APPLICATIONS EN JAVA EN UTILISANT SWING

1 Ecriture d'une application

Une application est un programme qui peut fonctionner en autonome (sans devoir être intégrée dans une page HTML). Elle 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, parfois, **javax.swing.event.*** en début de programme.

2.1 Réalisation d'une interface graphique

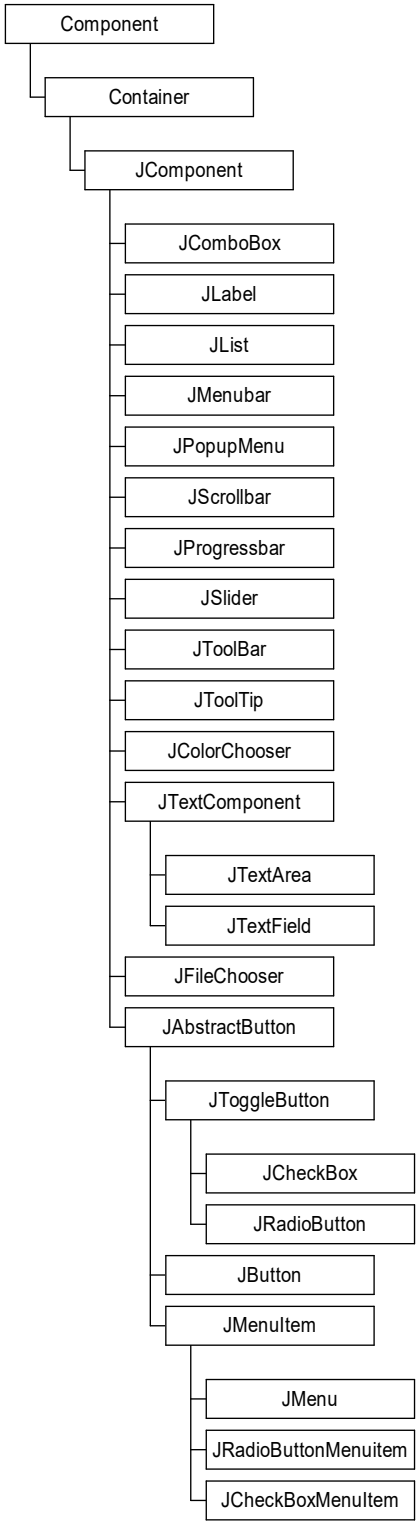
Il faut tout d'abord définir un contenant (**Container**) de cette interface graphique. On choisit en général de personnaliser par héritage la classe **JFrame** qui correspond à une fenêtre de type Windows ou Xwindows. Il est à remarquer que lorsque l'on écrit des applets plutôt que des applications 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 on 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 getPreferedSize() 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)
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**.

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 créer 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)
JTextField : une zone de texte éditable d'une ligne
JTextArea : une zone de texte éditable multi lignes
JCheckBox : une case à cocher
JComboBox : une liste déroulante
JList : une liste dans laquelle on peut sélectionner un ou plusieurs éléments par double clic
JScrollBar : un ascenseur horizontal ou vertical
JMenubar : une barre de menu
JFileChooser : une fenêtre de dialogue permettant de choisir un fichier
JSlider : un curseur linéaire
JProgressBar : 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 icone 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 seIcon(ImageIcon) qui ajoute ou change l'image contenue 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 ou des images fixes. Ses principales méthodes sont :

JLabel(String) qui construit le JLabel avec un contenu textuel
JLabel(ImageIcon) qui construit le JLabel avec une image
void setText(String) qui (re)définit le texte
String getText() qui retourne le texte
void setIcon(ImageIcon) qui (re)définit l'image

2.1.2.4 La classe JComboBox

Elle définit 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 selection 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 selectionné. 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 annulle toutes les sélections
int getModel().getSize() retourne le nombre d'éléments dans la liste
Object getModel().getElementAt(int) retourne l'objet de la liste correspondant au rang donné en paramètre
void addListSelectionListener(ListSelectionListener) pour associer l'objet qui traitera les sélections dans la liste (voir 2.1.6.3)

Remarque : le constructeur de JList avec un tableau d'objets permet de créer une liste initialisée mais pas de la modifier dynamiquement. Pour créer une liste modifiable dynamiquement (ajout/suppression d'éléments) il faut lui associer un contenu sous forme d'un modèle :

```
DefaultListModel contenu = new DefaultListModel ();
JList listeDynamique = new JList(contenu);
```

On pourra alors ajouter des éléments à la liste en les ajoutant au modèle par :

```
contenu.addElement(Object)
```

ou `contenu.insertElementAt(Object,int)`

Et en enlever par :

```
contenu.removeElementAt(int)
```

ou `contenu.removeElement(Object)`

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 positon 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 un graduation plus longue.
void setMinorTickSpacing(int) qui détermine le nombre d'unités correspondant à un 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 paramètres 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. La classe **Hastable** (qui ressemble à **HashMap** peut être utilisée comme paramètre pour définir des étiquettes personnalisées)
void setPaintLabels(boolean) qui détermine si les valeurs correpondant aux graduations longues sont ou non écrites.
int getValue() qui retourne la positon 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 positon 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, JTextArea et JTextPane

Elles permettent de définir des zones de texte éditable 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 (voir ci-dessous). 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, elle est obtenue par la méthode *getDocument*. Ses principales méthodes sont :

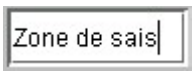
int getLength() qui retourne le nombre de caractères du document
String getText(int,int) qui retourne la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre
void removeText(int,int) qui supprime la partie du texte qui commence à la position donnée en premier paramètre et dont la longueur est donnée par le second paramètre

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

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

La classe JTextField

Elle permet de définir des zones de texte modifiables sur une seule ligne. 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 colones
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.**setPreferedSize(**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 :

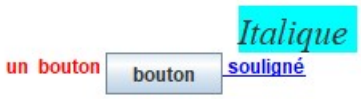
JTextArea(String) qui crée une zone de texte avec un contenu initial
JTextArea(String, int, int) qui crée une zone de texte avec un contenu initial et précise le nombre de lignes et de colonnes de la zone de texte
void append(String) qui ajoute la chaîne à la fin du texte affiché
void insert(String, 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.

La classe JTextPane

Elle permet de définir des zones de texte formaté 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 :



JTextPane() qui crée une zone de texte sans contenu initial.
StyledDocument getStyledDocument() renvoie le document contenu dans le **JTextPane**.
insertComponent(Component) insère un composant à l'emplacement du curseur (voir *setCaretPosition* en 2.1.2.9). Le composant peut être tout objet de classe **Component** ou dérivée (bouton, case à cocher …). Il est considéré comme un seul caractère du texte.
insertIcon(Icon) insère une image à l'emplacement du curseur (voir *setCaretPosition* en 2.1.2.9). L'image est considérée comme un seul caractère du texte.
void setContentType(String) définit le type mime du contenu par exemple : « text/plain » ou « text/html ». Par défaut le type mime est « text/plain ».
void setPage(String) affiche dans le **JTextPane** la page html désignée par l'url contenu dans le paramètre. Cette méthode peut lever une **IOException** si l'url désignée est incorrecte ou inaccessible.

