6. Interfaces graphiques

6.1. Compteurs

On vous demande de réaliser une application pour compter (des étoiles, des passagers embarqués, des bactéries, etc.). Cela se présente (voir la figure 1) comme un panneau comportant un titre, un nombre entier et un bouton. Chaque fois que l'utilisateur appuie sur le bouton, le nombre augmente de une unité.

A. Pour commencer, réalisez un programme très minimaliste : une classe **`Compteur`** avec la méthode **`main`** et deux variables statiques (la **`valeur`** du nombre et le **`JLabel`** chargé de son **`affichage`**).

Fig. 1

Il vous faudra aussi une classe auxiliaire **`AuditeurBouton`** pour représenter l'objet qui détecte et dispatche les pressions sur le bouton ; faites-en une *classe interne* à **`Compteur`**, ce qui lui permettra d'accéder aux variables **`valeur`** et **`affichage`** (Pour des raisons techniques - mais compréhensibles - cette classe devra elle aussi être qualifiée **`static`**).

B. [Légère amélioration du code] Remplacez la classe interne **`AuditeurBouton`** par une classe *anonyme*.

Fig. 2

C. [Légère amélioration de l'aspect] Faites en sorte que le bouton « ++ » ait sa largeur préférée (voyez la figure 2), au lieu d'occuper toute la largeur du cadre. Pour cela, *intercalez* un panneau entre le bouton et le cadre.

D. [Grosse amélioration du code] Faites les choses comme il faut les faire : réorganisez le code précédent afin de définir une classe **`Compteur`**, sous-classe de **`JPanel`**. Elle est munie d'un constructeur prenant le titre pour argument, et chacune de ses instances représente un panneau supportant un triplet (*titre*, *nombre affiché*, *bouton*).

Pour essayer cette classe, écrivez une méthode **`main`** (soit dans la classe **`Compteur`**, soit dans une autre classe définie à cet effet) qui crée un cadre et y place un compteur.

E. Pour vous convaincre du bien fondé de la classe précédente, modifiez la méthode **`main`**

précédente afin qu'elle crée un cadre avec, par exemple, *quatre* compteurs indépendants :

Fig. 3

6.2. Afficher l'heure

L'objet de cet exercice est la réalisation d'une classe **`Horloge`** servant à afficher l'heure courante - rafraîchie toutes les secondes - dans une interface graphique.

Un objet **`Horloge`** doit pouvoir être utilisé à tout endroit où un objet **`JLabel`** est permis, et doit supporter toutes les sortes de personnalisation (changement de taille, de couleur, de police, etc.) que peut supporter un **`JLabel`**.

Cette classe possédera un constructeur

```
public Horloge(String texte,
String format)
```

où **`texte`** est une chaîne qu'on souhaite voir affichée devant l'heure et **`format`** une chaîne spécifiant la présentation de l'heure (au besoin, revoyez l'exercice 1.3). Par exemple, dans l'illustration ci-contre, **`texte`** est la chaîne **`"Il est"`** et **`format`** la chaîne **`"HH:mm:ss"`**.

Dans cet exemple, un objet **`Horloge`** a été ajouté comme composant inférieur (« **`SOUTH`** ») d'un cadre ayant par ailleurs une image comme composant central.

A - Récupérez le code de **`TestHorloge.java`**. Il devrait fonctionner. Rajoutez une seconde horloge en modifiant la classe **`TestHorloge`**. Dans la classe **`Horloge`**, modifiez seulement le constructeur pour pouvoir paramétrer les couleurs de fond et du textede l'horloge. Vous rendez-vous compte que ce code a un problème ? Lequel ?

B - Pour obtenir le rafraîchissement de l'heure de chaque horloge, faites donc en sorte qu'en fin de programme principal, celui-ci produise la mise en route d'un **`Thread`** séparé (voir l'API de la classe **`Thread`**), associé à un objet **`Runnable`** dont la méthode **`run`** peut être ainsi décrite :

- répéter indéfiniment :
    - construire la date courante et la définir comme texte du label en question
    - dormir pendant 1000 millisecondes (en anglais, dormir se dit **`sleep`**...)

Vous avez de la chance, votre méthode **`run()`** fait exactement ce qui est demandé.

N.B. L'objet **`Runnable`** requis par la construction du **`Thread`** peut être l'objet **`Horloge`** lui-même.

