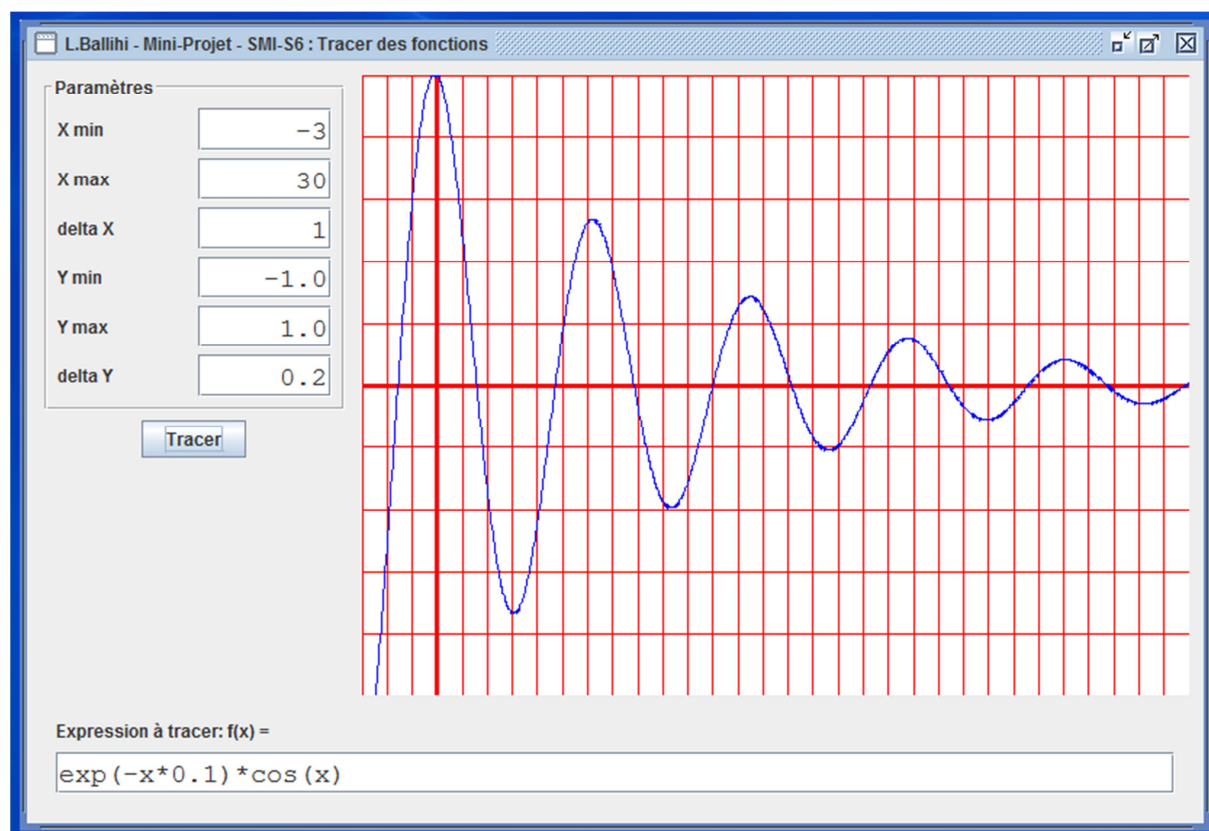




## Mini-Projet : Tracer des fonctions

**Objectif** : L'objectif de ce mini-projet est l'implémentation d'une application qui trace la représentation graphique d'une fonction donnée au moment de l'exécution. L'application finale ressemblera à la figure ci-dessous :



*Tracé d'une fonction*

L'utilisateur écrit dans un champ (en bas du cadre) un texte qui est l'expression d'une fonction réelle d'une variable réelle. Il donne aussi quatre nombres ( $X_{min}$ ,  $X_{max}$ ,  $Y_{min}$  et  $Y_{max}$ ) qui définissent les limites du rectangle de tracé, et deux autres nombres ( $\delta X$  et  $\delta Y$ ) qui définissent les intervalles entre les lignes du quadrillage sur lequel la fonction est tracée. Dans ce quadrillage, deux lignes plus épaisses représentent les axes de coordonnées.