La classe **StyledDocument** permet de gerer le contenu du **JTextPane** (voir *getStyledDocument* ci-dessus) ses principales méthodes sont :
Style addStyle(String, Style) qui permet de créer un style de texte ou de paragraphe, le 1er paramètre est le nom du style ajouté, le second est celui du style dont il hérite. Quand un style hérite d'un autre il en a toutes les propriétés mais si certaines sont redéfinies elles surchargent celles du style parent. Cette méthode renvoie le nouveau style créé. Si le second paramètre est **null**, le style défini n'hérite de rien. En général après avoir créé un nouveau style on en définit les propriétés grâce à la classe **StyleConstants** (voir ci-dessous).
Style getStyle(String) renvoie le style créé pour ce **JTextPane** (voir addStyle) désigné par son nom
void insertString(int, String, Style) insère le texte donnée en 2ème paramètre à la position donnée en 1er paramètre en lui applicant le style donné en dernier paramètre. Le dernier paramètre peut être obtenu par la méthode *getStyle*.
void setParagraphAttributes(int, int, Style, boolean) définit le style d'un paragraphe (en général son alignement). Ce style est appliqué au texte se situant entre la caratère désigné par le 1er paramètre et celui désigné par le 2ème paramètre. Le style appliqué peut être obtenu par la méthode *getStyle*. Le dernier paramètre indique si ce style vient remplacer (**true**) ou s'ajouter (**false**) au style déjà appliqué à cette zone du texte.

La classe **StyleConstants** permet de définir des styles en utilisant les méthodes statiques suivantes qui reçoivent en 1er paramètre l'objet de classe **Style** à définir :
void StyleConstants.setAlignment(Style, int) définit un alignement de paragraphe. Le second paramètre peut prendre les valeurs **StyleConstants.CENTER**, **StyleConstants.JUSTIFIED**, **StyleConstants.RIGHT** ou **StyleConstants.LEFT.**
void StyleConstants.setBackground(Style, Color) définit la couleur du fond du texte
void StyleConstants.setForeground(Style, Color) définit la couleur du texte lui même
void StyleConstants.setBold(Style, boolean) définit si le texte est en gras (**true**) ou pas (**false**)
void StyleConstants.setItalic(Style, boolean) définit si le texte est en italiques (**true**) ou pas (**false**)
void StyleConstants.setUnderline(Style, boolean) définit si le texte est souligné (**true**) ou pas (**false**)
void StyleConstants.setSubscript(Style, boolean) définit si le texte est en indice (**true**) ou pas (**false**)
void StyleConstants.setSuperScript(Style, boolean) définit si le texte est en exposant (**true**) ou pas (**false**)
void StyleConstants.setFontFamily(Style, String) définit la police du texte
void StyleConstants.setFontSize(Style, int) définit la taille de la police du texte

Exemple d'utilisation de JTextPane :

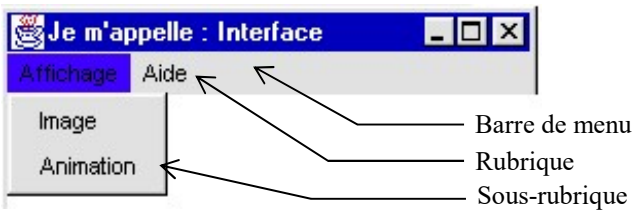
Le code suivant permet de définir un JTextPane contenant du texte et un bouton (identique à celui présenté sur l'image ci-dessus) :

```
texte = new JTextPane();
// Document affiché et styles créés
StyledDocument doc = texte.getStyledDocument();
Style styleTexte, styleParagraphe; // Styles créés

// Préparation des styles utilisés :
// Texte en italiques sur fond cyan en police 22
styleTexte = doc.addStyle("italBleu", null);
StyleConstants.setItalic(styleTexte, true);
StyleConstants.setBackground(styleTexte, Color.CYAN);
StyleConstants.setFontFamily(styleTexte, "Serif");
StyleConstants.setFontSize(styleTexte, 22);
// Paragraphe aligné à droite
styleParagraphe = doc.addStyle("droite", null);
StyleConstants.setAlignment(styleParagraphe, StyleConstants.ALIGN_RIGHT);
// Texte en gras et rouge
styleTexte = doc.addStyle("grasrouge", null);
StyleConstants.setBold(styleTexte, true);
StyleConstants.setForeground(styleTexte, Color.red);
// Paragraphe centré
styleParagraphe = doc.addStyle("centre", null);
StyleConstants.setAlignment(styleParagraphe, StyleConstants.ALIGN_CENTER);
// Texte en gras bleu et souligné par héritage du précédent
//  avec ajout de souligné et surcharge de la couleur
styleTexte = doc.addStyle("grassoulbleu", doc.getStyle("grasrouge"));
StyleConstants.setUnderline(styleTexte, true);
StyleConstants.setForeground(styleTexte, Color.blue);
// Ajout de textes et composants
int taille = 0;
try {
   // Ajout 1er texte style 'italiques' et cadré à droite
   doc.insertString(taille, "Italique \n", doc.getStyle("italBleu"));
   doc.setParagraphAttributes(taille, doc.getLength(), doc.getStyle("droite"), false);
   taille = doc.getLength();
   // Ajout 2ème texte style 'grasrouge' et centré
   doc.insertString(taille, "un  bouton ", doc.getStyle("grasrouge"));
   doc.setParagraphAttributes(taille, doc.getLength(), doc.getStyle("centre"), false);
   taille = doc.getLength();
   // Insertion d'un composant
   JButton bout = new JButton("bouton");
   texte.setCaretPosition(taille); // position bouton à la fin du
   texte.insertComponent(bout);
   taille = doc.getLength(); // Le composant compte comme un caractère
   // Ajout 3ème texte style 'grassoulbleu' après le bouton sans
   //  changement du style de paragraphe
   doc.insertString(taille, " souligné\n",doc.getStyle("grassoulbleu"));
   taille = doc.getLength();
}
catch (BadLocationException e) {
   System.err.println("Insertion hors du texte");
}
```

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 les éléments d'une rubrique peuvent être eux mêmes des rubriques.

*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.

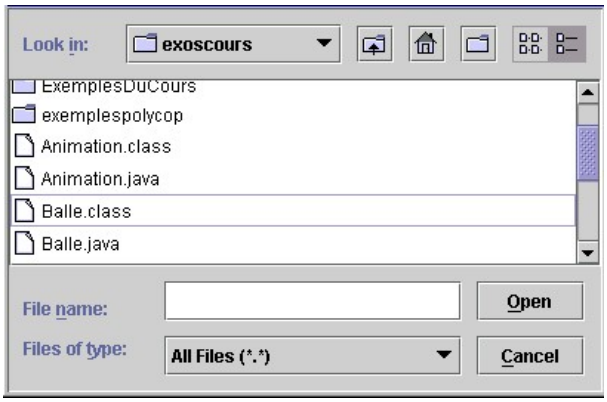
void setEnabled(boolean) valide ou invalide la possibilité d'utiliser la sous-rubrique.

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. Le 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. Par défaut **FILES_AND_DIRECTORIES** est choisi.

void setMultipleSelectionEnabled(boolean) qui autorise ou interdit les sélections multiples (interdit par défaut).

int showOpenDialog(Component) fait apparaître la fenêtre de choix de fichier et attend qu'elle soit fermée. La valeur de retour est : **JFileChooser.APPROVE_OPTION** si un choix a été fait, **JFileChooser.CANCEL_OPTION** si aucun choix n'a été fait et **JFileChooser.ERROR_OPTION** en cas d'erreur. Le paramètre est la fenêtre qui fait apparaître le FileChooser.

void setFileFilter(FileFilter) permet de définir un filtre de sélection de fichiers. Le paramètre peut être new **FileNameExtensionFilter**(description, extension …) pour filtrer les fichiers proposés par extension. Le 1er paramètre est une chaîne de caractères décrivant la sélection (par exemple "Fichiers texte"), les suivants sont les extensions par lesquelles les fichiers sont filtrés (par exemple "txt", "doc", "docx").

