

APPLICATIONS EN JAVA EN UTILISANT SWING

1 Ecriture d'une application

Une application est un programme qui peut fonctionner en autonome (sans besoin d'être intégré dans une page HTML). Il doit comporter une classe contenant une méthode **main** déclarée comme suit :

```
public static void main(String arguments[]) {  
    // corps du programme principal  
}
```

Remarque : une méthode déclarée **static** est en fait une fonction (comme en C) indépendante de toute classe. Il est possible de déclarer d'autres fonctions que **main** de type **static** toutefois elles doivent quand même être placées dans une classe. Elle seront appelées par : `nomDeClasse.nomDeMethode(paramètres)`.

arguments est un tableau de chaînes de caractères correspondant aux paramètres d'appel de l'application (comme en C). On peut connaître la taille de ce tableau en utilisant **arguments.length**

2 Interfaces graphiques avec SWING

SWING offre toute une panoplie de classes pour la création d'interfaces graphiques. Chacun des objets de ces classes est susceptible de réagir à des événements provenant de la souris ou du clavier. Lorsque l'on utilise de telles classes il faut prévoir d'importer les bibliothèques **javax.swing.*** et **javax.swing.event.*** en début de programme.

2.1 Réalisation d'une interface graphique

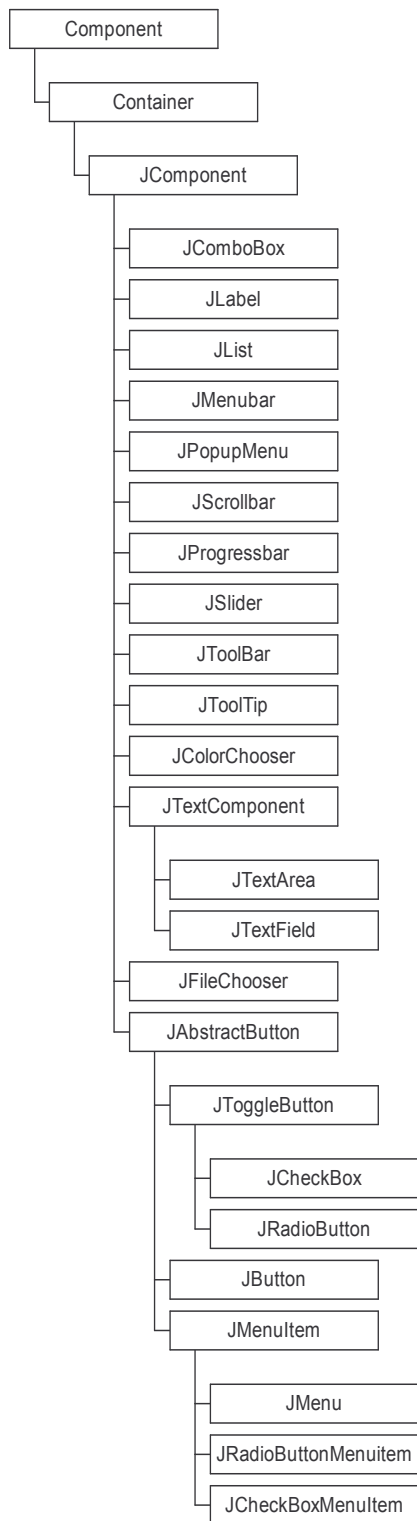
Il faut tout d'abord définir un contenant (**Container**) de cette interface graphique. On choisit en général de personnaliser par héritage la classe **JFrame** qui correspond à une fenêtre de type Windows ou Xwindows. Il est à remarquer que lorsque l'on écrit des applets plutôt que des applications cette étape n'est plus nécessaire puisqu'une applet est déjà obtenue par héritage de la classe **Container** ce qui lui permet de se placer dans une page HTML. On placera ensuite dans cette fenêtre une barre de menu et des composants comme des boutons, des zones de saisie etc. Tous ces composants appartiennent à des classes héritant de la classe **JComponent** dont nous allons maintenant décrire les principales méthodes.

2.1.1 Les classes Component et JComponent

C'est de là que dérivent tous les composants de l'interface graphique. La classe **JComponent** hérite de **Component** et propose quelques méthodes supplémentaires. Les principales méthodes de ces classe sont les suivantes :

Celles propres à la classe **Component** :

- boolean isShowing()** indique si le composant est visible à l'écran
- void setVisible(boolean)** rend le composant visible ou non à l'écran
- void requestFocus()** rend le composant actif (les saisies au clavier le concernent). Il faut que le composant soit visible.
- boolean isEnabled()** indique si le composant est sensible aux événements
- void setEnabled(boolean)** permet de rendre le composant sensible ou pas aux événements
- Color getForeground()** retourne la couleur utilisée pour dessiner et écrire dans ce composant
- void setForeground(Color)** définit la couleur de tracé
- Color getBackground()** retourne la couleur de fond de ce composant
- void setBackground(Color)** définit la couleur de fond



Font **getFont()** retourne la fonte de caractères utilisée par ce composant
void setFont(Font) définit la fonte de caractères
Cursor **getCursor()** retourne le curseur de souris utilisé sur ce composant
void setCursor(Cursor) définit le curseur de souris pour ce composant
int getWidth() et **int getHeight()** retournent la largeur et la hauteur du composant.
void setSize(int , int) définit la taille du composant
Dimension getPreferredSize() retourne la taille préférentielle du composant
int getX() et **int getY()** retournent les coordonnées de ce composant
void setLocation(int , int) définit la position du composant
Toolkit getToolkit() retourne un objet boîte à outils permettant de gérer les images, les fontes de caractères etc. (voir 2.2)
Graphics getGraphics() retourne un objet permettant de dessiner (voir 2.2.1)
boolean prepareImage(Image, ImageObserver) provoque le chargement d'une image. Le premier paramètre une image (par exemple retournée par **getImage()** voir 2.2.1) et en second le composant qui la gère (en général c'est celui dont on utilise la méthode **prepareImage** que l'on désignera donc par **this**). Ce chargement se fait en parallèle et il peut ne pas être terminé quand on veut utiliser l'image. La valeur de retour indique si le chargement est ou non terminé.
void repaint() redessine le composant

Celles propres à la classe **JComponent** :

void setPreferredSize(Dimension) définit la taille préférentielle du composant
void setMinimumSize(Dimension) définit la taille minimale du composant
void setMaximumSize(Dimension) définit la taille maximale du composant
void setToolTipText(String) définit le texte qui sera affiché lorsque la souris passera sur ce composant. Ce texte disparaît lorsque la souris s'éloigne du composant ou au bout d'un certain délai.



JRootPane getRootpane() retourne l'objet de classe RootPane qui contient ce composant ou null s'il n'y en a pas.

Remarque : se reporter à la documentation de Java pour en savoir plus sur les classes **Dimension**, **Toolkit** et **Cursor**. Les classes **Font** et **Color** seront décrites plus loin.

2.1.2 Les composants de l'interface

La fenêtre (héritée de **JFrame**) que l'on va définir contiendra en tant que membres les noms des composants d'interface (boutons, zones de texte ou de dessin ...) dont on souhaite disposer.

Il conviendra ensuite de les initialiser et de définir leur disposition. Pour cela, le constructeur de la classe d'interface devra se charger :

- de créer les objets associés (new)
- de les initialiser par leurs constructeurs ou par tout autre moyen disponible
- de les placer

- de leur associer les actions liées aux événements qu'ils peuvent recevoir
 Il se chargera par ailleurs de dimensionner la fenêtre et de la rendre visible à l'écran.
 Les principales classes de composants sont les suivants :

- JButton** : un bouton à cliquer avec un texte
- JLabel** : juste un titre (un texte non éditable)
- TextField** : une zone de texte éditable d'une ligne
- TextArea** : une zone de texte éditable multi lignes
- CheckBox** : une case à cocher
- ComboBox** : une liste déroulante
- JList** : une liste dans laquelle on peut sélectionner un ou plusieurs éléments par double clic
- Scrollbar** : un ascenseur horizontal ou vertical
- MenuBar** : une barre de menu
- FileChooser** : une fenêtre de dialogue permettant de choisir un fichier
- Slider** : un curseur linéaire
- ProgressBar** : une barre indiquant la progression d'une opération

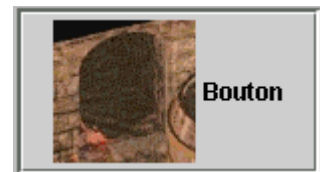
Remarque : Cette liste n'est pas exhaustive. Il existe aussi des groupes de cases à cocher, des boîtes de dialogue, des sous menus etc.

Ces composant possèdent les méthodes communes contenues dans la classe **JComponent** (voir 2.1.1) auxquelles viennent s'ajouter leurs méthodes propres. Nous allons maintenant en décrire les principales :

2.1.2.1 La classe JButton

Elle permet de définir des boutons sur lesquels on peut cliquer. Ses principales méthodes sont :

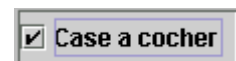
- JButton(String)** construction avec définition du texte contenu dans le bouton
- JButton(ImageIcon)** construction avec une icône dans le bouton
- JButton(String,ImageIcon)** construction avec définition du texte et d'une icône dans le bouton
- String getText()** qui retourne le texte contenu dans le bouton
- void setText(String)** qui définit le texte contenu dans le bouton
- void addActionListener(ActionListener)** pour associer l'objet qui traitera les clics sur le bouton (voir 2.1.6.3)



2.1.2.2 La classe JCheckBox

Elle permet de définir des cases à cocher. Ses principales méthodes sont :

- JCheckBox(String)** construction avec définition du texte contenu dans la case à cocher
- JCheckBox(String,boolean)** construction avec en plus définition de l'état initial de la case à cocher
- boolean isSelected()** qui retourne l'état de la case à cocher (cochée ou non).
- void setSelected(boolean)** qui définit l'état de la case à cocher (cochée ou non).
- String getText()** qui retourne le texte contenu dans la case à cocher
- void setText(String)** qui définit le texte contenu dans la case à cocher
- void addActionListener(ActionListener)** pour associer l'objet qui traitera les actions sur la case à cocher (voir 2.1.6.3)



2.1.2.3 La classe JLabel

Elle permet de définir des textes fixes. Ses principales méthodes sont :

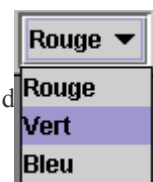
- JLabel(String)** qui construit le titre avec son contenu
- void setText(String)** qui définit le texte
- String getText()** qui retourne le texte



2.1.2.4 La classe JComboBox

Elle permet de définir des boîtes permettant de choisir une valeur parmi celles proposées. Ses principales méthodes sont :

- JComboBox(Object[])** construction avec définition de la liste. On peut utiliser un tableau de chaînes de caractères (**String**) ou de toute autre classe d'objets.
- void addItem(Object)** qui ajoute une valeur possible de choix
- int getSelectedIndex()** qui retourne le numéro du choix actuel



Object getSelectedItem() qui retourne l'objet associé au choix actuel. Attention l'objet retourné est de classe **Object**, il faudra utiliser l'opérateur de coercition pour le transformer en sa classe d'origine (**String** par exemple).

void setSelectedIndex(int) qui sélectionne un élément défini par son numéro

void addActionListener(ActionListener) pour associer l'objet qui traitera les choix faits (voir 2.1.6.3)

2.1.2.5 La classe JList

Elle permet de définir des listes non déroulantes (pour disposer de listes avec ascenseur il faut faire appel à un contenant possédant des ascenseurs comme **JScrollPane**). La sélection dans ces listes peut porter sur 1 ou plusieurs objets. Ses principales méthodes sont :

JList(Object[]) construction avec définition de la liste. On peut utiliser un tableau de chaînes de caractères (**String**) ou de toute autre classe d'objets.

setListData(Object[]) définition de la liste. On peut utiliser un tableau de chaînes de caractères (**String**) ou de toute autre classe d'objets.

void setVisibleRowCount(int) qui définit le nombre d'éléments visibles sans ascenseur

int getSelectedIndex() qui retourne le numéro du premier élément sélectionné

int[] getSelectedIndices() qui retourne les numéros des éléments sélectionnés

Object getSelectedValue() qui retourne le premier objet sélectionné. Attention l'objet retourné est de classe **Object**, il faudra utiliser l'opérateur de coercition pour le transformer en sa classe d'origine (**String** par exemple).

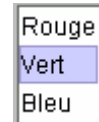
Object[] getSelectedValues() qui retourne les objets actuellement sélectionnés.

void setSelectedIndex(int) qui sélectionne l'élément désigné par son numéro

void setSelectedIndices(int[]) qui sélectionne les éléments désignés par leurs numéros

void clearSelection() qui annule toutes les sélections

void addListSelectionListener(ListSelectionListener) pour associer l'objet qui traitera les sélections dans la liste (voir 2.1.6.3)



2.1.2.6 La classe JScrollbar

Elle permet de définir des ascenseurs horizontaux ou verticaux. Ses principales méthodes sont :

JScrollbar(int,int,int,int,int) qui définit l'ascenseur avec dans l'ordre des paramètres son orientation (on peut y mettre les constantes **JScrollbar.HORIZONTAL** ou **JScrollbar.VERTICAL**), sa position initiale, le pas avec lequel on le déplace en mode page à page, ses valeurs minimales et maximales.

int getValue() qui retourne la position actuelle de l'ascenseur

void setValue(int) qui définit la position de l'ascenseur

int getBlockIncrement() qui retourne la valeur utilisée pour le pas en mode page à page

void setBlockIncrement (int) qui définit la valeur utilisée pour le pas en mode page à page

int getUnitIncrement() qui retourne la valeur utilisée pour le pas unitaire

void setUnitIncrement (int) qui définit la valeur utilisée pour le pas unitaire

int getMaximum() qui retourne la valeur maximale actuelle

void setMaximum (int) qui définit la valeur maximale actuelle

int getMinimum() qui retourne la valeur minimale actuelle

void setMinimum (int) qui définit la valeur minimale actuelle

void addAdjustmentListener(AdjustmentListener) pour associer l'objet qui traitera les déplacements de l'ascenseur (voir 2.1.6.3)



2.1.2.7 La classe JSlider

Elle permet de définir des curseurs horizontaux ou verticaux gradués. Ses principales méthodes sont :

JSlider(int,int,int,int,int) qui définit l'ascenseur avec dans l'ordre des paramètres son orientation (on peut y mettre les constantes **JSlider.HORIZONTAL** ou **JSlider.VERTICAL**), ses valeurs minimales et maximales et sa position initiale.

void setMajorTickSpacing(int) qui détermine le nombre d'unités correspondant à une graduation plus longue.

void setMinorTickSpacing(int) qui détermine le nombre d'unités correspondant à une graduation courte.

void setPaintTicks(boolean) qui détermine si les graduations sont ou non dessinées.

void setPaintTrack(boolean) qui détermine si la piste du curseur est ou non dessinée.