6.3. Le Taquin

Récupérez le code du fameux jeu du Taquin. Modifiez-le de façon à :

- ce que la taille du jeu soit fonction de la taille de l'image (et non codée en dur)
- pouvoir passer le nom du fichier image en ligne de commande
- pouvoir passer la taille du jeu (2x2, 2x3, 3x3, 3x4, 4x4, …) en ligne de commande

6.4. Boîtes de dialogue standard

Un certain nombre de *boîtes de dialogue simples* sont utilisées très fréquemment dans toutes sortes d'applications : les boîtes d'information (figure 2), les « questions » à deux et trois boutons (figure 3), la saisie d'une chaîne, soit libre (figure 4), soit en la choisissant dans une liste de possibilités (figure 5).

Bien entendu, pour des boîtes de dialogue plus complexes il n'y a pas de solution toute prête, et le programmeur doit construire sa propre boîte dialogue. Cela se fait en définissant une sous-classe de **`JDialog`**, et sera étudié dans un autre exercice.

En Java les boîtes de dialogue simples s'obtiennent très facilement à l'aide des méthodes statiques **`show`*`XXX`*`Dialog`** de la classe **`JOptionPane`**.

Fig. 1

A - Dans cet exercice on vous demande d'écrire une application (voyez la figure 1) qui est une sorte de « présentoir » des diverses sortes de boîtes de dialogue qu'on peut obtenir de cette manière.

Cela se présente comme un cadre contenant une colonne de boutons. La pression sur un bouton affiche la boîte de dialogue en question. Lorsque la chose est pertinente, l'information obtenue à travers la boîte de dialogue est affichée pour contrôle à l'aide d'une boîte d'information.

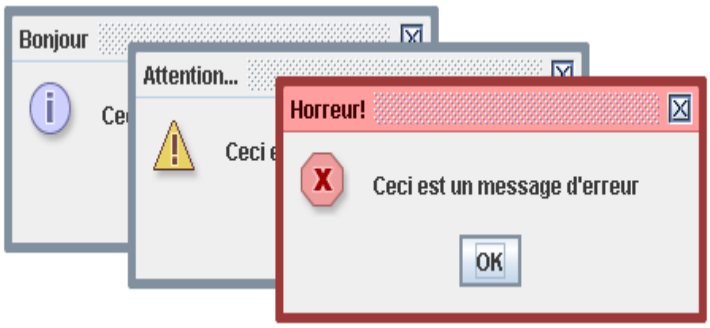


Fig. 2

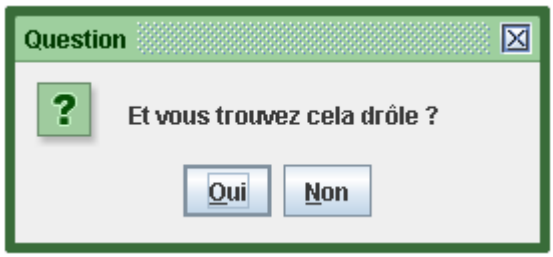


Fig. 3

Fig. 4

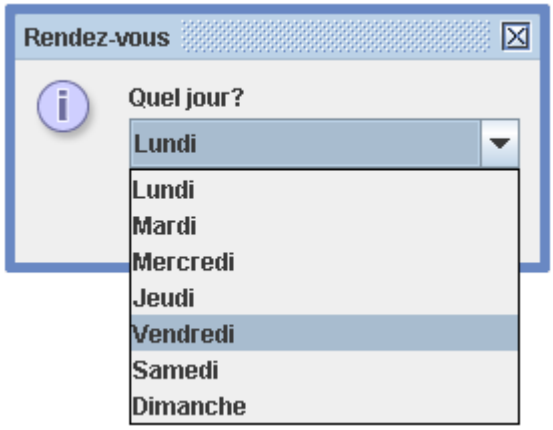


Fig. 5

Note. Observez la différence d'aspect des cadres et des boîtes de dialogue selon qu'on ait mis, avant toute création d'une instance de la classe **`JFrame`** ou de la classe **`JDialog`** (les méthodes **`showXxxDialog`** créent de telles instances) les deux lignes

```
JFrame.setDefaultLookAndFeelDecorated(true);
JDialog.setDefaultLookAndFeelDecorated(true);
```

B - Rajoutez au début de l'application une fenêtre de dialogue permettant de lister puis de sélectionner l'un des *LookAndFeel* disponibles sur votre système (classe **`UIManager`**, méthode **`getInstalledLookAndFeels()`**, etc.)

6.5. Dessine-moi une spline...!

Vous voulez épater votre petit neveu, qui vous tient pour un héros, en lui montrant une section de spline cubique, c'est-à-dire un segment de courbe définie par un polynôme de degré 3 dont les extrémités sont données et les tangentes aux extrémités aussi (voyez la figure 2). Cela tombe bien, de telles courbes sont prêtes à l'emploi, disponibles dans la bibliothèque Java2D.

