)` est identique à `!C1 || !C2`

`!(C1 || C2)` est identique à `!C1 && !C2`

Exemple illustratif :

Papa tond la pelouse, s'il fait beau **ET** la pelouse est trop haute.
 Papa **ne** tond **pas** la pelouse, s'il fait mauvais **OU** la pelouse est courte.

Exemples :

`!((a==0) || (b==0))` est identique à `(a!=0) && (b!=0)`

`!((a>b) && (b>c))` est identique à `(a<=b) || (b<=c)`

3.5. La structure répétitive « tant que »

La **structure répétitive « tant que »**, encore appelée structure **« while »**, permet de répéter certaines instructions tant qu'une condition donnée est vraie.

La syntaxe de la **structure répétitive « tant que »** est la suivante :

```
while (<condition>)  
{  
    <instructions>  
}
```

- Tant que la **<condition>** est vraie, les **<instructions>** sont exécutées. Dès que la **<condition>** est fausse, la boucle s'arrête.
- Si la **<condition>** est fausse dès le début, le corps de la boucle n'est jamais exécuté.
- Une boucle dont la **<condition>** est toujours vraie ne s'arrête jamais. On parle alors d'une « boucle infinie ». Bien entendu, il faut éviter une telle boucle, puisqu'elle fait en sorte que le programme bloque et déstabilise tout l'ordinateur.

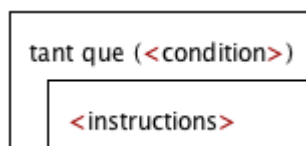
Si le corps de la **structure répétitive « tant que »** ne contient qu'une seule instruction, on peut omettre les accolades :

```
while (<condition>)  
    <instruction>;
```

Remarques :

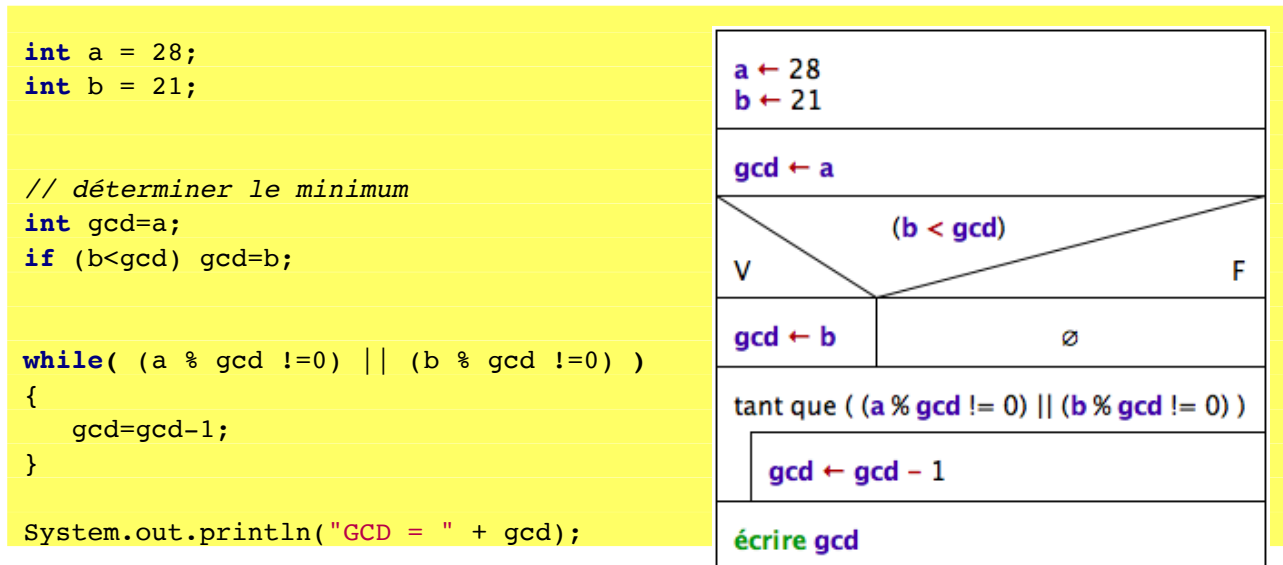
- Il n'y a jamais de point-virgule ';' derrière une accolade fermante '}' d'un bloc d'instructions.
- Il n'y a pas de point-virgule ';' derrière la parenthèse fermante ')' de la condition. En effet, dans une construction de la forme **while(...); {...}** le point-virgule serait interprété comme une instruction vide et la structure **while** se rapporterait uniquement à cette instruction vide. Comme une instruction vide n'effectue pas de changement dans l'état de la condition, une fois que la condition a été remplie, elle reste remplie et la boucle est infinie. (Le bloc d'instructions qui suit n'est pas lié à la boucle et ne sera jamais atteint si la condition est **true**. Si la condition est **false**, la boucle n'aura pas d'effet, mais le bloc d'instructions sera exécuté une seule fois.).

La représentation d'une **structure répétitive « tant que »** dans un structogramme est la suivante :



Exemple 4:

Voici l'exemple d'une boucle qui recherche le plus grand commun diviseur de deux nombres (angl.: **GCD** *Greatest Common Divisor*) :

**Remarque pratique :**

Dans ce cours nous allons éviter de quitter une boucle **while** par un **return** dans le bloc d'instructions de la boucle. Pour terminer une boucle, nous allons définir une condition de parcours appropriée. De cette façon, le code reste plus clair car il ne faut pas analyser le corps entier de la boucle pour être sûr qu'il n'y a pas "d'issues cachées" dans la boucle. En plus on évite des erreurs du type '*missing return statement*'.

4 Site vous aidant à visualiser les passages d'une boucle: <https://www.cs.virginia.edu/~lat7h/cs1/JavaVis.html>

3.6. La structure répétitive « pour »

La **structure répétitive « pour »**, encore appelée boucle **« for »** ou **boucle de comptage** (all. : « *Zählschleife* »), permet de répéter un bloc d'instructions un nombre précis de fois. En effet, le nombre d'itérations (de répétitions) est **connu à l'avance**.

Exemple :

Pour commencer, voici une simple boucle **for** qui affiche 10 fois "Hello" à l'écran.

```
for (int i=0 ; i<=9 ; i++)
{
    System.out.println("Hello");
}
```

pour i ← 0 à 9
écrire "Hello"

Voici la syntaxe d'une **structure répétitive « pour »** en Java :

```
for ( <initialisation> ; <condition> ; <incrémentation> )
{
    <instructions>
}
```

- Dans la partie **<initialisation>** le compteur (p.ex. une variable i) est initialisé. Souvent cette variable est même déclarée dans cette partie (voir exemple).
- La condition **<condition>** est évaluée *avant* chaque itération. Si elle est vraie, une nouvelle itération est faite, sinon la boucle se termine. Typiquement dans une boucle de comptage, la condition a la forme :

$$i \leq \text{valeur_maximale} \quad \text{ou bien} \quad i \geq \text{valeur_minimale}$$
- Le code dans **<incrémentation>** est exécuté *après* chaque itération. Dans ce cours, nous utilisons ou bien l'incrémentation positive (**i++** ou **i=i+1**) ou bien l'incrémentation négative (**i--** ou **i=i-1**), c.-à-d. à chaque passage de la boucle, le compteur est ou bien augmenté de 1 ou bien diminué de 1.
- Le bloc **<instructions>** forme le corps de la boucle et est exécuté à chaque répétition.

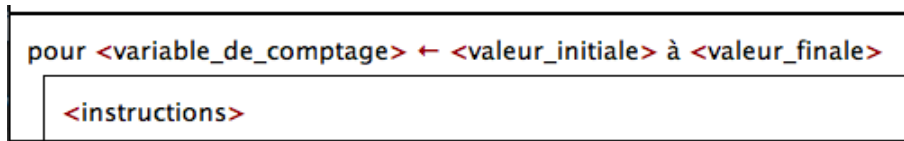
Si le corps de la **structure répétitive « pour »** ne contient qu'une seule instruction, on peut omettre les accolades :

```
for ( <initialisation> ; <condition> ; <incrementation> )
    <instruction>;
```

Remarques :

- Il n'y a jamais de point-virgule ';' derrière une accolade fermante '}' d'un bloc d'instructions.
- Il n'y a pas de point-virgule ';' derrière la parenthèse fermante ')' de l'instruction **for**. En effet, dans une construction de la forme **for(...); {...}** le point-virgule serait interprété comme une instruction vide et la structure **for** se rapporterait uniquement à cette instruction vide. Ainsi cette instruction vide serait 'exécutée' le nombre de fois fixé par la boucle **for**. Ensuite le bloc d'instructions qui suit serait exécuté une seule fois.

Représentation d'une **structure répétitive « pour »** dans un structogramme :



En comparant avec le code source ...

- c'est dans la partie **<initialisation>** que la **<valeur_initiale>** est affectée à la **<variable_de_comptage>**.
- c'est dans la partie **<condition>** que la valeur de la **<variable_de_comptage>** est comparée à la **<valeur_finale>**.
- c'est dans la partie **<incrémentations>** que la **<variable_de_comptage>** est augmentée ou diminuée. Dans le structogramme nous pouvons optionnellement indiquer le pas de l'incrémentations (**pas=1** ou **pas=-1**). Par défaut (c.-à-d. si aucun pas n'est indiqué), nous utilisons l'incrémentations positive (**i++**).

Remarques pratiques :

- Il ne faut jamais modifier la variable de comptage dans le corps de la boucle. Nous n'allons pas non plus modifier les valeurs initiales et finales de la boucle dans le corps de la boucle.
- Dans ce cours nous allons éviter de quitter une boucle **for** par un **return** dans le bloc d'instructions de la boucle. Si nous devons terminer une boucle de comptage avant qu'elle ne soit arrivée à la valeur finale, nous allons utiliser une boucle **while** avec une condition de parcours appropriée.
- En respectant les deux règles ci-dessus, il suffira toujours de regarder l'en-tête de la boucle **for** pour savoir exactement combien de fois elle sera parcourue. De cette façon, la boucle **for** garde son avantage principal : une bonne lisibilité et une interprétation facile. En plus on évite des erreurs du type '*missing return statement*'.

3.7. Les blocs et la durée de vie des variables

Dans ce chapitre, nous avons traité différentes structures de contrôle et dans ce contexte, nous avons introduit la notion de bloc d'instructions.

Un bloc d'instructions regroupe une suite d'instructions en les plaçant entre accolades { ... } .
Un bloc d'instructions peut être utilisé partout où on pourrait placer une seule instruction.

Les blocs servent à marquer le corps d'une méthode mais aussi à regrouper des instructions derrière **if**, **else**, **for**, **while**, etc.

En relisant le chapitre traitant les variables, nous voyons que **les variables déclarées dans un bloc d'instructions ne sont visibles que dans ce bloc d'instructions**.

La 'vie' d'une variable commence donc à l'endroit de sa déclaration et s'arrête à la fin du bloc dans lequel elle a été déclarée.