Hashtable createStandardLabels(int,int) qui permet de créer une table d'étiquettes constituée de nombres entiers. Le premier paramètre est le pas, le second est la valeur de départ. Une telle table pourra être associée au curseur par sa méthode **setLabelTable** (la classe **Hashtable** est une classe directement héritée de **Dictionary**).

void setLabelTable(Dictionary) qui associe une table d'étiquettes aux graduations longues du curseur

void setPaintLabels(boolean) qui détermine si les valeurs correspondant aux graduations longues sont ou non écrites.

int getValue() qui retourne la position actuelle de l'ascenseur

void setValue(int) qui définit la position de l'ascenseur

int getUnitIncrement() qui retourne la valeur utilisée pour le pas unitaire

void setUnitIncrement (int) qui définit la valeur utilisée pour le pas unitaire

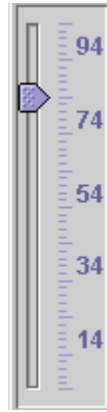
int getMaximum() qui retourne la valeur maximale actuelle

void setMaximum (int) qui définit la valeur maximale actuelle

int getMinimum() qui retourne la valeur minimale actuelle

void setMinimum (int) qui définit la valeur minimale actuelle

void addChangeListener(ChangeListener) pour associer l'objet qui traitera les déplacements du curseur (voir 2.1.6.3)



Le curseur présenté ci contre ici est défini par :

```
curseur=new JSlider (JSlider.VERTICAL,0,100,80);
curseur.setMajorTickSpacing(10);
curseur.setMinorTickSpacing(2);
curseur.setPaintTicks(true);
curseur.setPaintTrack(true);
curseur.setLabelTable (curseur.createStandardLabels (20,14));
curseur.setPaintLabels (true);
```

2.1.2.8 La classe JProgressBar

Elle permet de dessiner une barre dont la longueur représente une quantité ou un pourcentage. Ses principales méthodes sont :

JProgressBar (int,int,int) qui définit la barre avec dans l'ordre des paramètres son orientation (on peut y mettre les constantes **JProgressBar.HORIZONTAL** ou **JProgressBar.VERTICAL**) et ses valeurs minimales et maximales.

int getValue() qui retourne la position actuelle de la barre

void setValue(int) qui définit la position de la barre

int getMaximum() qui retourne la valeur maximale actuelle

void setMaximum (int) qui définit la valeur maximale actuelle

int getMinimum() qui retourne la valeur minimale actuelle

void setMinimum (int) qui définit la valeur minimale actuelle

void addChangeListener(ChangeListener) pour associer l'objet qui traitera les déplacements de la barre (voir 2.1.6.3)



2.1.2.9 Les classes JTextField et JTextArea

Elles permettent de définir des zones de texte éditables sur une ou plusieurs lignes. Elles ont en commun un certain nombre de méthodes dont voici les principales :

void copy() qui copie dans le bloc-notes du système la partie de texte sélectionnée

void cut() qui fait comme **copy** puis supprime du texte la partie sélectionnée

void paste() qui fait copie dans le texte le contenu du bloc-notes du système

String getText() qui retourne le texte contenu dans la zone de texte

void setText(String) qui définit le texte contenu dans la zone de texte

int getCaretPosition() qui retourne la position du curseur d'insertion dans le texte (rang du caractère)

int setCaretPosition(int) qui place le curseur d'insertion dans le texte au rang indiqué (rang du caractère)

int moveCaretPosition(int) qui déplace le curseur d'insertion dans le texte en sélectionnant le texte depuis la position précédente.

setEditable(boolean) qui rend la zone de texte modifiable ou pas

int getSelectionStart() qui retourne la position du début du texte sélectionné (rang du caractère)

int getSelectionEnd() qui retourne la position de la fin du texte sélectionné (rang du caractère)

void setSelectedTextColor(Color c) qui définit la couleur utilisée pour le texte sélectionné

String getSelectedText() qui retourne le texte sélectionné

void select(int,int) qui sélectionne le texte compris entre les deux positions données en paramètre

void selectAll() qui sélectionne tout le texte

Document getDocument() qui retourne le gestionnaire de document associé à cette zone de texte. C'est ce gestionnaire qui recevra les événements de modification de texte

void addCaretListener(CaretListener) pour associer l'objet qui traitera les déplacements du curseur de saisie dans le texte (voir 2.1.6.3)

La classe Document

Prend en charge l'édition de texte. Ses principales méthodes sont :

int getLength() qui retourne le nombre de caractères du document

String getText(int,int) qui retourne la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre

void removeText(int,int) qui supprime la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre

void addDocumentListener(DocumentListener) pour associer l'objet qui traitera les modifications du texte (voir 2.1.6.3)

Remarque : Ces deux dernières méthodes peuvent lever une exception de classe **BadLocationException** si les paramètres sont incorrects

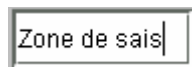
La classe JTextField

Elle permet de définir des zones de texte sur une seule ligne modifiables Ses principales méthodes sont :

JTextField(String) qui la crée avec un contenu initial

JTextField(String,int) qui la crée avec un contenu initial et définit le nombre de colonnes

void addActionListener(ActionListener) pour associer l'objet qui traitera les modifications du texte (voir 2.1.6.3)



Remarque : Lorsque l'on saisit du texte dans un tel composant celui-ci adapte sa taille au texte saisi au fur et à mesure. Ce comportement n'est pas toujours souhaitable, on peut l'éviter en lui définissant une taille préférentielle par sa méthode **setPreferredSize** et une taille minimale par sa méthode **setMinimumSize** (voir 2.1.1). On procédera comme suit : `texte.setPreferredSize(texte.getPreferredSize()) ;`
`texte.setMinimumSize(texte.getPreferredSize()) ;`

La classe JTextArea

Elle permet de définir des zones de texte sur plusieurs lignes modifiables sans ascenseurs (pour disposer d'ascenseurs il faut faire appel à un contenant en possédant voir 2.1.3.4). Ses principales méthodes sont :

JTextArea(String) qui crée une zone de texte avec un contenu initial

JTextArea(String,int,int) qui crée une zone de texte avec un contenu initial et précise le nombre de lignes et de colonnes de la zone de texte

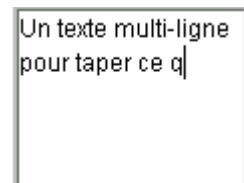
void append(String) qui ajoute la chaîne à la fin du texte affiché

void insert(String,int) qui insère la chaîne au texte affiché à partir du rang donné

void setTabSize(int) qui définit la distance entre tabulations.

void setLineWrap(boolean) qui détermine si les lignes longues doivent ou non être repliées.

void setWrapStyleWord(boolean) qui détermine si les lignes sont repliées en fin de mot (true) ou pas.



Remarque : Lorsque l'on saisit du texte dans une zone de texte celle-ci adapte sa taille au texte saisi au fur et à mesure. Ce comportement n'est pas toujours souhaitable, on peut l'éviter en mettant ce composant dans un **JScrollPane** pour disposer d'ascenseurs.

2.1.2.10 La classe JMenuBar

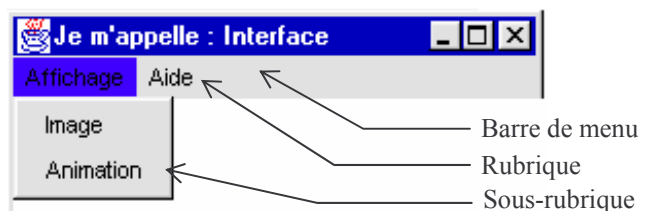
On peut doter une fenêtre d'une barre de menu qui s'affichera en haut et permettra d'accéder à des rubriques et sous rubriques. On utilisera pour cela les classes **JMenuBar**, **JMenu** et **JMenuItem**.

La classe JMenuBar

Elle permet de définir une barre de menu. Ses principales méthodes sont :

void add(JMenu) qui ajoute une rubrique dans la barre

JMenu getMenu(int) qui retourne la rubrique dont le numéro est spécifié



On l'associe à la fenêtre (classe **JFrame**, **JApplet** ou **JInternalFrame**) en utilisant la méthode **setJMenuBar** de celle-ci (voir 2.1.3.2)

La classe **JMenu**

Elle permet de définir une rubrique dans une barre de menu. Ses principales méthodes sont :

JMenu(String) qui crée et définit le texte de la rubrique

JMenuItem add(JMenuItem) qui ajoute une sous rubrique (elles apparaîtront dans l'ordre où elles sont ajoutées). La valeur de retour est cette sous-rubrique.

JMenuItem add(String) qui crée une sous_rubrique (**JMenuItem**) à partir de la chaîne de caractères et l'ajoute. La valeur de retour est cette sous-rubrique.

void addSeparator() qui place un séparateur entre sous rubriques

void insert(JMenuItem,int) qui ajoute une sous rubrique au rang spécifié

void insertSeparator(int) qui place un séparateur entre sous rubriques au rang spécifié.

void addActionListener(ActionListener) pour associer l'objet qui traitera l'événement de sélection de cette rubrique (voir 2.1.6.3). En général inutile s'il y a des sous_rubriques puisque chacune traitera ses propres événements (voir **JMenuItem**)

Remarque : la classe **Jmenu** hérite de **JMenuItem** de sorte que l'un des éléments d'une rubrique peut être lui-même une rubrique.

La classe **JMenuItem**

Elle permet de définir une sous-rubrique dans une barre de menu. Ses principales méthodes sont :

JMenuItem(String) construction avec définition du texte de la sous-rubrique.

JMenuItem (ImageIcon) construction avec une icône dans la sous-rubrique.

JMenuItem (String,ImageIcon) construction avec définition du texte et d'une icône dans la sous-rubrique.

void setEnabled(boolean) valide ou invalide la possibilité d'utiliser la sous-rubrique.

String getText() qui retourne le texte contenu dans la sous-rubrique.

void addActionListener(ActionListener) pour associer l'objet qui traitera l'événement de sélection de cette sous-rubrique (voir 2.1.6.3)

2.1.2.11 La classe **JFileChooser**

Elle permet de faire apparaître une zone de choix de fichier capable de gérer le parcours des répertoires. Ses principales méthodes sont :

JFileChooser() construction avec le répertoire courant comme point de départ.

JFileChooser(String) construction avec comme point de départ le répertoire donné en paramètre

File getSelectedFile() qui retourne le fichier sélectionné

File[] getSelectedFiles() qui retourne les fichiers sélectionnés quand on peut en sélectionner plusieurs

Void setDialogTitle(String) qui définit le titre de la zone de choix de fichier

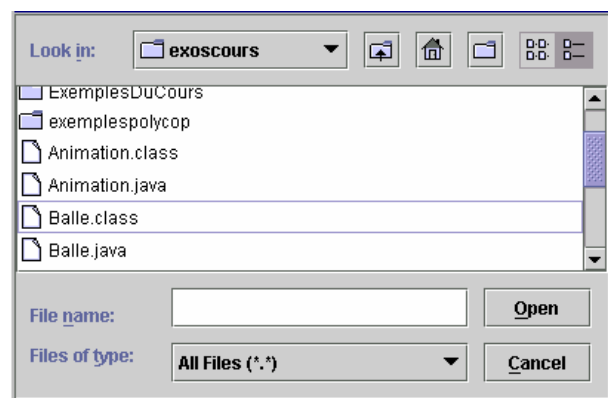
void setFileSelectionMode(int) qui définit le mode de sélection de fichier. La paramètre peut être

JFileChooser.FILES_ONLY, **JFileChooser.DIRECTORIES_ONLY** ou

FileChooser.FILES_AND_DIRECTORIES pour autoriser seulement les fichiers, seulement les répertoires ou les deux.

Void setMultipleSelectionEnabled(boolean) qui autorise ou interdit les sélections multiples.

void addActionListener(ActionListener) pour associer l'objet qui traitera l'événement de sélection d'un fichier (voir 2.1.6.3)



2.1.3 Placement des objets

Le placement des composants fait appel à des classes spéciales permettant de définir où se situent les divers composants de l'interface graphique et comment ils doivent se comporter lorsque l'on modifie les dimensions de la fenêtre les contenant.

2.1.3.1 Définition de l'interface

Le principe de définition d'une interface est de doter une fenêtre d'un contenant dans lequel viendront se placer les composants de l'interface (boutons, cases à cocher ...) ou de nouveaux contenants. Cet emboîtement de contenants les uns dans les autres permet de définir des interfaces aussi complexes que nécessaire. Le contenant définit la façon dont seront présentés les éléments (avec ascenseurs, par onglets ...). Le placement des éléments eux mêmes dans le contenant est régi par un objet de placement que l'on associe au contenant. Cet objet de placement permet de choisir la façon dont les éléments se situent dans le contenant et les uns par rapport aux autres (en tableau, en ligne ...).

