# Structure générale de Swing

On peut distinguer dans Swing deux types d'objets, un premier type nommé *conteneurs*, destinés, comme leur nom l'indique, à contenir les objets du second type nommés *composants*. Les choses se compliquent un peut lorsque l'on sait que les conteneurs sont également ... des composants ! En effet, tous ces objets héritent de la même classe javax.swing.JComponent, ce qui signifie notamment qu'il va être possible de placer des conteneurs dans des conteneurs, les choses ne diffèrent que très peu de ce qui se fait avec AWT pour l'instant.

## Conteneurs de premier niveau (Top-Level Containers)

Il existe cependant un groupe de conteneurs un peu particuliers, nommés *conteneurs de premier niveau (Top-Level Containers)* qui ne sont pas des sous classes de JComponent. Ces objets doivent leur nom au fait qu'ils constituent toujours les objets de base d'une interface Swing. Un conteneur de premier niveau peut contenir des objets d'une des sous-classes de JComponent, mais n'est pas destiné à être contenu dans un autre conteneur de premier niveau. Ce groupe comprend les applets (JApplet), les dialogues (JDialog), et les cadres (JFrame).

## Conteneurs intermédiaires

Tout comme les conteneurs de premier niveau, les *conteneurs intermédiaires* sont destinés à contenir des objets d'une sous-classe de JComponent.Faisant eux-mêmes partie de cette dernière catégorie, il est possible d'imbriquer des conteneurs intermédiaires dans d'autres conteneurs intermédiaires. Le but étant, comme dans AWT de combiner les différents *gestionnaires de mise en page (Layout Managers)*, pour élaborer des interfaces complexes.
Swing comporte tout d'abord un premier type de conteneurs intermédiaires dit à *vocation générale*, que nous étudierons en détail, et dont font partie :

les **panneaux** (JPanel), conteneur le plus simple, il est destiné à recevoir d'autres composants dont l'organisation est commandée par le gestionnaire de mise en page qui lui est associé ;
les **panneaux défilants** (JScrollPane), qui contiennent une zone utile chargée de recevoir un autre composant, comme une image ou du texte. Lorsque ce dernier est plus grand que la zone affichable couverte par le panneau, deux ascenseurs, horizontaux et verticaux, permettent de parcourir l'ensemble de ce composant ;
les **panneaux divisables** (JSplitPane), qui comportent une barre de séparation, horizontale ou verticale, fixe ou ajustable, créant deux zones distinctes qui peuvent recevoir des composants ;
les **panneaux à onglets** (JTabbedPane), dont l'intérêt est de permettre de multiplier le nombre de composants accessibles sur une surface affichable réduite, en structurant les composants en fiches. Chaque fiche comporte un onglet toujours visible, qui permet à la fiche de passer au premier plan lorsque l'utilisateur clique dessus ;
les **barre d'outils** (JToolBar), dont les outils sont généralement constitués de boutons, elles peuvent être positionnées sur n'importe quel bord d'une fenêtre, ou encore déplacées et détachées dans une fenêtre propre, ce qui permet de la positionner n'importe-où sur l'écran.

Les conteneurs intermédiaires *spécialisés* sont destinés à des utilisations plus particulières, il sont au nombre de trois :

les **cadres internes** (JInternalFrame) permettent de transposer le concept de bureau-écran à la fenêtre. Cette dernière devient un bureau virtuel qui peut contenir des mini-fenêtres que l'on peut déplacer, redimensionner, fermer, ect... ;
les **panneaux superposés** (JLayeredPane) dont le principe reprend celui des feuille de celluloid en animation. Ces couches sont transparentes, ordonnées suivant leur profondeur. Un composant situé sur une couche donnée masquera les composants des couches plus profondes qu'il recouvre. Comme le modèle réel sur lequel elles reposent, elles sont particulièrement adaptées à la création d'animations, de

jeux, puisqu'il est possible par exemple de définir un décors sur le premier et le dernier plan, et de faire librement évoluer un personnage sur un plan intermédiaire sans se soucier des problèmes de recouvrements.
cette structure en couches est utilisée par Java pour l'animation de certains composants comme les barres de menus. Chaque conteneur de premier niveau (JApplet, JDialog, JFrame) est associé à un **panneau racine** (JRootPane). Ce panneau comporte quatre parties : le *panneau de verre* (glass pane), le *panneau de contenu* (content pane), un premier *panneau en couche* (layered pane), et la *barre de menu* optionnelle.

# Les composants Swing

Ce sont les briques qui vont servir à la construction de l'interface. Bien que chaque composant possède des fonctionnalités qui lui sont propre, certains possèdent entre eux des points communs qui permettent de les regrouper dans trois catégories : les *contrôles de base,* les a*ffichages informatifs non éditables*, et les *affichages éditables d'informations formatées*.

**Contrôles de base**

Ce sont tous les objets sur lesquels l'utilisateur peut agir pour fournir des informations au programme : lancer une action en cliquant sur un bouton ou en sélectionnant l'item approprié dans un menu, enter un texte dans un champ de texte éditable, choisir une option dans une liste, et bien d'autres éléments que l'on retrouve habituellement dans une interface graphique.

**Affichages informatifs non éditables**

Sous cette catégorie sont regroupés tous les composants destinés à fournir des informations à l'utilisateur, sans que celui-ci ne puisse les éditer. La liste suivante énumère les composants de ce type :

il peut s'agir d'une simple information, un **label** (JLabel) ;
d'une **barre de progression** (JProgressBar), qui renseigne sur l'état d'avancement d'une tâche ;
ou encore d'une **bulle d'aide** (JToolTip) qui s'affiche lorsque l'utilisateur passe sur un composant, pour lui fournir par exemple une information sur l'utilité de ce dernier.

**Affichages éditables d'informations formatées**

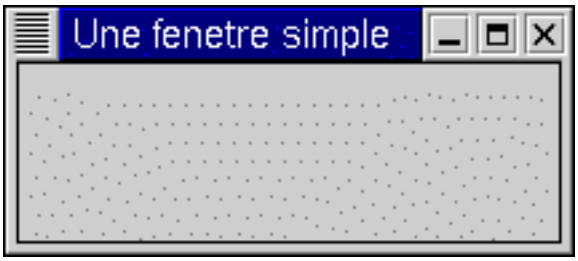
Ce terme désigne des composants aux fonctionnalités plus complexes, contenant des informations que l'utilisateur peut éventuellement modifier, qui possèdent la particularité d'être formatées d'une manière particulière :

le **sélecteur de couleur** (JColorChooser) proposant plusieurs modes pour choisir une couleur ;
le **sélecteur de fichier** (JFileChooser), qui est la célèbre boîte de sélection classique permettant de naviguer dans l'arborescence d'un disque et de sélectionner le nom du fichier à ouvrir ou à sauvegarder :
la **table** (JTable) qui regroupe les informations sous forme de cellules, de façon similaire à ce que l'on peut trouver dans un tableur ;
les divers **composants de texte** qui permettent la saisie et la mise en forme de texte sur plusieurs lignes.
les **arbres** (JTree), destinés à présenter des informations qui sont structurées de manière hiérarchique. Chaque noeud peut-être dans un état *étendu*, auquel cas il fait apparaître la liste de ses fils, ou encore *refermé,* dans ce cas il ne la montre pas.

# Les Cadres (JFrames)

Premier type de *conteneur de premier niveau*, ce qui signifie qu'ils peuvent contenir d'autres objets, les composants, mais ne peuvent pas être eux-même contenus dans d'autres conteneurs. Un cadre, implémenté comme une instance de la classe JFrame est tout simplement une fenêtre qui peut comporter un titre, des cases de fermeture et d'iconification....