Derrière son bloc de déclaration, la variable n'existe plus et on provoque une erreur du type *'cannot find symbol'* si on essaie d'y accéder.

Exemple :

```
public void setLimits(int pMin, int pMax)
{
    if (pMin > pMax)
    {
        int help = pMin;
        pMin = pMax;
        pMax = help;
    }
    min    = pMin;
    max    = pMax;
}
```

← ici, la variable **help** n'existe pas encore

} dans ce domaine, la variable **help** est accessible.

← ici, la variable **help** n'existe plus

Durée de vie de la variable de comptage for :

La variable de comptage est définie à l'intérieur du bloc **for**, donc sa 'vie' commence au début du bloc **for** et s'arrête à la fin du bloc **for**. Derrière la boucle, **i** n'existe plus et on provoque une erreur du type *'cannot find symbol'* si on essaie d'y accéder.

```
...
for (int i=0 ; i<=9 ; i++)
{
    System.out.println(i);
}
...
```

← ici, la variable **i** n'existe pas encore

} dans ce domaine, la variable **i** est accessible.

← ici, la variable **i** n'existe plus

4. Annexe : Automatisation des tests des classes

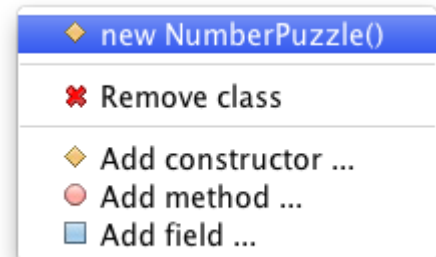
Cette annexe contient des notions qui ne figurent pas au programme de la 3GTG, mais qui peuvent être intéressantes pour créer des interfaces individuels qui nous permettront d'automatiser les tests des classes développées dans le cours.

4.1. Création d'objets avec «new»

Jusqu'ici, vous avez créé des objets en :

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

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



```
XYZ xyz = new XYZ();
```

Remarque: Les instances créées ainsi ne seront pas visibles dans l'interface Unimozzer. Elles existent dans la mémoire, mais nous ne pouvons pas les analyser directement.

4.2. Saisie de données au clavier en mode texte

Dans le cours, nous avons vu qu'il est possible d'afficher du texte dans la fenêtre des messages, mais il est aussi possible de saisir des données au clavier pendant le déroulement du programme. De cette façon, nous pouvons créer un petit interface texte pour tester nos classes. Dans la logique du MVC (*Model-View-Controller* → voir cours de 12eGE) ces classes portent de préférence le suffixe **Controller**.

Exemple (Ajoutez la classe **PointController** dans le projet de la classe **Point**) :

```
import java.util.Scanner;
public class PointController
{
    public void run()
    {
        Scanner input = new Scanner(System.in);
        System.out.print("Entrez X : ");
        double x = input.nextDouble();
        System.out.print("Entrez Y : ");
        double y = input.nextDouble();

        Point myPoint = new Point(x,y);           // création d'une instance
                                                // de la classe à tester

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

Explications

- L'instruction `import java.util.Scanner;` devant la définition de la classe **PointController**, fait en sorte que la classe **Scanner** du paquet `java.util` soit incluse dans ce programme.
- L'instruction `Scanner input = new Scanner(System.in)` crée un objet de la classe **Scanner**⁵ qui permet de lire une entrée de l'utilisateur via le clavier.
- La partie `input.nextDouble()` lit ce que l'utilisateur a entré au clavier, l'interprète comme une donnée du type **double** et affecte la valeur lue à la variable **x** respectivement **y**.
- A part `nextDouble`, la classe **Scanner** propose des méthodes pour les autres types numériques vus dans le cours : `nextFloat`, `nextInt`, `nextLong`, `nextByte`, `nextShort`.

4.3. Démarrage automatique du programme

Pour tester la classe **Point** dans l'exemple précédent, il faut toujours créer une instance de **PointController**, puis appeler manuellement la méthode `run()`. Il serait bien plus pratique si ces deux actions s'effectueraient automatiquement.

En effet, on peut créer en Java une méthode avec la déclaration suivante :

```
public static void main(String[] args)
```

L'effet en sera le suivant :

- On n'aura plus besoin de créer une instance de **PointController** (à cause de **static** dont l'explication apparaîtra bien plus tard dans le cours de programmation...).
- La méthode **main** sera appelée automatiquement lorsqu'on démarre l'application. Oui, en effet, maintenant en ajoutant la méthode **main**, vous avez créé une application que l'on peut démarrer (voir exemple ci-dessous).

En Unimozzer, vous pouvez créer une méthode **main** avec une nouvelle classe simplement en cochant la case ☒ **Main Method**.

Exemple :

Remplacez dans l'exemple précédent la déclaration