Le travail à faire se décompose en quatre parties :

- définition des structures de données pour la représentation interne de la fonction,
- écriture d'un *analyseur syntaxique* qui construit cette représentation à partir du texte tapé par l'utilisateur,
- mise en place du cadre et des autres composants graphiques,
- programmation du tracé, c'est-à-dire écriture de la méthode **paint** du panneau du dessin.



## 1 - Structures de données

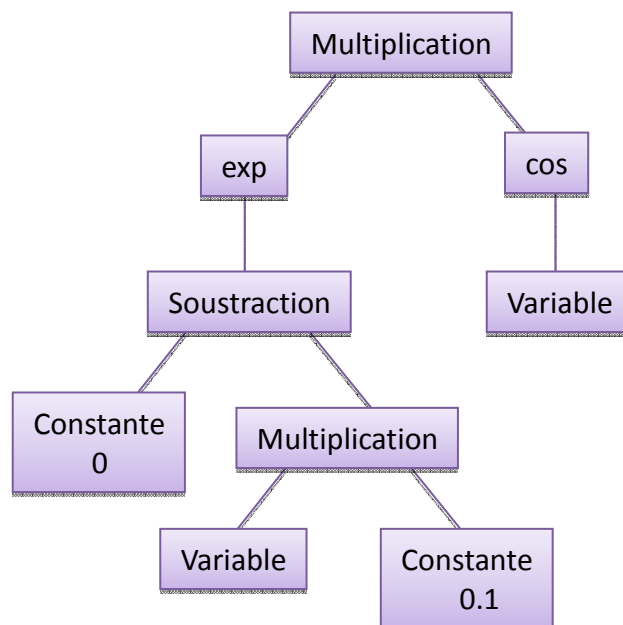
La fonction à tracer sera représentée dans votre programme par un arbre dont les nœuds correspondent aux divers éléments d'une expression arithmétique :

- des constantes, chacune avec sa valeur spécifique,
- diverses occurrences de la variable  $x$ ,
- des opérateurs binaires : addition, soustraction, multiplication, division, chacun portant des références sur ses deux opérandes,
- des opérateurs unaires, avec une référence sur leur argument ; dans notre cas, les seuls opérateurs unaires seront les fonctions prédéfinies :  $\sin$ ,  $\cos$ ,  $\exp$  et  $\log$ , etc.

Par exemple, l'arbre représentatif de l'expression

$$f(x) = \exp(-x * 0.1) * \cos(x)$$

sera quelque chose comme ceci :



### 1.1 - Classes pour réaliser les nœuds

Les nœuds de l'arbre représentatif de la fonction à tracer seront des instances des classes :

```
Fonction (interface)
  FonctionBinaire (classe abstraite)
    Addition (classe)
    Soustraction (classe)
    Multiplication (classe)
    Division (classe)
  FonctionUnaire (classe abstraite)
    Sin (classe)
```



```
Cos (classe)  
Exp (classe)  
Log (classe)  
Constante (classe)  
Variable (classe)
```

Dans la liste ci-dessus, la marge laissée à gauche exprime la relation d'héritage (relations **implements** et **extends**) existant entre ces interfaces et classes. Nous avons déjà vu quelque chose qui ressemble à cela en TD1-TP1, l'objectif de cette partie est d'écrire toutes ces classes. Pour fixer les idées, voici le texte intégral de l'interface **Fonction** :

```
public interface Fonction {  
    double getValue(double x);  
}
```

La signification de cette déclaration est la suivante : à tout objet qui prétend être une [implémentation de l'interface] **Fonction** on pourra demander la valeur qu'il prend lorsque la variable prend la valeur donnée **x**. Ainsi, un élément important de chaque classe concrète implémentant directement ou indirectement l'interface **Fonction** sera la définition d'une méthode **getValue** adaptée à ce que la classe représente.