Le plus dur à faire sera de mettre en place un panneau portant quatre points (les deux extrémités de la courbe et les deux extrémités restantes des tangentes) qu'on pourra déplacer avec la souris pour observer les changements de la courbe.

A. Définissez la classe **`Cubique`**, sous-classe de **`JPanel`**, qui représente un panneau montrant quatre points, extrémités de deux segments, voyez la figure 1. A la création ces points ont des positions convenues, ensuite l'utilisateur peut les attraper avec la souris et les déplacer comme il veut.

Indications. Définissez une classe interne pour représenter les points, chacun ayant sa propre

couleur. Un **`Point`** est fait de trois variables d'instance, deux (**`x`** et **`y`**) de type **`int`** et une de type `Color`. Votre classe **`Cubique`** contiendra comme variable d'instance un tableau de quatre de tels objets.

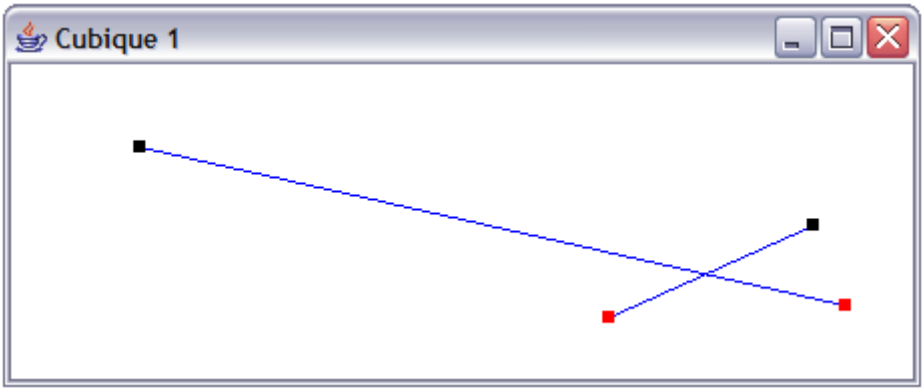


Fig. 1

La méthode **`void paint(Graphics g)`** de votre classe **`Cubique`** consistera essentiellement à tracer les deux segments (méthode **`drawLine`**) *puis* à dessiner les quatre points sous forme de petits carrés (méthode **`fillRect`**).

Pour le déplacement des points à la souris déclarez que votre classe implémente les interfaces **`MouseListener`** et **`MouseMotionListener`**. Cela vous oblige à écrire sept méthodes, dont seules trois vous intéressent :

- **`mousePressed`** - lorsque cet événement se produit, il faut chercher le point près duquel on a cliqué et en faire la valeur d'une certaine variable d'instance **`pointSélectionné`**
- **`mouseDragged`** - si **`pointSélectionné`** n'est pas **`null`**, il faut lui donner pour coordonnées celles de l'événement souris, et redessiner le panneau
- **`mouseReleased`** - ici il suffit de remettre **`pointSélectionné`** à **`null`**.

B. Pour montrer la spline cubique définie par les quatre points (les noirs jouent le rôle d'extrémités de la courbe, les rouges définissent les tangentes) il faut déclarer une variabe d'instance de type **`CubicCurve2D.Double`**, créée lors de la construction du panneau et, surtout, positionnée à chaque appel de la méthode **`paint`** par un appel de la méthode **`setCurve`**. Cette méthode requiert quatre objets de type **`Point2D.Double`** (des points aux coordonnées **`double`**) qu'il faudra construire à partir des coordonnées (entières) des quatre points dessinés.