```
public void run()
```

par

```
public static void main(String[] args)
```

Sauvegardez, puis pour démarrez l'application :

- **En Unimozzer :** choisissez **Project → Run** ou poussez la touche **F6**

Dans les deux cas, la méthode **main** s'exécutera et on vous demandera des arguments (→ **args**) pour l'application. Comme nos applications n'ont pas besoin d'arguments de l'extérieur, vous pouvez confirmer simplement en laissant la liste des arguments vide.

⁵ <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

5. Quickstart avec NetBeans

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

Étape 1 : Créer un nouveau projet vierge dans NetBeans :

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

Étape 2 : Ajouter deux classes au projet :

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

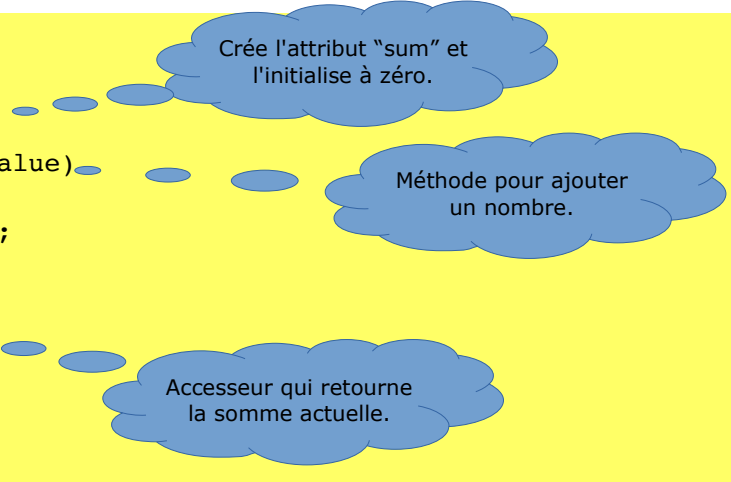
Étape 3 : Programmer la classe **Adder** :

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

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

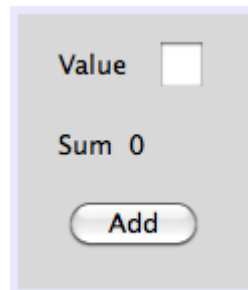
    public void add(int pValue)
    {
        sum = sum + pValue;
    }

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



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

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



Variable Name: valueTextField
Text:

Variable Name: sumLabel
Text: 0

Variable Name: addButton
Text: Add

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

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

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

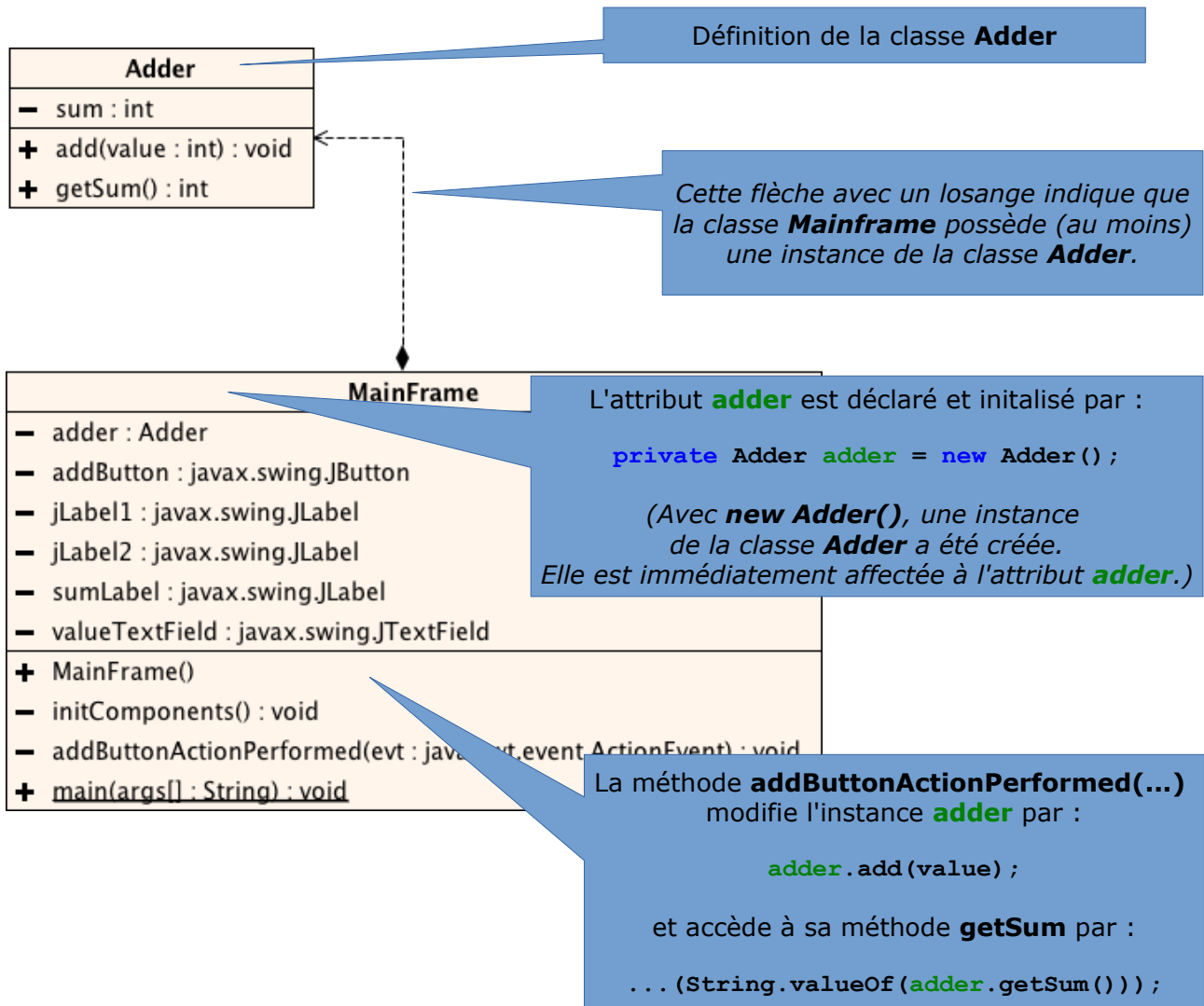
Récupère le texte du champ d'entrée,
convertit ce texte en un entier et
sauvegarde cette valeur entière dans la variable "value".

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

Ajoute la valeur
au calculateur.

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

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



Questions :

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

6. Manipulation de NetBeans

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

6.1. *Installation*

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

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

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

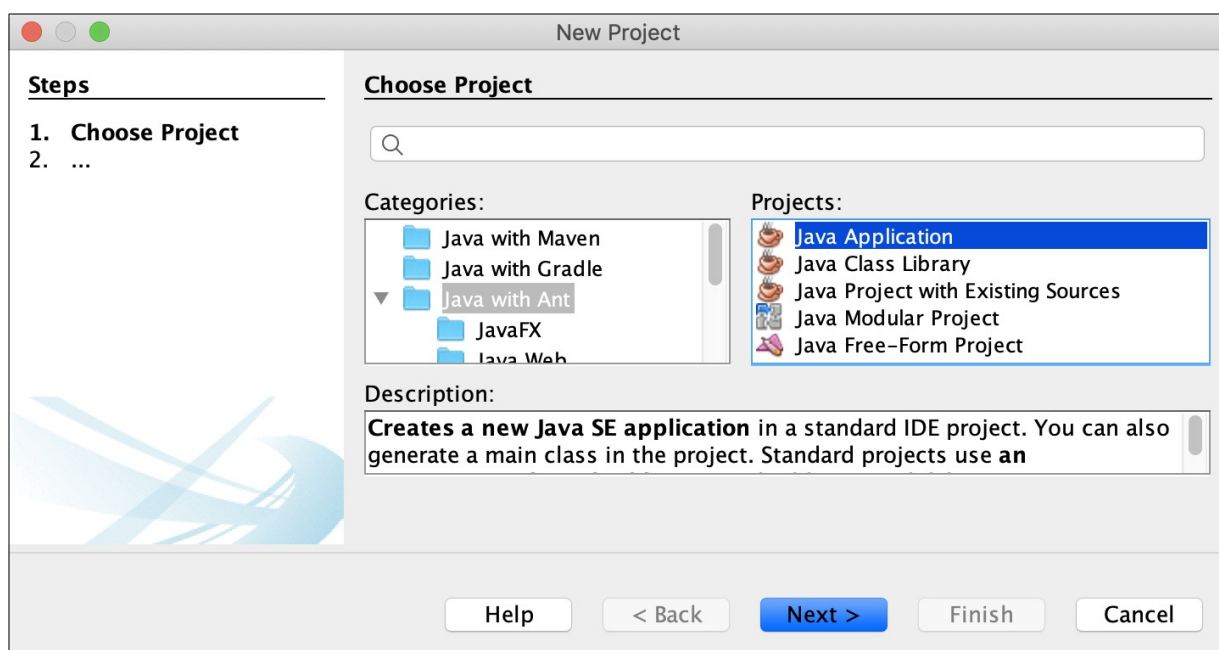
6.2. *Les projets*

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

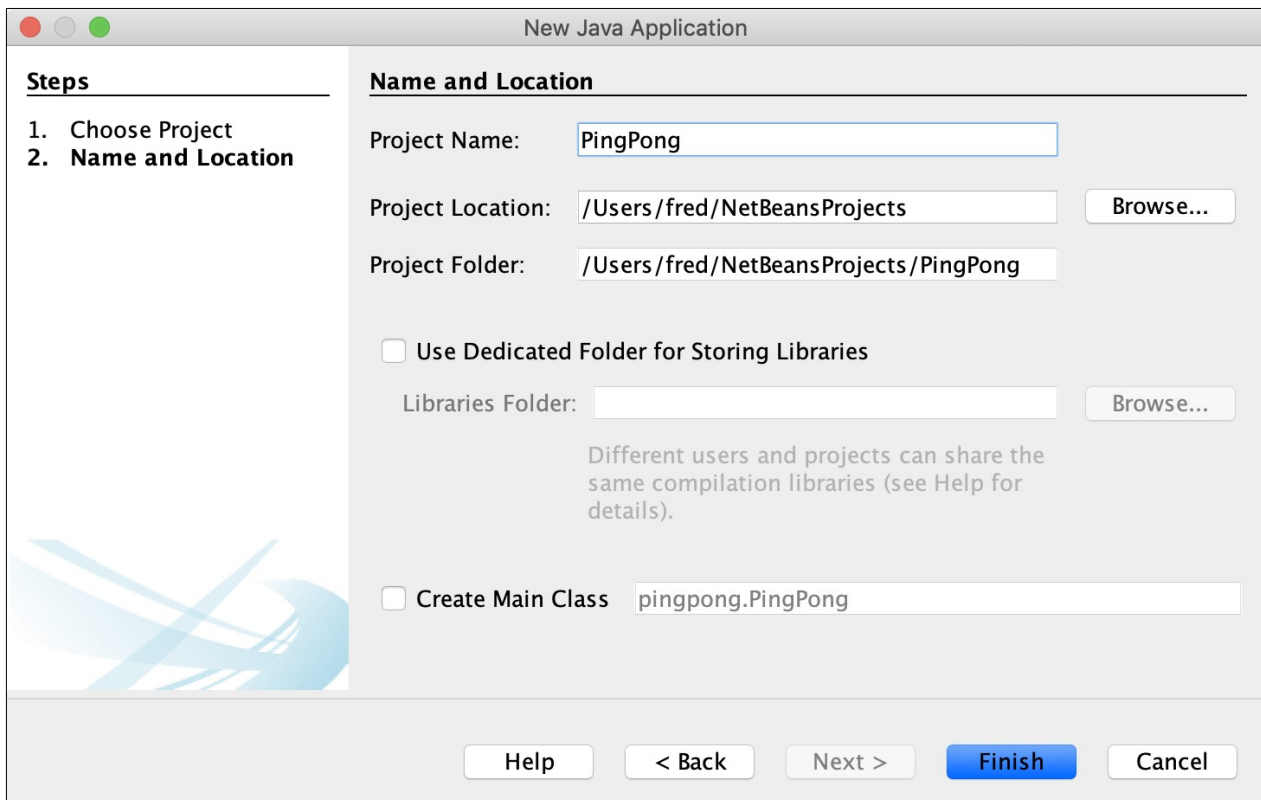
6.2.1. *Créer un nouveau projet*

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

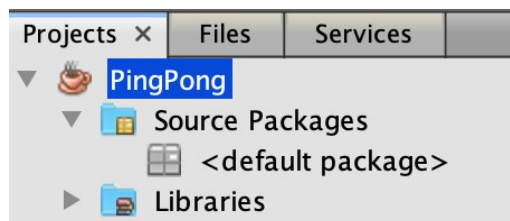
1. Choisissez dans le menu principal « File », puis « New project ... ».
2. Dans le dialogue suivant, sélectionnez dans la liste intitulée « Categories » l'élément « Java with Ant », puis dans la liste intitulée « Projects » l'élément « Java Application ».



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



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

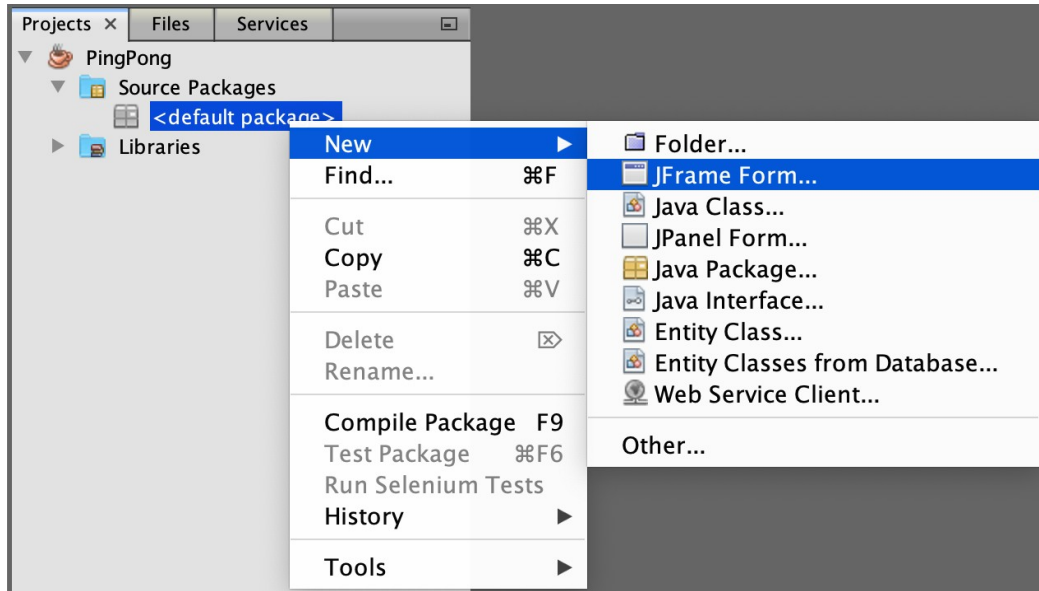


6. Vous venez de mettre en place un projet vide, qui ne contient encore aucune classe. Le chapitre suivant montre comment ajouter des classes.

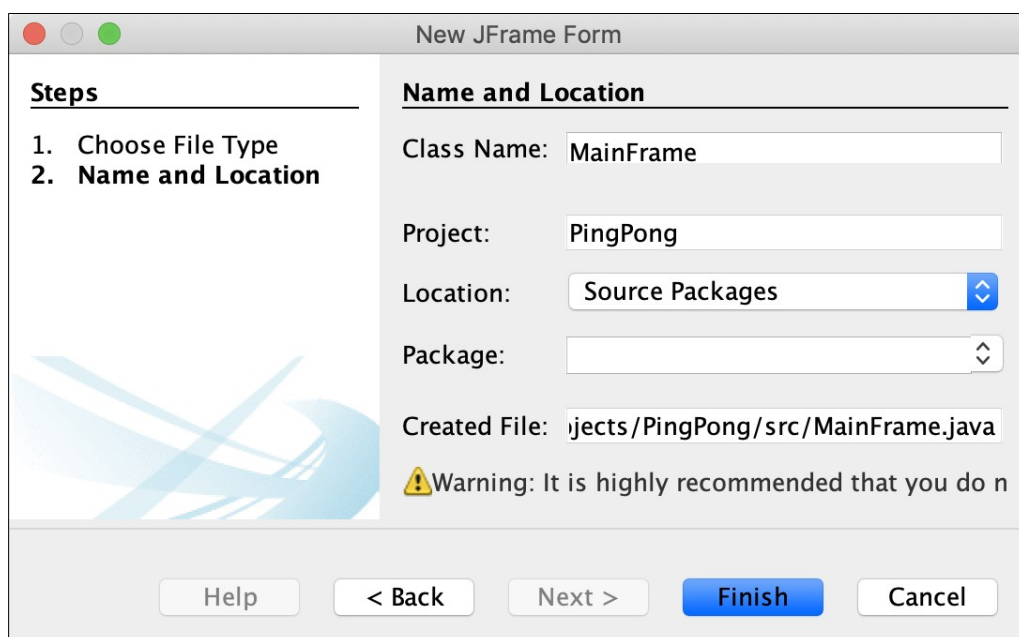
6.2.2. Ajouter des classes à un projet

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

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



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



6.3. Importer un projet de Unimozer dans NetBeans

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

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

6.4. Distribuer une application

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

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

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

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

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

7. Notions importantes

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

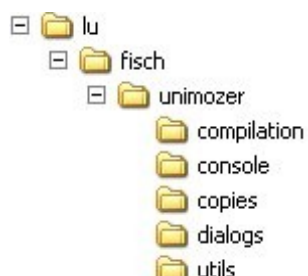
7.1. *Les paquets*

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

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

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

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



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

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

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

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



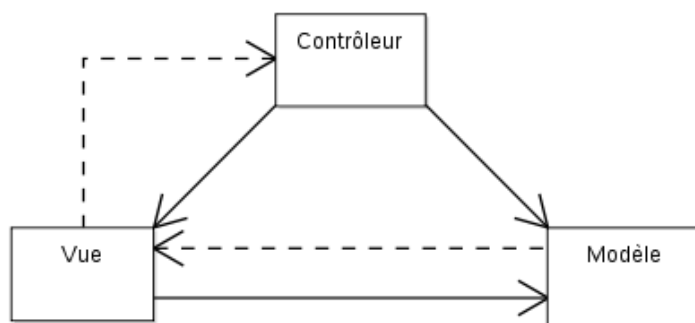
Il faut utiliser la touche **<Ctrl><Space>** pour activer la **complétion de code** lors de l'écriture du code, puis valider l'entrée par **<Enter>**. Ensuite, le programme placera automatiquement l'instruction « **import** » correspondante au début du code.

7.2. Le modèle « MVC »

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

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

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



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

7.2.1. Le modèle

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

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

7.2.2. La vue

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

7.2.3. Le contrôleur

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

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

7.2.4. Flux de traitement

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

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

7.2.5. Avantages du MVC

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

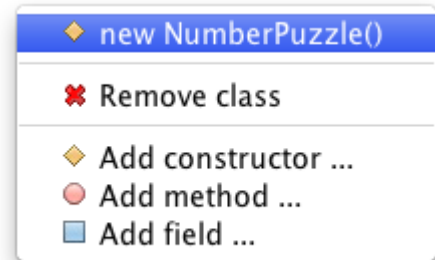
Exemples :

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

7.3. Création de nouveaux objets avec 'new'

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

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



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

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

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

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

Remarques:

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

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

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

Par exemple :