La démarche suivie peut donc être la suivante :

1. Faire un dessin de l'interface en y plaçant tous les composants
2. Créer une classe héritée de **JFrame** (voir 2.1.3.2) ou de **JDialog** (voir 2.1.3.3)
3. Définir , si nécessaire, les composants de la barre de menu, des rubriques et sous-rubriques (voir 2.1.2.10)
4. Ajouter, cette barre de menu à la fenêtre par la méthode **setMenuBar(JMenuBar)** de cette dernière (voir 2.1.3.2).
5. Récupérer le contenant associé à la fenêtre par la méthode **getContentPane** de cette dernière (voir 2.1.3.2). Ce contenant est de classe **JPanel**, (on peut le modifier en utilisant la méthode **setContentPane** de la fenêtre).
6. Choisir l'objet de placement le plus approprié : **BorderLayout**, **FlowLayout**, **GridLayout**, **GridBagLayout**, **BoxLayout** ou **OverlayLayout** (voir 2.1.3.5) et l'associer au contenant par la méthode **setLayout** de ce dernier (voir 2.1.3.4)
7. A l'aide de la méthode **add** du contenant, placer les composants d'interface dans les zones définies par l'objet de placement choisi à l'étape 6. Ceci uniquement pour les composants qui sont seuls dans une zone. On mettra un contenant dans les zones dans lesquelles doivent se trouver plusieurs composants. Ce contenant est un objet de classe **JPanel**, **JScrollPane**, **JLayeredPane**, **JSplitPane**, **JTabbedPane**, **Box** ou **JInternalFrame** selon le comportement souhaité (voir 2.1.3.4).
8. Pour chacun de ces contenants, refaire les étapes 6 et 7 de façon à y positionner des composants ou de nouveaux contenants pour lesquels on suivra la même démarche. Lorsque c'est terminé ajouter ce contenant à son propre contenant par la méthode **add** de ce dernier.
9. Continuer jusqu'à ce que tous les composants aient trouvé leur place.

2.1.3.2 La classe JFrame

Elle permet de réaliser des interfaces en devenant le réceptacle où viendront se placer les divers composants d'interface. Elle hérite de la classe **Component** et ajoute les méthodes propres à la gestion des fenêtres suivantes :

JFrame(String) qui construit la fenêtre avec son titre

Container getContentPane() qui retourne le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants). Si on ne l'a pas modifié, ce contenant est de classe **JPanel**.

setContentPane(Container) qui redéfinit le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants)

void dispose() supprime la fenêtre, elle disparaît de l'écran

Image getIconImage() retourne l'image affichée comme icône de cette fenêtre

void setIconImage(Image) définit l'image utilisée comme icône de cette fenêtre

String getTitle() retourne le titre de la fenêtre

void setTitle(String) définit le titre de la fenêtre

void toBack() place la fenêtre derrière les autres

void toFront() place la fenêtre devant les autres

boolean isResizable() indique si la fenêtre peut être redimensionnée ou pas

void setResizable(boolean) autorise ou non le redimensionnement de la fenêtre

void setJMenuBar(MenuBar) définit la barre de menu de la fenêtre (voir 2.1.2.10)

void pack() donne à chaque élément contenu dans la fenêtre sa taille préférentielle.

void setVisible() qui rend la fenêtre visible à l'écran.

void addWindowListener(WindowListener) permet de prendre en compte les événements concernant la fenêtre (fermeture ...)

Ce n'est pas directement dans un objet de classe **JFrame** que l'on place les composants mais dans un contenant qui lui est associé (voir 2.1.3.1).

2.1.3.3 La classe JDialog

Elle est utilisée pour faire apparaître des fenêtres temporaires permettant d'afficher un message ou de faire une saisie. Il est possible de rendre ces fenêtres prioritaires dans le sens où on ne peut rien faire d'autre tant qu'on ne les a pas fermées. Ceci permet d'obliger la lecture d'un message ou une saisie avant toute autre utilisation de l'interface principale. Elle hérite de la classe **Component** et ajoute les méthodes propres à la gestion des fenêtres suivantes :

JDialog(Frame,boolean) qui construit une fenêtre de dialogue sans titre attachée à la fenêtre passée en premier paramètre. Le second paramètre indique si cette fenêtre est ou non prioritaire (s'il est omis, sa valeur est false et la fenêtre de dialogue est non prioritaire)

JDialog(Frame,String,boolean) qui construit une fenêtre de dialogue avec titre attachée à la fenêtre passée en premier paramètre. Le deuxième paramètre est le titre de la fenêtre. Le dernier paramètre indique si cette fenêtre est ou non prioritaire (s'il est omis, sa valeur est false et la fenêtre de dialogue est non prioritaire)

Container getContentPane() qui retourne le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants). Si on ne l'a pas modifié, ce contenant est de classe **JPanel**.

setContentPane(Container) qui redéfinit le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants)

void dispose() supprime la fenêtre, elle disparaît de l'écran

String getTitle() retourne le titre de la fenêtre

void setTitle(String) définit le titre de la fenêtre

boolean isResizable() indique si la fenêtre peut être redimensionnée ou pas

void setResizable(boolean) autorise ou non le redimensionnement de la fenêtre

void setJMenuBar(MenuBar) définit la barre de menu de la fenêtre (voir 2.1.2.10)

void setLocationRelativeTo(Component) définit la position de la fenêtre de dialogue comme relative au composant passée en paramètre, elle sera centrée sur ce composant.

void pack() donne à chaque élément contenu dans la fenêtre sa taille préférentielle.

void setVisible() qui rend la fenêtre visible à l'écran.

void setDefaultCloseOperation(int) définit l'opération qui sera faite lorsque l'utilisateur tentera de fermer la fenêtre de dialogue. les valeurs du paramètre peuvent être :

JDialog.DO_NOTHING_ON_CLOSE pour que rien ne soit automatiquement fait. Dans ce cas, il faudra que la fermeture de la fenêtre soit traitée par un contrôleur d'événements (voir **addWindowListener** ci-dessous).

JDialog.HIDE_ON_CLOSE pour que la fenêtre soit automatiquement cachée. Les traitements d'événements de fermeture éventuellement associés à cette fenêtre par **addWindowListener** seront effectués.

JDialog.DISPOSE_ON_CLOSE pour que la fenêtre soit automatiquement fermée. Les traitements d'événements de fermeture éventuellement associés à cette fenêtre par **addWindowListener** seront effectués.

void addWindowListener(WindowListener) permet de prendre en compte les événements concernant la fenêtre (fermeture ...)

Ce n'est pas directement dans un objet de classe **JDialog** que l'on place les composants mais dans un contenant qui lui est associé (voir 2.1.3.1).

2.1.3.4 Les contenants

Les contenants sont des objets appartenant à des classes qui héritent de **Container** qui, elle même, hérite de **Component** (voir 2.1.1). Les principaux contenants définis par SWING sont **JPanel**, **JScrollPane**, **JLayeredPane**, **JSplitPane**, **JTabbedPane** et **JInternalFrame**.

Dans ces contenants on ajoute les composants de l'interface ou d'autres contenants (voir 2.1.3.1).

Voici les principales méthodes communes à toutes ces classes :

void setLayout(LayoutManager) qui associe au contenant l'objet de placement passé en paramètre.

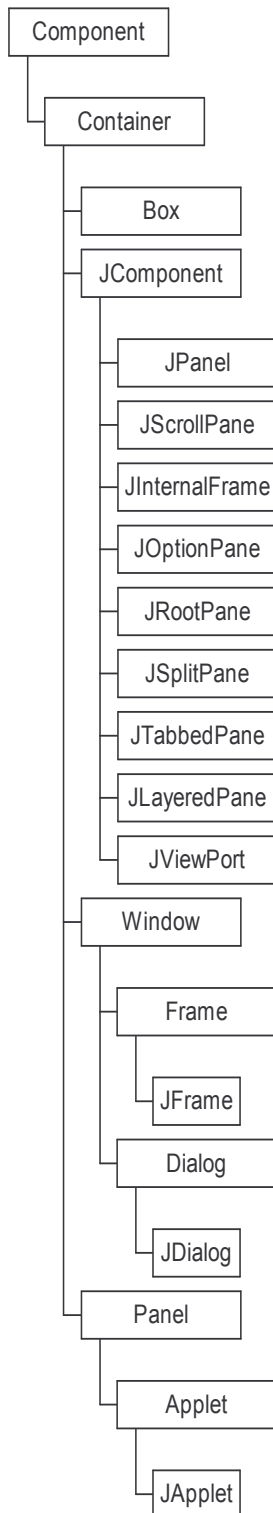
void add(Component) qui ajoute ce composant au contenant sans désignation de zone lorsque l'objet de placement n'en a pas besoin (**FlowLayout**, **GridLayout** ou **GridBagLayout**) (voir 2.1.3.5)

void add(Component,Object) qui ajoute ce composant au contenant en utilisant une désignation de zone qui dépend de l'objet de placement qui a été associé à ce contenant (**int** pour un **BorderLayout**) (voir 2.1.3.5).

void remove(Component) qui enlève ce composant au contenant

void removeAll() qui enlève tous les composants au contenant

Component locate(int,int) qui retourne le composant situé aux coordonnées données en paramètre



La classe Box

Elle permet de placer des éléments les uns à côté des autres sur une ligne ou une colonne comme dans des barres de boutons. Ses principales méthodes sont :

Box(int) qui crée une boîte horizontale ou verticale selon que le paramètre est `BoxLayout.X_AXIS` ou `BoxLayout.Y_AXIS`.

Component CreateGlue() qui crée un composant pouvant être ajouté à la boîte. Ce composant permet aux autres composants contenus dans la boîte de garder leur taille lorsque l'on redimensionne la boîte. En effet c'est lui qui récupère la place restante quand on agrandit la boîte et fournit la place demandée quand on la rapetisse.

Component CreateHorizontalGlue() qui crée un composant pouvant être ajouté à la boîte. Ce composant a un comportement comparable à celui décrit pour **CreateGlue** mais seulement pour les modifications de taille horizontale.

Component CreateVerticalGlue() qui crée un composant pouvant être ajouté à la boîte. Ce composant a un comportement comparable à celui décrit pour **CreateGlue** mais seulement pour les modifications de taille verticale.

Component CreateHorizontalStrut(int) qui crée un composant pouvant être ajouté à la boîte. Ce composant constitue un séparateur horizontal entre les autres composants. Le paramètre est l'épaisseur de ce séparateur en pixels.

Component CreateVerticalStrut(int) qui crée un composant pouvant être ajouté à la boîte. Ce composant constitue un séparateur vertical entre les autres composants. Le paramètre est la hauteur de ce séparateur en pixels.

Remarque : La classe `Box` est associée à un objet de placement de classe **BoxLayout** qui convient parfaitement à son usage. Il n'est donc pas nécessaire de lui associer un objet de placement par la méthode **setLayout**.

La classe JPanel

Elle correspond au contenant le plus simple. Elle offre deux constructeurs :

JPanel() qui se contente de créer l'objet.

JPanel(LayoutManager) qui accepte en paramètre un objet de placement et construit le **JPanel** associé à cet objet.

La classe JScrollPane

C'est une version améliorée de **JPanel** qui possède des ascenseurs de défilement vertical et horizontal. Ses principales méthodes sont :

JScrollPane() qui crée un **JScrollPane** vide

JScrollPane(Component) qui crée un **JScrollPane** contenant un seul composant (celui passé en paramètre).

JScrollPane(int,int) qui crée un **JScrollPane** vide en précisant le comportement des ascenseurs (voir ci-dessous).

JScrollPane(Component,int,int) qui crée un **JScrollPane** contenant un seul composant (celui passé en paramètre) en précisant le comportement des ascenseurs.

Le premier entier définit le comportement de l'ascenseur vertical, il peut prendre les valeurs :

`JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED` (l'ascenseur n'apparaît que s'il est nécessaire),

`JScrollPane.VERTICAL_SCROLLBAR_NEVER` (pas d'ascenseur) ou

`JScrollPane.VERTICAL_SCROLLBAR_ALWAYS` (l'ascenseur est toujours présent).

Le dernier entier définit le comportement de l'ascenseur horizontal, il peut prendre les valeurs :

`JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED` (l'ascenseur n'apparaît que s'il est

nécessaire), `JScrollPane.HORIZONTAL_SCROLLBAR_NEVER` (pas d'ascenseur) ou `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS` (l'ascenseur est toujours présent).

La classe `JLayeredPane`

Elle permet de définir des couches superposées. Quand on ajoute un élément à un `JLayeredPane`, on précise à quelle profondeur il se situe. Les couches de profondeur moindre cachent les autres. Ses principales méthodes sont :

- `void add(Component,Integer)`** qui ajoute ce composant au contenant dans une couche donnée. Plus le second paramètre est élevé moins la couche est profonde.
- `int getLayer(Component)`** qui retourne le numéro de la couche où se trouve le composant passé en paramètre.
- `void setLayer(Component,int)`** qui déplace le composant dans la couche dont le numéro est passé en second paramètre.
- `int lowestLayer()`** qui retourne le numéro le plus élevé correspondant à une couche occupée.
- `int highestLayer()`** qui retourne le numéro le plus faible correspondant à une couche occupée.

La classe `JSplitPane`

Elle permet de définir deux zones adjacentes avec une barre de séparation pouvant être déplacée à la souris. Ces zones peuvent être côte à côte ou l'une sous l'autre. Ses principales méthodes sont :

- `JSplitPane(int)`** qui construit une `JSplitPane` avec une indication d'orientation. Le paramètre peut être `JSplitPane.VERTICAL_SPLIT` ou `JSplitPane.HORIZONTAL_SPLIT`
- `JSplitPane(int,Component,Component)`** qui construit une `JSplitPane` avec une indication d'orientation en premier paramètre (`JSplitPane.VERTICAL_SPLIT` ou `JSplitPane.HORIZONTAL_SPLIT`), et les deux composants à y placer.

Remarque : On peut ajouter à ces deux constructeurs un dernier paramètre booléen qui indique si, lors du déplacement de la barre de séparation des deux zones, leurs contenus doivent être redessinés en permanence (true) ou seulement lorsque la barre cessera de bouger (false).