Exemple :

```
JFileChooser choix = new JFileChooser(".") ;
choix.setFileFilter(new FileNameExtensionFilter("Fichiers texte", "txt")) ;
int codeRetour = choix.showOpenDialog(this) ;
if (codeRetour = JFileChooser.APPROVE_OPTION) {
   File choisi = choix.getSelectedFile();
   …
}
```

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 éventuellement 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 et à la fenêtre elle même 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 …)

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

2.1.3.3 La classe JDialog

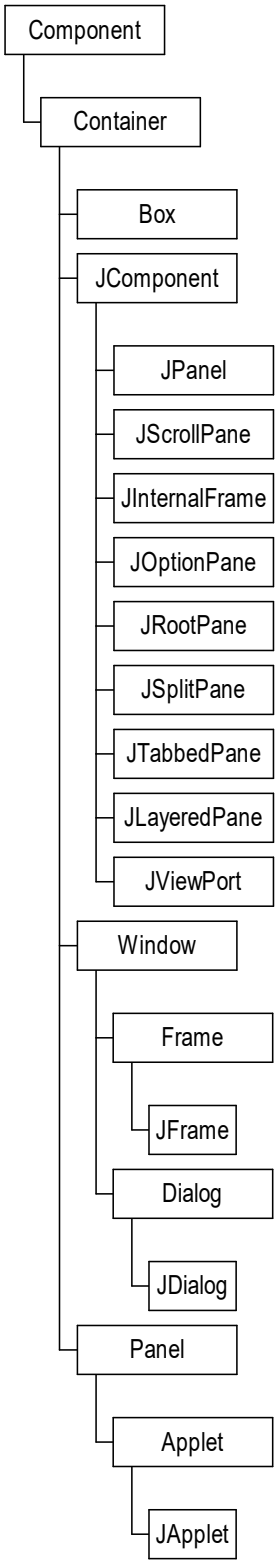
Elle est utilisée pour faire apparaître des fenêtres temporaires permettant d'afficher un message ou de faire une saisie. Il est possible de rendre ces fenêtres prioritaires dans le sens où on ne peut rien faire d'autre tant qu'on ne les a pas fermées. Ceci permet d'obliger la lecture d'un message ou 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 de dialogue 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 on 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 …)

Attention : Ce n'est pas directement dans un objet de classe **JDialog** que l'on place les composants mais dans le 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'autre 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 coté 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 paramètres 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/ou 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 highesterLayer() 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 cote à cote 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 à1.0.

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

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)

Remarque : les méthodes **removeTabAt**, **setIconAt**, **setSelectedIndex**, **getTitleAt**, **isEnabledAt**, **setEnabledAt**, **setBackgroundAt** et **setForegroundAt** peuvent lever une exception de classe **ArrayIndexOutOfBoundsException** si la position de l'onglet n'est pas correcte.

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 1er 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 et à la fenêtre elle même 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, **s**i 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.
Remarque : lors d'un redimensionnement de la fenêtre c'est la zone CENTER qui est redimensionnée.

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. Le troisième paramètre donne l'espacement horizontal entre composants (en pixels). Le dernier paramètre donne l'espacement vertical entre composants (en pixels).
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 les tailles des lignes et des colonnes sont ajustées 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 colonne.
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.**

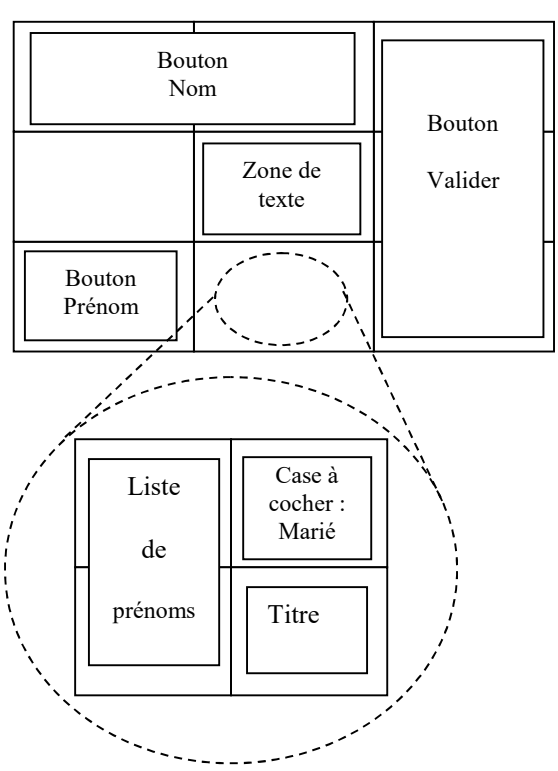
insets pour définir les marges externes du composant. Ce champ est positionné en faisant **new Insets(h,g,b,d)** où h définit la marge haute, b la marge basse, g la marge gauche et d la marge droite.

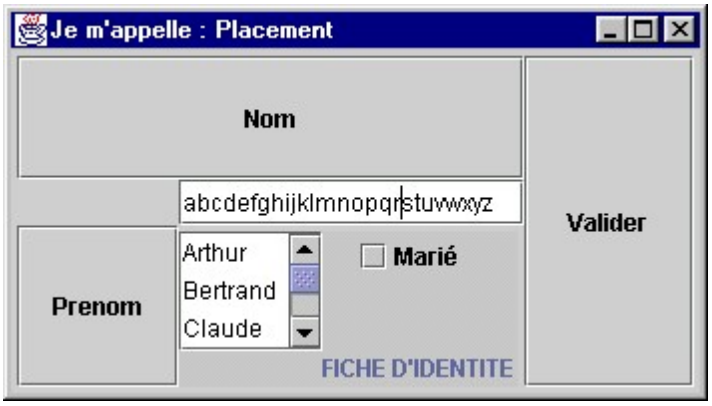
weigthx et **weighty** permettent d'indiquer quelle part de surface est distribuée ou retirée à ce composant lors d'un redimensionnement de la fenêtre. Il expriment un pourcentage de répartition de largeur (respectivement hauteur) pour ce composant.

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	Redimension-nement
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	Redimension-nement
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 prenoms;        // 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"};
        prenoms=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
        Panel caseDuBas=new Panel(panelPlaceur); //création du Panel avec son objet de placemt
        // placement des 3 composants dans le Panel
                // placement de la liste dans un JscrollPane pour avoir des ascenseurs
        prenoms.setVisibleRowCount(3); // 3 lignes visibles dans la liste
        JScrollPane defile=new JScrollPane(prenoms);
        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 associées 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** **FocusListener**	**FocusEvent**	**focusGained**
	devient non visible			**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**	**MouseEvent**	**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 getButton() qui retourne le numéro du bouton dont l'état a changé
int getClickCount() qui retourne le nombre de clics effectués

La méthode **void consume()** peut être 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 associées 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 ne 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**
Elément	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 :

Evénement / Composant	Action	Ajustement	Changement	Elément	Document	Curseur	Sélection
JButton	**X**		X	X			
JCheckBox	**X**		X	X			
JComboBox				**X**			
JList							**X**
JSrollBar		**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 associées 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**	**adjustmentValue Changed**
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 **JLabel**. 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();
```

Cet objet est en réalité de classe **Graphics2D** mais il faut utiliser la coercition (cast) pour le préciser :

```
Graphics2D zoneDeDessin2D = (Graphics2D)nomDuComposant.getGraphics();
```

Chaque fois que le composant devra être dessiné ou redessiné à l'écran java fera appel à sa méthode **paintComponent**. Il est possible de surcharger, dans une classe dérivée de celle du composant, la méthode **paintComponent** pour les doter d'un comportement particulier. Elle est définie comme suit :

```
public void paintComponent (Graphics g)
```

Lorsque l'on désire rafraîchir l'affichage d'un composant en dehors des instants où java le fait, on peut appeler sa méthode **repaint()**.

2.3 La classe ImageIcon

Elle permet de définir des images 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 .png** ou **.jpg** dont on passe le nom en paramètre.

ImageIcon(URL) qui construit l'icône à partir d'un fichier **.gif .png** 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 de 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** de 2 façons :

1. Par la classe **AudioClip** de **Applet** de la bibliothèque **java.applet** :
Créer l' audio clip :