```
Clock myClock; //l'objet myClock est déclaré mais n'existe pas encore
...
myClock = new Clock(); //un objet du type Clock est créé et affecté à myClock
```

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

Exemple :

Soit la classe suivante :

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

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

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

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

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

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

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

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

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

Rappel :

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

7.4. La valeur 'null'

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

Exemples :

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

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

myPoint

null

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

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

myPoint

x: 12
y: 50

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

Point(...,....)
toString()

En ajoutant maintenant une ligne fatale ...

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

myPoint

x: 12
y: 50

Point(...,....)
toString()

Objet perdu/inaccessible dans la mémoire.
Un objet sans référence sera supprimé
automatiquement par le 'garbage collector'

Autre exemple :

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

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

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

7.5. Égo-référence 'this'

Parfois lors qu'on est en train d'écrire le code pour une classe, il faut faire référence à l'instance actuelle qui existera en mémoire lors de l'exécution. Au moment de la rédaction du code, on ne connaît cependant pas le nom de cette instance, et comme il peut exister beaucoup d'instances avec des noms différents il est même impossible d'en connaître le nom. C'est pour cette raison qu'existe la variable spéciale **this**. Cette variable pointe toujours sur l'instance actuelle, c.-à-d. l'objet lui-même.⁶

Exemple :

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

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

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

Pour ceux qui ne l'auraient pas vu,
ce sont les noms des paramètres
qui ont changé

Problème !!

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

Le problème consiste dans le fait que les noms des paramètres du constructeur sont identiques aux noms des attributs, les noms de paramètres « cachent » alors les attributs, c.-à-d : à l'intérieur du constructeur, les attributs ne sont plus accessibles par leur nom direct !⁷

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

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

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

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

⁶ **this** n'existe pas dans les méthodes statiques

⁷ C'est d'ailleurs pour cette raison que nous avons ajouté le préfixe 'p' au paramètres dans le cours de 11e.

8. Types primitifs et classes enveloppes

En Java, les types **int**, **float**, **double**, **boolean** sont appelés types "primitifs" ou aussi types "simples". Les attributs ou variables de ces types ne sont pas des objets et ils sont utilisables dès leur déclaration, c.-à-d. on n'a pas besoin d'en créer des instances à l'aide de **new**.

(Tout le cours de 11^e a été basé sur des attributs et variables de ces types primitifs, c'est pourquoi dans nos programmes, nous n'avons jamais eu besoin de créer explicitement des instances à l'aide de **new**).

Comme toutes les autres données en Java sont des objets, et la majorité des mécanismes prédéfinis en Java sont prévus pour des objets, il est parfois nécessaire de travailler avec des valeurs numériques qui sont des objets. De cette façon, on peut par exemple profiter de structures de données composées pour créer des listes de nombres (voir : listes de données → chapitre 12).

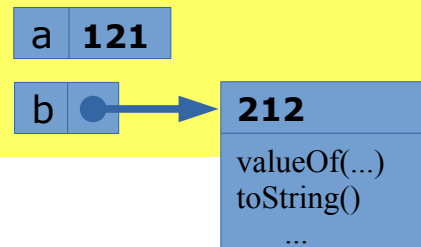
En Java, il existe pour chaque type primitif une classe **enveloppe** (*angl* : **wrapper class**) qui fournit une version objet des données :

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

Exemples :

```
int a = 121 ;
```

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



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

Un autre avantage des classes enveloppes est qu'elles possèdent des méthodes utiles qui ne sont pas disponibles pour les types primitifs. La plus utile de ces méthodes est la méthode **valueOf** qui nous permet de convertir des nombres en textes et vice-versa (→ voir chapitre 9)

9. Les chaînes de caractères « String »

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

9.1.1. Déclaration et affectation

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

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

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

9.1.2. Conversions de types

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

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

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

Exemples

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

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

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

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

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

9.1.3. Autre méthode importante

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

Exemples

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

10. Comparaison d'objets

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

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

10.1. Comparaison deux objets du type String

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

Exemples

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

10.2. Comparaison d'instances de classes enveloppes

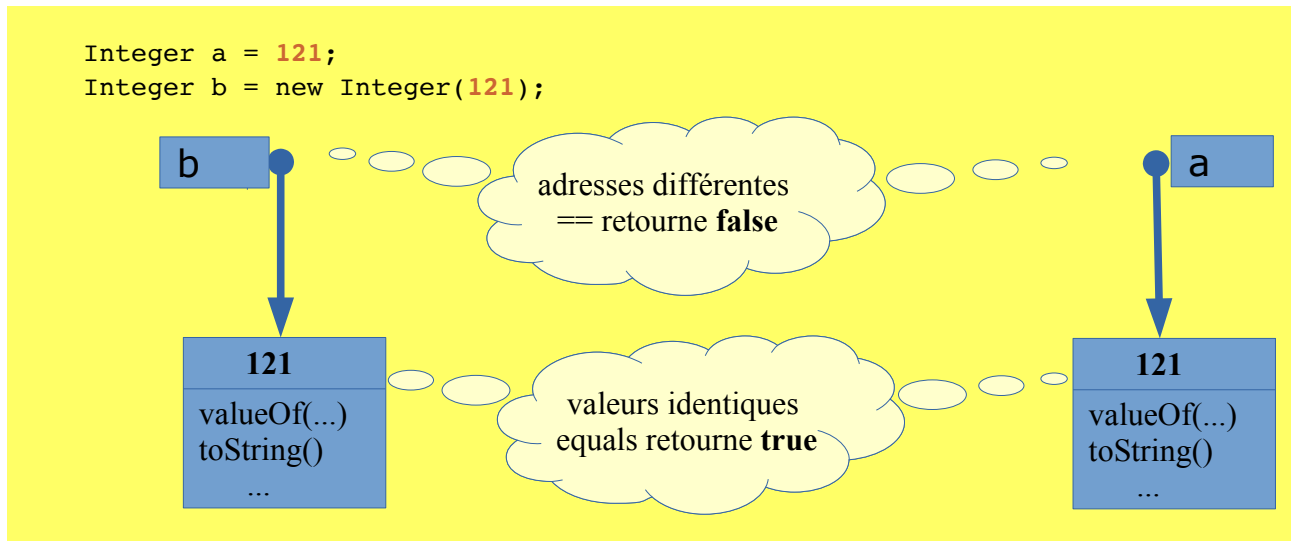
Pour comparer deux instances de classes enveloppes, il faut aussi employer **equals** et **compareTo**.

Exemples

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

10.3. *Explications pour avancés*

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



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

11. Interfaces graphiques simples

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

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



NetBeans possède deux modes d'édition :

Source

Design

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

11.1. Les fenêtres « JFrame »

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

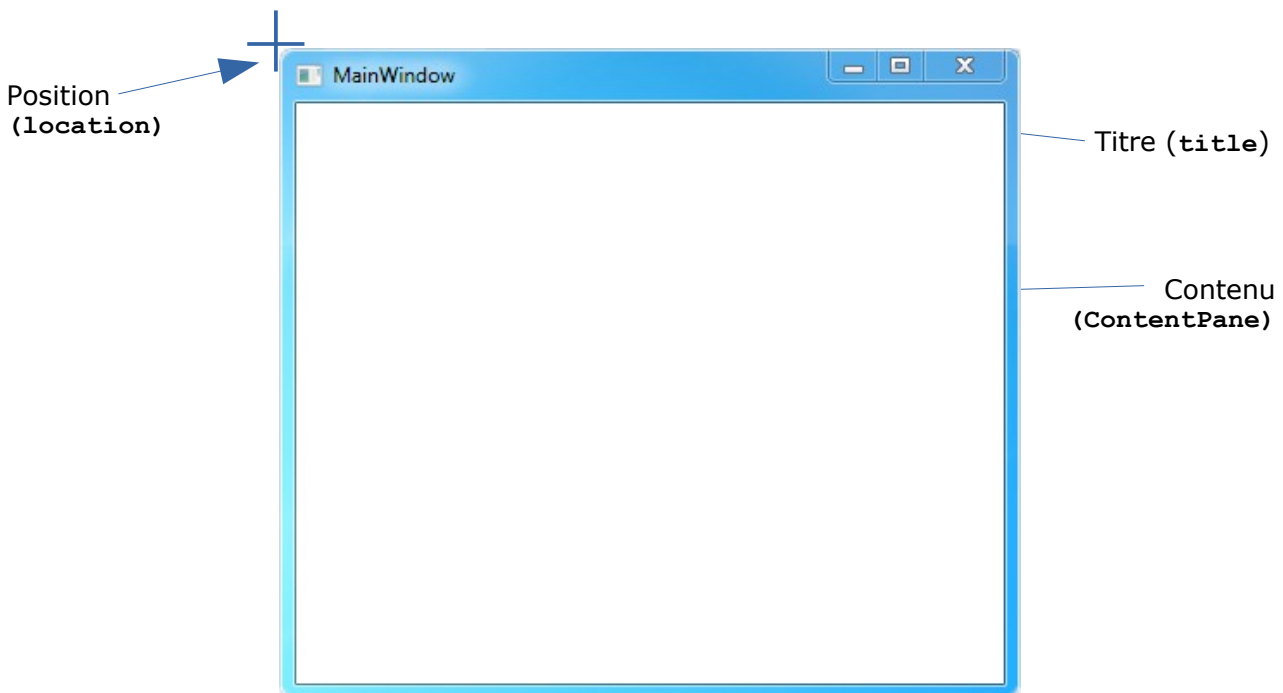
11.1.1. Attributs

Voici les éléments les plus importants d'une fenêtre :

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



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



11.1.2. Initialisations

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

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

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

```
public class MainFrame extends javax.swing.JFrame {  
    // ajoutez vos déclarations de champs ici  
    ...  
  