- `Component getLeftComponent()`** qui retourne le composant de gauche (ou celui de dessus).
- `Component getRightComponent()`** qui retourne le composant de droite (ou celui de dessous).
- `Component getTopComponent()`** qui retourne le composant de dessus (ou celui de droite).
- `Component getBottomComponent()`** qui retourne le composant de dessous (ou celui de gauche).
- `int getDividerSize()`** qui retourne la taille en pixels de la séparation.
- `int getDividerLocation()`** qui retourne la position de la séparation (en pixels).
- `int getMaximumDividerLocation()`** qui retourne la position maximale possible de la séparation (en pixels).
- `int getMinimumDividerLocation()`** qui retourne la position minimale possible de la séparation (en pixels).
- `int setDividerLocation(double)`** qui positionne la séparation. La valeur est un réel représentant un pourcentage de 0.0 à 1.0.
- `void setDividerSize(int)`** qui définit l'épaisseur de la barre de séparation (en pixels).

La classe `JTabbedPane`

Elle permet de placer les éléments selon des onglets. Ses principales méthodes sont :

- `JTabbedPane(int)`** crée l'objet en précisant où se placent les onglets. Le paramètre peut prendre les valeurs : `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT` ou `JTabbedPane.RIGHT`.
- `void addTab(String,Component)`** ajoute un onglet dont le libellé est donné par le premier paramètre et y place le composant donné en second paramètre
- `void addTab(String,ImageIcon,Component)`** ajoute un onglet dont le libellé est donné par le premier paramètre et auquel est associée une icône (second paramètre) et y place le composant donné en dernier paramètre
- `void addTab(String,ImageIcon,Component,String)`** ajoute un onglet dont le libellé est donné par le premier paramètre et auquel est associée une icône (second paramètre) et y place le composant donné en troisième paramètre. Le dernier est une bulle d'aide qui apparaîtra lorsque la souris passera sur l'onglet.
- `void insertTab(String,ImageIcon,Component,String,int)`** insère un onglet dont le libellé est donné par le premier paramètre et auquel est associée une icône (second paramètre) et y place le composant donné en troisième paramètre. Le quatrième paramètre est une bulle d'aide qui apparaîtra lorsque la souris passera sur l'onglet. Le dernier indique la position à laquelle doit être inséré l'onglet.
- `void removeTabAt(int)`** qui supprime l'onglet désigné.
- `void setIconAt(int ,ImageIcon)`** associe une icône à l'onglet désigné
- `int getSelectedIndex()`** qui retourne le numéro de l'onglet sélectionné
- `void setSelectedIndex(int)`** qui sélectionne l'onglet dont le numéro est passé en paramètre

int getTabCount() qui retourne le nombre d'onglets disponibles.
int indexOfTab(ImageIcon) retourne le premier onglet associé à l'icône passée en paramètre
int indexOfTab(String) retourne le premier onglet associé au nom passé en paramètre

Remarque : les deux méthodes précédentes peuvent lever une exception de classe **ArrayIndexOutOfBoundsException** si la position de l'onglet n'est pas correcte.

String getTitleAt(int) qui retourne le libellé de l'onglet désigné
void setTitleAt(int,String) qui définit le libellé de l'onglet désigné.
boolean isEnabledAt(int) qui indique si l'onglet désigné est accessible ou pas.
boolean setEnabledAt(int,boolean) qui rend accessible ou pas l'onglet désigné.
int getTabCount() qui retourne le nombre d'onglets.
int indexOfTab(String) qui retourne le numéro correspondant à l'onglet désigné par son libellé.
void setBackgroundAt(int,Color) qui définit la couleur de fond pour la page correspondant à l'onglet désigné par le premier paramètre.
void setForegroundAt(int,Color) qui définit la couleur de tracé pour la page correspondant à l'onglet désigné par le premier paramètre.
void addChangeListener(ChangeListener) pour associer l'objet qui traitera les changements d'onglet (voir 2.1.6.3)

La classe JInternalFrame

C'est une version allégée de **JFrame** qui permet de définir des sous-fenêtres qui se comportent comme des fenêtres normales mais sont liées à leur contenant. Ses principales méthodes sont :

JInternalFrame(String,boolean,boolean,boolean,boolean) qui crée une fenêtre avec un titre passé en 1^{er} paramètre. Le deuxième paramètre indique si elle peut être redimensionnée ou pas. Le troisième paramètre indique si elle peut être fermée ou pas. Le quatrième paramètre indique si elle peut être mise en pleine taille ou pas. Le dernier paramètre indique si elle peut être mise en icône ou pas. Si le dernier, les deux derniers, les trois derniers ou les quatre derniers paramètres sont omis, ils sont tous considérés comme false.
Container getContentPane() qui retourne le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants). Si on ne l'a pas modifié, ce contenant est de classe **JPanel**.
setContentPane(Container) qui redéfinit le contenant associé à cette fenêtre qui servira de base à tout ce que l'on y placera (composants ou autres contenants)
void dispose() supprime la fenêtre, elle disparaît de l'écran
String getTitle() retourne le titre de la fenêtre
void setTitle(String) définit le titre de la fenêtre
void toBack() place la fenêtre derrière les autres
void toFront() place la fenêtre devant les autres
void setJMenuBar(MenuBar) définit la barre de menu de la fenêtre (voir 2.1.2.10)
void pack() donne à chaque élément contenu dans la fenêtre sa taille préférentielle.
void setIcon() met la fenêtre en icône.
void setClosed() ferme la fenêtre.
void setSelected(boolean) sélectionne ou dé-sélectionne la fenêtre.

Remarque pour les trois précédentes méthodes, si l'opération n'est pas possible, une exception de classe **PropertyVetoException** est levée.

void setVisible() qui rend la fenêtre visible à l'écran.
void addInternalFrameListener(InternalFrameListener) pour associer l'objet qui traitera les événements concernant la fenêtre (fermeture ...)



2.1.3.5 Les objets de placement

Le placement des composants fait appel à des classes spéciales permettant de définir où se situent les divers composants de l'interface graphique et comment ils doivent se comporter lorsque l'on modifie les dimensions de la fenêtre les contenant.

Pour cela on fait appel à un objet de l'une des classes disponibles pour organiser le placement :

BorderLayout pour un placement simple en 5 zones (Nord, sud, Est, Ouest et Centre)

FlowLayout pour un placement ligne à ligne. Dès qu'une ligne est pleine on passe à la suivante.
GridLayout pour un placement en tableau avec une case par élément. Le tableau est rempli ligne par ligne.
GridBagLayout pour un placement en grille avec une ou plusieurs cases par élément.

La classe BorderLayout

Cette classe définit 5 zones. Les composants y sont ajoutés par la méthode **add(Composant,int)** dont le second paramètre indique dans quelle zone doit être placé le composant. Ce paramètre peut prendre les valeurs : **BorderLayout.NORTH**, **BorderLayout.SOUTH**, **BorderLayout.EAST**, **BorderLayout.WEST** ou **BorderLayout.CENTER**

La classe FlowLayout

Cette classe permet un placement ligne par ligne. Elle possède un constructeur qui définit la façon dont les composants seront ajoutés par la suite à l'aide de la méthode **add(Component)**. Ce constructeur accepte trois paramètres selon le modèle suivant **FlowLayout(int,int,int)**. Le premier paramètre indique comment seront alignés les composants, il peut prendre les valeurs suivantes : **FlowLayout.CENTER**, **FlowLayout.LEFT** ou **FlowLayout.RIGHT**. Le deuxième paramètre donne l'espacement horizontal entre composants (en pixel). Le dernier paramètre donne l'espacement vertical entre composants (en pixel).

La classe GridLayout

Cette classe permet un placement sur un tableau. Elle possède un constructeur qui définit les dimensions de ce tableau et les espacements entre les composants. Ce constructeur accepte quatre paramètres selon le modèle suivant **GridLayout(int,int,int,int)**. Le premier paramètre est le nombre de lignes du tableau tandis que le deuxième est le nombre de colonnes du tableau. Le troisième paramètre donne l'espacement horizontal entre composants (en pixel). Le dernier paramètre donne l'espacement vertical entre composants (en pixel). Les composants seront ajoutés par la suite à l'aide de la méthode **add(Component)** qui remplira le tableau ligne par ligne.

La classe GridBagLayout

Cette classe permet un contrôle plus précis du placement des composants grâce à l'utilisation d'un objet de classe **GridBagConstraints** qui permet de définir les règles de placement pour chaque composant. **GridBagLayout** permet de placer les composants dans une grille dont les cases sont numérotées de gauche à droite et de haut en bas à partir de 0. Chaque composant peut occuper une ou plusieurs cases et la taille des cases est ajustée en fonction des composants qu'elles contiennent. Toutes les cases d'une même ligne ont la même hauteur qui correspond à la hauteur nécessaire pour placer le composant le plus haut de cette ligne. De même toutes les cases d'une même colonne ont la même largeur qui correspond à la largeur nécessaire pour placer le composant le plus large de cette ligne.

Pour placer un composant il faut suivre la procédure suivante :

- appeler la méthode **setLayout** avec en paramètre le nom de l'objet de placement de classe **GridBagLayout**
- préparer un objet de classe **GridBagConstraints** permettant de définir la position, la taille et le comportement de l'objet à placer
- utiliser la méthode **setConstraints** de l'objet de placement en lui donnant en paramètre le composant d'interface graphique à placer et l'objet de classe **GridBagConstraints** précédemment préparé.
- ajouter ce composant à l'interface graphique par la méthode **add**

Les objets de la classe **GridBagConstraints** comportent les champs suivants :

gridx et **gridy** pour définir les coordonnées de la case en haut à gauche du composant

gridwidth et **gridheight** pour définir le nombre de cases qu'il occupe horizontalement et verticalement

ipadx et **ipady** pour définir les marges internes du composant

fill pour indiquer dans quelle(s) direction(s) le composant se déforme pour s'adapter à la taille de la (des) cellule(s). Ce champ peut prendre les valeurs suivantes : **GridBagConstraints.NONE** , **BOTH** , **HORIZONTAL** , **VERTICAL**

anchor pour indiquer comment le composant se place dans la (les) cellule(s). Ce champ peut prendre les valeurs suivantes : **GridBagConstraints.NONE**, **CENTER** , **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST** et **NORTHWEST**.

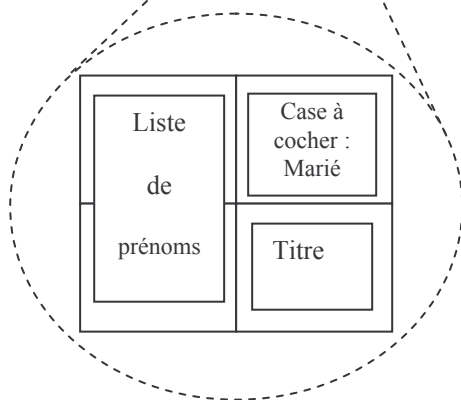
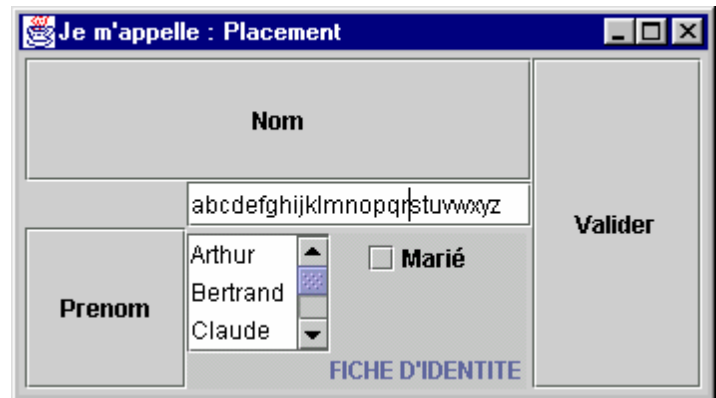
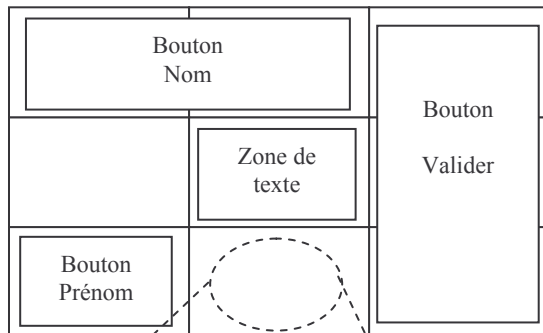
insets pour définir les marges externes du composant. Ce champ est positionné en faisant **new Insets(h,g,b,d)** où h définit la marge haute, b la marge basse, g la marge gauche et d la marge droite.

weightx et **weighty** permettent d'indiquer quelle part de surface est distribuée ou retirée à ce composant lors d'un redimensionnement de la fenêtre. Ils expriment un pourcentage de répartition de largeur (respectivement hauteur) pour ce composant.

2.1.4 Exemple de placement de composants dans une interface graphique :

On veut définir une interface ayant l'aspect ci-dessous.