```
AudioClip son = Applet.newAudioClip(new File("nom").toURI().toURL());
```

Jouer le son :

```
son.play() ;
```

ou

```
son.loop() ;
```

Arrêter le son :

```
monClip.stop();
```

2. L'autre façon d'utiliser des sons en java est d'utiliser les classes **Clip** et **AudioSystem** de la bibliothèque **javax.sound.sampled** :
Créer le clip :

```
Clip monClip = AudioSystem.getClip();
monClip.open(AudioSystem.getAudioInputStream(new File("nom")));
```

Jouer le son :

```
monClip.start();
```

Arrêter le son :

```
monClip.stop();
```

3 Un petit d'exemple d'application

Afin d'illustrer tout ceci voici une petite application en java qui :

- Crée une interface permettant d'onvrir un fichier d'image et de l'afficher
- Permet de modifier la taille de l'imge par un curseur
- Permet de déplacer l'image à la souris
- Permet de lancer/arrêter une animation qui fait tourner cette image

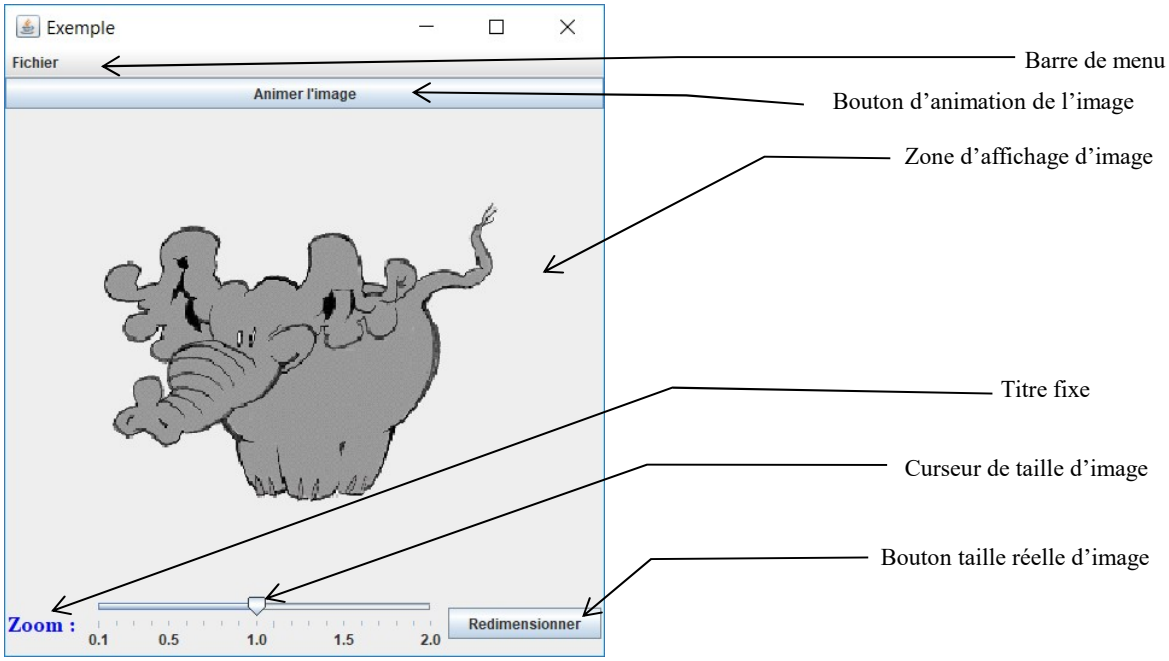
De plus, la modification de la taille d'image peut également être faite à distance soit par client/serveur (sockets) soit par RMI.

Le but de cet exemple est de monter comment :

- On crée une interface (*Exemple* et *InterfaceDeCommande*)
- On crée des composants d'interface personnalisés (*JLabelImageMobile*)
- On gère des événements de boutons, de menu, de curseur et de souris (*Exemple*)
- On utilise un FileChooser pour choisir un fichier (*Exemple*)
- On utilise une interface java pour définir un comportement (*ICurseurPilotable*)
- On utilise un Timer pour lancer une tâche répétitive (*Animation* dans *Exemple*)
- On crée/lance/arrête un Thread (*ServeurDeZoom*)
- On crée et utilise une classe abstraite (*InterfaceDeCommande*)
- On crée un objet pouvant être envoyé dans un flot (ici sur réseau par socket) (*Commande*)
- On met en place un dialogue client/serveur par socket (*ServeurDeZoom* et *ZoomParSocket*)
- On met en place et utilise un service RMI (*Exemple*, *ZoomParRMI, ServiceZoomRMIInterface* et *IServiceZoomRMIInterface*)

Vous pourez y puiser des exemples pour vos applications.

L'aspec de cette interface est le suivant :



Le code java de catte application est le suivant :

Tout d'abord l'application elle-même :

```
package exemple;
// imports pour placement dans l'interface
import java.awt.BorderLayout;
import java.awt.FlowLayout;
// imports pour gestion des événements
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
// imports pour fontes, couleurs, images et tailles
import java.awt.Font;
import java.awt.Image;
import javax.swing.ImageIcon;
import java.awt.Color;
import java.awt.Dimension;
// imports pour widgets
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JSlider;
import javax.swing.JLabel;
// imports pour menus
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
// import pour collection pour étiquettes du curseur
import java.util.Hashtable;
// imports pour tâche répétitive
import java.util.Timer;
import java.util.TimerTask;
// imports pour fichiers
import java.io.File;
```

```
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileNameExtensionFilter;
// imports pour serveur RMI
import java.rmi.AlreadyBoundException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
// imports d'autres paquetages (objet transmis par socket)
import commandeParSocket.Commande;

/*
 * Interface de démo qui propose :
 *    - Une zone d'affichage d'image
 *    - Un curseur de zoom d'image
 *    - Un bouton pour remettre l'image en taille réelle
 *    - Un bouton pour lancer/arrêter une mise en rotation continue de l'image
 * La souris permet de déplacer l'image en tenant le bouton appuyé
 *
 * Cette application permet à des clients de piloter à distance la position du curseur de zoom d'image :
 *    - Soit en se connectant par socket sur le port 5555
 *    - Soit par le service RMI de nom "Zoom_par_RMI" sur le port 5556
 */

@SuppressWarnings("serial")
//Interface : hérite de JFrame et propose un curseur pilotable à distance
public class Exemple extends JFrame implements ICurseurPilotable {
    // Eléments constituant l'interface
    private final String TOURNER = "Animer l'image"; // texte du bouton pour faire tourner l'image
    private final String ARRETER = "Arrêter l'animation de l'image"; // Texte du bouton pour arrêter
    private JButton anime; // Bouton pour faire tourner l'image
    private JLabelImageMobile photo; // Zone pour afficher l'image (composant personnalisé)
    private JSlider zoom; // Réglage du zoom de l'image
    private final int POSITION_INITIALE = 10; // position initiale du curseur de zoom
    private JButton redim; // Retour à la taille réelle de l'image (curseur en position initiale)
    // Propriétés de l'application
    private ImageIcon imageAffichee; // Image visible (fichier ouvert)
    private final int TAILLE_MINIMALE = 10; // taille minimale de l'image lors de zooms
    private int coordX, coordY; // Coordonnées de souris auxquelles a commencé le déplacement d'image
    private final int DELAI_ROTATION = 100; // délai entre 2 rotations unitaires d'image
    private Timer animation; // Timer qui fait tourner l'image
    // Pilotage du zoom de l'image à distance par socket
    private ServeurDeZoom serveur; // Thread qui assure le service permettant le zoom à distance
    private final int PORT_TCP = 5555; // Port du serveur par socket
    // Pilotage du zoom de l'image à distance par RMI
    private static final int PORT_RMI = 5556; // Port du serveur RMI
    private static final String SERVICE_RMI = "Zoom_par_RMI"; // Nom du service RMI
    private Registry annuaire; // Annuaire de services RMI

    public Exemple() { // Mise en place de l'interface
        super("Exemple"); // Création de la JFrame avec titre
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE); // On ne ferme que par le menu