    /** Creates new form MainFrame */  
    public MainFrame()  
    {  
        initComponents();  
        // ajoutez vos initialisations ici  
        ...  
    }  
    ...  
}
```

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

11.2. Les composants standards

Par composant standard on entend :

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

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

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

11.2.1. Attributs

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

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

non traité
dans le cadre
de ce cours

11.2.2. Événement

Nom dans NetBeans	Description
actionPerformed	<p>Cet événement est déclenché si :</p> <ul style="list-style-type: none"> • l'utilisateur clique sur le bouton ou • appuie <Enter> dans un champ d'édition. <p>Un libellé ne possède pas cet événement.</p>

11.3. Manipulation des composants visuels

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

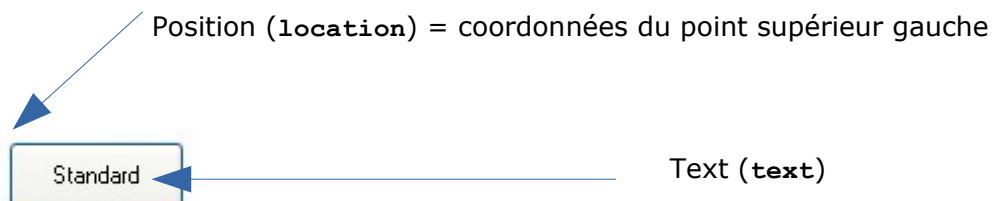
11.3.1. Placer un composant visuel sur une fiche



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

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

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



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

11.3.2. Affichage et lecture de données

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

Exemple :

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

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

Exemple :

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

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

Exemple :

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

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

Exemples :

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

11.3.3. Édition des propriétés

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

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

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

11.3.4. Noms des composants visuels

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

- Les noms de tous les composants visuels se terminent par leur type (en omettant le

préfixe 'J'). Par exemple les noms des boutons (**JButton**) se terminent par **Button**, les noms des libellés (**JLabel**) se terminent par **Label**, les noms des champs texte (**JTextField**) se terminent par **TextField** etc.

- La première partie du nom indique leur fonctionnalité.
- La première lettre du nom est une minuscule.

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

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

Remarque : `<Ctrl>+<Space>`

Il peut sembler fastidieux de devoir entrer des noms aussi 'longs', mais NetBeans (tout comme Unimozzer) vous assiste en complétant automatiquement un nom que vous avez commencé à entrer si vous tapez `<Ctrl>+<Space>`. → voir 16Annexe C - Assistance et confort en NetBeans.

11.3.5. Événements et méthodes de réaction

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

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

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

11.3.6. Attacher une méthode de réaction à un événement

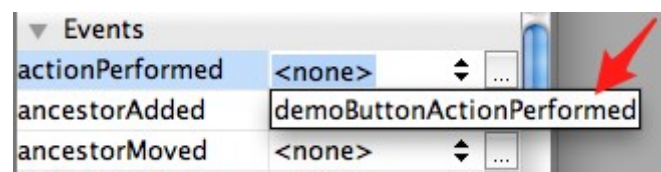
Afin d'attacher une méthode de réaction à un composant, sélectionnez le composant à l'aide de la souris, puis rendez vous dans l'onglet « Events » de l'éditeur des propriétés. Cliquez sur les petites flèches derrière l'événement, puis cliquez sur l'entrée présentée dans le menu contextuel se composant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

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

Exemple :

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

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



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

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

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



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

Exemple 1 :

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

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

Pour avancés / 12GI :

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

Pourquoi est-ce qu'on ne peut pas écrire `'MainFrame.setTitle'` (en supposant que la fiche s'appelle **MainFrame**) ? Quel est le rôle du mot clé `'this'` (→ voir chapitre 7.5) ?

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

12GI :

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

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

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

Que reconnaissez-vous dans ces informations ?

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

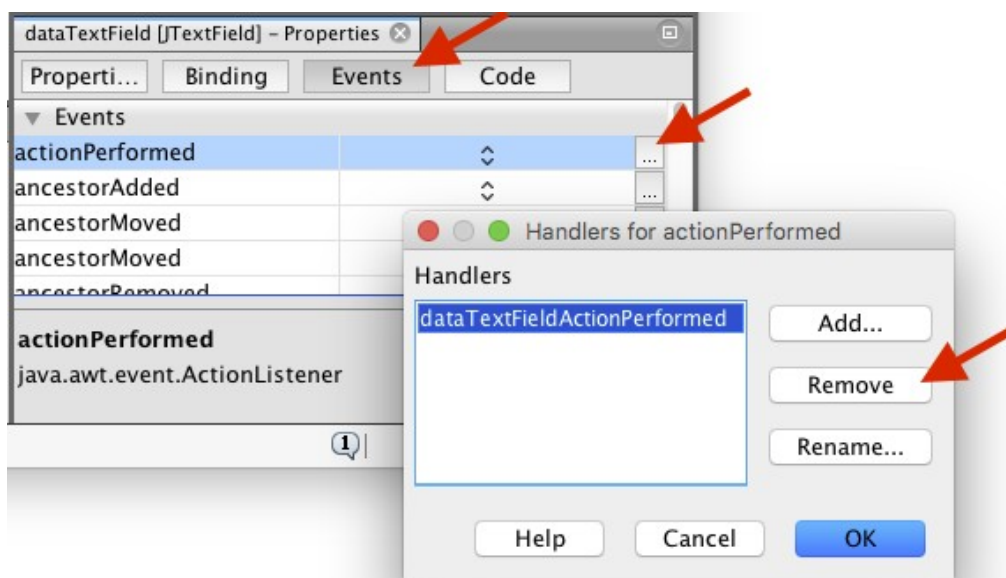
Exemple 2 :

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

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

11.3.7. Comment supprimer un événement

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



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

11.4. Les champs d'entrée « *TextField* »

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

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

11.4.1. Attributs

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

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

Exemples :

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

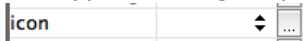
11.4.2. Événements

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



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

11.5. Afficher une image

Les libellés (**JLabel**) peuvent être utilisés pour afficher une image sur une fiche. On peut choisir l'un des types d'images que voici : **JPG**, **PNG** ou **GIF**. Modifiez pour ceci la propriété **icon** du libellé :  en cliquant sur le trois points. Ensuite, vous pouvez choisir une image qui se trouve déjà dans le projet (**Image Within Project**) ou choisir une image sur le disque que vous importez dans le projet (**External Image** → **Import to Project ...**).

11.6. Autres composants utiles

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

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



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

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

Voici les éléments les plus importants de ces composants :

11.6.1. Attributs

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

11.6.2. Événements

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



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

11.7. Les panneaux « **JPanel** »

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

1. regrouper d'autres composants visuels et
2. faire des dessins (cf. chapitre 13).

11.7.1. Attributs

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



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

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

11.7.2. Accès au canevas

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

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

11.8. Confort et ergonomie de la saisie

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

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

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

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

En conséquence, l'utilisateur n'aura plus besoin de passer à la souris lors de la saisie consécutive et répétitive de données. Par ces petites améliorations de l'ergonomie⁸, il économisera un temps considérable et son travail sera bien plus confortable.

Pour déterminer les possibilités d'améliorer l'ergonomie, le programmeur doit (faire) effectuer un certain nombre de tests pratiques avec son produit.

⁸ L'**ergonomie** est « l'étude scientifique de la relation entre l'homme et ses moyens, méthodes et milieux de travail » et l'application de ces connaissances à la conception de systèmes « qui puissent être utilisés avec le maximum de confort, de sécurité et d'efficacité par le plus grand nombre. » (source : Wikipedia.fr)

12. Les listes

12.1. Les listes « *ArrayList* »

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

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

Exemple

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

...

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

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

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

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

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

Depuis Java 7, il n'est plus nécessaire de spécifier le type des éléments contenus dans la liste



lors de l'appel au constructeur. Il suffit donc d'indiquer `<>` lors de l'initialisation.⁹

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

Exemples :

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

alDemoList =

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

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

alDemoList =

Hello	World!
-------	--------