Il n'est pas possible d'ajouter directement un composant à un objet de la classe JFrame, il est nécessaire de passer par l'intermédiaire d'un conteneur spécial appelé *panneau de contenu* (content pane). Les cadres possèdent tous un panneau de contenu par défaut, auquel on fait référence à l'aide de la méthode getContentPane(). Il est également possible de remplacer ce panneau par défaut par un conteneur intermédiaire de type **JPanel** par exemple.




# Création d'un cadre

**JFrame()**
**JFrame(String title)**
Crée un cadre initialement invisible. Le paramètre **title** permet de fixer le titre du cadre lors de sa crétion. On peut aussi utiliser **setTitle** pour modifier ce titre.

# Affichage d'un cadre

**void pack()**
Hérité de java.awt.Window
Ajuste la taille du cadre en fonction des tailles des composants qu'il contient.

**void setVisible(boolean b)**
Hérité de java.awt.Component.
Rend le cadre visible ou invisible, suivant la valeur du paramètre **b** :
**true** : rend le cadre visible ;
**false** : masque le cadre.

# Gestion du panneau de contenu

**void setContentPane(Container contentPane)**
**Container getContentPane()**
Fixe ou retourne le panneau de contenu du cadre.

**void setJMenuBar(JMenuBar menubar)**
**JMenuBar getJMenuBar()**
Fixe ou retourne la barre de menu du cadre.

# Fermeture d'un cadre

**void addWindowListener(WindowListener l)**
Hérité de java.awt.Window
Permet d'ajouter un écouteur d'événement de fenêtre au cadre.

**void setDefaultCloseOperation(int operation)**
**int getDefaultCloseOperation()**

Fixe ou retourne le comportement du cadre lorsque l'utilisateur clique sur la case de fermeture. Les divers choix sont :

DO_NOTHING_ON_CLOSE , ne fait rien, le programmeur prend totalement en charge la gestion du cadre, et doit fournir à ce cadre un écouteur d'événement de fenêtre (windowListener) qui implémente la méthode **windowClosing()** ;
HIDE_ON_CLOSE, c'est la valeur par défaut, le cadre existe toujours mais est rendu invisible, **après invocation** d'un éventuel objet de la classe WindowListener associé au cadre ;
DISPOSE_ON_CLOSE, masque et détruit le cadre **après invocation** d'un éventuel objet de la classe WindowListener associé au cadre. Ce dernier peut alors être pris en charge par le garbage collector qui se chargera de libérer les ressources qu'il occupe.

Ces constantes sont définies dans l'interface WindowConstants, que JFrame implémente.

# Les dialogues (JDialog, JOptionPane)

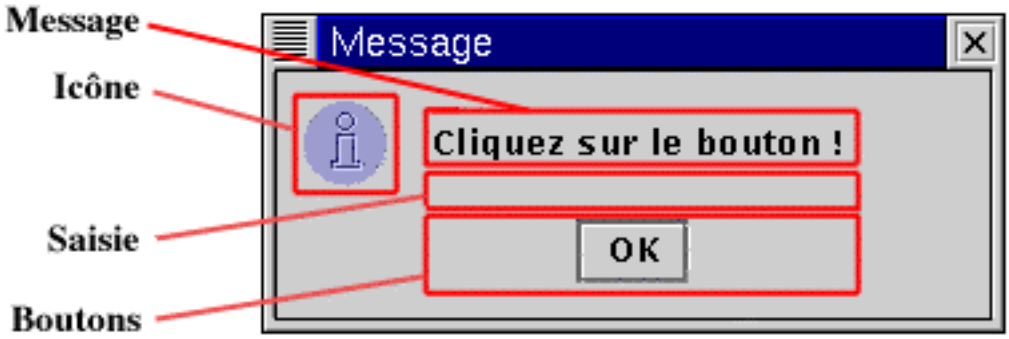
## Anatomie des dialogues

Tout comme les cadres (JFrame), les dialogues de la classe JDialog sont des *conteneurs de premier niveau* (top-level container), ils ne peuvent donc pas être contenus dans d'autres cadres. Cependant, ils sont toujours dépendants d'un autre cadre. Lorsque ce dernier est fermé, tout dialogue qui lui est associé l'est de même. Lorsqu'un cadre est réduit à l'état d'icône, ses dialogues associés disparaissent de l'écran, et redeviennent visibles quand il est désicônifié.

Un dialogue peut être *modal.* Auquel cas, lorsqu'il est affiché, il bloque toute entrée utilisateur destinée aux autres fenêtres du programme. Les dialogues affichés à l'aide des méthodes showXxxDialog de la classe JOptionPane sont tous de type modaux. Pour créer des dialogues non modaux, il faut utiliser la classe JDialog.

La classe JOptionPane fournit plusieurs façons de créer et d'utiliser des boîtes de dialogues, très simplement. Il suffit d'utiliser l'une des méthodes showXxxxDialog. Dans le cas où vous auriez besoin d'un dialogue non modal, ou si vous avez besoin de contrôler le comportement de fermeture de la fenêtre du dialogue de façon particulière, alors vous devrez créer une instance de JOptionPane et l'ajouter à un objet JDialog. Invoquez ensuite setVisible(true) sur le JDialog pour le faire apparaître.

La figure ci-dessous présente la structure générale des boîtes de dialogue.




## Détail des paramètres communs aux méthodes d'affichage et de création de dialogues

Les diverses méthodes de la classe JOptionPane qui permettent d'afficher (showXxxDialog) ou de créer (JOptionPane) des boîtes de dialogue possèdent pratiquement toutes les mêmes séries de paramètres, nous détaillons le rôle de chacun d'eux ci-dessous :

**Component parentComponent**
Dans les méthodes showXxxDialog :
Le composant auquel est rattaché le dialogue. Il doit s'agir d'un cadre, d'un composant dans un cadre, ou null. S'il s'agit d'un cadre ou d'un composant situé dans un cadre, le dialogue sera centré sur ce cadre ou ce composant. S'il s'agit de null, la position du dialogue sera dépendante du Look&Feel sélectionné -- généralement au centre de l'écran -- et le dialogue ne sera rattaché à aucun cadre visible.

Dans les constructeurs JOptionPane :
Ce paramètre n'existe pas, le cadre auquel est rattaché le dialogue est spécifié lors de la création du JDialog qui contient l'objet JOptionPane. La position du dialogue doit alors être fixée à l'aide de la méthode setLocationRelativeTo() de la classe JDialog.

**Object message**
Spécifie ce qui doit être affiché dans la zone principale du dialogue. Il s'agit généralement une chaîne de caractères, que l'on peut étendre sur plusieurs lignes en utilisant la séquence \n, comme en langage C.
**String title**
Titre de la fenêtre du dialogue.
**int optionType**
Définit quel ensemble de boutons apparaîtra dans le dialogue. Les différents choix possibles sont : DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION.
**int messageType**
Permet de définir le type de dialogue, et donc quel icône sera utilisée dans le dialogue. Les choix possibles sont présentés dans le tableau suivant, les icônes correspondent au Look&Feel Java.