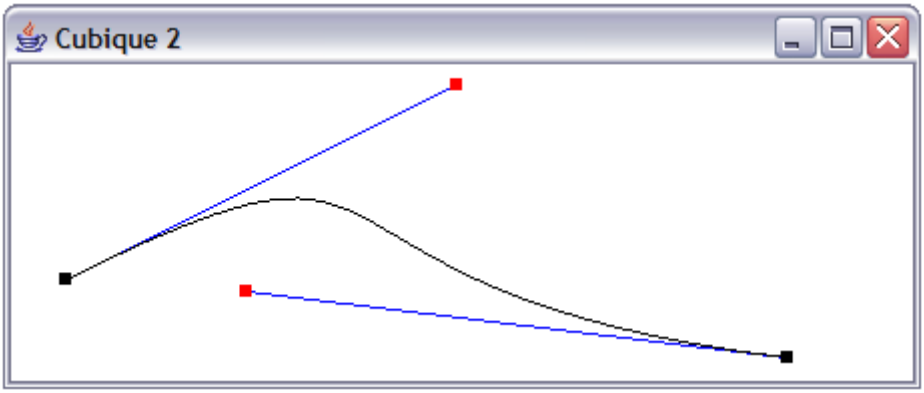
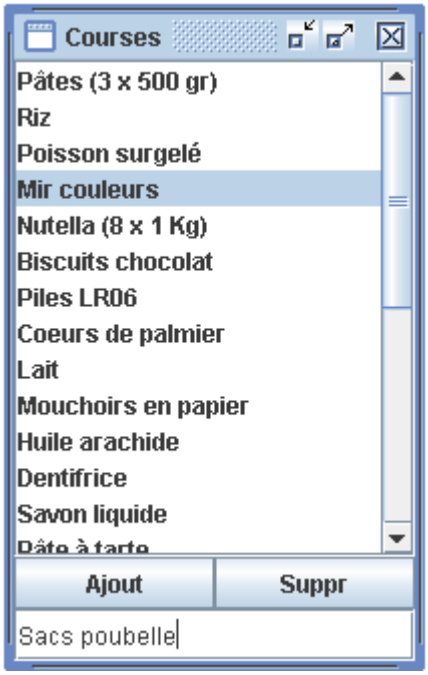


Fig. 2

6.6. Une liste (très) simple

Comme divers autres composants, les listes (classe **`JList`**) de *Swing* sont moins faciles à utiliser qu'on ne pourrait le penser (et que ne le sont les **`List`**, leurs homologues d'*AWT*) à cause de la séparation qui est maintenue entre le composant lui-même, qui est une *vue*, et le *modèle* sous-jacent, qui se charge de la gestion des données que la vue est censée montrer. Cette séparation a de nombreux avantages, mais entraîne toute une machinerie avec laquelle il faut bien négocier.



Pour pratiquer cela nous allons réaliser une application qui construit et affiche – on se contentera de cela, mais on pourrait imaginer des fonctions ultérieures d'impression, de sauvegarde dans un fichier, etc. – une liste simple, par exemple la *liste des courses au supermarché*, voyez la figure ci-contre.

Pour ajouter un item à cette liste il faut le taper dans le champ de texte tout en bas de l'interface, puis actionner le bouton « *Ajout* ». Pour supprimer un item il faut le sélectionner dans la liste puis agir sur le bouton « *Suppr* ».

A. Pour commencer ne vous occupez que de la vue (c.-à-d. l'aspect : ignorez donc le comportement associé aux composants). Écrivez une classe **`ListeSimple`**, sous classe de **`JPanel`**, comportant un constructeur et une méthode **`main`** qui, classiquement, se limite à créer un cadre (classe **`JFrame`**) et y placer comme panneau de contenu un objet **`ListeSimple`**.

Une **`ListeSimple`** est un **`JPanel`** piloté par un **`BorderLayout`** qui a :

- comme composant **`CENTER`** un **`JScrollPane`** contenant un **`JList`** initialisé comme indiqué plus loin,
- comme composant **`SOUTH`** un **`JPanel`** contrôlé par un **`GridLayout`** à une colonne, contenant :
  - un **`JPanel`** géré par un **`GridLayout`** à une ligne, contenant deux boutons
  - un **`JTextArea`**

B. Dans un deuxième temps intéressons-nous à la détection et au traitement des actions de l'utilisateur sur cette interface.

Pour l'utilisateur, les moyens d'action principaux sont les deux boutons *Ajout* et *Suppr*. Or, l'ajout n'a de sens que si un mot est tapé dans la zone de texte ; de même, la suppression n'est pertinente que si un item de la liste est couramment sélectionné. On pourrait traiter ces « préconditions » par des tests placés à l'entrée des fonction qui traitent les actions sur les boutons. Mais on préfère généralement, et c'est ce que nous ferons ici, traiter ces questions en amont même de l'action de l'utilisateur, en désactivant (estompant) les boutons lorsque l'opération qu'ils exprimente n'a pas de sens (méthode **`setEnabled(boolean)`**).