On va pour cela tracer une grille correspondant à la fenêtre associée à l'application et déterminer la position et la taille de chaque composant :



Placement des composants dans la fenêtre principale (celle de l'applet) :

Composant	Coordonnées	taille	marges internes	remplissage	positionnement	marges externes	Redimensionnement
Bouton Nom	0 , 0	2 , 1	2 , 2	BOTH	CENTER	3 , 3 , 0 , 0	0 , 50
Bouton Prénom	0 , 2	1 , 1	2 , 2	BOTH	CENTER	0 , 3 , 3 , 0	0 , 0
Zone de texte	1 , 1	1 , 1	2 , 2	NONE	WEST	0 , 0 , 0 , 0	100 , 0
Sous le texte	1 , 2	1 , 1	2 , 2	BOTH	CENTER	0 , 3 , 3 , 0	100 , 50
Bouton Valider	2 , 0	1 , 3	10 , 3	BOTH	CENTER	3 , 0 , 3 , 3	0 , 0

Si on redimensionne cette fenêtre, la hauteur se répartira entre les lignes 0 et 2 désignées par le boutons Nom et la zone sous le texte (50% pour chacun) et la largeur ira à la colonne 1 désignée par le texte ou la zone qui est dessous (100%)

Placement des composants dans la case sous la zone de texte :

Composant	Coordonnées	taille	marges internes	remplissage	positionnement	marges externes	Redimensionnement
Liste de prénoms	0 , 0	1 , 2	2 , 2	BOTH	WEST	0 , 0 , 0 , 0	100 , 100
Case à cocher	1 , 0	1 , 1	2 , 2	NONE	NORTH	0 , 0 , 0 , 0	0 , 0
Titre	1 , 1	1 , 1	2 , 2	NONE	NORTH	0 , 0 , 0 , 0	0 , 0

Le fichier correspondant est le suivant :

```
import java.awt.*;
import javax.swing.*;

class FenetrePlacement extends JFrame {
    // definition des noms de composants
    private JButton afficheNom; // Le bouton "Nom"
    private JButton affichePrenom; // Le bouton "Prénom"
    private JTextField affichage; // La zone ou s'affiche l'identité
}
```

```

private JButton valider;           // Le bouton de validation
private JList prenom;             // La liste proposant des prénoms
private JCheckBox marie;          // La case à cocher "Marié"
private JLabel titre;             // Le titre de la fiche

public FenetrePlacement(String nomFenetre) {
    super("Je m'appelle : "+nomFenetre); // création de la fenêtre avec son nom
    setSize(370,250); // dimensionnement de cette fenêtre

    // définition des noms d'objets d'interface
    afficheNom=new JButton("Nom");
    affichePrenom=new JButton("Prenom");
    valider=new JButton("Valider");
    affichage=new JTextField("abcdefghijklmnopqrstuvwxyz",40);
    String[] donnees = {"Arthur","Bertrand","Claude","David","Emile","François"};
    prenom=new JList(donnees);
    marie=new JCheckBox("Marié");
    titre=new JLabel("FICHE D'IDENTITE");

    // définition des objets utilisés pour placer les composants
    GridBagLayout placeur=new GridBagLayout(); // objet de placement des composants
    GridBagConstraints contraintes=new GridBagConstraints(); // regles de placement
    getContentPane().setLayout(placeur); // utiliser cet objet de placement pour la fenêtre

    // placement du bouton afficheNom
    contraintes.gridx=0; contraintes.gridy=0; // coordonnées 0,0
    contraintes.gridwidth=2; contraintes.gridheight=1; // 2 cases en largeur
    contraintes.fill=GridBagConstraints.BOTH;
    contraintes.anchor=GridBagConstraints.CENTER;
    contraintes.weightx=0; contraintes.weighty=50;
    contraintes.insets=new Insets(3,3,0,0);
    contraintes.ipadx=2; contraintes.ipady=2;
    placeur.setConstraints(afficheNom, contraintes);
    getContentPane().add(afficheNom);

    // placement du bouton affichePrenom
    contraintes.gridx=0; contraintes.gridy=2; // coordonnées 0,2
    contraintes.gridwidth=1; contraintes.gridheight=1; // occupe 1 case
    contraintes.fill=GridBagConstraints.BOTH;
    contraintes.anchor=GridBagConstraints.CENTER;
    contraintes.weightx=0; contraintes.weighty=0;
    contraintes.insets=new Insets(0,3,3,0);
    contraintes.ipadx=2; contraintes.ipady=2;
    placeur.setConstraints(affichePrenom, contraintes);
    getContentPane().add(affichePrenom);

    // placement de la zone de texte
    contraintes.gridx=1; contraintes.gridy=1; // coordonnées 1,1
    contraintes.gridwidth=1; contraintes.gridheight=1; // occupe 1 case
    contraintes.fill=GridBagConstraints.BOTH;
    contraintes.anchor=GridBagConstraints.WEST;
    contraintes.weightx=100; contraintes.weighty=0;
    contraintes.insets=new Insets(0,0,0,0);
    contraintes.ipadx=2; contraintes.ipady=2;
    placeur.setConstraints(affichage, contraintes);
    getContentPane().add(affichage);

    // placement du bouton valider
    contraintes.gridx=2; contraintes.gridy=0; // coordonnées 2,0
    contraintes.gridwidth=1; contraintes.gridheight=3; // 3 cases en hauteur
    contraintes.fill=GridBagConstraints.BOTH;
    contraintes.anchor=GridBagConstraints.CENTER;
    contraintes.weightx=0; contraintes.weighty=0;
    contraintes.insets=new Insets(3,0,3,3);
    contraintes.ipadx=10; contraintes.ipady=3;
    placeur.setConstraints(valider, contraintes);
    getContentPane().add(valider);

    // Ajout d'un JPanel pour placer les autres composants dans la case
    // du milieu en bas et création d'un objet de placement pour ce JPanel
    GridBagLayout panelPlaceur=new GridBagLayout(); // objet de placement pour le panel
    GridBagConstraints panelContraintes=new GridBagConstraints(); // regles de placement
    JPanel caseDuBas=new JPanel(panelPlaceur); //création du Panel avec son objet de placement
    // placement des 3 composants dans le Panel
    // placement de la liste dans un JScrollPane pour avoir des ascenseurs
    prenom.setVisibleRowCount(3); // 3 lignes visibles dans la liste
    JScrollPane defile=new JScrollPane(prenom);
    panelContraintes.gridx=0; contraintes.gridy=0; // coordonnees 0,0
    panelContraintes.gridwidth=1; contraintes.gridheight=2; // occupe 2 cases
    panelContraintes.fill=GridBagConstraints.BOTH;
    panelContraintes.anchor=GridBagConstraints.WEST;

```

```

panelContraintes.weightx=100; contraintes.weighty=100;
panelContraintes.insets=new Insets(0,0,0,0);
panelContraintes.ipadx=2; contraintes.ipady=2;
panelPlaceur.setConstraints(defile, panelContraintes);
caseDuBas.add(defile);
    // placement de la case à cocher
panelContraintes.gridx=1; contraintes.gridy=0; // coordonnees 1,0
panelContraintes.gridwidth=1; contraintes.gridheight=1; // occupe 1 case
panelContraintes.fill=GridBagConstraints.NONE;
panelContraintes.anchor=GridBagConstraints.NORTH;
panelContraintes.weightx=0; contraintes.weighty=0;
panelContraintes.insets=new Insets(0,0,0,0);
panelContraintes.ipadx=2; contraintes.ipady=2;
panelPlaceur.setConstraints(marie, panelContraintes);
caseDuBas.add(marie);
    // placement du titre
panelContraintes.gridx=1; contraintes.gridy=1; // coordonnees 1,1
panelContraintes.gridwidth=1; contraintes.gridheight=1; // occupe 1 case
panelContraintes.fill=GridBagConstraints.NONE;
panelContraintes.anchor=GridBagConstraints.NORTH;
panelContraintes.weightx=0; contraintes.weighty=0;
panelContraintes.insets=new Insets(0,0,0,0);
panelContraintes.ipadx=2; contraintes.ipady=2;
panelPlaceur.setConstraints(titre, panelContraintes);
caseDuBas.add(titre);

// placement du panel dans l'applet
contraintes.gridx=1; contraintes.gridy=2; // coordonnees 1,2
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case en hauteur
contraintes.fill=GridBagConstraints.BOTH;
contraintes.anchor=GridBagConstraints.CENTER;
contraintes.weightx=100; contraintes.weighty=50;
contraintes.insets=new Insets(3,0,3,3);
contraintes.ipadx=10; contraintes.ipady=3;
placeur.setConstraints(caseDuBas, contraintes);
getContentPane().add(caseDuBas);

setVisible(true); // rendre visible cette fenetre
}
}

```

2.1.6 Traitement des événements

Chacun des constituants de l'interface est susceptible, selon sa classe, de réagir à certains types d'événements. On peut associer à chacun un contrôleur d'événement qui est un objet contenant une ou plusieurs méthodes de traitement des événements. Ces méthodes seront alors automatiquement exécutées lorsque l'événement se produira.

On définit ces objets de traitement par des classes obtenues, selon le cas, par implémentation d'une interface de contrôle ou par héritage d'un modèle de contrôleur. Ces classes peuvent être des classes internes à celle qui définit l'interface, elles peuvent être privées.

Une instance de contrôleur (instance d'un objet de la classe définie précédemment) est associé à chaque constituant de l'interface par sa méthode **addxxxListener** (se reporter à la description des méthodes des différents constituants d'interface en 2.1.2, 2.1.3.2 et 2.1.3.4)

2.1.6.1 Les événements élémentaires

Tous les constituants de l'interface sont sensibles à ces événements. Il s'agit des événements du clavier, de la souris et de visibilité (perte ou gain de visibilité).

Les événements correspondants sont les suivants :

Type d'événement	événement	classe d'événement
Clavier	touche appuyée, touche lâchée ou touche tapée	KeyEvent
Souris	clic, entrée ou sortie du curseur de la souris dans le composant, bouton de la souris appuyé ou lâché. Souris déplacée éventuellement avec un bouton appuyé	MouseEvent
Visibilité	devient visible ou devient non visible	FocusEvent

Les contrôleurs seront obtenus par héritage de la classe modèle. Les classes modèles et les méthodes associés à ces événements sont décrits par le tableau ci-dessous :

Type d'événement	événement	classe modèle interface modèle	classe d'événement	nom de méthode associée
Visibilité	devient visible	FocusAdapter	FocusEvent	focusGained
	devient non visible	FocusListener		focusLost
Clavier	touche appuyée	KeyAdapter KeyListener	KeyEvent	keyPressed
	touche lâchée			keyReleased
	touche tapée			keyTyped
Souris	clic	MouseAdapter MouseListener	MouseEvent	mouseClicked
	le curseur rentre dans le composant			mouseEntered
	le curseur sort du composant			mouseExited
	un bouton de la souris appuyé			mousePressed
	un bouton de la souris lâché			mouseReleased
	souris déplacée avec un bouton appuyé	MouseMotionAdapter MouseMotionListener	MouseMotionEvent	mouseDragged
	souris déplacée			mouseMoved

ATTENTION : Lorsqu'un événement clavier se produit, il est envoyé au composant actif. Un composant peut être rendu actif par l'utilisateur (à la souris) ou en utilisant sa méthode **requestFocus()** voir (2.1.1)

Pour les événements clavier ou souris il est possible de tester le contexte précis de l'événement (touches modificatrices du clavier et boutons de la souris). Pour cela on utilise la méthode **getModifiers()** de l'événement qui retourne un entier représentant ce contexte. Les tests suivants indiquent comment interpréter cette valeur (le nom evt désigne l'événement clavier ou souris testé, il est de classe **KeyEvent** ou **MouseEvent**) :

```
if ((evt.getModifiers() & InputEvent.SHIFT_MASK)!=0) // la touche shift est appuyée
if ((evt.getModifiers() & InputEvent.ALT_MASK)!=0) // la touche alt est appuyée
if ((evt.getModifiers() & InputEvent.CTRL_MASK)!=0) // la touche control est appuyée
if ((evt.getModifiers() & InputEvent.BUTTON1_MASK)!=0) // le bouton gauche de la souris est enfoncé
if ((evt.getModifiers() & InputEvent.BUTTON3_MASK)!=0) // le bouton droit de la souris est enfoncé
```

La méthode suivante ne concerne que les événements liés au clavier
char getKeyChar() qui retourne la caractère tapé

Les méthodes suivantes ne concernent que les événements liés à la souris

int getX() qui retourne la coordonnée horizontale du curseur de la souris au moment de l'événement
int getY() qui retourne la coordonnée verticale du curseur de la souris au moment de l'événement
int getClickCount() qui retourne le nombre de clics effectués

La méthode **void consume()** est utilisée pour les événements clavier et souris. Elle permet de signaler à java que l'événement a été traité et qu'il n'est donc plus nécessaire de le diffuser.

2.1.6.2 Les événements concernant les fenêtres

Les fenêtres sont sensibles à des événements concernant leur ouverture, fermeture, mise en icône etc. On distingue le cas des fenêtres normales (**JFrame**, **JDialog** et **JApplet**) des fenêtres internes **JInternalFrame**. Il existe donc deux classes d'événement et de contrôleur d'événements mais leur comportement est similaire.

Les événements correspondants sont les suivants :

Fenêtre	événement	classe d'événement
normales (JFrame , JDialog et JApplet)	activation, désactivation, ouverture, fermeture, mise en icône et restitution	WindowEvent
internes (JInternalFrame)		InternalFrameEvent

Les contrôleurs seront obtenus par héritage de la classe modèle. Les classes modèles et les méthodes associés à ces événements sont décrits par le tableau ci-dessous :

Fenêtre	événement	classe modèle interface modèle	classe d'événement	nom de méthode associée
normales : JFrame , JDialog et JApplet	La fenêtre devient active (elle recevra les saisies au clavier)	WindowAdapter WindowListener	WindowEvent	windowActivated
	La fenêtre devient inactive			windowDeactivated
	La fenêtre est fermée			windowClosed
	La fenêtre est en cours de fermeture (pas encore fermée)			windowClosing
	La fenêtre est mise en icône			windowIconified
	La fenêtre est restaurée			windowDeiconified
	La fenêtre est visible pour la première fois			windowOpened
internes : JInternalFrame	La fenêtre devient active (elle recevra les saisies au clavier)	InternalFrameAdapter InternalFrameListener	InternalFrameEvent	InternalFrameActivated
	La fenêtre devient inactive			InternalFrameDeactivated
	La fenêtre est fermée			InternalFrameClosed
	La fenêtre est en cours de fermeture (pas encore fermée)			InternalFrameClosing
	La fenêtre est mise en icône			InternalFrameIconified
	La fenêtre est restaurée			InternalFrameDeiconified
	La fenêtre est visible pour la première fois			InternalFrameOpened

2.1.6.3 Les événements de composants d'interface

Ce sont ceux qui se produisent lorsque l'on coche une case, on sélectionne un élément dans une liste etc. Les événements correspondants sont les suivants :

Type d'événement	événement	classe d'événement
Action	action sur le composant (clic sur un bouton, choix dans un menu ...)	ActionEvent
Ajustement	déplacement d'un ascenseur	AdjustmentEvent
Changement	modification de la position d'un composant	ChangeEvent
Élément	modification de l'état d'un élément (liste, case à cocher ...)	ItemEvent
Document	modification dans un texte	DocumentEvent
Curseur	déplacement du curseur d'insertion dans un texte	CaretEvent
Sélection	sélection d'un élément dans une liste	ListSelectionEvent

Chaque composant est susceptible de réagir à certains de ces événements mais pas à tous. Le tableau suivant donne, pour chaque composant, les événements auxquels il est sensible :

Événement Composant	Action	Ajustement	Changement	Élément	Document	Curseur	Sélection
JButton	X		X	X			
JCheckBox	X		X	X			
JComboBox				X			
JList							X
JScrollBar		X					
JSlider			X				
JProgressBar			X				
JTextField	X				X ⁽¹⁾	X	
JTextArea					X ⁽¹⁾	X	
JMenu	X		X	X			
JMenuItem	X		X	X			
FileChooser	X						

(1) Les composants JTextField et JTextArea ne sont pas directement sensibles à l'événement de document, mais ils sont associés à un objet de classe Document qui l'est (voir 2.1.2.9)

Remarque : Les croix marquées en gras correspondent aux événements généralement traités pour ce type de composant. Par exemple l'événement 'changement' pour un JButton est produit quand on enfonce le bouton et quand on le lâche alors que l'événement 'action' est produit quand on clique le bouton ce qui est généralement l'utilisation normale d'un tel composant.