Toujours pour fixer les idées, voici le texte intégral de la classe **FonctionBinaire** :

```
abstract class FonctionBinaire implements Fonction {  
    protected Fonction filsGauche, filsDroite;  
  
    public FonctionBinaire(Fonction g, Fonction d) {  
        filsGauche = g;  
        filsDroite = d;  
    }  
}
```

Cette classe doit être déclarée abstraite car elle ne définit pas la méthode **getValue** « promise » par l'interface **Fonction** du code ci-dessus il découle que les constructeurs des classes **Addition**, **Soustraction**, etc., devront avoir pour arguments les deux opérandes de l'opération en question.

Il se révélera commode de pouvoir construire une opération unaire en connaissant ou sans connaître son argument (dans le second cas, il faudra revenir sur la fonction unaire pour redéfinir l'argument). Voici une suggestion pour la classe **FonctionUnaire** :

```
abstract class FonctionUnaire implements Fonction {  
    protected Fonction argument;  
  
    void FonctionUnaire (Fonction a) {  
        argument = a;  
    }  
}
```



Si les explications précédentes vous laissent de marbre même-ci ça devrait pas être le cas, car on a presque tout vu en TD1-TP1, voici, à titre d'exemple, deux des classes qui restent à écrire : un opérateur binaire, l'addition, et un opérateur unaire, la fonction sinus. Il ne vous reste plus qu'à vous en inspirer de (très) près pour écrire les autres classes :

```
class Addition extends FonctionBinaire {  
  
    public Addition(Fonction g, Fonction d) {  
        super(g, d);  
    }  
  
    public double valeur(double x) {  
        return filsGauche.getValue(x) + filsDroite.getValue(x);  
    }  
  
    public String toString() {  
        return filsGauche + " + " + filsDroite;  
    }  
}  
  
class Sin extends FonctionUnaire {  
  
    public double getValue(double x) {  
        return Math.sin(argument.getValue(x));  
    }  
  
    public String toString() {  
        return "sin(" + argument + ")";  
    }  
}
```

---

## 2 – Analyseur syntaxique

---

L'analyseur prend en entrée un texte (c'est-à-dire une chaîne de caractères) et produit en sortie l'arbre représentatif de la fonction dont ce texte est l'expression écrite. Durant ce travail, les erreurs de syntaxe dans le texte sont détectées et provoquent l'affichage d'un message et l'abandon de l'analyse (une seule erreur est donc détectée chaque fois).

Le travail d'analyse se compose de deux couches distinctes :

- L'analyse lexicale prend en entrée des caractères et fournit en sortie des « mots », appelés unités lexicales ou tokens.

Par exemple, les 16 caractères de la chaîne "cos ( 0.25 \* x )" sont successivement transformés par l'analyseur lexical dans les 7 unités lexicales "cos", "(", "0.25", "\*", "x", ")", fin-du-texte.

La suite ci-dessus n'est pas homogène, car elle mélange des caractères, c.-à-d. des entiers comme '(', '\*', ')' (n'oubliez pas que les caractères sont représentés par leurs codes ASCII) et des chaînes comme "sin", "0.25", "x", etc. En réalité, nous utiliserons



comme analyseur lexical un objet `StreamTokenizer`, qui donnera les unités lexicales suivantes, toutes de type `int` : `TT_WORD`, `'('`, `TT_NUMBER`, `'*'`, `TT_WORD`, `')`, `TT_EOF`.

Dans cette suite, `TT_WORD`, `TT_NUMBER`, etc., sont des constantes entières conventionnelles de la classe `StreamTokenizer`. Lorsque l'analyseur est placé sur une unité de type `TT_NUMBER`, le membre `nval` permet d'accéder à la valeur numérique (de type `double`) dont il s'agit. Si l'analyseur est placé sur une unité de type `TT_WORD`, le membre `sval` permet d'accéder à la chaîne de caractères en question.

En définitive, l'analyse lexicale de la chaîne `"cos (0.25 * x)"` par un objet `StreamTokenizer` donnera la suite d'entités :

- – `TT_WORD`, avec `sval = "cos"`
- – `'('`
- – `TT_NUMBER`, avec `nval = 0.25`
- – `'*'`
- – `TT_WORD`, avec `sval = "x"`
- – `')`
- – `TT_EOF`

L'analyseur lexical se charge en outre de faire disparaître les caractères blancs (c.-à-d. : espaces, tabulations, fins de ligne).