```
alDemoList.add("Some");
```

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

alDemoList =

Hello	World!	Some	1 2 3
-------	--------	------	-------

12.1.1. Listes de types primitifs

Si on désire créer des listes contenant des éléments de types primitifs, donc par exemple `int` ou `double`, il faut passer obligatoirement par les classes enveloppes ! (cf. chapitre 8)

Déclaration d'une liste de nombres entiers :

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

Déclaration d'une liste de nombres décimaux :

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

12.1.2. Méthodes

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

Méthode	Description
<code>boolean add(Object)</code>	Permet d'ajouter un élément à la liste. (Retourne toujours true – le résultat est en général ignoré.)
<code>void clear()</code>	Vide la liste en supprimant tous les éléments.
<code>boolean contains(Object)</code>	Teste si la liste contient un élément donné.
<code>Object get(int)</code>	Retourne l'élément à la position indiquée dans le paramètre.
<code>int indexOf(Object)</code>	Retourne la position de l'élément indiqué dans le paramètre ou -1 si l'élément ne se trouve pas dans la liste.
<code>Object remove(int)</code>	Supprime l'élément à la position indiquée dans le paramètre. Les éléments qui suivent l'élément supprimé avancent automatiquement d'une position. (Retourne l'élément supprimé)

⁹ En Java 6 et dans les versions antérieures, il faut répéter le type d'éléments lors de l'initialisation. P.ex. : `ArrayList<Person> alMyList = new ArrayList<Person>();`

	comme résultat – le résultat est en général ignoré.)
Object set(int, Object)	Remplace l'élément à la position spécifiée par celui passé en tant que paramètre. (Retourne l'élément supprimé comme résultat – le résultat est en général ignoré.)
int size()	Retourne la taille de la liste, donc le nombre d'éléments y contenus.
boolean isEmpty()	Teste si la liste est vide (c.-à-d. test identique à size()==0)
Object[] toArray()	Transforme la liste en un « Array ». Cette méthode est uniquement nécessaire pour la visualisation des éléments à l'aide d'un composant « JList » (voir chapitre suivant).

Remarques :

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

Exemples :

Soit la liste suivante :

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

Effet	Code
Ajouter les noms "Jean", "Anna" et "Marc" à la liste.	<pre>allList.add("Jean"); allList.add("Anna"); allList.add("Marc");</pre>
Supprimer le premier nombre de la liste (celui à la position 0).	<pre>allList.remove(0);</pre>
Sauvegarder le nombre d'éléments dans la variable count .	<pre>int count = allList.size();</pre>
Sauvegarder la position de l'élément "Marc" dans la variable pos .	<pre>int pos = allList.indexOf("Marc");</pre>
Remplacer l'élément à la position pos (ici le nom "Marc") avec le nom "Michelle".	<pre>allList.set(pos, "Michelle");</pre>
Sauvegarder le dernier élément de la liste dans la variable last .	<pre>String last = allList.get(allList.size()-1);</pre>
Tester si la liste contient la valeur "Marc".	<pre>if (allList.contains("Marc")) System.out.println("contained"); else System.out.println("not contained");</pre>
Vider la liste.	<pre>allList.clear();</pre>
Tester si la liste est vide.	<pre>if (allList.isEmpty()) ... ou bien : if (allList.size()==0) ...</pre>

Conseil :

Consultez l'annexe C : 16.3 Ajout de propriétés et de méthodes <Insert Code...> pour simplifier la rédaction d'une classe contenant une **ArrayList**.

12.2. Les listes « JList »

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

12.2.1. Attributs

Désignation	Propriété dans NetBeans	Accesseurs
les éléments contenus dans la liste	<code>model</code>	<code>void setListData(Object[])</code>
le mode de sélection de la liste	<code>SelectionMode</code> Remarque: Veuillez toujours choisir la valeur « SINGLE » ¹⁰ .	
la position de l'élément sélectionné de la liste	<code>selectedIndex</code>	<code>void setSelectedIndex(int)</code> <code>int getSelectedIndex()</code> <code>void clearSelection()</code>

Remarque :

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

12.2.2. Événements

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



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



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

¹⁰ **SelectionMode** a par défaut la valeur **MULTIPLE_INTERVAL** qui permet de sélectionner plusieurs groupes de données dans une **JList**. Dans notre cours, nous n'aurons besoin que d'une seule sélection par liste. Ainsi, nous allons toujours choisir l'option **SINGLE**, ce qui nous évitera aussi le traitement beaucoup plus complexe de sélections multiples.

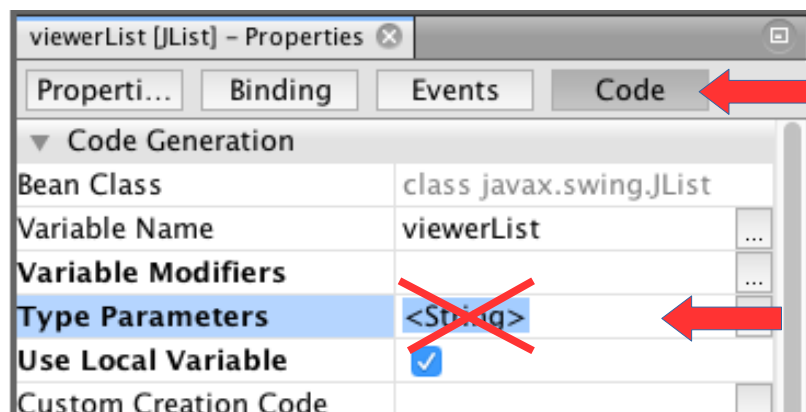
12.2.3. Préparer la liste pour accepter toute sorte d'objets

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

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

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

- Dans le mode *Design*, sélectionnez la **JList** en question et choisir '**Code**' dans l'éditeur de propriétés.
- Supprimez ensuite l'indication **<String>** de '**Type Parameters**' et confirmez par 'Enter'.

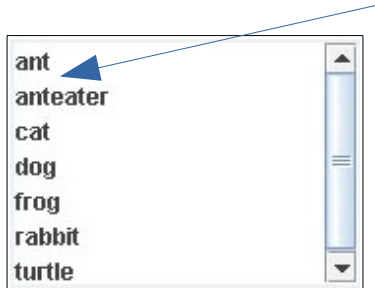


Maintenant, la déclaration dans la liste des composants à la fin du code dans **MainFrame** devrait se présenter comme suit :

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


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

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



Liste avec différents éléments.

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

Exemple :

Soit la structure de donnée suivante :

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

On initialise la liste de la manière suivante :

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

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

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

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

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

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

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

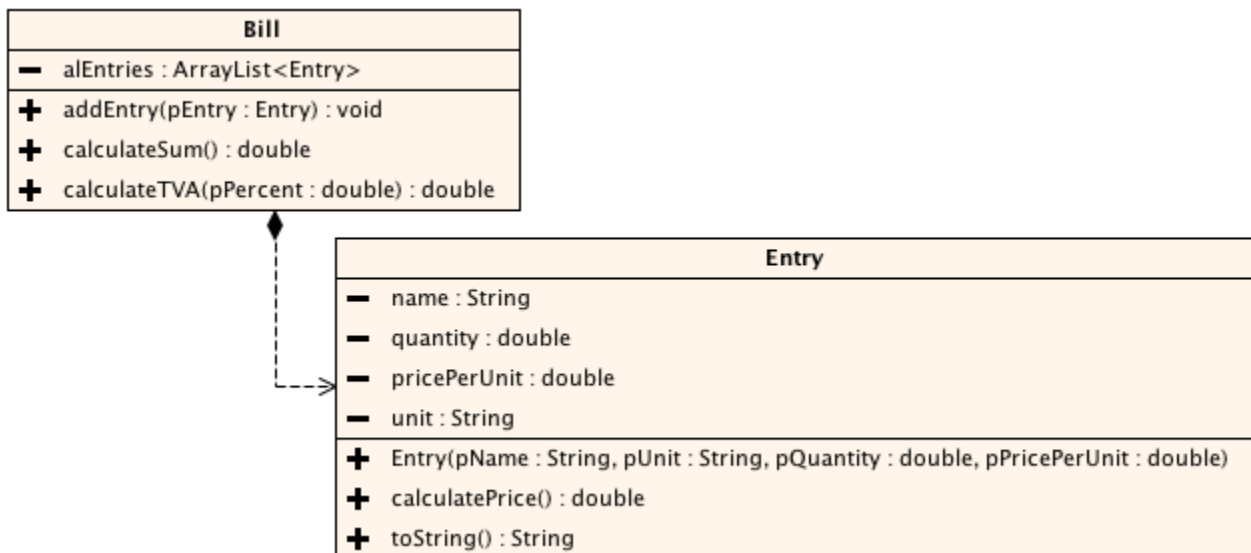
En modifiant le code de la manière suivante, ce n'est plus la position qui est affichée, mais l'élément lui-même :

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

12.3. Les listes et le modèle MVC

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

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



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

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

```

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

Remarquez que le résultat de **toArray()** est **Object[]**. En effet, la méthode **setListData(Object[])** de **JList** nécessite un paramètre du type **Object[]** et c'est précisément ce qu'on va lui offrir.¹¹