Les contrôleurs seront obtenus par une classe implémentant l'interface de contrôle. Les interfaces de contrôle et les méthodes associés à ces événements sont décrits par le tableau ci-dessous :

Type d'événement	événement	interface de contrôle	classe d'événement	nom de méthode associée
Action	activation de bouton, case cochée, choix dans un menu ou choix d'un fichier, saisie de texte	ActionListener	ActionEvent	actionPerformed
Ajustement	modification de la position de l'ascenseur	AdjustmentListener	AdjustmentEvent	adjustmentValueChanged
Changement	modification de la position d'un curseur ou d'une barre de progression	ChangeListener	ChangeEvent	stateChanged
Elément	sélection d'un élément	ItemListener	ItemEvent	itemStateChanged
Document	modification, insertion ou suppression de texte	DocumentListener	DocumentEvent	changedUpdate removeUpdate insertUpdate
Curseur	déplacement du curseur d'insertion	CaretListener	CaretEvent	caretUpdate
Sélection	sélection d'un ou plusieurs éléments	ListSelectionListener	ListSelectionEvent	valueChanged

Remarque : Dans le cas des sélections l'événement est produit lors d'une modification de la sélection. Ceci signifie que si l'on sélectionne un élément déjà sélectionné il ne se passe rien. De plus, quand on sélectionne un nouvel élément, l'événement se produit 2 fois (quand on appuie le bouton de la souris et quand on le lâche). On peut éviter de traiter cet événement intermédiaire en utilisant la méthode **getValueIsAdjusting()** de l'événement de classe **ListSelectionEvent** qui retourne true si l'événement est intermédiaire (on peut alors l'ignorer).

ATTENTION : lorsque l'interface de contrôle possède plusieurs méthodes, toutes doivent être écrites même si elles sont vides.

2.1.6.4 Associer un contrôleur à un événement

Pour chaque élément pour lequel on veut traiter les événements il faudra :

- créer une classe obtenue, selon le cas, par implémentation d'une interface de contrôle ou par héritage d'un modèle de contrôleur correspondant au type d'événement que l'on désire traiter.
- y écrire les actions associées aux événements dans les méthodes correspondantes. Si le contrôleur a été défini à partir d'une interface de contrôle toutes les méthodes doivent être écrites même si elles sont vides. S'il a été défini par héritage d'une classe modèle, seules les méthodes utiles sont écrites (les autres sont automatiquement vides).
- associer cette classe au composant par la méthode **addxxxListener** de celui-ci. On utilisera selon le cas **addActionListener**, **addMouseListener** ... (se reporter aux tableaux précédents pour les noms)

Par exemple pour définir la classe associée aux clics sur le bouton 'Valider' de notre interface graphique du 2.1.4 il faudra faire :

```
private class ActionValider implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) {
        // action associée au clic du bouton valider
        // le paramètre permet de savoir quel événement s'est produit et dans quelle conditions
    }
}
```

Remarque : Cette classe sera une classe interne à celle qui définit l'interface graphique (*fenetrePlacement* dans l'exemple)

Pour associer ceci au bouton il faudra ajouter, dans le constructeur de l'interface graphique (après la création de l'objet correspondant à ce bouton), la ligne suivante :

`valider.addActionListener(new actionValider());`

Pour aller plus loin il faudra se reporter à la documentation de java pour y trouver plus d'informations sur les classes de composants et d'événements ainsi que toutes les classes qui n'ont pas été évoquées ici.

2.2 Le graphique

On utilisera, en général, pour faire du graphique, un composant de classe **JPanel**. Chaque composant possède son propre système de coordonnées dont le point 0,0 est en haut à gauche et les axes sont orientés vers la droite (axe des x) et vers le bas (axe des y). La couleur du tracé peut être définie par la méthode **setForeground** et on peut connaître sa valeur par **getForeground** (voir 2.1.1).

Pour pouvoir dessiner il faut disposer d'un objet de classe **Graphics**. On obtient un tel objet par appel de la méthode **getGraphics** du composant.

`Graphics zoneDeDessin = nomDuComposant.getGraphics();`

Chaque fois que le composant devra être dessiné ou redessiné à l'écran java fera appel à sa méthode **update**. Cette méthode efface le fond puis le remplit avec sa couleur et enfin invoque la méthode **paint** du composant qui ne fait rien. Il est possible de réécrire, dans une classe dérivée de celle du composant, les méthodes **update** et/ou **paint** pour les doter d'un comportement particulier. Elles sont définies comme suit :

```
public void update(Graphics g)
public void paint(Graphics g)
```

Lorsque l'on désire rafraîchir l'affichage d'un composant en dehors des instants où java le fait, on peut appeler la méthode **repaint** dont le fonctionnement est le même que celui décrit ci-dessus (appel de **update** etc.).

L'utilisation directe d'un objet de classe **JPanel** pose le problème de la mise à jour de l'affichage puisque la méthode **paint** ne fait rien. Il est donc nécessaire de créer sa propre classe dérivée de **JPanel** pour la doter d'une méthode **paint**. La solution la plus générale est de créer une classe possédant une zone de mémoire dans laquelle seront faits les dessins. La méthode **paint** de cette classe se contentera alors d'afficher cette zone de mémoire à l'écran. On peut obtenir une telle zone de mémoire pour tout composant par la méthode **createImage**. De plus, afin d'éviter les clignotements, on peut réécrire la méthode **update** de façon à ce qu'elle n'efface pas le fond. La classe définie ci-dessous permet ce fonctionnement :

```
import java.awt.*;
import javax.swing.*;

class ZoneGraphique extends JPanel {
    // Extension de la classe JPanel de swing :
    // Contient un tampon en mémoire dans lequel on peut dessiner.
    // ce tampon est affiché à l'écran chaque fois que l'on appelle la méthode
    // repaint() ou chaque fois que java fait un paint() (la fenêtre apparaît, bouge ...).
    // Ce tampon est de la classe Graphics de AWT et on y accède par la méthode
    // obtenirGraphics() on peut lui appliquer toutes les méthodes de dessin définies
    // dans la classe Graphics (voir AWT).

    private Image dessin=null; // tampon de dessin
    public int taillex=0; // dimensions de la zone de dessin
    public int tailley=0; // ces valeurs sont accessibles de l'extérieur

    private void creerTampon() { // creation du tampon
        taillex=getSize().width; // taille du tampon
        tailley=getSize().height;
        dessin=createImage(taillex, tailley); // creation du tampon
    }

    public Graphics obtenirGraphics() {
        // retourne le tampon dans lequel on peut dessiner et qui sera affiché
        // à l'écran par paint() et repaint()
        // voir si la taille de la zone de dessin n'a pas change
        // c'est le cas lorsque l'utilisateur redimensionne la fenetre
        if ((taillex != getSize().width) || (tailley != getSize().height)) {
            creerTampon(); // si oui, il faut recréer le tampon
        }
        return dessin.getGraphics(); // retourne le tampon de dessin
    }

    public void update(Graphics g) {
        // Surdéfinition de update() pour qu'elle n'efface pas
        paint(g);
    }
}
```

```

    }

    public void paint(Graphics g) {
        // Surdéfinition de paint() pour qu'elle affiche le tampon à l'écran
        if (dessin != null) g.drawImage(dessin,0,0,this); // affichage du tampon
    }
}

```

2.3 La classe ImageIcon

Elle permet de définir des icônes que l'on peut ensuite placer dans divers composants. Ses principales méthodes sont :

ImageIcon(String) qui construit l'icône à partir d'un fichier **.gif** ou **.jpg** dont on passe le nom en paramètre.

ImageIcon(URL) qui construit l'icône à partir d'un fichier **.gif** ou **.jpg** dont on passe l'URL en paramètre.

ImageIcon(Image) qui construit l'icône à partir d'un objet Image.

Image getImage() qui crée un objet Image contenant l'icône

int getIconWidth() qui retourne la largeur de l'icône

int getIconHeight() qui retourne la hauteur de l'icône

void paintIcon(Component,Graphics,int,int) qui dessine l'icône dans un composant dont on précise le contexte graphique. Les deux derniers paramètres sont les coordonnées du coin supérieur gauche où placer l'icône dans le composant.

2.4 Le son

Il est possible de jouer des sons contenus dans des fichiers au format **au** ou **wav**. Pour cela il faut créer un objet de classe **AudioClip** à partir du fichier désigné par son URL.

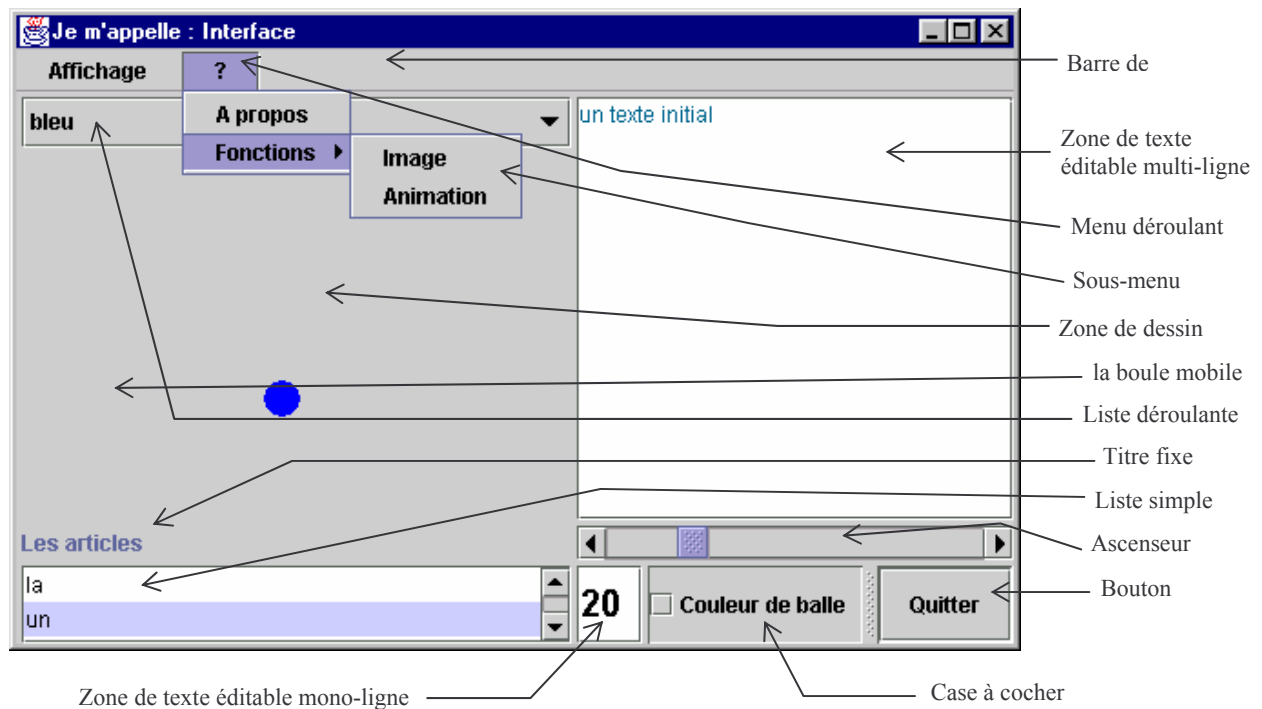
L'objet de classe **AudioClip** est obtenu grâce à la méthode **newAudioClip** de la classe **Applet** (cette méthode est statique et donc accessible directement sans qu'il soit nécessaire de créer d'objet de classe **Applet**). On écrira :

```
AudioClip son = Applet.newAudioClip(URL_où_se_trouve_le_son)
```

Après quoi le son peut être joué par la méthode **play** de l'objet de classe **AudioClip** obtenu par : son.**play()**.

3 Un petit d'exemple d'application

Afin d'illustrer tout ceci voici un petit programme qui crée une fenêtre avec divers composants. Une boule se dessine dans une zone graphique, lorsque l'on clique dans cette zone la boule se déplace jusqu'au point désigné (un thread est utilisé pour ce faire). Le programme est arrêté par le bouton 'Quitter'. Un menu déroulant permet de changer la couleur du texte éditable et celle de la balle quand la case appropriée est cochée. Un ascenseur permet de modifier la taille de la balle, sa valeur est affiché dans une ligne de texte éditable au dessous. Dans le menu 'Affichage' on dispose d'une rubrique 'Image' qui affiche une photo dans la zone graphique et une rubrique 'Animation' qui retourne au dessin de la balle. Dans le menu '?' on dispose d'une aide générale et d'un sous menu donnant accès à 2 aides spécifiques aux fonctions précédentes. Enfin une liste propose des articles qui seront insérés dans le texte éditable.