Pour une fois nous allons renoncer aux classes internes anonymes en déclarant que notre classe **`ListeSimple`** implémente les interfaces :

- **`ActionListener`**, pour détecter les événements sur les boutons (des pressions) ; cette interface impose une unique méthode **`actionPerformed(ActionEvent ae)`**,
- **`ListSelectionListener`**, pour détecter la sélection ou la dé-sélection d'un élément de la liste ; cette interface impose également une unique méthode **`valueChanged (ListSelectionEvent lse)`**,
- **`DocumentListener`**, pour détecter les caractères frappés ou effacés dans la zone de texte. Il s'agit ici d'une interface à trois méthodes, **`changedUpdate (DocumentEvent de)`**, **`insertUpdate (DocumentEvent de)`** et **`removeUpdate (DocumentEvent de)`**, dont seules les deux dernières nous intéressent.

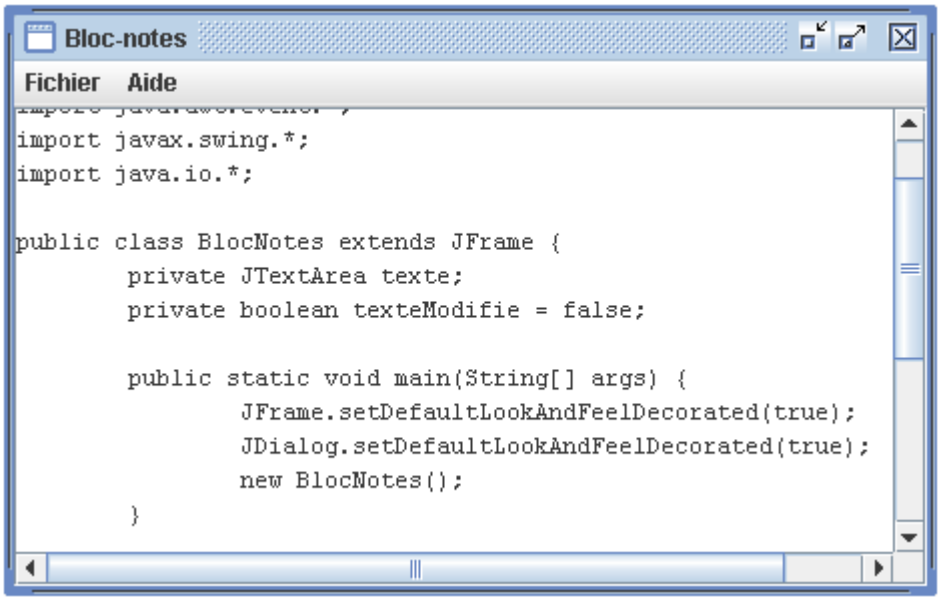
A divers endroits du constructeur de **`ListeSimple`** nous aurons donc les instructions

```
laListe.addListSelectionListener(this);
zoneTexte.getDocument().addDocumentListener(this);
boutonAjout.addActionListener(this);
boutonSuppr.addActionListener(this);
```

Écrivez les méthodes mentionnées plus haut, qui donnent à notre interface le comportement voulu. Les méthodes **`valueChanged`** (de **`ListSelectionListener`**), **`insertUpdate`** et **`removeUpdate`** (de **`DocumentListener`**) se chargent d'« allumer » et d'« éteindre » les boutons, la méthode **`actionPerformed`** de **`ActionListener`** s'occupe de faire le travail d'ajout ou de suppression d'un élément.

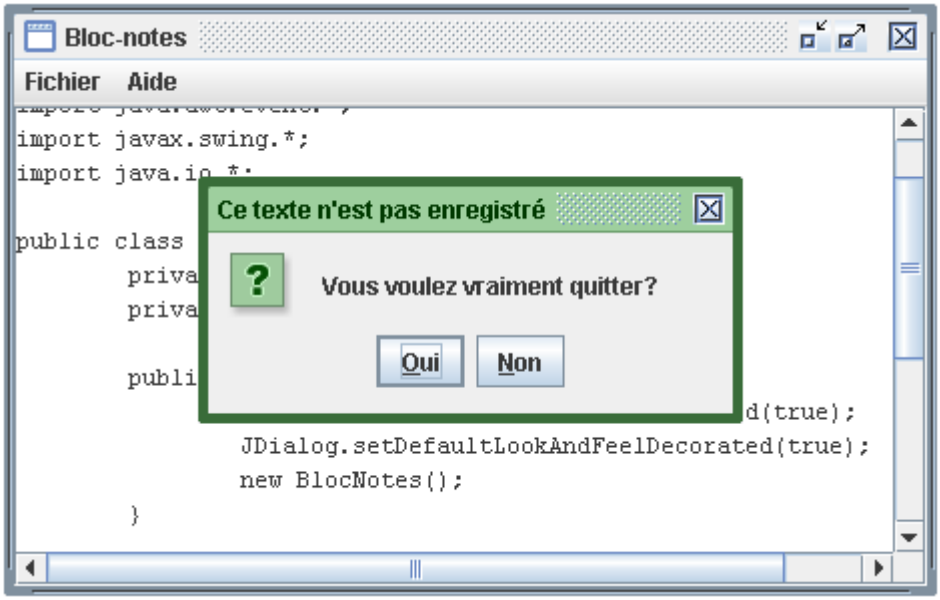
6.7. Un bloc-notes

On vous demande d'écrire un éditeur de texte simple, analogue au bloc-notes de Windows :