**Valeur de messageType Icône Affichée** PLAIN_MESSAGE Aucune icône. ERROR_MESSAGE

INFORMATION_MESSAGE

WARNING_MESSAGE

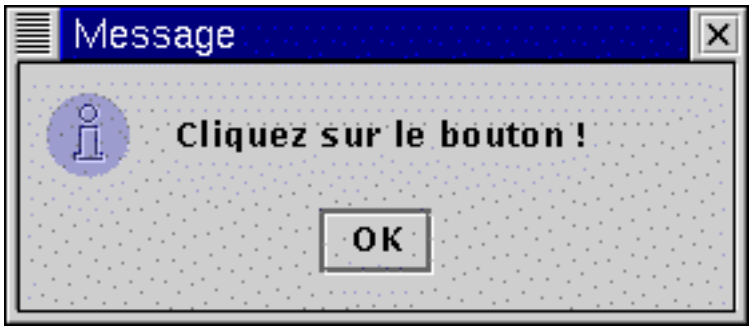
QUESTION_MESSAGE

**Icon icon**
Permet d'attribuer une icône personnalisée au dialogue.
**Object[] options**
Table d'objets (souvent des chaînes de caractères) qui permet de personnaliser le contenu des boutons

du dialogue.
**Object initialValue**
Spécifie la valeur par défaut, i.e. le bouton qui sera encadré et sélectionné lorsque l'utilisateur appuyera sur la touche retour. Ce paramètre est un des objets de la table précédente.

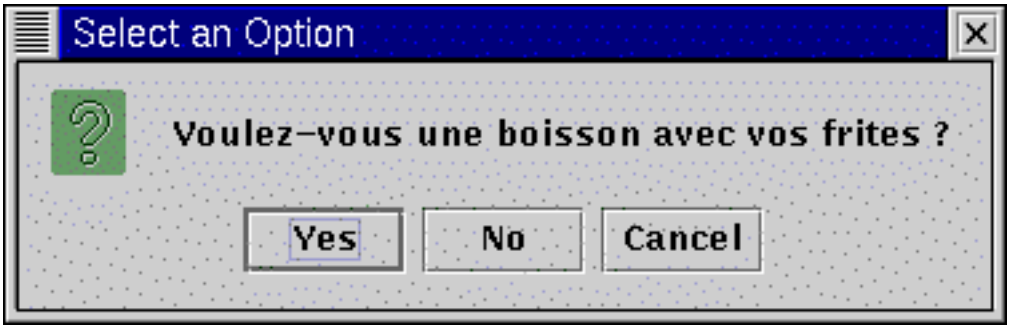
# **Dialogues modaux standards** (méthodes de la classe JOptionPane).

## Dialogues d'information



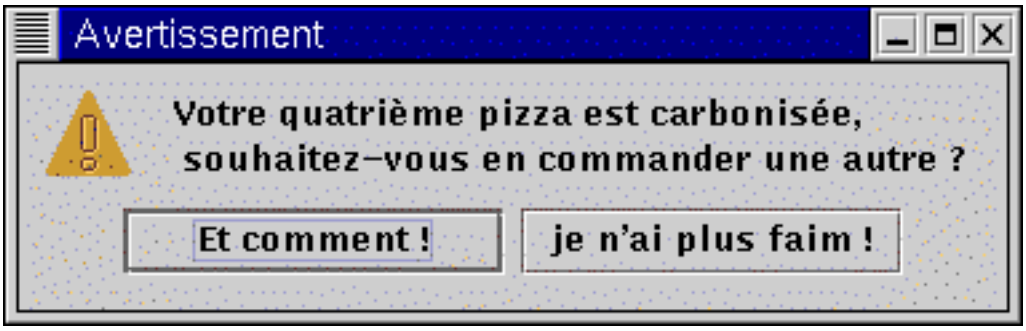

**public static void showMessageDialog(Component parentComponent, Object message)**
**public static void showMessageDialog(Component parentComponent, Object message, String title, int messageType)**
**public static void showMessageDialog(Component parentComponent, Object message, String title, int messageType, Icon icon)**
Affiche un dialogue modal contenant une information et un seul bouton de confirmation.

## Dialogues de confirmation



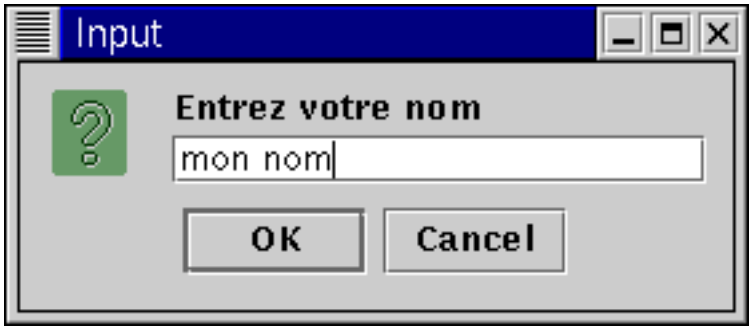

**public static int showConfirmDialog(Component parentComponent,Object message)**
**public static int showConfirmDialog(Component parentComponent,Object message,String title,int optionType)**
**public static int showConfirmDialog(Component parentComponent,Object message,String title,int optionType,int messageType)**
**public static int showConfirmDialog(Component parentComponent,Object message,String title,int optionType,int messageType, Icon icon)**
Affiche un dialogue modal destiné à poser une question à l'utilisateur.
**Valeur retournée** :
YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION, suivant le bouton sur lequel l'utilisateur a cliqué,
ou encore
CLOSED_OPTION : l'utilisateur a quitté le dialogue en cliquant sur la case de fermeture de la fenêtre.

# Dialogues modaux à boutons personnalisés




**public static int showOptionDialog(Component parentComponent, Object message, String title, int optionType,  int messageType, Icon icon, Object[] options, Object initialValue)**
Affiche un dialogue modal entièrement personnalisé.
**Valeur retournée** :
YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION, suivant le bouton sur lequel l'utilisateur a cliqué,
ou encore
CLOSED_OPTION : l'utilisateur a quitté le dialogue en cliquant sur la case de fermeture de la fenêtre.

# Dialogues modaux de saisie d'informations




**static String showInputDialog(Component parentComponent, Object message)**
**static String showInputDialog(Component parentComponent, Object message, String title, int messageType)**
**static String showInputDialog(Component parentComponent, Object message, String title, int messageType,Icon icon, Object[] selectionValues, Object initialSelectionValue)**
Affiche un dialogue modal contenant un champ de texte éditable.
**Valeur retournée** :
la chaîne entrée par l'utilisateur, s'il a confirmé la saisie en cliquant sur le bouton "OK" ;
**null** si l'utilisateur a annulé la saisie en cliquant sur le bouton "CANCEL".

# Dialogues personnalisables non standards, non modaux

## Création du panneau de contenu

**JOptionPane()**
**JOptionPane(Object message)**
**JOptionPane(Object message, int messageType)**
**JOptionPane(Object message, int messageType, int optionType)**
**JOptionPane(Object message, int messageType, int optionType, Icon icon)**

**JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options)**
**JOptionPane(Object message, int messageType, int optionType, Icon icon, Object[] options, Object initialValue)**
Crée un panneau de contenu de dialogue, qui devra être substitué au panneau de contenu d'un objet de la classe JDialog.

### Création du cadre du dialogue

**JDialog(Frame owner)**
**JDialog(Frame owner, boolean modal)**
**JDialog(Frame owner, String title)**
**JDialog(Frame owner, String title, boolean modal)**
Crée un cadre de premier niveau de type boîte de dialogue. Le booléen **modal** permet de spécifier si le dialogue est modal, et il est possible de définir le titre de la fenêtre à l'aide du paramètre **title**.

**void setContentPane(Container contentPane)**
**Container getContentPane()**
Fixe ou récupère le panneau de contenu du dialogue.