Tout d'abord le programme principal (c'est le strict minimum) :

```
// Programme qui cree la fenetre d'interface
class Exemple {
static public void main(String argv[]) {
    FenetreInterface f=new FenetreInterface("Interface");
}
}
```

L'objet que l'on va animer, c'est une balle pleine :

```
import java.awt.*;
import javax.swing.*;

// Définition d'un objet Balle permettant de positionner, dimensionner, placer,
// déplacer, afficher et effacer une balle dans une zone de dessin
class Balle {
public int sommetX, sommetY; //coordonnées de la balle (coin sup gauche du rectangle)
private int diametre; // son diamètre
private boolean active; // indique si la balle est utilisable
boolean enMouvement; /* indique si la balle bouge ou non cet indicateur
    sera utilise par les processus qui animent la balle pour éviter
    qu'elle ne soit modifiée par erreur pendant qu'elle est en mouvement */

public Balle(int x, int y, int r) { // construction de la balle
    activer(); arreter(); // active et immobile
    sommetX=x; sommetY=y; // placement de la balle
    definirTaille(r); // définition de son diamètre
}

public void deplacer(int dx, int dy) { // déplacement de la balle sans affichage
    if (active) {
        sommetX=sommetX+dx; // seulement si la balle est utilisable
        sommetY=sommetY+dy;
    }
}

public void definirTaille(int r) { // définition du diamètre de la balle
    if (active) { diametre=r; } // seulement si la balle est utilisable
}

public void activer() { active=true; } // rendre la balle utilisable
public void desactiver() { active=false; } // rendre la balle inutilisable
public void demarrer() { enMouvement=true; } // la balle est en mouvement
}
```

```

public void arreter() { enMouvement=false; } // la balle est immobile

public void tracer(ZoneGraphique f) { // dessin de la balle
    if (active) { // seulement si la balle est utilisable
        Graphics g=f.obtenirGraphics(); // pour dessiner
        g.setColor(f.getForeground()); // couleur de dessin = couleur de trace
        g.fillOval(sommetX,sommetY,diametre,diametre); // cercle plein
        f.repaint(); // mise a jour de l'écran
    }
}

public void effacer(ZoneGraphique f) {
    if (active) { // seulement si la balle est utilisable
        Graphics g=f.obtenirGraphics(); // pour effacer
        g.setColor(f.getBackground()); // couleur de dessin = couleur de fond
        g.fillOval(sommetX,sommetY,diametre,diametre); // cercle plein
        f.repaint(); // mise a jour de l'écran
    }
}
}

```

Maintenant la fenêtre d'interface (elle utilise la classe zoneGraphique définie en 2.2)

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

import java.applet.*;
import java.net.*;

/* Classe définissant une fenêtre permettant d'illustrer l'utilisation des
différents composants, des événements et de l'utilisation d'un thread */

/***** Définition des composants de l'interface *****/
class FenetreInterface extends JFrame { // la fenêtre principale
    private JMenuBar laBarre; // la barre de menu de cette fenêtre
    private JMenu menuAffiche, menuAide; // les deux rubriques de cette barre
    private JMenuItem image, animation; // les sous-rubriques du menu d'affichage
    private JMenu menuFonctions; // le sous-menu d'aide
    private JMenuItem aPropos, aideAnime, aideImage; // les sous-rubriques du menu d'aide et de son
    sous-menu
    private JComboBox choix; // la zone de choix de couleurs
    private JCheckBox couleurBalle; // le choix des couleurs concerne la balle
    private JTextArea texte; // la zone de texte multi-lignes éditable
    private JLabel titre; // juste un titre pour la liste
    private JList liste; // la liste proposant des mots à insérer dans le texte
    public ZoneGraphique zoneDeDessin; // l'endroit où l'on dessine
    private Balle balleAnimee; // la balle que l'on peut animer
    private Thread anime; // le processus qui fait bouger la balle
    private Image photo=null; // la photo affichée à la place de la balle
    private final int tailleInitiale=20; // taille initiale de la balle
    private JScrollBar ascenseur; // ascenseur permettant de modifier la taille de la balle
    private JTextField afficheur; // zone pour afficher et saisir la taille de la balle
    private JButton quitter; // le bouton pour sortir

    /***** La classe associée aux événements de la fenêtre *****/
    private class GestionFenetre extends WindowAdapter {
        public synchronized void windowClosing(WindowEvent e) { // fermeture de la fenêtre
            System.exit(0); // arrêt du programme
        }
    }

    /***** Les classes associées aux événements des composant de l'interface *****/

    // classe associee aux événements de la rubrique 'Image' du menu 'Affichage'
    private class ActionImage implements ActionListener {
        public synchronized void actionPerformed(ActionEvent e) { // choix de la rubrique
            balleAnimee.desactiver(); // la balle n'est plus utilisable
            // on affiche la photo à la place
            zoneDeDessin.obtenirGraphics().drawImage(photo,2,2,zoneDeDessin);
            zoneDeDessin.repaint(); // mise a jour de l'écran
        }
    }
}

```

```

    }
}

// classe associée aux événements de la rubrique 'Animation' du menu 'Affichage'
//*****
private class ActionAnimation implements ActionListener {
public synchronized void actionPerformed(ActionEvent e) { // choix de la rubrique
    balleAnimee.activer(); // la balle est à nouveau utilisable
    // effacer l'image
    zoneDeDessin.obtenirGraphics().clearRect(0,0,
        zoneDeDessin.getSize().width,zoneDeDessin.getSize().height);
    balleAnimee.tracer(zoneDeDessin); // faire re-apparaître la balle
}
}

// classe associée aux événements de la rubrique 'Aide' du menu '?'
//*****
private class ActionAPropos implements ActionListener {
    // on définit un message afficher
    private final String messageDAide=new String("Ceci est une démonstration d'interface\n pour
illustrer les divers composants\n et leur utilisation");
    public synchronized void actionPerformed(ActionEvent e) { // choix de la rubrique
        texte.setText(messageDAide); // on affiche ce message dans la zone de texte
    }
}

// classe associée aux évts de la rubrique 'Image' du sous-menu 'Fonctions' du menu 'Aide'
//*****
private class ActionAideImage implements ActionListener {
    // on définit un message afficher
    private final String messageDAide=new String("Affiche une photo dans la zone graphique\n");
    public synchronized void actionPerformed(ActionEvent e) { // choix de la rubrique
        texte.setText(messageDAide); // on affiche ce message dans la zone de texte
    }
}

// classe associée aux évts de la rubrique 'Anime' du sous-menu 'Fonctions' du menu 'Aide'
//*****
private class ActionAideAnime implements ActionListener {
    // on définit un message afficher
    private final String messageDAide=new String("Réalise une animation dans la zone
graphique\n");
    public synchronized void actionPerformed(ActionEvent e) { // choix de la rubrique
        texte.setText(messageDAide); // on affiche ce message dans la zone de texte
    }
}

// classe associée aux événements du bouton 'quitter'
//*****
private class ActionQuitter implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) { // clic sur le bouton
        System.exit(0); // arrêt du programme
    }
}

// classe associée aux événements de la zone de choix de couleurs
//*****
private class ActionChoix implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) { // un choix est fait
        String coul=(String)choix.getSelectedIndex(); // le texte qui a été choisi
        Color couleurChoisie=new Color(255,255,255); // on va créer la couleur correspondante
        if (coul.equals("rouge")) couleurChoisie=new Color(255,0,0); // c'est du rouge
        if (coul.equals("bleu")) couleurChoisie=new Color(0,0,255); // c'est du bleu
        if (coul.equals("vert")) couleurChoisie=new Color(0,255,0); // c'est du vert
        if (coul.equals("noir")) couleurChoisie=new Color(0,0,0); // c'est du noir
        texte.setForeground(couleurChoisie); // le texte prend la couleur choisie
        if (couleurBalle.isSelected()) { // la case 'couleur de balle' est cochée
            zoneDeDessin.setForeground(couleurChoisie); // la balle aussi change de couleur
            balleAnimee.tracer(zoneDeDessin); // on la redessine de la nouvelle couleur
        }
    }
}

// classe associée aux événements de modification du texte multi-lignes
//*****
private class ActionTexte implements DocumentListener {
    public synchronized void insertUpdate(DocumentEvent e) { // insertion
        texte.setForeground(new Color(0,100,150)); // on change la couleur de ce texte
    }
    public synchronized void removeUpdate(DocumentEvent e) { // suppression
        texte.setForeground(new Color(0,100,150)); // on change la couleur de ce texte
    }
}

```

```

    }
    public synchronized void changedUpdate(DocumentEvent e) { // modification
        // aucun traitement n'y est associé
    }
}

// classe associée aux événements de sélection d'un élément dans la liste
//*****
private class ActionListe implements ListSelectionListener {
    public synchronized void valueChanged(ListSelectionEvent e) { // sélection d'un élément
        if (!e.getValueIsAdjusting()) { // si la sélection est terminée
            // on insère cet élément dans le texte à la position du curseur
            texte.insert((String)liste.getSelectedValue()+" ",texte.getCaretPosition());
        }
    }
}

// classe associée aux événements de modification de l'affichage de la taille de la balle
//*****
private class ActionAfficheur implements ActionListener {
    public synchronized void actionPerformed(ActionEvent e) { // modification du texte
        int valeur; // la future taille de la balle
        try {valeur=Integer.parseInt(afficheur.getText());} // conversion de la valeur saisie
        catch (NumberFormatException enf) {valeur=tailleInitiale;} // en cas d'erreur
        ascenseur.setValue(valeur); // l'ascenseur est ajusté pour correspondre à cette valeur
        balleAnimee.effacer(zoneDeDessin); // on efface la balle actuelle
        balleAnimee.definirTaille(valeur); // on en modifie la taille
        balleAnimee.tracer(zoneDeDessin); // on la re-dessine
    }
}

// classe associée aux événements de modification de la position de l'ascenseur
//*****
private class ActionAscenseur implements AdjustmentListener {
    public synchronized void adjustmentValueChanged(AdjustmentEvent e) { // modification
        // affichage de la nouvelle valeur dans la zone de texte mono-ligne associée
        afficheur.setText(String.valueOf(ascenseur.getValue()));
        balleAnimee.effacer(zoneDeDessin); // on efface la balle actuelle
        balleAnimee.definirTaille(ascenseur.getValue()); // on en modifie la taille
        balleAnimee.tracer(zoneDeDessin); // on la re-dessine
    }
}

// Classe associée aux événements souris dans la fenêtre de dessin
//*****
private class ClicDeSouris extends MouseAdapter {
    public synchronized void mouseEntered(MouseEvent e) {
        // un son est joué lors de l'entrée de la souris dans la zone de dessin
        URL fichierSon=null;
        try { fichierSon=new URL("file:ding.wav"); } // accès au fichier
        catch (MalformedURLException mue) {System.out.println("url incorrecte"); }
        AudioClip son = Applet.newAudioClip(fichierSon); // créer un clip audio
        son.play(); // le jouer
        e.consume(); /* l'événement a été pris en compte */
    }
    public synchronized void mousePressed(MouseEvent e) { // bouton appuyé
        if (! balleAnimee.enMouvement) { // si la balle est actuellement arrêtée
            // on crée un processus (thread) qui va l'animer
            anime=new Thread(new BougeBalle(zoneDeDessin, balleAnimee, e.getX(), e.getY()));
            anime.start(); // on lance ce processus d'animation
        }
        e.consume(); /* l'événement a été pris en compte */
    }
}

//*****
//**** Le constructeur de la fenêtre qui prépare les composants et les place ****/
//*****
public FenetreInterface(String nomFenetre) {
    super("Je m'appelle : "+nomFenetre); // création de la fenêtre avec son nom
    setSize(500,420); // dimensionnement de cette fenêtre
    addWindowListener(new GestionFenetre()); // traitements des événements de la fenêtre

    // Ajout de la barre de menu et association des actions rubriques
    menuAffiche = new JMenu("Affichage"); // rubrique des fonctions d'affichage
    image = new JMenuItem("Image"); // affichage d'une image
    animation = new JMenuItem("Animation"); // affichage d'une animation
    menuAffiche.add(image); menuAffiche.add(animation); // ajoutées à la rubrique
    image.addActionListener(new ActionImage()); // traitement lié à 'Image'
    animation.addActionListener(new ActionAnimation()); // traitement lié à 'Animation'
    menuAide = new JMenu("?"); // rubrique d'aide

```

```

aPropos = new JMenuItem("A propos"); // fonction d'aide générale
aPropos.addActionListener(new ActionAPropos()); // traitement associé à l'aide générale
menuAide.add(aPropos); // ajoutée à la rubrique d'aide
menuFonctions = new JMenu("Fonctions"); // sous-menu d'aide sur les fonctions
aideImage = new JMenuItem("Image"); // sa rubrique sur l'image
aideImage.addActionListener(new ActionAideImage()); // traitement associé à l'aide image
aideAnime = new JMenuItem("Animation"); // sa rubrique sur l'animation
aideAnime.addActionListener(new ActionAideAnime()); // traitement associé à l'aide animation
menuFonctions.add(aideImage); menuFonctions.add(aideAnime); // ajoutées au sous-menu
menuAide.add(menuFonctions); // ajout du sous menu d'aide à la rubrique d'aide
laBarre = new JMenuBar(); // barre de menu
laBarre.add(menuAffiche); laBarre.add(menuAide); // ajout des rubriques
setJMenuBar(laBarre); // ajout de la barre dans la fenêtre