- L'analyse syntaxique prend en entrée les unités lexicales produites par la couche précédente et s'en sert pour construire l'arbre qui représente l'expression analysée.

C'est dans cette étape que les erreurs de syntaxe sont détectées. Chacune provoquera le lancement d'une exception comportant un texte explicatif de l'erreur.

## Principe de l'analyse syntaxique

Le point de départ d'un analyseur syntaxique est la grammaire du langage considéré. Dans notre cas, elle est définie par trois règles, *expression*, *terme* et *facteur*, qu'on peut représenter par les figures Figure1, Figure2 et Figure3.

Pour interpréter ces figures, il faut les parcourir dans le sens des flèches, en faisant des choix lorsque des embranchements se présentent. Les éléments écrits dans des cercles ou des boîtes aux extrémités arrondies représentent des unités lexicales (ou *tokens*) qui doivent se présenter tels quels dans le texte analysé. Les éléments écrits dans des rectangles représentent des constructions décrites par d'autres figures.

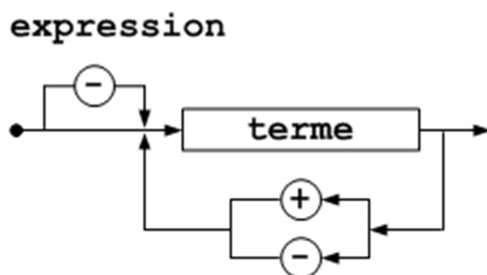


Figure 1 - Syntaxe d'une expression

La figure 1 exprime ceci : une *expression* est constituée par une suite de *termes* séparés entre eux par des opérateurs + ou -. La suite elle-même peut être précédée d'un signe -.

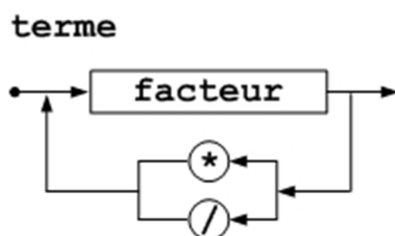


Figure 2 - Syntaxe d'un terme

La figure 2 dit : un *terme* est une suite de *facteurs*, séparés entre eux par des opérateurs \* ou /.

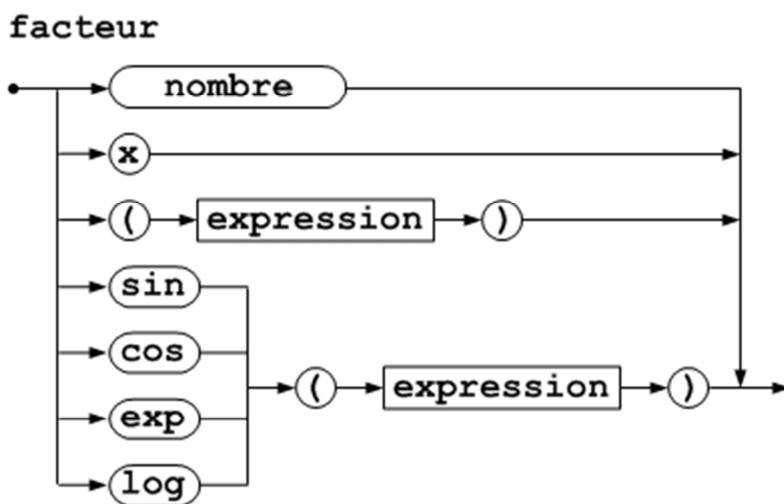


Figure 3 - Syntaxe d'un facteur

Enfin, la figure 3 affirme : un *facteur* est soit un *nombre*, soit une occurrence de la *variable*  $x$ , soit une *expression* entre parenthèses, soit enfin le nom d'une *fonction* (connue) suivi d'une *expression* placée entre parenthèses.