**void setDefaultCloseOperation(int operation)**
**int getDefaultCloseOperation()**
Fixe ou récupère le comportement du dialogue lorsque l'utilisateur clique sur la case de fermeture. Les divers choix sont :

DO_NOTHING_ON_CLOSE , ne fait rien, le programmeur prend totalement en charge la gestion du cadre, et doit fournir à ce cadre un écouteur d'événement de fenêtre (windowListener) qui implémente la méthode windowClosing() ;
HIDE_ON_CLOSE, c'est la valeur par défaut, le cadre existe toujours mais est rendu invisible, **après invocation** d'un éventuel objet de la classe WindowListener associé au cadre ;
DISPOSE_ON_CLOSE, masque et détruit le cadre **après invocation** d'un éventuel objet de la classe WindowListener associé au cadre. Ce dernier peut alors être pris en charge par le garbage collector qui se chargera de libérer les ressources qu'il occupe.

Ces constantes sont définies dans l'interface WindowConstants, que JDialog implémente.

**void setLocationRelativeTo(Component c)**
Centre le dialogue sur le composant passé en paramètre.

## Conteneurs intermédiaires

# Panneaux (JPanel)

Les panneaux sont les *conteneurs intermédiaires* les plus simples. Destinés à contenir les composants dont l'agencement est régie par le gestionnaire de mise en page (layout manager). Le gestionnaire par défaut étant un FlowLayout. Il est possible de placer les panneaux dans d'autres conteneurs, qu'ils soient de premier niveau ou intermédiaires, ce permet de combiner les différents gestionnaires de mise en page.

## Construction d'un panneau

**JPanel()**
**JPanel(LayoutManager layout)**
Crée le panneau avec le gestionnaire de mise en page éventuellement spécifié.

## Gérer les composants d'un  panneau (méthodes de java.awt.Container)

**Component add(Component comp)**
**Component add(Component comp, int index)**
Ajoute le composant **comp** au panneau. L'argument **index** représente l'indice du composant dans le panneau, le premier indice étant 0.

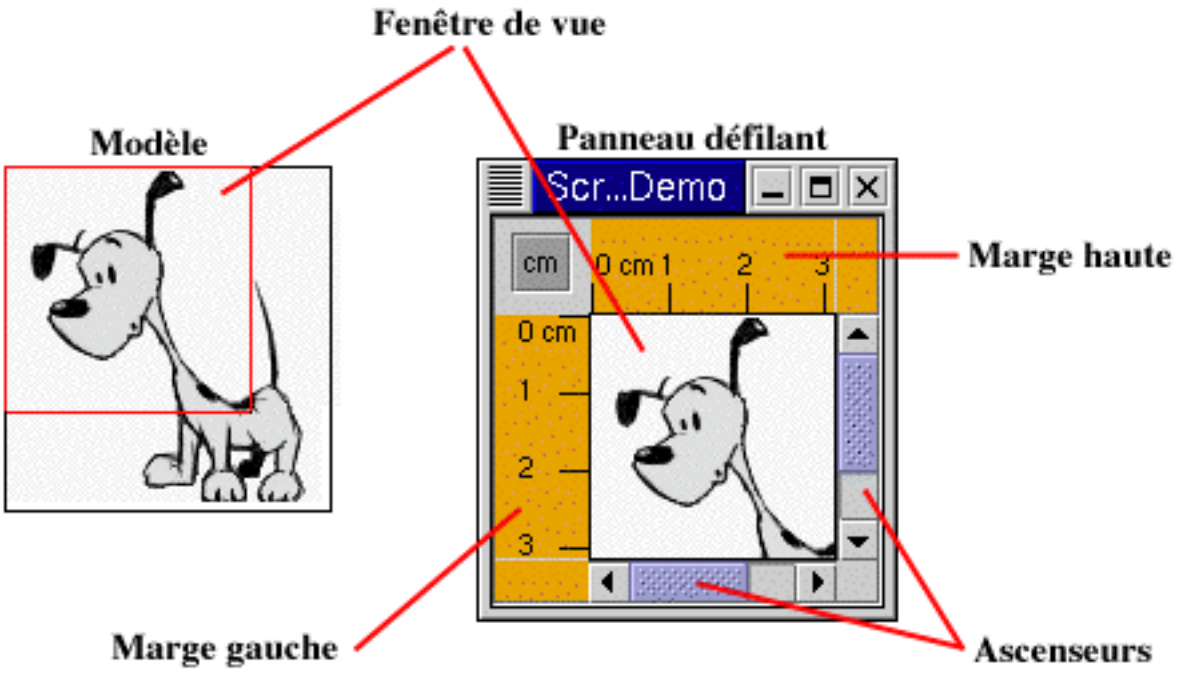
**void remove(int index)**
**void remove(Component comp)**
**void removeAll()**
Supprime le composant **comp**, ou d'indice **index** du conteneur.

## Gestionnaire de mise en page d'un panneau

**void setLayout(LayoutManager mgr)**
**LayoutManager getLayout()**
Fixe ou retourne le gestionnaire de mise en page (manager du panneau).

# Panneaux défilants (JScrollPane)

Ce sont des *conteneurs intermédiaires* (qui peuvent donc s'intégrer dans d'autres conteneurs et contenir des composants) destinés à la visualisation de composants de grande taille. Pour cet usage, ils peuvent posséder deux barres de défilement, verticale et/ou horizontale, permettant de déplacer le composant dans la fenêtre de visualisation.




## Création d'un panneau et gestion des barres de défilement

À proprement parler, il n'y a rien à faire pour gérer les barres de défilement. Elles s'ajustent automatiquement à la taille de la fenêtre de visualisation et celle du composant situé dedans. Inversement, la mise à jour du contenu de la fenêtre de visualisation est assurée automatiquement lorsque l'utilisateur déplace les curseurs des barres.
La gestion des barres de défilement se limite donc principalement à fixer la politique d'apparition de ces barres. Les arguments **hsbPolicy** et **vsbPolicy** des méthodes ci-dessous peuvent prendre respectivement les valeurs suivantes :

Pour la barre verticale (**vsbPolicy**) :

VERTICAL_SCROLLBAR_AS_NEEDED la barre apparaît/disparaît selon les besoins
VERTICAL_SCROLLBAR_NEVER la barre n'apparaît jamais
VERTICAL_SCROLLBAR_ALWAYS la barre est constamment présente

Pour la barre horizontale (**hsbPolicy**) :

HORIZONTAL_SCROLLBAR_AS_NEEDED la barre apparaît/disparaît selon les besoins
HORIZONTAL_SCROLLBAR_NEVER la barre n'apparaît jamais
HORIZONTAL_SCROLLBAR_ALWAYS la barre est constamment présente

**JScrollPane()**
**JScrollPane(Component view)**
**JScrollPane(Component view, int vsbPolicy, int hsbPolicy)**
**JScrollPane(int vsbPolicy, int hsbPolicy)**
Crée un panneau défilant dont le client est view, et aux politiques de gestion des barres de défilement spécifiées.

**void setViewportView(Component view)**
Fixe le client du panneau, i.e. le composant à placer dans la zone utile.

**void setHorizontalScrollBarPolicy(int hsbPolicy)**
**int getHorizontalScrollBarPolicy()**
Fixe ou récupère la politique de gestion de la barre de défilement horizontale.

**int getVerticalScrollBarPolicy()**
**void setVerticalScrollBarPolicy(int bspPolicy)**
Fixe ou récupère la politique de gestion de la barre de défilement verticale.

# Décoration d'un panneau défilant

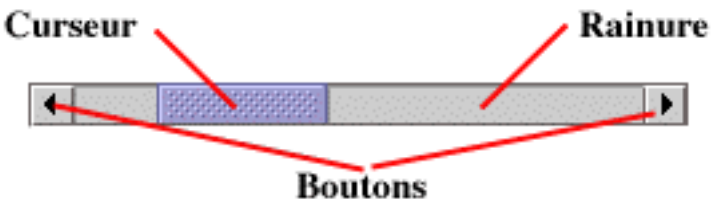
Le panneau défilant possède six zones personnalisables, quatre coins et deux marges, horizontale et verticale. La marge horizontale peut recevoir n'importe quel composant (une règle par exemple), qui sera déplacé horizontalement lors des actions sur la barre horizontale. Le même comportement s'applique à la marge verticale.