        // Mise en place de la barre de menu
        JMenuBar barreDeMenu = new JMenuBar(); // Barre de menu
        JMenu menuFichier = new JMenu("Fichier"); // Rubrique Fichier
        menuFichier.setToolTipText("Ouvrir une image et Quitter"); // aide
        barreDeMenu.add(menuFichier);
        JMenuItem ouvrir = new JMenuItem("Ouvrir"); // Item Ouvrir de la rubrique Fichier
        menuFichier.add(ouvrir);
        ouvrir.addActionListener(new Ouvrir()); // Ecouteur d'événement pour Ouvrir
        JMenuItem enregistrer = new JMenuItem("Quitter"); // Item Quitter de la rubrique Fichier
        menuFichier.add(enregistrer);
        enregistrer.addActionListener(new Quitter()); // Ecouteur d'événement pour Quitter
        setJMenuBar(barreDeMenu); // Ajout de la barre de menu à la fenêtre
```

```
        // Mise en place de l'interface
        getContentPane().setLayout(new BorderLayout()); // Placement des widgets en BorderLayout

        anime = new JButton(TOURNER); // Bouton pour faire tourner l'image
        anime.setToolTipText("Lancer/arrêter la rotation de l'image"); // aide
        anime.setEnabled(false); // Tant qu'on n'a pas d'image il est inactif
        anime.addActionListener(new Animer()); // Lancement du Timer de rotation d'image
        getContentPane().add(anime, BorderLayout.NORTH); // Placement en haut de l'interface

        photo = new JLabelImageMobile(); // Zone d'affichage d'image (composant personnalisé)
        getContentPane().add(photo, BorderLayout.CENTER); // Ajout au centre de l'interface
        photo.addMouseMotionListener(new BougeImage()); // Ecouteur pour les mouvements de souris
        photo.addMouseListener(new RetourImage()); // Ecouteur pour les boutons de souris

        JPanel bas = new JPanel(); // Le bas de l'interface est divisé
        getContentPane().add(bas, BorderLayout.SOUTH); // Bas de l'interface
        bas.setLayout(new FlowLayout(FlowLayout.LEFT, 2, 2)); // Placement en FlowLayout

        JLabel titre = new JLabel("Zoom : "); // Titre du curseur
        titre.setFont(new Font("Serif", Font.BOLD,18)); // Police de ce titre
        titre.setForeground(Color.BLUE); // Couleur du texte
        bas.add(titre); // Mise en place du titre

        zoom = new JSlider(JSlider.HORIZONTAL, 1,20,1); // Curseur de zoom
        zoom.setToolTipText("Modification de la taille de l'image"); // aide
        zoom.setPreferredSize(new Dimension(300, 50)); // Pour qu'il ne soit pas trop petit
        zoom.setValue(POSITION_INITIALE); // Valeur initiale (10 = taille réelle)
        zoom.setMajorTickSpacing(10); // Grandes graduations chaque 10
        zoom.setMinorTickSpacing(1); // Petites graduations chaque 1
        zoom.setPaintTicks(true); // Dessiner les graduations
        zoom.setPaintTrack(true); // dessiner la piste
        Hashtable<Integer, JLabel> labels = new Hashtable<>(); // Collection d'étiquettes
        labels.put(1, new JLabel("0.1")); // La 1ere étiquette est 0,1
         for (int i=1; i<=20; i++) { // Les étiquettes suivantes sont 0,5 1.0 1.5 et 2.0
              labels.put(i*5, new JLabel(String.valueOf(5*i/10.0)));
         }
        zoom.setLabelTable(labels); // Associer ces étiquettes au curseur
        zoom.setPaintLabels(true); // Afficher les étiquettes
        zoom.setEnabled(false); // tant qu'on n'a pas d'image le zoom ne marche pas
        zoom.addChangeListener(new Zoomer()); // Ecouteur d'événement du zoom
        bas.add(zoom); // Ajout du curseur

        redim = new JButton("Redimensionner"); // Bouton de retour à la taille réelle
        redim.setToolTipText("Retour à la taille normale de l'image"); // aide
        redim.setEnabled(false); // tant qu'on n'a pas d'image le bouton ne marche pas
        redim.addActionListener(new Redimensionne()); // 2couteur d'événement du bouton
        bas.add(redim); // Ajout du bouton

        // Initialisations des propriétés
        imageAffichee = null; // pas d'image affichée pour le moment
        animation = null; // Pas de Timer de rotation d'image créé

        // Lancement du serveur pour pour commande de zoom à distance par socket
        serveur = new ServeurDeZoom(this, PORT_TCP);
        // lancement du serveur pour commande de zoom à distance par service RMI
        lancerServeurRMI();

        // Visualisation de l'interface
        pack(); // Taille de fenêtre suffisante
        setVisible(true); // fenêtre visible
    }

    // Lancement de l'application
    public static void main(String[] args) { // Cette classe est aussi l'application
```

```
        new Exemple(); // Au démarrage du programme l'interface est créée
    }

    // lancement du serveur pour commande de zoom à distance par service RMI
    private void lancerServeurRMI() {
        try {
            // Créer l'annuaire de services
            annuaire = LocateRegistry.createRegistry(PORT_RMI);
            // Crer l'objet qui assure le service
            ServiceZoomRMI service = new ServiceZoomRMI(zoom);
            // Enregistrer le service avec un nom
            annuaire.bind(SERVICE_RMI, service);
            System.out.println("Serveur pret");
        }
        catch(RemoteException re) {
            System.err.println("Erreur : Impossible d'enregistrer le service RMI");
            re.printStackTrace(); // Trace complète de l'erreur
         }
        catch(AlreadyBoundException abe) {
            // le service est déjà enregistré
        }
    }

    // Déplacer le curseur selon la commande reçue par le serveur
    // Méthode de l'interface ICurseurPilotable => doit être définie
    @Override
    public void positionneZoom(Commande commande) { // Positionner le curseur selon la commande reçue
        // Si le curseur est actif et que la valeur est correcte
        if ((zoom.isEnabled()) && (commande.valeurAcceptable(zoom))) {
            zoom.setValue(commande.valeurZoom()); // on positionne le curseur
        }
    }

    // Ecouteur pour l'item de menu : Quitter
    private class Quitter implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (animation != null) animation.cancel(); // Arrêt de l'animation s'il y en a une
            serveur.arreter(); // Arrêt du serveur par socket
            try { // Désinscription du service RMI
                annuaire.unbind(SERVICE_RMI);
            }
            catch (NotBoundException nbe) {
                // Le service n'était pas enregistré : ce n'est pas une erreur
            }
            catch (RemoteException re) { // Le service ne peut pas être enlevé
                System.err.println("Erreur d'arret du service RMI");
            }
            System.exit(0); // Arrêt du programme
        }
    }