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

```

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

¹¹ **Object[]** est un tableau d'objets. La structure 'tableau' (array) est une structure de base de Java. Son utilité correspond à celle de **ArrayList**, mais elle est beaucoup moins confortable que **ArrayList**. Dans ce cours, nous ne traitons pas ce type de tableaux. Il suffit que vous reteniez qu'il faut noter des crochets **[]** à la fin.

13. 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.

13.1. Le canevas « Graphics »

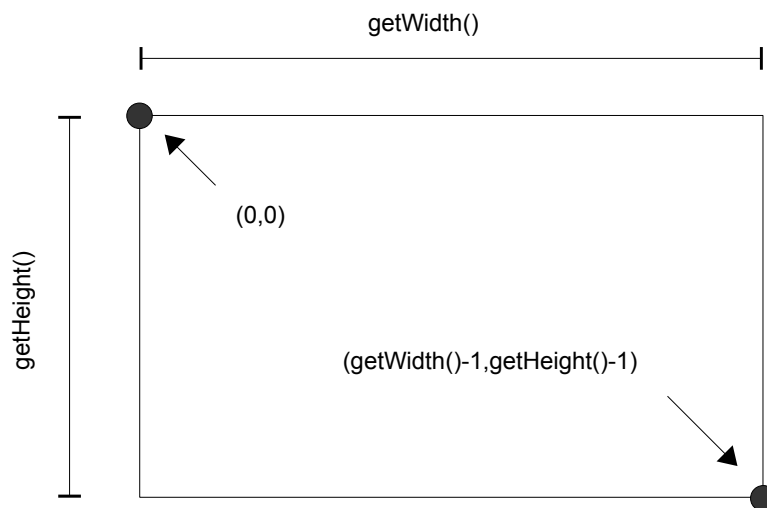
Un artiste peintre dessine sur un canevas (DE : *die Leinwand*). En informatique l'image de l'écran est aussi appelée canevas et elle consiste en une grille composée de pixels minuscules, des points qui peuvent prendre des millions de couleurs différentes.

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.

13.1.1. 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.



13.1.2. 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 (x1,y1) et (x2,y2) .
<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 (x,y) représente le point supérieur gauche, tandis que width et height 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 (x,y) représente le point supérieur gauche, tandis que width et height représentent sa largeur respectivement sa hauteur.
<code>void drawString(String s, int x, int y)</code>	Dessine le texte s à la position (x,y) tel que (x,y) représente le point inférieur de l'alignement de base du texte.

Remarque :

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

13.1.3. 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 :

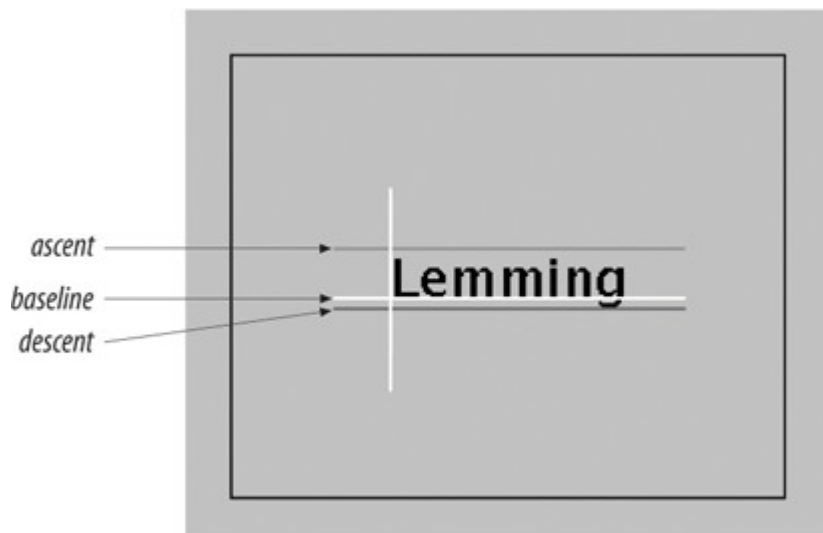
```
/**
 * Méthode qui dessine une _____
 */
public void paintComponent(Graphics g)
{
    g.setColor(Color.RED);
    g.drawLine(0,0,getWidth(),getHeight());
    g.drawLine(0,getHeight(),getWidth(),0);
}
```

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 13.2).

13.1.4. 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>

13.2. 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**'.

13.2.1. 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 alpha permet de fixer le degré de transparence pour la couleur. Exemples : alpha=0 => la couleur est complètement transparente alpha=127 => la couleur est transparente à 50% alpha=255 => la couleur est opaque (non transparente)

Exemples :

<code>Color color1 = new Color(0,0,0);</code>	crée une couleur _____
<code>Color color2 = new Color(0,0,128);</code>	crée une couleur _____
<code>Color color3 = new Color(200,200,0);</code>	crée une couleur _____
<code>Color color4 = new Color(255,0,255);</code>	crée une couleur _____
<code>Color color5 = new Color(64,64,64);</code>	crée une couleur _____
<code>Color color6 = new Color(0,0,0,127);</code>	crée une couleur _____
<code>Color color7 = new Color(0,255,0,64);</code>	crée une couleur _____

13.2.2. 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>

13.3. 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.

13.3.1. Constructeur

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

13.3.2. Attributs

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

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

13.3.3. Méthodes

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

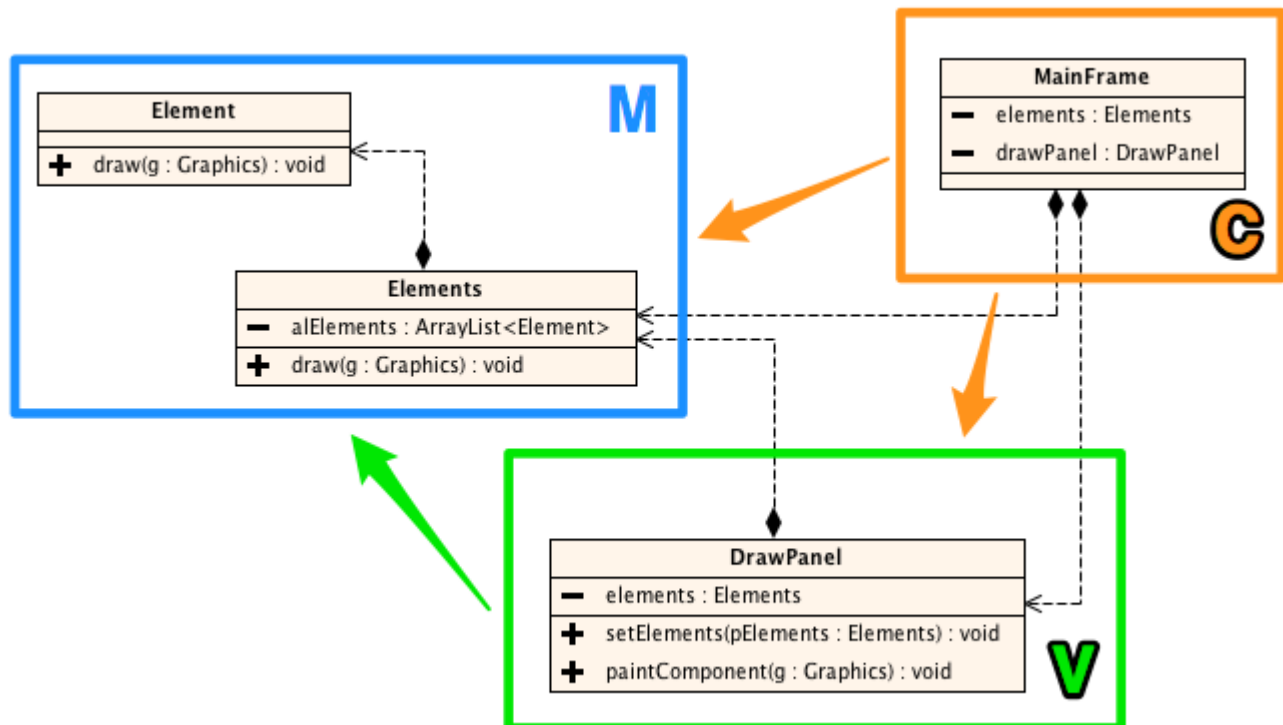


Voici le lien direct vers la page JavaDoc :

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

13.4. 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 classes¹². 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.
- 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.

¹² 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.

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);
    }
}
```

14. 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).

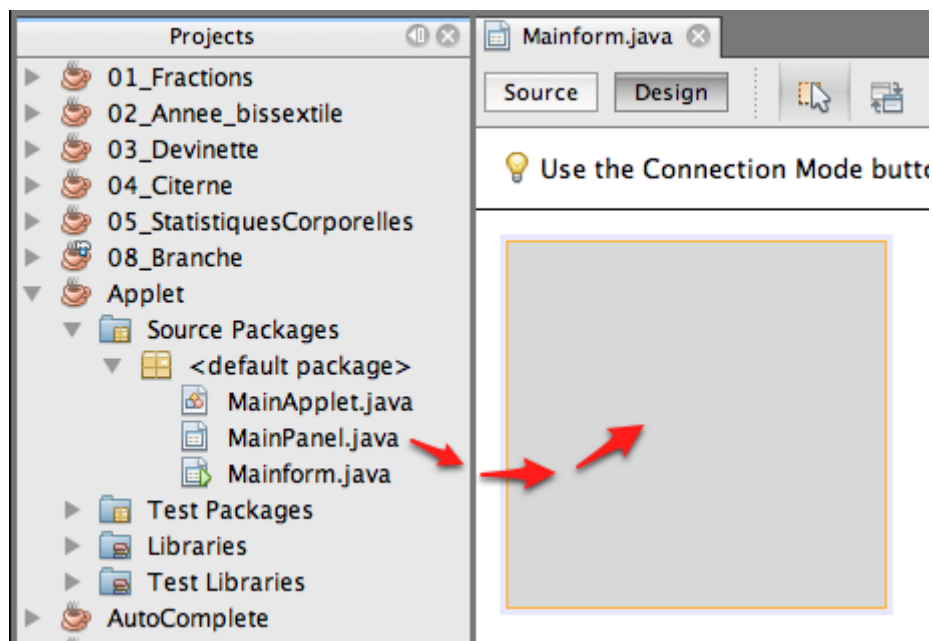
14.1. *Java Applet*

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 :

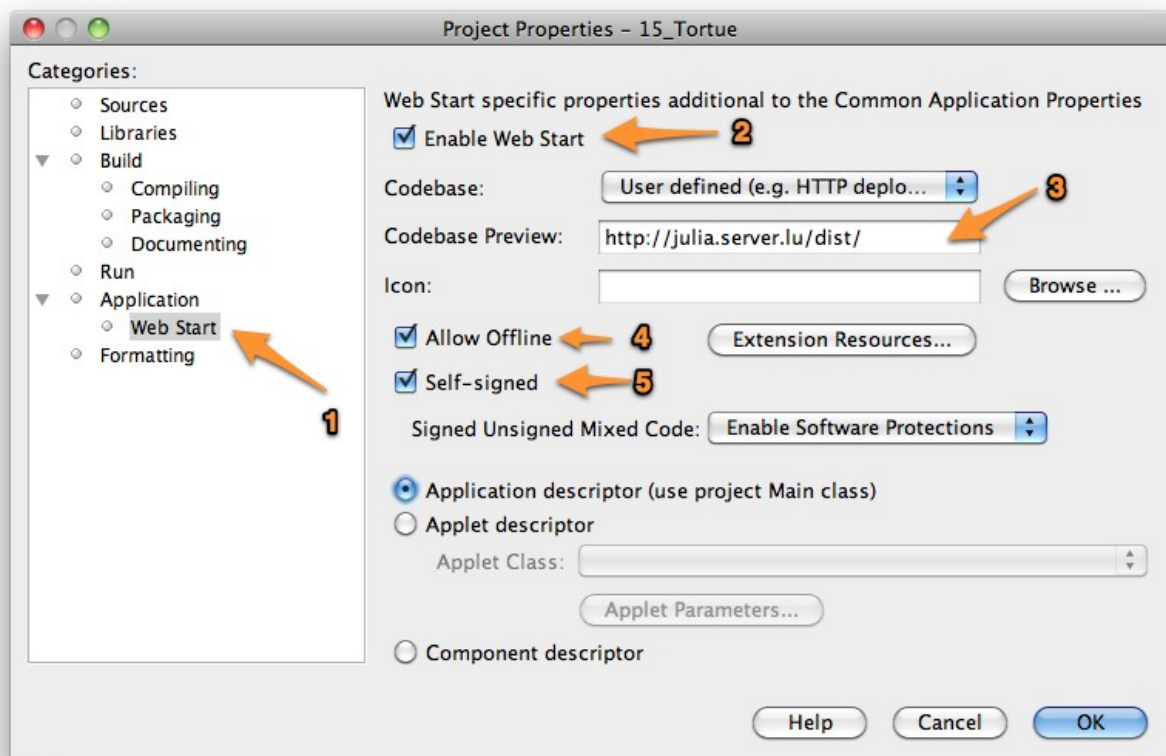
```
<html>
  <head>
    <title>Titre de votre application</title>
  </head>
  <body>
    <applet archive="Applet.jar,lib/swing-layout-1.0.4.jar"
      code="MainApplet"
      width="680" height="500"></applet>
  </body>
</html>
```

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.

14.2. 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 Projet**
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.

15. 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.