**void setColumnHeaderView(Component view)**
**void setRowHeaderView(Component view)**
Fixe le composant à placer dans la marge verticale ou horizontale du panneau défilant.

**void setCorner(String key, Component corner)**
**Component getCorner(String key)**
Fixe ou récupère le composant situé dans le coin défini par l'argument key. Ce dernier peut prendre l'une des valeurs suivantes : LOWER_LEFT_CORNER , LOWER_RIGHT_CORNER , UPPER_LEFT_CORNER , UPPER_RIGHT_CORNER

# Implémenter l'interface Scrollable

Il est possible de personnaliser la façon dont un composant client va interagir avec le panneau défilant qui le contient si le composant implémente d'interface Scrollable. L'objet précise alors de combien de pixels il doit être déplacé lorsque l'utilisateur clique sur les boutons ou dans la rainure de la barre, ainsi que la taille de la fenêtre de visualisation.




**Dimension getPreferredScrollableViewportSize()**
Doit retourner la taille de la fenêtre de vue souhaitée.

**int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)**
**int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)**
Doit retourner le nombre de pixels à déplacer lorsque l'utilisateur clique respectivement sur un bouton ou dans la rainure de la barre.
Arguments :
**visibleRect** : définit la zone actuellement visible du composant,
**orientation** : est soit SwingConstants.VERTICAL ou SwingConstants.HORIZONTAL suivant la barre sollicitée,
**direction** : une valeur positive pour un déplacement vers le bas/droite, et négative pour un déplacement vers le haut/gauche.

**boolean getScrollableTracksViewportWidth()**
**boolean getScrollableTracksViewportHeight()**
Doit retourner true si le panneau défilant doit forcer le composant client à entrer en totalité dans la fenêtre de vue, i.e. le composant doit alors ajuster sa taille lorsque le panneau est redimensionné.

# Méthodes d'autres classes en rapport avec les panneaux défilants

**void scrollRectToVisible(Rectangle aRect)**, dans **JComponent**
Provoque le défilement du panneau de manière à rendre le rectangle **aRect** visible dans la fenêtre de visualisation.

**void setAutoscrolls(boolean autoscrolls)**, dans **JComponent**
**boolean getAutoscrolls()**
Fixe ou récupère si le composant peut être déplacé directement en cliquant et en glissant le curseur dans la fenêtre de visualisation, i.e. sans passer par les barres de défilement.

**void setVisibleRowCount(int visibleRowCount),** dans **JList**
**int getVisibleRowCount()**
Fixe ou récupère combien de lignes de la liste sont visibles dans la fenêtre de visualisation.

**void ensureIndexIsVisible(int index)**, dans **JList**
Provoque le défilement de manière à faire apparaître dans la fenêtre de visualisation l'élément d'indice **index** de la liste . **void setVisibleRowCount(int newCount),** dans **JTree**
**int getVisibleRowCount()**
Fixe ou récupère combien de lignes de l'arbre sont visibles dans la fenêtre de visualisation.

**void scrollPathToVisible(TreePath path),** dans **JTree**
**void scrollRowToVisible(int row)**

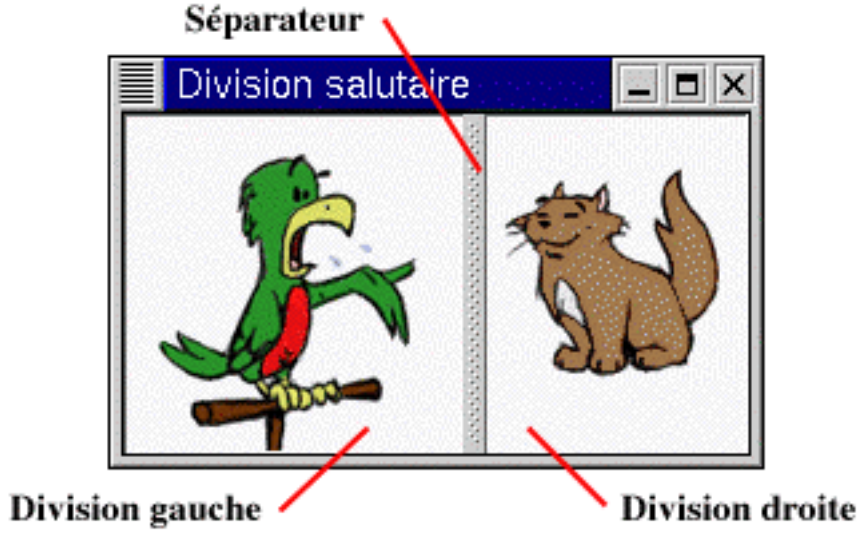
Provoque le défilement de manière à faire apparaître l'élément identifié par **path** ou **row** .

**void setScrollsOnExpand(boolean newValue),** dans **JTree**
**boolean getScrollsOnExpand()**
Fixe ou récupère si le défilement doit s'effectuer automatiquement lorsqu'un noeud de l'arbre est étendu.

**void setPreferredScrollableViewportSize(Dimension size),** dans **JTable**
Fixe la taille préférée de la fenêtre de visualisation de la table

# Panneaux divisés (JSplitPane)

Un panneau divisé est un composant intermédiaire comportant deux surfaces d'affichage distinctes, séparées par une barre horizontale ou verticale.




## Création d'un panneau divisé, fixer ses propriétés

**JSplitPane()**
**JSplitPane(int newOrientation)**
**JSplitPane(int newOrientation, boolean newContinuousLayout)**
**JSplitPane(int newOrientation, Component newLeftComponent, Component newRightComponent)**
**JSplitPane(int newOrientation, boolean newContinuousLayout, Component newLeftComponent, Component newRightComponent)**
Crée un panneau divisé de manière indiquée par **newOrientation** (JSplitPane.HORIZONTAL_SPLIT ou JSplitPane.VERTICAL_SPLIT). **newContinuousLayout** indique si les composants doivent être redessinés en continu lorsque l'utilisateur déplace la barre de division. **newLeftComponent** et **newRightComponent** spécifient les composants à placer dans les parties gauche et droite, ou haute et basse.

**void setOrientation(int orientation)**
**int getOrientation()**
Fixe ou retourne l'orientation de la division :
JSplitPane.VERTICAL_SPLIT : division gauche/droite
JSplitPane.HORIZONTAL_SPLIT : division haut/bas

**void setDividerSize(int newSize)**
**int getDividerSize()**
Fixe ou retourne la taille en pixels de la barre de division.

**void setContinuousLayout(boolean newContinuousLayout)**
**boolean isContinuousLayout()**
Fixe ou retourne si les composants situés dans les deux parties du panneau doivent être redessinés en continu lorsque le séparateur est déplacé.

**void setOneTouchExpandable(boolean newValue)**
**boolean isOneTouchExpandable()**
Fixe ou retourne si la barre de division comporte deux contrôles pour fermer d'un clic l'une ou l'autre des divisions, l'autre occupant alors la totalité du panneau.

## Gestion du contenu du panneau divisé

**void setTopComponent(Component comp)**
**void setBottomComponent(Component comp)**
**void setLeftComponent(Component comp)**
**void setRightComponent(Component comp)**
**Component getLeftComponent()**
**Component getRightComponent()**
**Component getTopComponent()**
**Component getBottomComponent()**
Fixe ou retourne le composant de la division indiquée.