    // Ecouteur pour l'item de menu : Ouvrir (afficher une image depuis un fichier)
    private class Ouvrir implements ActionListener {
        @Override
        public synchronized void actionPerformed(ActionEvent e) {
            // Créer un FileChooser pour sélectionner l'image à afficher
            JFileChooser choixFichier = new JFileChooser("."); // Répertoire de départ = actuel
            // Types de fichiers acceptés (images "jpg", "png", "gif")
        FileNameExtensionFilter filtre = new FileNameExtensionFilter("Fichiers Images", "jpg", "png", "gif");
        choixFichier.setFileFilter(filtre); // Mise en place du filtre par extensions de fichiers
        int codeRetour = choixFichier.showOpenDialog(Exemple.this); // ouverture du dialogue
        if(codeRetour == JFileChooser.APPROVE_OPTION) { // Un fichier a été sélectionné
            File fich = choixFichier.getSelectedFile(); // Récupérer le fichier sélectionné
            imageAffichee = new ImageIcon(fich.getAbsolutePath()); // Créer l'image à partir de ce fichier
```

```
                photo.setImageIcon(new ImageIcon(fich.getAbsolutePath())); // L'image récupérée est affichée
                zoom.setEnabled(true); // Le zoom devient utilisable
                redim.setEnabled(true); // Le bouton de retour à la taille réelle aussi
                anime.setEnabled(true); // Le bouton d'animation de l'image aussi
                pack(); // Redimensionnement de la fenêtre en fonction de l'image si nécessaire
            }
        }
    }

    // Ecouteur des actions du bouton de souris sur l'image
    private class RetourImage extends MouseAdapter {
        @Override
        public void mousePressed(MouseEvent e) { // Bouton appuyé => début de déplacement
            coordX = e.getX(); coordY = e.getY(); // garder les coordonnées de souris au début du déplacement
        }

        @Override
        public void mouseReleased(MouseEvent e) { // Bouton lâché => fin du déplacement
            photo.remetImageDroite(); // L'image revient à sa place initiale
        }
    }

    // Ecouteur des déplacements de souris sur l'image avec bouton appuyé
    private class BougeImage extends MouseMotionAdapter {
        @Override
        public synchronized void mouseDragged(MouseEvent e) { // déplacements de souris avec bouton appuyé
            int coordXAct = e.getX(); // Coordonnées actuelles de la souris
            int coordYAct = e.getY();
            photo.deplaceImage(coordXAct-coordX, coordYAct-coordY); // L'image bouge pour suivre la souris
            coordX = coordXAct; // Conserver les coordonnées actuelles de la souris
            coordY = coordYAct;
        }
    }

    // Ecouteur du curseur de zoom
    private class Zoomer implements ChangeListener {
        @Override
        public void stateChanged(ChangeEvent e) { // Le curseur a été déplacé
            if (imageAffichee != null) { // s'il y a une image affichée on la redimensionne
                Image img = imageAffichee.getImage(); // Image affichée pour redimensionnement
                int valZoom = imageAffichee.getIconWidth()*zoom.getValue()/10; // taille calculée
                // On ne descend pas au dessous de la TAILLE_MINIMALE sinon on ne voit rien
                if (valZoom < TAILLE_MINIMALE) valZoom = TAILLE_MINIMALE;
                // Afficher l'image redimensionnée
                photo.setImageIcon(new ImageIcon(img.getScaledInstance(valZoom, -1, Image.SCALE_DEFAULT)));
            }
        }
    }

    // Ecouteur du bouton de retour à la taille réelle
    private class Redimensionne implements ActionListener {
        @Override
        public synchronized void actionPerformed(ActionEvent e) {
            // On remet le curseur en position initiale => déclenche l'écouteur du curseur
            zoom.setValue(POSITION_INITIALE);
        }
    }

    // Ecouteur du bouton de lancement/arrêt de rotation d'image
    private class Animer implements ActionListener {
        @Override
        public synchronized void actionPerformed(ActionEvent e) {
            // L'action du bouton (rotation ou arrêt) dépend du texte qu'il affiche
            if (anime.getText().equals(TOURNER)) { // Le bouton lance la rotation
                animation = new Timer(); // Créer et lancer un Timer pour faire tourner l'image
```

```
                animation.scheduleAtFixedRate(new Animation(), DELAI_ROTATION, DELAI_ROTATION);
                anime.setText(ARRETER); // Le bouton sert maintenant à arrêter
            }
            else { // Le bouton arrête la rotation
                animation.cancel(); // On arrête le Timer de rotation d'image
                animation = null; // Le Timer n'existe plus (apres un cancel il faut le recréer)
                photo.remetImageDroite(); // On remet l'image en position droite
                anime.setText(TOURNER); // Le bouton sert maintenant à lancer la rotation
            }
        }
    }

    // Tâche de Timer qui fait tourner l'image de PI/20 lancée chaque DELAI_ROTATION
    private class Animation extends TimerTask {
        // A chaque étape l'image tourne d'un angle de UNITE_DE_ROTATION
        private final double UNITE_DE_ROTATION = Math.PI/20; // unité de rotation de 9°

        @Override
        public void run() { // Ce que fait le Timer chaque DELAI_ROTATION
            photo.tourneImage(UNITE_DE_ROTATION); // Tourner l'image de UNITE_DE_ROTATION
        }
    }

}
```

Création d'un élément d'interface personnalisé (JLabel pour afficher une image offrant des méthodes dea translation et de rotation de l'image) :

```
package exemple;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.ImageIcon;
import javax.swing.JLabel;

// Classe de widget personnalisé
// C'est un JLabel utilisé pour afficher des images
// Il en permet la rotation et le déplacement (translation)

@SuppressWarnings("serial")
public class JLabelImageMobile extends JLabel {

    private final int LARGEUR = 400; // Largeur préférentielle du JLabel
    private final int HAUTEUR = 400; // Hauteur préférentielle du JLabel
    private double tx, ty; // déplacement de l'image (translation) en x et en y
    private double angle; // angle de rotation
    private ImageIcon contenu; // Image contenue dans le JLabel

    // Construction du JLabel vide
    public JLabelImageMobile() {
        super(); // Création du JLabel vide
        tx = 0.0; // Déplacement en x
        ty = 0.0; // Déplacement en y
        angle = 0.0;  // Angle de rotation
        contenu = null; // On n'a pas encore d'image dans le JLabel
        this.setPreferredSize(new Dimension(LARGEUR,HAUTEUR)); // Taille car initialement vide
        this.setHorizontalAlignment(JLabel.CENTER); // L'image se placera au centre
        this.setVerticalAlignment(JLabel.CENTER);
    }

    // Création du JLabel avec une image (non utilisé dans cet exemple)
    public JLabelImageMobile(ImageIcon img) {
        this(); // Appel du constructeur de JLabelImageMobile vide
        setImageIcon(img); // Ajout d'une image dans le JLabel
    }
```

```
    // Place ou remplace l'image dans le JLabel
    public void setImageIcon(ImageIcon img) {
        contenu = img; // Conserver l'image à afficher
        this.repaint(); // Afficher cette image
    }

    @Override
    // Méthode de JLabel surchargée appelée lors de son affichage à l'écran
    public void paintComponent(Graphics g) { // Affichage du JLabel
        super.paintComponent(g); // Appel de l'affichage de JLabel
        if (contenu != null) { // Si on a une image
            Graphics2D g2D = (Graphics2D)g; // Récupération de l'objet Graphics2D du JLabel
            g2D.rotate(angle, getWidth()/2, getHeight()/2); // Rotation autour du centre
            g2D.translate(tx, ty); // Translation de tx et ty
            // Dessin de l'image centrée après rotation et translation
            g2D.drawImage(contenu.getImage(),(this.getWidth()-contenu.getIconWidth())/2,
                    (this.getHeight()-contenu.getIconHeight())/2, this);
        }
    }

    // Getters (en réalité ils ne servent à rien dans cet exemple)
    public double getTranslationX() { return tx; }
    public double getTranslationY() { return ty; }
    public double getRotation() { return angle; }
    public ImageIcon getImageIcon() { return contenu; }