// définition des composants d'interface et association des actions
quitter=new JButton("Quitter"); // le bouton 'Quitter'
quitter.addActionListener(new ActionQuitter()); // traitement associé
choix = new JComboBox(); // la zone de choix
choix.addItem("bleu");choix.addItem("noir"); // les choix proposés
choix.addItem("rouge"); choix.addItem("vert");
choix.addActionListener(new ActionChoix()); // le traitement associé
couleurBalle=new JCheckBox("Couleur de balle",false);
couleurBalle.setToolTipText("Applique le choix de couleur à la balle"); // bulle d'aide
texte=new JTextArea("texte initial\n",10,20); // la zone de texte multi-lignes
texte.setLineWrap(true); texte.setWrapStyleWord(true); // repliement en frontière de mot
texte.getDocument().addDocumentListener(new ActionTexte()); // le traitement associé
titre=new JLabel("Les articles"); // le titre pour la liste (pas de traitement)
String[] articles = {"le","la","un","une","les"}; // les éléments de la liste
liste = new JList(articles); // la liste contenant ces éléments
liste.setVisibleRowCount(2); // seulement 2 lignes visibles à la fois
liste.addListSelectionListener(new ActionListe()); // le traitement associé
ascenseur=new JScrollBar(JScrollBar.HORIZONTAL,tailleInitiale,5,0,100); // l'ascenseur
ascenseur.addAdjustmentListener(new ActionAscenseur()); // le traitement associé
afficheur=new JTextField(String.valueOf(tailleInitiale),2); // la zone de texte mono-ligne
afficheur.setFont(new Font("Times",Font.BOLD,20)); // choix de la police pour ce texte
afficheur.setColumns(2); // le texte est sur 2 caractères
afficheur.addActionListener(new ActionAfficheur()); // traitement associé
zoneDeDessin=new ZoneGraphique(); // la zone de dessin en double buffer
zoneDeDessin.setForeground(new Color(0,0,250)); // sa couleur de trace
zoneDeDessin.addMouseListener(new ClicDeSouris()); // les actions souris associées

// définition des objets utilisés pour placer les composants
GridBagLayout placeur=new GridBagLayout(); // objet de placement
GridBagConstraints contraintes=new GridBagConstraints(); //objet de définition des contraintes
getContentPane().setLayout(placeur); // utiliser cet objet de placement pour la fenêtre
contraintes.fill=GridBagConstraints.BOTH; // remplissage complet des cases
contraintes.anchor=GridBagConstraints.CENTER; // centrage des composant
contraintes.insets=new Insets(3,3,0,0); // marges externes
contraintes.ipadx=2; contraintes.ipady=2; // marges internes

// Placement des composants
// le bouton quitter et la case à cocher
JSplitPane partage=new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,couleurBalle,quitter);
contraintes.gridx=2; contraintes.gridy=3; // coordonnées (2,3)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case
contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(partage, contraintes);
getContentPane().add(partage); // placement du bouton et de la case à cocher
// la liste de choix de couleurs
contraintes.gridx=0; contraintes.gridy=0; // coordonnées (0,0)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case
contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(choix, contraintes);
getContentPane().add(choix); // placement du choix
// La zone d'édition de texte
JScrollPane defileTexte = new JScrollPane(texte); // ascenseurs pour le texte
contraintes.gridx=1; contraintes.gridy=0; // coordonnées (1,0)
contraintes.gridwidth=2; contraintes.gridheight=2; // 4 cases
contraintes.weightx=0; contraintes.weighty=100;
placeur.setConstraints(defileTexte, contraintes);
getContentPane().add(defileTexte); // placement du texte + ascenseurs
// Le titre de la liste
contraintes.gridx=0; contraintes.gridy=2; // coordonnées (0,2)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case
contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(titre, contraintes);
getContentPane().add(titre); // placement du titre
// la liste de mots
JScrollPane defileListe = new JScrollPane(liste); // ascenseurs pour la liste
contraintes.gridx=0; contraintes.gridy=3; // coordonnées (0,3)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case

```

```

contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(defileListe, contraintes);
getContentPane().add(defileListe); // placement de la liste + ascenseurs
// l'ascenseur de taille de balle
contraintes.gridx=1; contraintes.gridy=2; // coordonnées (1,2)
contraintes.gridwidth=2; contraintes.gridheight=1; // 2 cases
contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(ascenseur, contraintes);
getContentPane().add(ascenseur); // placement de l'ascenseur
// l'affichage/modification de taille de balle
afficheur.setMinimumSize(afficheur.getPreferredSize());
contraintes.gridx=1; contraintes.gridy=3; // coordonnées (1,3)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case
contraintes.weightx=0; contraintes.weighty=0;
placeur.setConstraints(afficheur, contraintes);
getContentPane().add(afficheur); // placement de l'afficheur
// la zone graphique
zoneDeDessin.setPreferredSize(new Dimension(300,200));
contraintes.gridx=0; contraintes.gridy=1; // coordonnées (0,1)
contraintes.gridwidth=1; contraintes.gridheight=1; // 1 case
contraintes.weightx=100; contraintes.weighty=100;
placeur.setConstraints(zoneDeDessin, contraintes);
getContentPane().add(zoneDeDessin); // placement de la zone de dessin

photo=getToolkit().getImage("Planetes.jpg"); // image qui remplace la balle
prepareImage(photo,this); // chargement de cette image

pack(); // mise en place des composants
setVisible(true); // rendre visible la fenêtre principale

balleAnimee=new Balle(10,15,tailleInitiale); // création de la balle
balleAnimee.tracer(zoneDeDessin); // elle apparaît
}
}

```

Et enfin le thread d'animation de la balle :

```

import javax.swing.*;

class BougeBalle extends Thread { // thread d'animation de la balle

    private ZoneGraphique zone; // composant de dessin de l'interface
    private Balle balle; // la balle à animer
    private int jusquax, jusquay; // point d'arrivée de la balle

    public BougeBalle(ZoneGraphique fenetre, Balle balleAnimee, int jX, int jY) {
        zone=fenetre; // le constructeur recupere le composant de dessin
        balle=balleAnimee; // l'objet à animer
        jusquax=jX; jusquay=jY; // et les coordonnées de son point d'arrivée
    }

    public void run() {
        balle.demarrer(); // la balle est en mouvement
        int pas; // on va la déplacer par pas successifs d'abord horizontalement puis verticalement
        // Le programme de dessin commence ici
        if (balle.sommetX<jusquax) pas=1; else pas=-1; // sens de déplacement horizontal
        for (int i=balle.sommetX; i!=jusquax; i=i+pas) { // jusqu'à la verticale du point d'arrivée
            balle effacer(zone); // on efface la balle pour pouvoir la redessiner plus loin
            balle.deplacer(pas, 0); // nouvelle position de la balle
            balle.tracer(zone); // on trace la balle dans cette nouvelle position
            // Petit delai pour que l'animation ne soit pas trop rapide
            try {sleep(10);} catch (InterruptedException e) {};
        }
        if (balle.sommetY<jusquay) pas=1; else pas=-1; // sens de déplacement vertical
        for (int j=balle.sommetY; j!=jusquay; j=j+pas) { // jusqu'au point d'arrivée
            balle effacer(zone); // on efface la balle pour pouvoir la redessiner plus loin
            balle.deplacer(0, pas); // nouvelle position de la balle
            balle.tracer(zone); // on trace la balle dans cette nouvelle position
            // Petit delai pour que l'animation ne soit pas trop rapide
            try {sleep(10);} catch (InterruptedException e) {};
        }
        balle.arreter(); // la balle est maintenant immobile
    }
}

```

4 Les Beans

4.1 Principe : ce sont des composants qui peuvent être vus par un éditeur externe (par exemple la beanbox de java) et qui peuvent communiquer par événements avec d'autres objets pour leur signaler des modifications de propriétés (membres). Ces objets peuvent alors prendre en compte ces événements et, dans certains cas, opposer un veto à la modification.

4.2 Création d'un bean :

- Il faut que la classe soit publique
- Il faut un constructeur sans paramètres
- A chaque propriété (membre) que l'on veut rendre accessible par un éditeur extérieur il faut associer des méthodes getxxx et setxxx pour en récupérer et modifier la valeur (xxx est le nom de la propriété). De plus cette propriété doit avoir été créée et initialisée dans le constructeur.

4.3 Compilation d'un bean :

Créer un fichier manifeste (fichier texte) contenant :

Name: NomDeLaClasse.class

Java-Bean: True

Puis générer une archive par la commande :

jar cfm nomDeLArchive.jar nomDuFichierManifeste.txt NomDeLaClasse.class

4.4 Gestion des événements simples dans un bean :

Un bean peut signaler certains événements à d'autres objets (des modifications d'état d'une propriété bean). Pour cela il faut :

- Que les objets se soient enregistrés auprès du bean
- Que le bean signale explicitement les événements

4.4.1 Pour autoriser l'enregistrement :

- On ajoute aux propriétés (membres) du bean un objet de classe **PropertyChangeSupport** initialisé à **null**.
- On crée cet objet dans le constructeur par **new PropertyChangeSupport(this);**
- On ajoute au bean les méthodes permettant à d'autres objets de s'enregistrer auprès de lui :

```
public void addPropertyChangeListener(PropertyChangeListener p) {  
    if (nomDuMembrePropertyChangeSupport != null) {  
        nomDuMembrePropertyChangeSupport.addPropertyChangeListener(p);  
    }  
}  
et  
public void removePropertyChangeListener(PropertyChangeListener p) {  
    nomDuMembrePropertyChangeSupport.removePropertyChangeListener(p); }  
}
```

4.4.2 Pour signaler un événement :

Lorsque l'on souhaite signaler un événement il suffit de faire :

nomDuMembrePropertyChangeSupport.firePropertyChange(info,ancien,nouveau);

où info est une chaîne (classe String) de caractères décrivant l'événement
ancien est la valeur de la propriété avant modification
nouveau est valeur de la propriété après modification

4.4.3 Les objets qui sont informés de ces événements simples :

- Ils doivent implémenter l'interface **PropertyChangeListener**
- Ils s'enregistrent auprès du bean par la méthode **addPropertyChangeListener** de celui-ci
- Ils surchargent la méthode : **public void PropertyChange(PropertyChangeEvent)** pour y mettre ce qu'ils font quand l'événement se produit.

La classe **PropertyChangeEvent** :

Elle offre les méthodes :

Object getNewValue() qui retourne la nouvelle valeur de la propriété (3^{ème} paramètre de **firePropertyChange**)

Object getOldValue() qui retourne l'ancienne valeur de la propriété (2^{ème} paramètre de **firePropertyChange**)

String getPropertyNames() qui retourne la chaîne décrivant l'événement (1^{er} paramètre de **firePropertyChange**)
Object getSource() qui retourne l'objet qui a créé l'événement

4.5 Gestion des événements auxquels on peut s'opposer dans un bean :

Un bean peut signaler certains événements à d'autres objets (des modifications d'état d'une propriété bean) et vérifier que ces objets ne s'y opposent pas. Pour cela il faut :

- Que les objets se soient enregistrés auprès du bean
- Que le bean signale explicitement les événements

4.5.1 Pour autoriser l'enregistrement :

- On ajoute aux propriétés (membres) du bean un objet de classe **VetoableChangeSupport**
- On crée cet objet dans le constructeur par **new VetoableChangeSupport(this);**
- On ajoute au bean les méthodes permettant à d'autres objets de s'enregistrer auprès de lui :

```
public void addVetoableChangeListener(VetoableChangeListener v) {  
    nomDuMembreVetoableChangeSupport.addVetoableChangeListener(v); }  
et  
public void removeVetoableChangeListener(VetoableChangeListener v) {  
    nomDuMembreVetoableChangeSupport.removeVetoableChangeListener(v); }
```

4.5.2 Pour signaler un événement :

Lorsque l'on souhaite signaler un événement il suffit de faire :

nomDuMembreVetoableChangeSupport.fireVetoableChange(info,ancien,nouveau);

où info est une chaîne de caractères (classe **String**) décrivant l'événement

ancien est la valeur de la propriété avant modification

nouveau est valeur de la propriété après modification

Cette méthode peut lever une exception de classe **PropertyVetoException**. Cette exception sera levée si l'un des objets enregistré auprès de lui s'oppose à cette modification.

4.5.3 Les objets qui sont informés de ces événements et peuvent s'y opposer :

- Ils doivent implémenter l'interface **VetoableChangeListener**
- Ils s'enregistrent auprès du bean par la méthode **addVetoableChangeListener** de celui-ci
- Ils surchargent la méthode :

```
public void VetoableChange(PropertyChangeEvent) throws PropertyVetoException
```

 pour y mettre ce qu'ils font quand l'événement se produit. Pour lever une exception et donc opposer leur veto il suffit de faire : **throw(new PropertyVetoException(message, evenement));**
où message est une chaîne de caractères (classe **String**) expliquant les raisons du veto
evenement est un objet de classe **PropertyChangeEvent** (en général celui que l'on a reçu en paramètre)

SOMMAIRE

1 Ecriture d'une application	1
2 Interfaces graphiques avec SWING	1
2.1 Réalisation d'une interface graphique	1
2.1.1 Les classes Component et JComponent	1
2.1.2 Les composants de l'interface	2
2.1.2.1 La classe JButton	3
2.1.2.2 La classe JCheckBox	3
2.1.2.3 La classe JLabel	3
2.1.2.4 La classe JComboBox	3
2.1.2.5 La classe JList	4
2.1.2.6 La classe JScrollBar	4
2.1.2.7 La classe JSlider	4
2.1.2.8 La classe JProgressBar	5
2.1.2.9 Les classes JTextField et JTextArea	5
2.1.2.10 La classe JMenuBar	6
2.1.2.11 La classe JFileChooser	7
2.1.3 Placement des objets	7
2.1.3.1 Définition de l'interface	8
2.1.3.2 La classe JFrame	8
2.1.3.3 La classe JDialog	9
2.1.3.4 Les contenants	9
2.1.3.5 Les objets de placement	12
2.1.4 Exemple de placement de composants dans une interface graphique :	14
2.1.6 Traitement des événements	16
2.1.6.1 Les événements élémentaires	16
2.1.6.2 Les événements concernant les fenêtres	17
2.1.6.3 Les événements de composants d'interface	18
2.1.6.4 Associer un contrôleur à un événement	19
2.2 Le graphique	20
2.3 La classe ImageIcon	21
2.4 Le son	21
3 Un petit d'exemple d'application	21
4 Les Beans	27
4.2 Création d'un bean :	28
4.3 Compilation d'un bean :	28
4.4 Gestion des événements simples dans un bean :	28
4.4.1 Pour autoriser l'enregistrement :	28
4.4.2 Pour signaler un événement :	28
4.4.3 Les objets qui sont informés de ces événements simples :	28
4.5 Gestion des événements auxquels on peut s'opposer dans un bean :	29
4.5.1 Pour autoriser l'enregistrement :	29
4.5.2 Pour signaler un événement :	29
4.5.3 Les objets qui sont informés de ces événements et peuvent s'y opposer :	29