## 2.1 - Ecriture de l'analyseur

L'analyse lexicale sera prise en charge par la classe **java.io.StreamTokenizer**, fournie dans la bibliothèque standard. Un objet **StreamTokenizer** fait l'analyse lexicale d'un flot (*stream*) de caractères, alors que nous aurons ici une chaîne (*string*) de caractères : il faudra donc transformer cette dernière en un flot, cela est le rôle d'un objet **StringReader**.

(Attention. La bibliothèque Java offre aussi une classe **java.util.StringTokenizer**. Mais elle n'est pas équivalente à **StreamTokenizer**, en fait elle ne couvre pas nos besoins.)

L'essentiel du travail à faire ici est donc l'écriture d'une classe **Analyseur**. Elle comporte un constructeur public **Analyseur(String texte)**, une méthode publique **Fonction analyser()** qui représente l'analyse de la chaîne donnée au constructeur, et trois méthodes privées, appelées par exemple **analyserExpression**, **analyserTerme** et **analyserFacteur**.

Pour fixer les idées, voici un programme de test, une application Java qui analyse le texte donné en premier argument, affiche l'expression obtenue ainsi que les valeurs que cette expression prend pour des valeurs de **x** données comme autres arguments (vous ajouterez cette méthode à votre classe **Analyseur** afin de la tester) :

```
public static void main(String[] args) {  
    try {  
        Analyseur analyseur = new Analyseur(args[0]);  
        Expression expression = analyseur.analyser();  
  
        System.out.println("f(x) = " + expression);  
  
        for (int i = 1; i < args.length; i++) {  
            double x = Double.parseDouble(args[i]);  
            System.out.println("f(" + x + ") = " +  
expression.valeur(x));  
        }  
    } catch (Exception exception) {  
        exception.printStackTrace();  
    }  
}
```

Exécution de ce programme sous la ligne de commande :

```
C:\>java Analyseur "x * x - 1" -2 -1 0 1 2  
f(x) = (x)*(x)-(1.0)  
f(-2.0) = 3.0  
f(-1.0) = 0.0  
f(0.0) = -1.0  
f(1.0) = 0.0  
f(2.0) = 3.0
```

**Indications.** Voici le constructeur de la classe **Analyseur** :



```
private StreamTokenizer lexical;  
  
public Analyseur(String texteSource) throws IOException {  
    lexical = new StreamTokenizer(new StringReader(texteSource));  
    lexical.ordinaryChar('/');  
    lexical.ordinaryChar('-');  
}
```

(Par défaut, un **StreamTokenizer** considère que le caractère '/' est le début d'un commentaire et que '-' peut faire partie d'un identificateur ; cela explique les deux réglages ci-dessus).

Une fois l'objet **StreamTokenizer** créé, on le « fait avancer » (c.-à-d. on le positionne sur l'unité suivante) en appelant sa méthode **nextToken()**. Pour connaître l'unité courante on consulte la variable d'instance **ttype**. Egalement à titre d'exemple, voici le texte intégral de la méthode **analyser** :

```
public Fonction analyser() throws IOException, ErreurDeSyntaxe {  
    lexical.nextToken();  
    Fonction resultat = analyserExpression();  
    if (lexical.ttype != StreamTokenizer.TT_EOF)  
        throw new ErreurDeSyntaxe("caractère inattendu à la fin du  
texte");  
    return resultat;  
}
```

Quand on écrit un analyseur syntaxique il est important de se donner une règle précise définissant le positionnement de l'analyseur lexical. Ici, on se donne la règle suivante : juste avant et juste après l'appel de chacune des méthodes **analyserExpression**, **analyserTerme** et **analyserFacteur**, l'unité lexicale courante est la prochaine unité à examiner.

La méthode **analyser** illustre cela. D'une part, c'est pour respecter cette règle qu'on fait précéder l'appel de **analyserExpression** par un appel de **nextToken**. D'autre part, c'est parce qu'on fait confiance à cette règle qu'au retour de **analyserExpression** on peut tester **ttype** sans qu'il faille le faire avancer.