    // Déplacement de l'image en x et y
    public void deplaceImage(double dx, double dy) {
        tx = tx+dx; // Décalage en x de dx
        ty = ty+dy; // Décalage en y de dy
        this.repaint(); // Rafraîchissement de l'affichage après décalage
    }

    // Rotation de l'image d'un angle da
    public void tourneImage(double da) {
        angle = angle + da; // Angle de rotation augmenté de da
        if (angle >= 0) { // Angle positif => ajustement entre 0 et 2*PI
            while (angle > 2*Math.PI) angle = angle - 2*Math.PI;
        }
        else { // Angle négatif => ajustement entre 0 et 2*PI
            while (angle < 0) angle = angle + 2*Math.PI;

        }
        this.repaint(); // Rafraîchissement de l'affichage après rotation
    }

    // Remet l'image en position initiale
    public void remetImageDroite() {
        tx = 0.0; // pas de décalage en x
        ty = 0.0; // pas de décalage en y
        angle = 0.0; // pas de rotation
        this.repaint(); // Rafraichissement de l'affichage
    }
}
```

Interface java implémentée par l'application pour pouvoir piloter le curseur à distance :

```
package exemple;
//imports d'autres paquetages (objet transmis par socket)
import commandeParSocket.Commande;

// Interface implémentée par les applications qui proposent un curseur pilotable à distance
public interface ICurseurPilotable {
    public void positionneZoom(Commande commande); // Pilotage du curseur par commande
}
```

Serveur (Thread) lancé par l'application pour piloter le curseur à distance par socket :

```
package exemple;
//imports pour sockets et flot d'envoi d'objets
import java.io.IOException;
import java.io.ObjectInputStream;
// import pour serveur TCP
import java.net.ServerSocket;
import java.net.Socket;
//imports d'autres paquetages (objet transmis par socket)
import commandeParSocket.Commande;

// Thread serveur : reçoit des commandes de zoom de clients et les exécute
// en déplaçant le curseur de zoom grâce à la méthode positionneZoom
// définie dans l'interface ICurseurPilotable
public class ServeurDeZoom extends Thread { // Le serveur est un thread qui surveille les connexions

    private ServerSocket ecoute; // Socket d'écoute du serveur
    private volatile boolean enMarche; // Pour arrêter proprement le serveur
    private ICurseurPilotable fenetrePilotable; // Interface pouvant être pilotée par le serveur

    public ServeurDeZoom(ICurseurPilotable pilotable, int port) {
        fenetrePilotable = pilotable; // Interface pouvant être pilotée par le serveur
        try {
            ecoute = new ServerSocket(port); // Socket d'écoute du serveur
            enMarche = true; // Le serveur est en marche
            start(); // Lancer le thread serveur
        }
        catch(IOException ioe) { // Erreur de création de la socket d'écoute
            ecoute = null; // pas de socket créée
            System.err.println("Impossible de lancer le serveur par socket");
        }
    }

    // Arrêter le serveur => fermer la socket d'écoute
    public void arreter() {
        enMarche = false; // Indicateur d'arrêt
        try {
            if (ecoute != null) ecoute.close(); // Fermer la socket d'écoute => IOException dans accept
        }
        catch(IOException ioe) { // Erreur de fermeture de la socket d'écoute
            System.err.println("Impossible d'arreter le serveur");
        }
    }

    // Boucle de reception des commandes de clients
    public void run() {
        while(enMarche) {
            try {
                // Attente d'un client (provoquera une IOException lors de l'arrêt du serveur)
                Socket duClient = ecoute.accept();
                // Flot pour lire des objets
                ObjectInputStream lecture = new ObjectInputStream(duClient.getInputStream());
                Commande commande = (Commande)lecture.readObject(); // réception de la commande
                lecture.close(); // Fermeture de connexion avec le client
                duClient.close();
                fenetrePilotable.positionneZoom(commande); // L'interface traite la commande reçue
            }
            catch (ClassCastException e) { // L'objet reçu n'est pas de classe Commande
                System.err.println("Commande recue incorrecte");
            }
            catch (ClassNotFoundException e) { // L'objet reçu est de classe inconnue
                System.err.println("Commande recue incorrecte");
            }
            catch(IOException ioe) {
                // Cette exception se produit en cas d'erreur de connexion avec le client
```

```
                    // Mais aussi quand on arrête le serveur alors qu'il est en attente de client (accept)
                    if (enMarche) System.err.println("Erreur de connexion avec un client");
                }
            }
        }
}
```

Objet sérialisable utilisé pour le dialogue client/serveur par socket pour piloter le curseur à distance :

```
package commandeParSocket;
//import pour que l'objet puisse être envoyé par socket
import java.io.Serializable;
//import pour pilotage de curseur
import javax.swing.JSlider;

// Classe permettant d'envoyer des commandes de déplacement de curseur en client/serveur
public class Commande implements Serializable { // Cette classe est sérialisable

    private static final long serialVersionUID = 1L; // n° de version pour sérialisation

    private int positionZoom; // Position de curseur demandée

    public Commande(int valZoom) { // Création d'une commande avec valeur de position de curseur
        positionZoom = valZoom;
    }

    public int valeurZoom() { // Renvoie la position demandée
        return positionZoom;
    }

    // Indique si la valeur est acceptable par un JSlider donné
    public boolean valeurAcceptable(JSlider curs) {
        return ((positionZoom >= curs.getMinimum()) && (positionZoom <= curs.getMaximum()));
    }
}
```

Objet (assurant un service accessible à distance) pour piloter le curseur à distance par RMI :

```
package exemple;
//imports pour RMI
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
// import pour le curseur piloté par ce service
import javax.swing.JSlider;

// Classe offrant un service RMI de déplacement d'un curseur distant
@SuppressWarnings("serial")
public class ServiceZoomRMI extends UnicastRemoteObject implements ServiceZoomRMIInterface {

    private JSlider curseur; // Le curseur piloté par ce service

    public ServiceZoomRMI(JSlider curs) throws RemoteException {
        curseur = curs; // Le curseur piloté par ce service
    }

    // Méthode offerte par ce service : déplacer le curseur
    @Override
    public void reglerCurseur(int positionZoom) throws RemoteException {
        if ((positionZoom >= curseur.getMinimum()) && (positionZoom <= curseur.getMaximum())
                && (curseur.isEnabled())) { // Si le curseur est actif et la valeur est correcte
            curseur.setValue(positionZoom); // Déplacer le curseur
        }
        else throw new RemoteException("Déplacement de zoom impossible"); // Sinon lever une exception
    }
}
```

Interface de l'objet de service distant permettant de piloter le curseur à distance par RMI. Cette Interface sera fournie aux utilisateurs (clients RMI) :

```
package exemple;
//imports pour RMI
import java.rmi.Remote;
import java.rmi.RemoteException;

// Interface offerte au client RMI pour lui permettre de déplacer un curseur à distance
public interface ServiceZoomRMIInterface extends Remote {

    // La seule méthode offerte par ce service RMI est celle qui déplace le curseur distant
    public void reglerCurseur(int val) throws RemoteException;
}
```

Classe abstraite définissant une interface de pilotage du curseur à distance utilisée par l'application cliente par socket et par l'application cliente par RMI :

```
package interfaceDeCommande;
// imports pour l'interface
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JSlider;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.util.Hashtable;
//imports pour les événements d'interface
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

// Interface permettant de bouger le curseur de zoom d'image à distance
// Utilisée pour commander le zoom par client/serveur (socket) ou par service RMI
// Cette classe est abstraite car incomplète : la méthode traiterEvenementCurseur
// qui gère les événements du curseur doit être définie dans les classes filles
// pour, selon le cas, déplacer le curseur distant par RMI ou par socket
@SuppressWarnings("serial")
public abstract class InterfaceDeCommande extends JFrame {

    private final int POSITION_INITIALE = 10; // position initiale du curseur
    protected JSlider zoomDeporte; // Réglage du zoom de l'image (accessible par les classes filles)
    private JButton redimDeporte; // Retour à la taille réelle de l'image

    public InterfaceDeCommande(String titreFenetre) {
        super(titreFenetre); // Création de la JFrame avec titre
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Arrêt du programme par fermeture de la fenêtre