**void remove(int index)**
**void removeAll()**
Supprime l'un l'autre ou les deux composants du panneau. L'argument index vaut 1 pour le composant gauche/haut, et 2 pour le composant droit/bas.

**Component add(Component comp)**
Ajoute le composant au panneau divisé. Le premier composant sera placé dans la partie haute/gauche, le suivant dans la partie droite/basse. L'ajout de plus de deux composants provoque une exception.

## Positionner le séparateur

Par défaut le panneau divisé est dimensionné en tenant compte des tailles souhaitées des composants (javax.swing.JComponent.setPreferredSize()). L'utilisateur peut déplacer interactivement le séparateur, mais il ne lui sera pas possible de faire descendre la taille d'une des deux cellules en dessous de la taille souhaitée du composant qu'elle contient (il est possible d'y parvenir en redimensionnant la fenêtre).

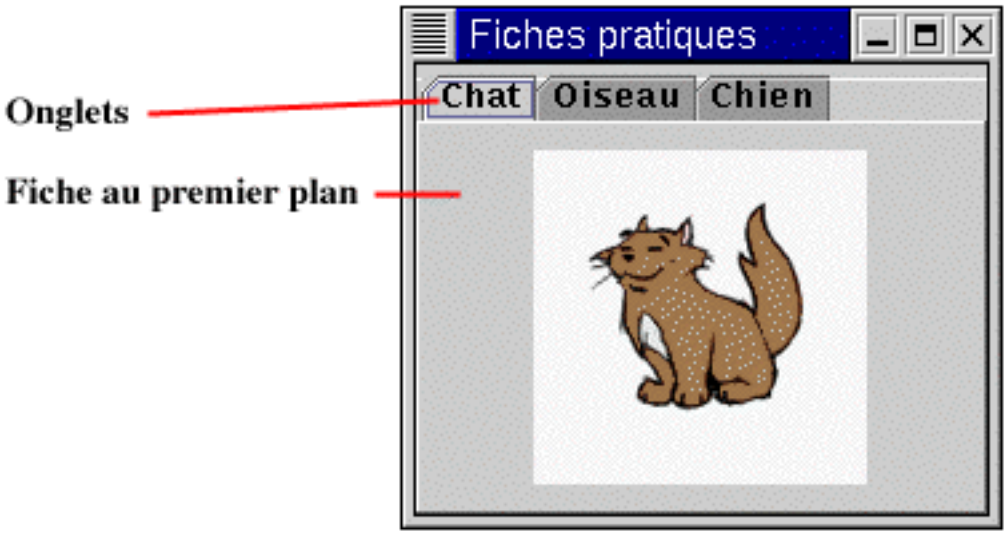
**void setDividerLocation(int location)**
**void setDividerLocation(double proportionalLocation)**
**int getDividerLocation()**
Fixe ou retourne la position de la barre de séparation. Cette position peut être spécifiée sous forme de pourcentage (**proportionalLocation**), ou de nombre de pixels (**location**).

**void resetToPreferredSizes()**
Positionne la barre de séparation de manière à ce que les composants contenus dans les deux cellules soient à leur taille souhaitée.

**void setLastDividerLocation(int newLastLocation)**
**int getLastDividerLocation()**
Fixe ou retourne la dernière position de la barre de séparation, i.e. le nombre de pixels qui sépare le bord du bas (gauche) du panneau du bord droit(haut) du séparateur.

# Panneaux à onglets (JTabbedPane)

Ce sont des panneaux qui peuvent comporter plusieurs fiches qui se partagent le même espace d'affichage. Chaque fiche possède un onglet qui permet de la faire passer au premier-plan.




# Création d'un panneau à onglets

**JTabbedPane()**
**JTabbedPane(int tabPlacement)**
Crée un panneau à onglets. L'argument **tabPlacement** permet de spécifier la position des onglets, les valeurs possibles sont TOP, BOTTOM, LEFT, ou RIGHT.

# Ajouter, supprimer trouver des fiches à onglets

Les différentes fiches du panneau peuvent être référencées par un indice, allant de 0 à **getTabCount()** -1.

**void addTab(String title, Component component)**
**void addTab(String title, Icon icon, Component component)**
**void addTab(String title, Icon icon, Component component, String tip)**
Ajoute une fiche à onglet au panneau. Les arguments **title** et **icon** spécifient le texte et l'icône à placer sur l'onglet, **component** est le composant à afficher lorsque l'onglet est sélectionné (i.e. contenu dans la fiche), **tip** est une chaine placée dans une bulle d'aide qui apparaît lorsque le pointeur de la souris passe sur l'onglet.

**void insertTab(String title, Icon icon, Component component, String tip, int index)**
Insère une nouvelle fiche à la position spécifiée par **index**. Les autres arguments sont identiques à **addTab().**

**void remove(Component component)**
**void removeTabAt(int index)**
Supprime la fiche du panneau qui possède l'indice **index**, ou contient le composant **component**. **void removeAll()**
Supprime toutes les fiches du panneau.

**int indexOfComponent(Component component)**
**int indexOfTab(String title)**
**int indexOfTab(Icon icon)**
Retourne d'indice de la fiche qui comporte le composant, titre ou icône spécifié.

**int getTabCount()**

Retourne le nombre de fiches présentes dans le panneau.

## Gérer l'apparence et le contenu des fiches

**void setComponentAt(int index, Component component)**
**Component getComponentAt(int index)**
Fixe ou récupère le composant associé à la fiche d'indice **index**.

**void setTitleAt(int index, String title)**
**void setIconAt(int index, Icon icon)**
**void setDisabledIconAt(int index, Icon disabledIcon)**
**String getTitleAt(int index)**
**Icon getIconAt(int index)**
**Icon getDisabledIconAt(int index)**
Fixe ou récupère le titre, l'icône ou l'icône d'apparence désactivée de l'onglet de la ficher d'indice **index**.

**void setForegroundAt(int index, Color foreground)**
**Color getForegroundAt(int index)**
**void setBackgroundAt(int index, Color background)**
**Color getBackgroundAt(int index)**
Fixe ou récupère les couleurs de premier et d'arrière plan de la fiche d'indice **index**.

**void setEnabledAt(int index, boolean enabled)**
**boolean isEnabledAt(int index)**
Fixe ou retourne si la fiche d'indice index est activée (sélectionnable) ou non.

# Barres d'outils (JToolBar)

Les barres d'outils sont des conteneurs intermédiaires destinés à regrouper des composants, généralement des boutons comportant des icônes, disposés en ligne (ou en colonne suivant la position de la barre). Cette barre peut être déplacée interactivement sur l'un des quatre bords de la fenêtre à laquelle elle est associée. Lorsqu'elle en est sortie, elle crée sa propre fenêtre, on se retrouve alors avec une palette d'outils flottante.
**Important :** de manière à fonctionner correctement, la barre d'outils doit être disposée dans un conteneur qui utilise le gestionnaire de mise en page BorderLayout, et être le seul autre composant contenu dans ce conteneur, l'autre composant étant placé au centre.

## Créer une barre d'outils, ajouter des composants, etc...

**JToolBar()**
**JToolBar(int orientation)**
Crée une barre d'outils, dont l'orientation initiale est éventuellement spécifiée (les valeurs valides sont HORIZONTAL/VERTICAL).

**JButton add(Action a)**
**Component add(Component comp)**
Ajoute le composant à la barre d'outils. Si le composant est un objet de la classe Action, la barre crée automatiquement un bouton et l'ajoute.

**void addSeparator()**
Ajoute un séparateur en fin de barre.

**void setFloatable(boolean b)**
**boolean isFloatable()**
Fixe ou permet de savoir s'il est possible de sortir la barre dans une fenêtre séparée pour créer une palette flottante.