Encore à titre d'exemple, voici la méthode **analyserTerme** (observez comment cette méthode « colle » à la figure 2) :

```
private Fonction analyserTerme() throws IOException, ErreurDeSyntaxe  
{  
  
    Fonction resultat = analyserFacteur();  
  
    while (lexical.ttype == '*' || lexical.ttype == '/') {  
        boolean estUnProduit = (lexical.ttype == '*');  
        lexical.nextToken();  
        Fonction facteur = analyserFacteur();
```





```
        if (estUnProduit)
            resultat = new Multiplication(resultat, facteur);
        else
            resultat = new Division(resultat, facteur);
    }
    return resultat;
}
```

Il ne vous reste plus qu'à écrire **analyserExpression** et **analyserFacteur**, et à tester le tout le plus exhaustivement possible.

N.B. A certains endroits de l'analyseur des erreurs de syntaxe sont détectées (parenthèse manquante, identificateur non reconnu, etc.). Il faut alors lancer une exception, qui sera une instance d'une classe spécialement définie à cet effet

```
public class ErreurDeSyntaxe extends Exception {
    ErreurDeSyntaxe(String message) {
        super(message);
    }
}
```

Comme exemple d'utilisation, voici un court extrait de **analyserFacteur** :

```
...
else if (lexical.ttype == '(') {
    lexical.nextToken();
    resultat = analyserExpression();
    if (lexical.ttype != ')')
        throw new ErreurDeSyntaxe(") attendue");
    lexical.nextToken();
}
...
```

---

## 3 - Interface graphique

---

### 3.1 - Cadre principal

---

Cette partie consiste à mettre en place les éléments de l'interface graphique sans nous occuper du tracé effectif de la fonction (pour commencer, un simple **JPanel** vide prendra la place du panneau de tracé).

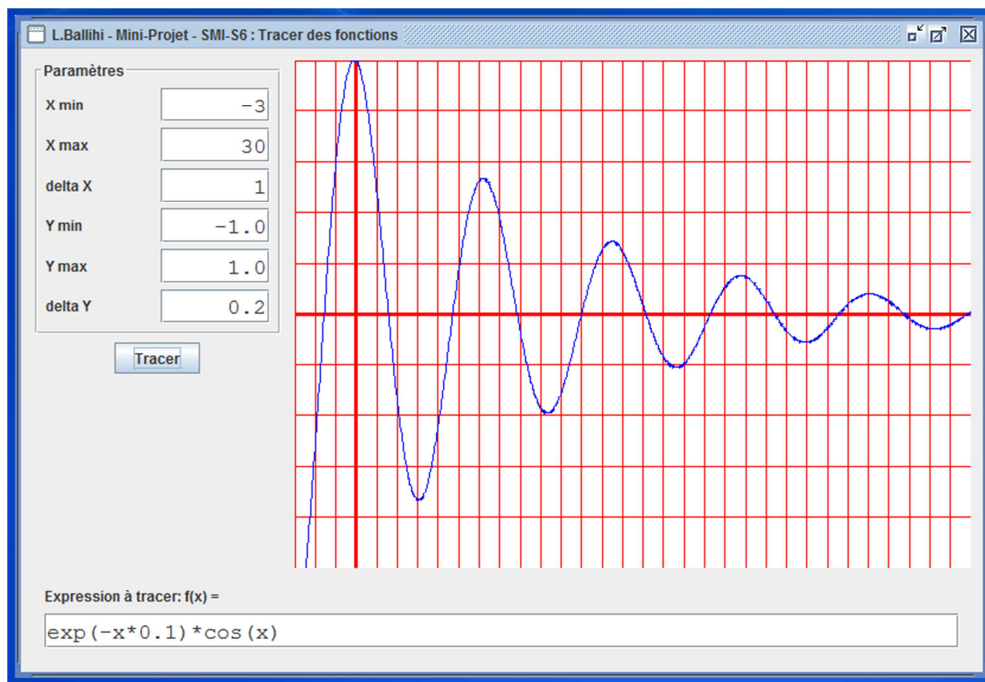
*Indications.* Le cadre principal est défini par une sous-classe de **JFrame**, nommée par exemple **CadrePrincipal**, munie d'un certain nombre de variables d'instance :