        // Mise en place de l'interface (c'est une partie de celle du serveur)
        getContentPane().setLayout(new FlowLayout(FlowLayout.LEFT, 2, 2)); // Placement en FlowLayout

        JLabel titre = new JLabel("Zoom distant : "); // Titre du curseur
        titre.setFont(new Font("Serif", Font.BOLD,18)); // Police de ce titre
        titre.setForeground(Color.BLUE); // Couleur du texte
        getContentPane().add(titre); // Mise en place du titre

        zoomDeporte = new JSlider(JSlider.HORIZONTAL, 1, 20, 1); // Curseur de zoom
        zoomDeporte.setToolTipText("Modification de la taille de l'image sur le serveur");
        zoomDeporte.setPreferredSize(new Dimension(300, 50)); // Pour qu'il ne soit pas trop petit
        zoomDeporte.setValue(POSITION_INITIALE); // Valeur initiale (10 = taille réelle)
        zoomDeporte.setMajorTickSpacing(10); // Grandes graduations chaque 10
        zoomDeporte.setMinorTickSpacing(1); // Petites graduations chaque 1
        zoomDeporte.setPaintTicks(true); // Dessiner les graduations
```

```
        zoomDeporte.setPaintTrack(true); // dessiner la piste
        Hashtable<Integer, JLabel> labels = new Hashtable<>(); // Collection d'étiquettes
        labels.put(1, new JLabel("0.1")); // La 1ere étiquette est 0,1
         for (int i=1; i<=20; i++) { // Les étiquettes suivantes sont 0,5 1.0 1.5 et 2.0
             labels.put(i*5, new JLabel(String.valueOf(5*i/10.0)));
         }
        zoomDeporte.setLabelTable(labels); // Associer ces étiquettes au curseur
        zoomDeporte.setPaintLabels(true); // Afficher les étiquettes
        zoomDeporte.addChangeListener(new Zoomer()); // Ecouteur d'événements du zoom
        getContentPane().add(zoomDeporte); // Ajout du curseur

        redimDeporte = new JButton("Redimensionner"); // Bouton de retour à la taille réelle
        redimDeporte.setToolTipText("Retour à la taille normale de l'image sur le serveur");
        redimDeporte.addActionListener(new Redimensionne()); // 2couteur d'événement du bouton
        getContentPane().add(redimDeporte); // Ajout du bouton

        // Dimensionnement de l'interface
        pack(); // Taille de fenêtre suffisante
        setResizable(false); // Le redimensionnement de la fenêtre de commande est inutile
        // La fenêtre ne devient pas visible ce sera fait dans les classes héritées
    }

    // Ecouteur du bouton de retour à la taille réelle
    private class Redimensionne implements ActionListener {
        @Override
        public synchronized void actionPerformed(ActionEvent e) {
            // On remet le curseur au milieu => déclenche l'écouteur du curseur
            zoomDeporte.setValue(POSITION_INITIALE);
        }
    }

    // Ecouteur du curseur de zoom
    private class Zoomer implements ChangeListener {
        @Override
        public void stateChanged(ChangeEvent e) { // Le curseur a été déplacé
            traiterEvenementCurseur(); // Méthode à définir dans les classes filles
        }
    }

    // Méthode de traitement des événements de curseur doit être définie dans les classes filles
    protected abstract void traiterEvenementCurseur();
}
```

Application cliente par socket qui permet de déplacer le curseur distant :

```
package commandeParSocket;
//imports pour la connexion par socket au serveur
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.Socket;
//imports d'autres paquetages
import interfaceDeCommande.InterfaceDeCommande;

// Application cliente de l'application "Exemple"
// qui permet de bouger le curseur de zoom d'image à distance par connexion C/S par socket
@SuppressWarnings("serial")
public class ZoomParSocket extends InterfaceDeCommande { // Spécialisation de l'interface de commande

    private final String SERVEUR_TCP = "localhost"; // Pour tester : client et serveur sur la même machine
    private final int PORT_TCP = 5555; // Port du serveur

    public ZoomParSocket() { // Création de l'interface et définition de l'écouteur du curseur
        super("Commande par socket"); // Titre de la fenêtre
        setVisible(true); // la fenêtre devient visible
    }
```

```
    public static void main(String[] args) {
        new ZoomParSocket(); // Lancement du client
    }

    // Ecouteur du curseur de zoom : méthode abstraite définie
    public void traiterEvenementCurseur() {
        try {
            Socket connexion = new Socket(SERVEUR_TCP, PORT_TCP); // Se connecter au serveur
            // Flot pour envoyer une commande de déplacement du curseur
            ObjectOutputStream envoi = new ObjectOutputStream(connexion.getOutputStream());
            Commande commande = new Commande(zoomDeporte.getValue()); // Création de la commande à envoyer
            envoi.writeObject(commande); // envoi de la commande
            envoi.flush(); // Pour être sûr qu'elle est envoyée (normalement inutile mais prudent)
            envoi.close(); // fermeture de connexion avec le serveur
            connexion.close();
        }
        catch(IOException ioe) { // Erreur lors de la connexion ou du dialogue avec le serveur
            System.err.println("Erreur de connexion au serveur");
            //ioe.printStackTrace();
        }
    }
}
```

Et enfin application cliente par RMI qui permet de déplacer le curseur distant :

```
package commandeParRMI;
//imports pour RMI
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
// imports d'autres paquetages
import exemple.ServiceZoomRMIInterface;
import interfaceDeCommande.InterfaceDeCommande;

// Application cliente de l'application "Exemple"
// qui permet de bouger le curseur de zoom d'image à distance par service RMI
@SuppressWarnings("serial")
public class ZoomParRMI extends InterfaceDeCommande { // Spécialisation de l'interface de commande

    private final String SERVEUR_RMI = "localhost"; // Pour tester : client et serveur sur la même machine
    // Port et nom de service pour serveur RMI
    private final int PORT_RMI = 5556;
    private final String SERVICE_RMI = "Zoom_par_RMI";

    private ServiceZoomRMIInterface serviceDistant; // Service distant vu par son interface fournie

    public ZoomParRMI() { // Création de l'interface et définition de l'écouteur du curseur
        super("Commande par RMI"); // Titre de la fenêtre
        // Trouver le service et le récupérer
        try {
            // trouver l'annuaire sur le serveur
            Registry annuaire = LocateRegistry.getRegistry(SERVEUR_RMI, PORT_RMI);
            // Trouver le service (désigné par son nom)
            serviceDistant = (ServiceZoomRMIInterface)annuaire.lookup(SERVICE_RMI);
            setVisible(true); // si tout s'est bien passé la fenêtre devient visible
        }
        catch(RemoteException e) {
            System.err.println("Erreur : service non accessible");
            System.exit(0);
        }
        catch(NotBoundException e) {
            System.err.println("Erreur : service non enregistre");
            System.exit(0);
        }
    }
```

```
    public static void main(String[] args) {
        new ZoomParRMI(); // Lancement du client RMI
    }

    // Ecouteur du curseur de zoom : méthode abstraite définie
    public void traiterEvenementCurseur() {
        try { // Utilisation du service RMI
            serviceDistant.reglerCurseur(zoomDeporte.getValue());
        }
        catch (RemoteException re) { // Exception si erreur de connexion ou valeur incorrecte
            System.err.println(re.getMessage());
        }
    }
}
```

SOMMAIRE