# Boutons, boutons-radio, et cases à cocher.

Les trois grands types de boutons utilisables en Swing sont les boutons classiques (JButton), les boutons radio (JRadioButton) et les cases à cocher (JCheckBox). Tous héritent de la classe javax.swing.AbstractButton, qui fournit un grand nombre de méthodes communes pour personnaliser leur apparence.
En swing un bouton peut comporter du texte, et/ou une icône ou un en ensemble d'icônes destinées à représenter les différents états du bouton :

activé ou désactivé : définit si l'utilisateur peut cliquer dessus ;
pressé/sélectionné ou relaché/non sélectionné.

Swing comporte en outre la gestion du rollover, qui permet de modifier l'apparence (l'icône) du bouton lorsque le pointeur de la souris passe dessus.


## Méthodes communes aux boutons, boutons-radio et cases à cocher

### Définir le texte et les icônes

**void setText(String text)**
**String getText()**
Fixe ou récupère le texte affiché sur le bouton

**void setIcon(Icon defaultIcon)**
**Icon getIcon()**
Fixe ou retourne l'icône affichée sur le bouton lorsqu'il n'est pas pressé ou sélectionné.

**void setDisabledIcon(Icon defaultIcon)**
**Icon getDisabledIcon()**
Fixe ou retourne l'icône affichée sur le bouton lorsqu'il est désactivé, i.e. non cliquable.

**void setPressedIcon(Icon defaultIcon)**
**Icon getPressedIcon()**
Fixe ou retourne l'icône affichée sur le bouton lorsqu'il est pressé.

**void setSelectedIcon(Icon defaultIcon)**
**Icon getSelectedIcon()**
**void setDisabledSelectedIcon(Icon defaultIcon)**
**Icon getDisabledSelectedIcon()**
Fixe ou retourne l'icône affichée sur le bouton lorsqu'il est sélectionné, lorsqu'il est activé ou désactivé.

**boolean isRolloverEnabled()**
**void setRolloverEnabled(boolean b)**
Permet de tester ou d'activer/désactiver (**b**=true/false) l'effet de rollover.

**Icon getRolloverIcon()**
**void setRolloverIcon(Icon rolloverIcon)**

**Icon getRolloverSelectedIcon()**
**void setRolloverSelectedIcon(Icon rolloverSelectedIcon)**
Permet de fixer ou de récupérer l'icône affichée par un bouton sélectionné ou non, lorsque le curseur passe dessus (effet de rollover).

### Ajuster l'apparence des boutons

**int getHorizontalAlignment()**
**void setHorizontalAlignment(int alignment)**
Fixe ou retourne la politique de positionnement horizontal des éléments contenus dans le bouton.
Valeurs possibles d'**alignment** : LEFT, CENTER, RIGHT

**int getVerticalAlignment()**
**void setVerticalAlignment(int alignment)**
Fixe ou retourne la politique de positionnement vertical des éléments contenus dans le bouton. Valeurs possibles d'**alignment** : TOP, CENTER, BOTTOM.

**int getVerticalTextPosition()**
**void setVerticalTextPosition(int textPosition)**
**int getHorizontalTextPosition()**
**void setHorizontalTextPosition(int textPosition)**
Fixe ou retourne la position dans le bouton du texte relativement à celle de l'icône.
Valeurs possibles de **textPosition** : horizontalement, RIGHT, LEFT, CENTER, LEADING, TRAILING verticalement, TOP, CENTER, BOTTOM.

**void setMargin(Insets m)**
**Insets getMargin()**
Fixe ou retourne le nombre de pixels à placer entre le bord du bouton et son contenu.

**void setFocusPainted(boolean b)**
**boolean isFocusPainted()**
Fixe ou permet de savoir si le bouton doit être peint de façon différente lorsqu'il a le focus.

**void setBorderPainted(boolean b)**
**boolean isBorderPainted()**
Fixe ou permet de savoir si le bord du bouton doit être peint.

**boolean isContentAreaFilled()**
**void setContentAreaFilled(boolean b)**
Fixe ou permet de savoir si le fond du bouton doit être rempli lorsque l'on clique dessus.
**Note** : l'effet de cette méthode est susceptible de varier d'un composant à un autre et d'un Look&Feel à un autre.

## Implémenter la fonctionnalité des boutons

**void setMnemonic(int mnemonic)**
**int getMnemonic()**
Fixe ou retourne l'équivalent clavier du bouton. Utilisez les entiers statiques de la forme VK_X définis dans java.awt.event.KeyEvent.
**Note** : il existe également une méthode setMnemonic qui prend un argument de type char, son utilisation est déconseillée.

**void setActionCommand(String actionCommand)**
**String getActionCommand()**
Fixe ou retourne le nom de l'action effectuée par le bouton.

**void addActionListener(ActionListener l)**
**void removeActionListener(ActionListener l)**

Ajoute ou retire un écouteur d'événement d'action du bouton.

**void addItemListener(ItemListener l)**
**void removeItemListener(ItemListener l)**
Ajoute ou retire l'écouteur d'événement d'item du bouton.

**boolean isSelected()**
**void setSelected(boolean b)**
Fixe ou retourne l'état du bouton.

## Création des boutons ordinaires JButton

**JButton()**
**JButton(String text)**
**JButton(Icon icon)**
**JButton(String text, Icon icon)**
Crée un bouton, comportant éventuellement du texte et/ou une icône.

## Création des boutons radios

**JRadioButton()**
**JRadioButton(Icon icon)**
**JRadioButton(Icon icon, boolean selected)**
**JRadioButton(String text)**
**JRadioButton(String text, boolean selected)**
**JRadioButton(String text, Icon icon)**
**JRadioButton(String text, Icon icon, boolean selected)**
Crée un bouton radio comportant éventuellement un texte. Lorsque **icon** est spécifié, l'icône en question remplace le dessin habituel du bouton. Le paramètre **selected** précise s'il est coché, par défaut il ne l'est pas.

## Création des cases à cocher

**JCheckBox()**
**JCheckBox(Icon icon)**
**JCheckBox(Icon icon, boolean selected)**
**JCheckBox(String text)**
**JCheckBox(String text, boolean selected)**
**JCheckBox(String text, Icon icon)**
**JCheckBox(String text, Icon icon, boolean selected)**
Crée une case à cocher comportant éventuellement un texte. Lorsque **icon** est spécifié, l'icône en question remplace le dessin habituel de la case. Le paramètre **selected** précise si elle est cochée, par défaut elle ne l'est pas.

## Regroupement mutuellement exclusif de boutons radio

**ButtonGroup()**
Crée un nouveau groupe de boutons/boutons-radio mutuellement exclusifs.

**void add(AbstractButton b)**
**void remove(AbstractButton b)**
Ajoute ou supprime un bouton/bouton-radio d'un groupe.

# Etiquettes (JLabel)

Une étiquette est un capable d'afficher du texte, une icône, ou les deux ensemble. Une étiquette n'est pas capable de réagir aux événements utilisateur, elle ne peut donc pas obtenir le focus d'entrée. Par contre elle peut afficher un caractère ayant l'apparence d'un raccourci clavier, pour assister un autre composant qui ne serait pas capable de le faire.

# Création et initialisation du contenu d'un label

**JLabel()**
**JLabel(Icon image)**
**JLabel(Icon image, int horizontalAlignment)**
**JLabel(String text)**
**JLabel(String text, Icon icon, int horizontalAlignment)**
**JLabel(String text, int horizontalAlignment)**
Crée une instance de JLabel comportant éventuellement le texte ou l'image spécifiée. Il est possible de spécifier la politique d'alignement texte/image à l'aide de l'argument **horizontalAlignment,** dont les valeurs possibles sont définies dans l'interface SwingConstants : LEFT, CENTER, RIGHT, LEADING, TRAILING.