15.1. Impression à l'aide du logiciel « **JavaSourcePrinter** »

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

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

Sélection du dossier de base

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

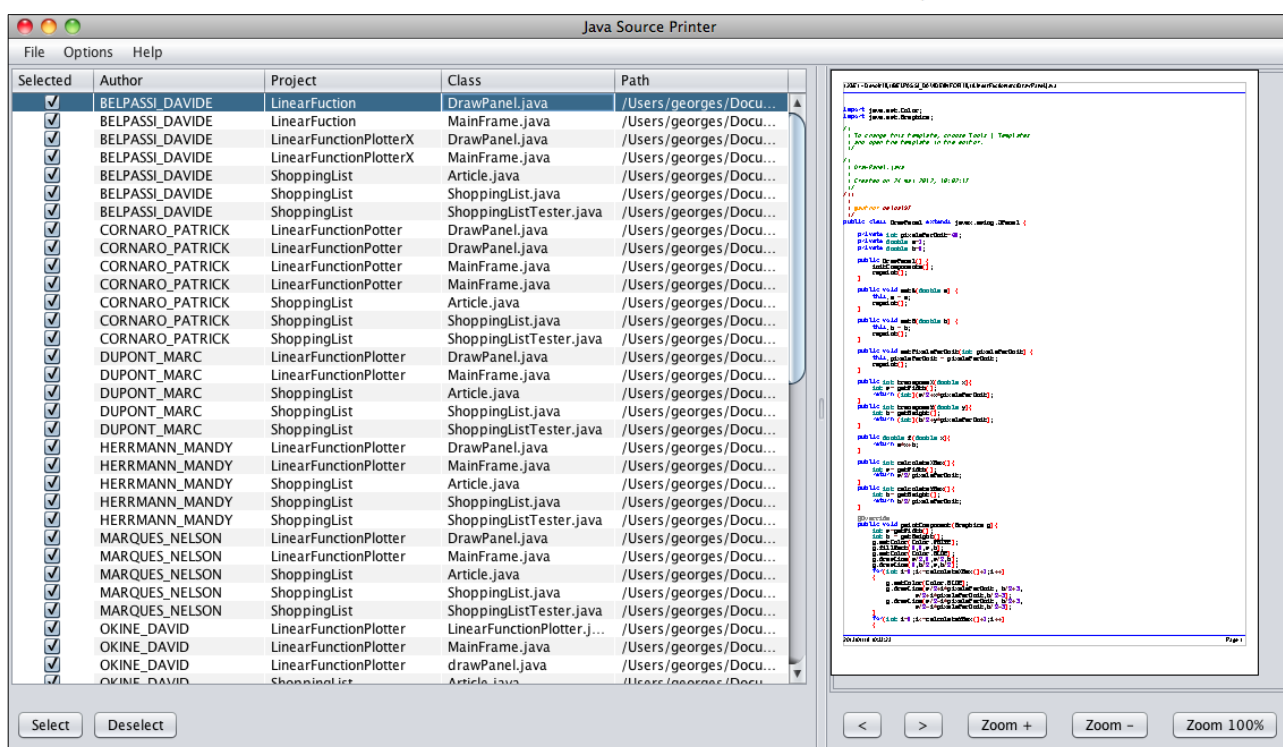
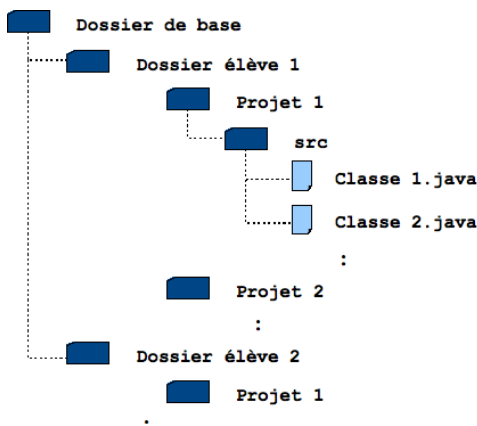


Table de sélection pour impression

Panneau de prévisualisation

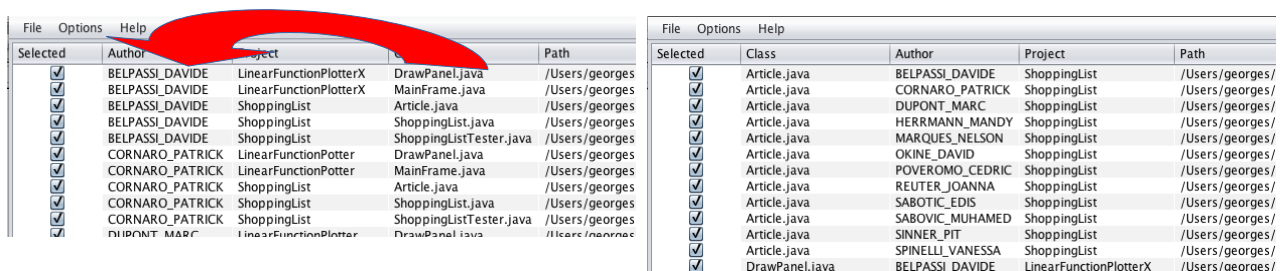
Sélection des classes pour impression

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	ShoppingListTester.java	/Users/georges/
<input checked="" type="checkbox"/>	CORNARO_PATRICK	ShoppingList	ShoppingListTester.java	/Users/georges/
<input checked="" type="checkbox"/>	DUPONT_MARC	LinearFunctionPlotter	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

15.2. *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.

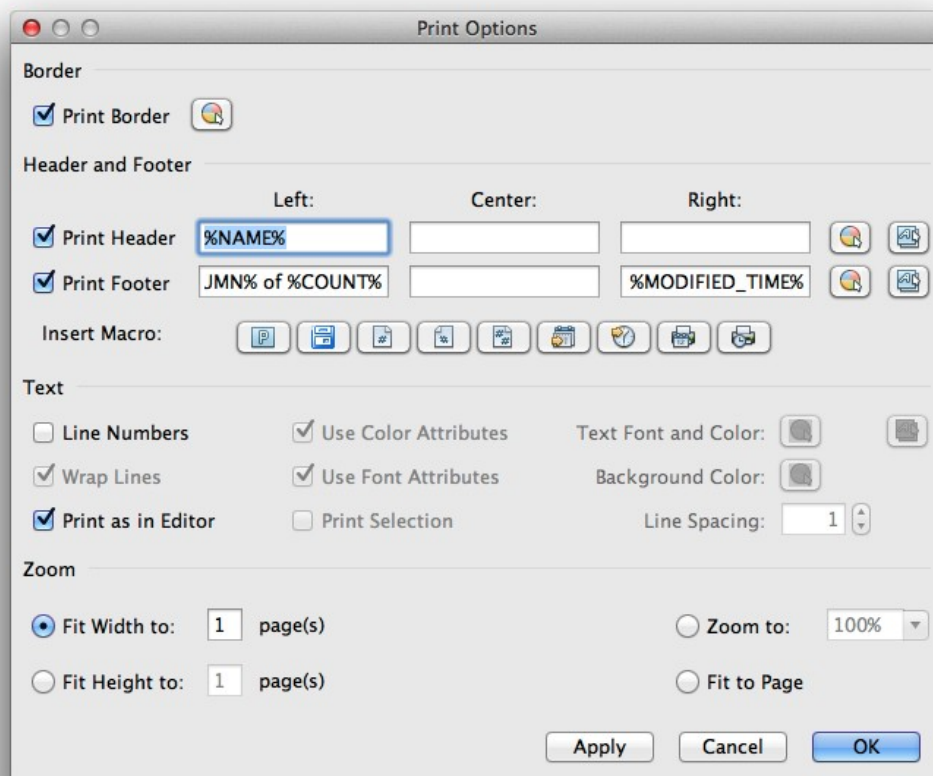
Cliquez d'abord sur **File** → **Print** et vous vous retrouvez dans le mode '**Print Preview**' qui vous permet de vérifier la disposition des pages.

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.

16. 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 ¹³:

16.1. 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.

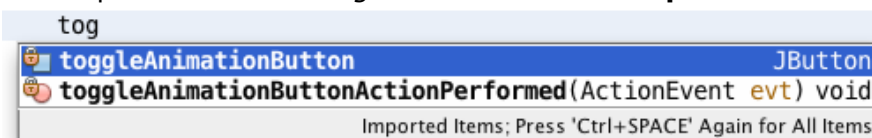
Mais attention : ces suggestions ne correspondent pas toujours à nos intentions ou besoins. Ceci compte surtout pour les valeurs des paramètres que NetBeans propose...

16.2. 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>.



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

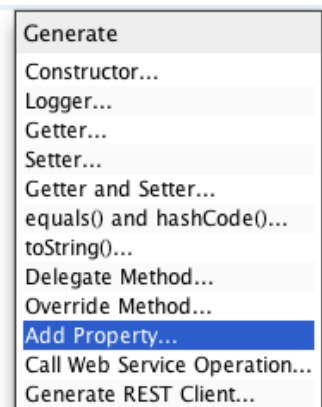
La définition d'une classe commence en général par un certain nombre de tâches monotones et répétitives (définition de propriétés, accesseurs, modificateurs, constructeurs, JavaDoc, méthode toString(), etc.).

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>

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.



¹³ Voir aussi: <https://netbeans.org/kb/docs/java/editor-codereference.html>

16.4. *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** **alLines** 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 :

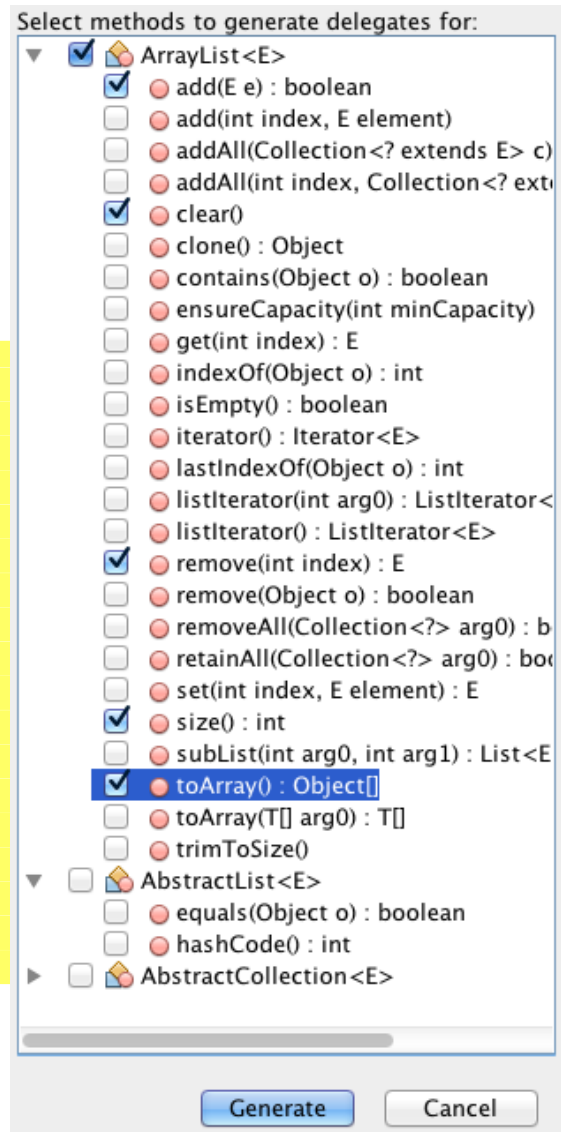
```
public int size() {
    return alLines.size();
}

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

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

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

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



16.5. 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.

16.6. 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.

16.7. 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 :