- un tableau de 6 champs de texte (**JTextField**) pour y saisir les paramètres du tracé,
- un tableau de 6 nombres (**double**) dans lesquels sont rangés, lorsque la saisie est validée - bouton *Tracer* - les valeurs des paramètres,
- un tableau constant de 6 chaînes (**String**) contenant les noms des paramètres ("**X min**", "**X max**", etc.),
- un champ de texte (**JTextField**) pour y saisir la fonction à tracer,

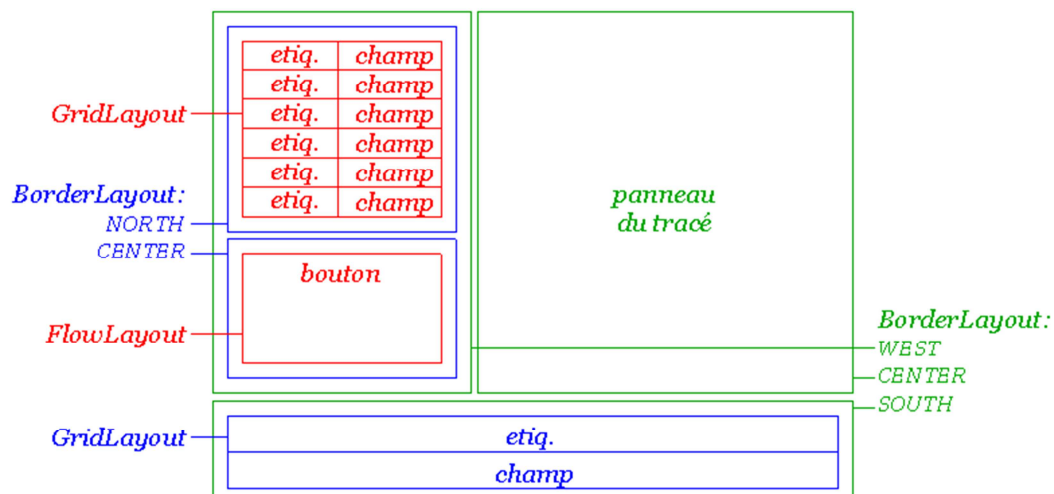


- un objet **Fonction** destiné à recevoir la représentation interne de la fonction à tracer, lorsqu'on a pu analyser avec succès la chaîne contenue dans le champ précédent,
- un panneau (**JPanel**) dans lequel, ultérieurement, sera tracée la fonction.

N.B. D'autres éléments sont nécessaires pour fabriquer l'interface voulue. S'ils ne sont pas déclarés comme variables d'instance c'est que leur portée est réduite au seul constructeur de la classe.



Pour obtenir une présentation comme celle montrée ci-dessus. Ah oui !!! La partie la plus agaçante du travail, je le savais, est la mise en place d'un système de panneaux imbriqués, chacun avec le gestionnaire de disposition adéquat, afin d'avoir l'aspect recherché quelle que soit la taille et la forme du cadre principal. Pour être gentille avec vous voici une suggestion dont vous n'êtes pas obligés de suivre :





---

### 3.2 - Événements

---

Un seul événement nous intéresse ici, l'appui sur le bouton *Tracer*. Cela devra provoquer l'appel d'une méthode nommée, par exemple, *preparerLaCourbe* qui doit

- obtenir les six valeurs des paramètres à partir des champs de texte correspondants,
- obtenir la représentation interne de la fonction à tracer à partir du contenu du champ Fonction à tracer.

Toutes ces opérations sont des analyses qui peuvent échouer de diverses manières (principalement : « champ vide » et « syntaxe incorrecte »). De telles erreurs, signalées au moyen d'exceptions, doivent provoquer l'affichage d'une boîte de dialogue de type *Information message* (voyez les méthodes *showMessageDialog* de la classe *JOptionPane*).

---

### 3.3 - Panneau de tracé

---

S'occuper du tracé c'est essentiellement écrire la méthode **paint** du panneau qui occupe la partie supérieure droite de l'interface graphique. Il faudra écrire une classe, nommée par exemple **PanneauCourbe**, sous-classe de **JPanel**, dans laquelle nous écrirons cette méthode **paint**. Le panneau de tracé sera un objet **PanneauCourbe**.