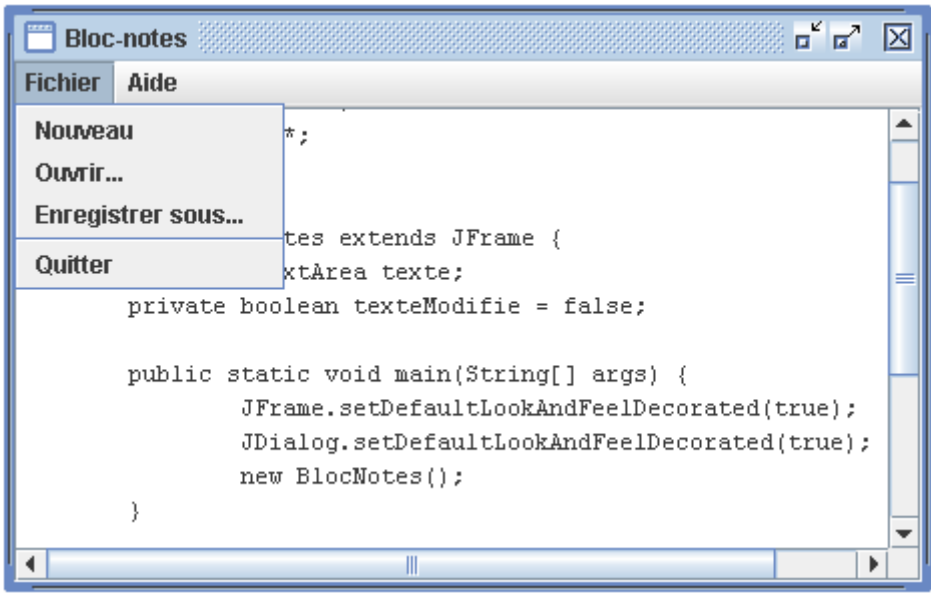


Fondamentalement, le bloc notes est un cadre (classe **`JFrame`**) avec une barre de menus (classe **`JMenuBar`**) et un panneau de contenu dans lequel on aura mis un panneau de défilement (classe **`JScrollPane`**) contenant une zone de texte (classe **`JTextArea`**).

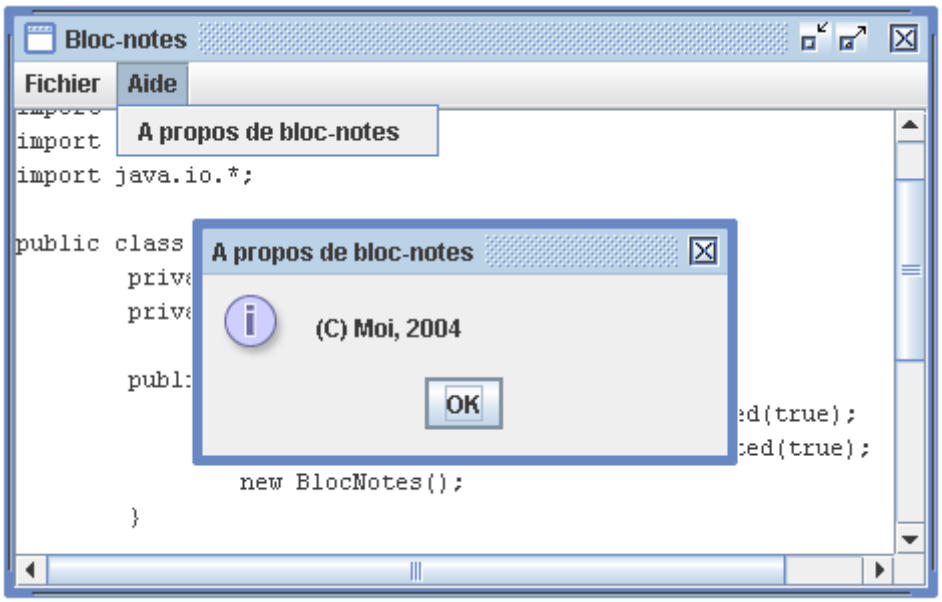
On veillera à ce que toute commande produisant la perte du texte édité soit précédée d'une confirmation :