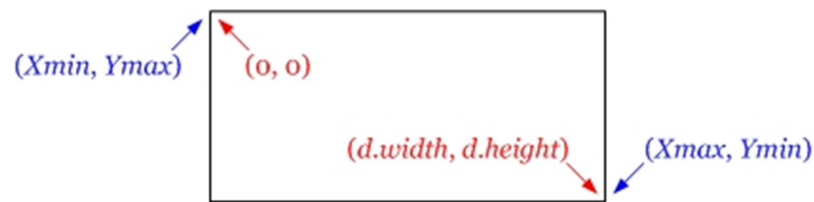
Selon votre préférence, vous écrirez la classe **PanneauCourbe** soit à l'intérieur de la classe **CadrePrincipal** soit à l'extérieur de cette classe, dans un fichier séparé. Dans le second cas il faudra que **PanneauCourbe** ait une variable d'instance de type **CadrePrincipal**, initialisé lors de la construction, lui permettant d'accéder aux divers membres du **CadrePrincipal**. Pour commencer ne vous occupez pas de tracer la grille et les axes, mais uniquement la fonction.

Le principe du tracé d'une fonction  $y = f(x)$  est simple : on se donne une constante entière *nbr* (par exemple, *nbr* = 1000), on considère la suite des *nbr* + 1 valeurs  $x_i = Xmin + i * h$ , avec  $h = (Xmax - Xmin) / nbr$  et les valeurs de *y* correspondantes  $y_i = f(x_i)$ . Pour tracer la fonction il suffit de tracer les segments  $[(x_{i-1}, y_{i-1}), (x_i, y_i)]$ .

Mais il faut régler le problème du changement de coordonnées : la fonction est exprimée en coordonnées « de l'utilisateur » ( $x_u, y_u$ ), il faut passer en coordonnées « écran » ( $x_e, y_e$ ). Pour cela on transforme les valeurs de *x* d'un côté et celles de *y* de l'autre par deux transformations affines :

$$\begin{aligned}x_e &= A_x * x_u + B_x \\ y_e &= A_y * y_u + B_y\end{aligned}$$

Pour trouver les coefficients  $A_x, B_x, A_y$  et  $B_y$  il suffit de considérer la figure suivante, dans laquelle les coordonnées de l'utilisateur sont bleues et les coordonnées écran rouges :



Ci-dessus,  $d$  représente la dimension du panneau de dessin (on l'obtient par la méthode `getSize()`).

### 3.4 - Grille et axes

Il ne reste plus qu'à ajouter le tracé de la grille et des axes de coordonnées.

Pour expliquer comment tracer la grille, examinons le cas des droites verticales (les droites horizontales se traitent de la même manière, en échangeant  $x$  et  $y$ ). L'écartement entre ces droites est défini par le paramètre  $\delta x$  : cela signifie qu'on doit tracer les droites d'équations  $x = 0, x = \delta x, x = 2 * \delta x$ , etc., c'est-à-dire les droites d'équation  $x = k * \delta x$  où  $k$  est un entier tel que cette droite soit visible à l'écran :

$$k = \dots -2, -1, 0, 1, 2, \dots \quad \text{et} \quad xMin \leq k * \delta x \leq xMax$$

ou, de manière équivalente

$$k = \dots -2, -1, 0, 1, 2, \dots \quad \text{et} \quad xMin / \delta x \leq k \leq xMax / \delta x$$

c'est-à-dire qu'on doit rechercher les entiers  $k = k_0, k_0 + 1, \dots, k_1 - 1, k_1$  avec

$k_0$  = le plus petit entier supérieur ou égal à  $xMin / \delta x$

$k_1$  = le plus grand entier inférieur ou égal à  $xMax / \delta x$

N.B. Dans beaucoup de langages, les deux entiers mentionnés ci-dessus s'obtiennent facilement avec des fonctions appelées souvent *ceil* et *floor*.