Le menu *Fichier* est formé des commandes les plus classiques :

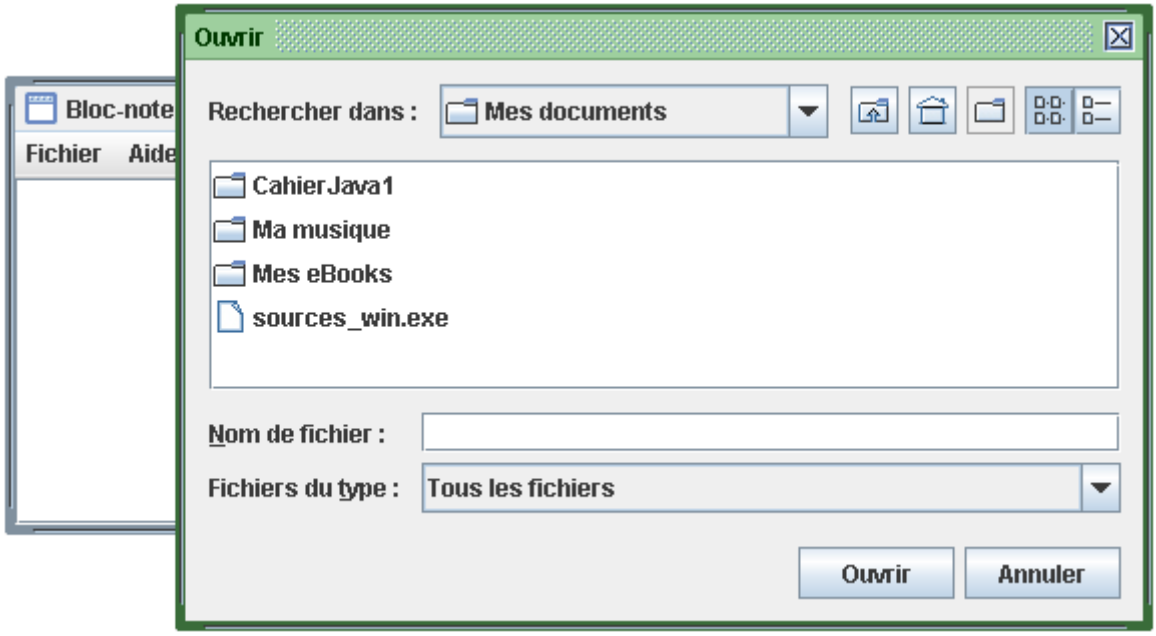


Le menu *Aide* se réduit à la boîte à propos :



Note 1. Pour déterminer s'il y a lieu de demander une confirmation avant de détruire le texte définissez une variable d'instance **`texteModifié`** que certaines opérations mettent à **`false`**, et qui est mise à **`true`** chaque fois qu'un caractère est tapé au clavier en visant la zone de texte (événement **`Key`**).

Note 2. La sauvegarde du texte dans un fichier ne pose pas de problème : il suffit d'écrire tout le contenu de la zone de texte dans un **`FileWriter`** ouvert à partir d'un objet **`File`** obtenu à l'aide d'un dialogue **`FileChooser`** :



La lecture du texte est un peu plus alambiquée, car les objets **`FileReader`** ne peuvent lire que dans un tampon de caractères, qu'il faudra allouer à partir de la taille donnée par une expression de la forme *unFichier*`.length()` (où *unFichier* est un objet **`File`** obtenu à l'aide d'un objet **`FileChooser`**).

6.8. Tracer une courbe

A. L'objet de l'exercice est l'écriture d'une classe simple effectuant la représentation graphique d'une fonction réelle d'une variable réelle. Par exemple, dans le cas de la fonction $y = \sin x$:

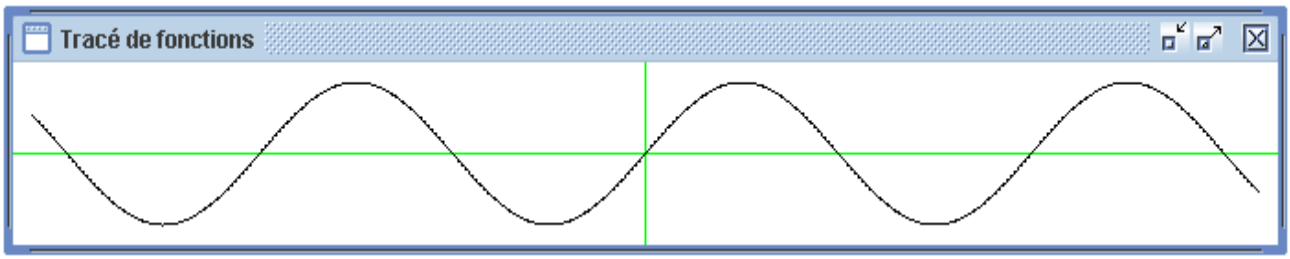


Fig. 1

La fonction est traceé de manière à remplir au mieux le panneau de dessin. Ainsi, les modifications de la taille ou de la forme du cadre entraînent un nouveau tracé, voyez la figure 2.

Écrivez une classe **`Traceur`**, sous-classe de **`JPanel`**. Elle est munie d'un constructeur

```
public Traceur(Fonction fonc);
```

où **`Fonction`** est une interface ainsi définie :

```
public interface Fonction {
    double fonction(double x);
    double xMin();
    double xMax();
}
```

`double funtion(double x)` représente la fonction en question, tandis que **`xMin()`** et **`xMax()`** sont deux méthodes qui renvoient toujours les mêmes valeurs (c.-à-d. des constantes). Ces valeurs déterminent l'intervalle de définition de la fonction [il est supposé que **`xMin()`** < **`xMax()`**].

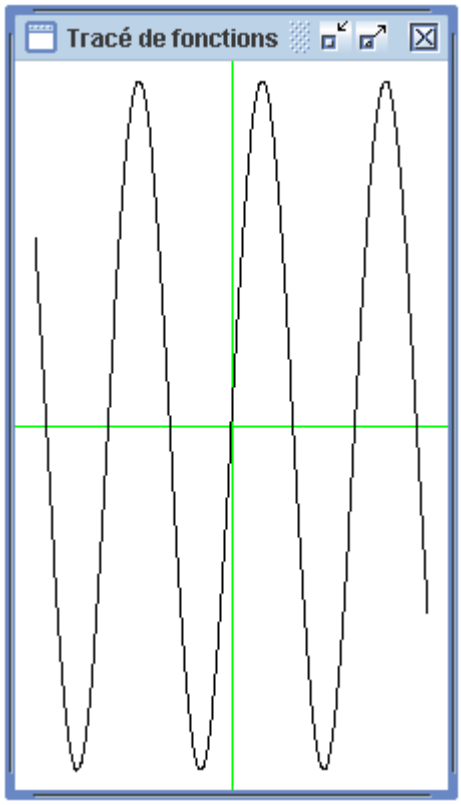


Fig. 2

Par exemple, le tracé montré sur les figures 1 et 2 est obtenu en construisant un panneau **`Traceur`** de la manière suivante :

```
Traceur panneau = new Traceur(new
Fonction() {
    public double fonction(double x) {
        return Math.sin(x);
    }
    public double xMin() {
        return -10;
    }
    public double xMax() {
        return 10;
    }
});
```

A partir des valeurs renvoyées par **`xMin()`** et **`xMax()`** le constructeur calcule les valeurs de **`yMin`** et **`yMax`**, ces quatre valeurs sont conservées dans des variables d'instance privée.

Le tracé lui-même est pris en charge par la rédéfinition de la méthode canonique

`paint(Graphics g)`, qui commence par obtenir la taille du panneau (méthode **`getSize()`**) et en déduire les quatre coefficients A_x, B_x, A_y et B_y permettant de convertir les coordonnées « utilisateur » (x_u, y_u) en des coordonnées « écran » (x_e, y_e) :

$$x_e = A_x \times x_u + B_x$$
$$y_e = A_y \times y_u + B_y$$

Notez que x_u et y_u sont des **`double`**, alors que x_e et y_e sont des **`int`**.

Si le cœur vous en dit, pour pouvez compléter ce programme par le tracé d'un quadrillage ou l'affichage de repères numériques..

B. Ajoutez au programme précédent la possibilité de « zoomer » sur une partie du tracé : lorsque l'utilisateur définit un rectangle avec la souris (par les gestes habituels : presser le bouton, déplacer la souris, relâcher le bouton) les coefficients A_x, B_x, A_y et B_y sont recalculés afin que la partie du tracé délimitée par ce rectangle remplisse tout le panneau.

On doit pouvoir revenir en arrière. Par exemple, si l'utilisateur fait un « Ctrl-clic » n'importe où dans le panneau, le tracé doit reprendre ses proportions précédentes.