**void setText(String text)**
**String getText()**
Fixe ou retourne le texte affiché par l'étiquette.

**void setIcon(Icon icon)**
**Icon getIcon()**
Fixe ou retourne l'image (icône par défaut) affichée par l'étiquette.

**void setDisplayedMnemonic(int key)**
**int getDisplayedMnemonic()**
Fixe ou retourne la lettre que l'étiquette doit afficher comme un raccourci clavier? **void setDisabledIcon(Icon disabledIcon)**
**Icon getDisabledIcon()**
Fixe ou retourne l'icône à afficher lorsque l'étiquette est désactivéé. Lorsqu'elle n'est pas spécifiée, le Look&Feel employé la calcule à partir de l'icône par défaut du label.

## Ajuster l'apparence des labels

**int getHorizontalAlignment()**
**void setHorizontalAlignment(int alignment)**
Fixe ou retourne la politique de positionnement horizontal des éléments contenus dans l'étiquette. Valeurs possibles d'alignment : LEFT (valeur par défaut pour un texte seul), CENTER(valeur par défaut pour une icône seule), RIGHT, LEADING, TRAILING.

**int getVerticalAlignment()**
**void setVerticalAlignment(int alignment)**
Fixe ou retourne la politique de positionnement vertical des éléments contenus dans l'étiquette. Valeurs possibles d'alignement : TOP, CENTER, BOTTOM.

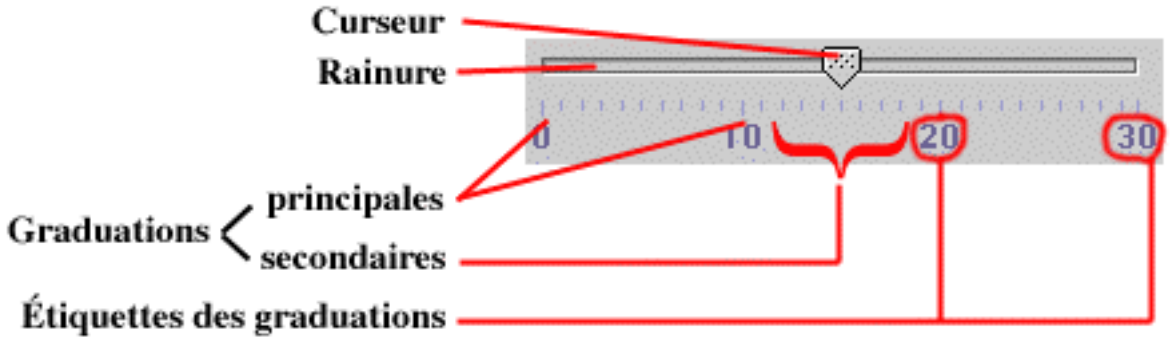
**int getVerticalTextPosition()**
**void setVerticalTextPosition(int textPosition)**
**int getHorizontalTextPosition()**
**void setHorizontalTextPosition(int textPosition)**
Fixe ou retourne la position dans l'étiquette du texte relativement à celle de l'icône.
Valeurs possibles de textPosition : horizontalement, RIGHT (par défaut), LEFT, CENTER, verticalement, TOP, CENTER (par défaut), BOTTOM.

**void setIconTextGap(int iconTextGap)**

**int getIconTextGap()**
Fixe ou retourne le nombre de pixels qui séparent le texte de l'image dans l'étiquette.

# Curseurs coulissants (JSlider)

Les curseurs permettent d'entrer une valeur numérique bornée par une valeur min et une valeur max. Par rapport aux champs de saisie de type texte, cette contrainte permet d'éliminer les erreurs de saisie.




## Création d'un curseur coulissant

**JSlider()**
**JSlider(int orientation)**
**JSlider(int min, int max)**
**JSlider(int min, int max, int value)**
**JSlider(int orientation, int min, int max, int value)**
Crée un curseur dont les bornes min et max la valeur courante initiale et l'orientation (JSlider.HORIZONTAL ou JSlider.VERTICAL) peuvent éventuellement être spécifiées parmis les valeurs . En l'absence de spécifications le curseur sera par défaut horizontal, les bornes min et max sont 0 et 100, et la valeur courante initiale 50.

## Personnaliser l'apparence d'un curseur coulissant

**void setValue(int n)**
**int getValue()**
Fixe ou retourne la valeur courante du curseur.

**void setMinimum(int minimum)**
**int getMinimum()**
**void setMaximum(int maximum)**
**int getMaximum()**
Fixe ou retourne la borne min ou max du curseur.

**void setOrientation(int orientation)**
**int getOrientation()**
Fixe ou retourne l'orientation (JSlider.HORIZONTAL ou JSlider.VERTICAL) du curseur.

**void setInverted(boolean b)**
**boolean getInverted()**
Fixe ou retourne si les bornes min et max du curseur sont inversées.

**void setMajorTickSpacing(int n)**
**int getMajorTickSpacing()**

Fixe ou retourne l'espace entre les graduations principales.

**void setMinorTickSpacing(int n)**
**int getMinorTickSpacing()**
Fixe ou retourne l'espace entre les graduations secondaires.

**void setPaintTicks(boolean b)**
**boolean getPaintTicks()**
Fixe ou retourne si les graduations doivent être peintes.

**void setPaintLabels(boolean b)**
**boolean getPaintLabels()**
Fixe ou retourne si les étiquettes des graduations doivent être peintes.

**void setLabelTable(Dictionary labels)**
**Dictionary getLabelTable()**
Permet de fournir un ensemble d'étiquettes personnalisées.

**Hashtable createStandardLabels(int increment)**
**Hashtable createStandardLabels(int increment, int start)**
Crée une table qui permettra au curseur créer automatiquement et d'afficher les étiquettes standard, calculées à partir de la borne min du curseur ou de l'argument **start**, et incrémentées de **increment.**

## Gérer les déplacements du curseur

**void addChangeListener(ChangeListener l)**
Associe un écouteur de changements au curseur.

**boolean getValueIsAdjusting()**
Retourne si l'utilisateur a terminé de déplacer le curseur.

# Listes (JList)

Les listes JList présentent un ensemble d'items les uns au dessus des autres. L'utilisateur peut ensuite sélectionner, suivant le cas, un ou plusieurs éléments dans la liste. Etant donné qu'une liste peut être plus grande que ce qu'il est possible d'afficher à l'écran, elles sont souvent placées dans la surface utile d'objets de classe JScrollPane pour leur associer un ascenseur.

Tout comme les listes de choix, elle peuvent être personnalisées pour afficher n'importe quel type d'objet, mais dans ce cas il est nécessaire de fournir le module de rendu approprié. Par défaut, elles peuvent afficher du texte et des icônes.

## Création et remplissage de listes statiques

**JList(Object[] listData)**
**JList(Vector listData)**
Crée une liste contenant les données éventuellement spécifiées.
**Note** : les deux derniers constructeurs créent une statique.

**public void setListData(Object[] listData)**
**public void setListData(Vector listData)**
Permet de fixer le contenu d'un objet JList à partir d'un tableau ou d'un vecteur d'objets.

## Création de listes dynamiques

Les listes sont associées à un *modèle de liste* (interface ListModel et classe DefaultListModel), chargé de stocker le contenu de la liste. La création de listes dynamiques, i.e. modifiables après création, passe par l'instanciation séparée d'un modèle de liste, associé à une liste. Les modifications s'effectuent ensuite sur le modèle, les répercutions sur l'apparence de la liste se faisant alors automatiquement.

**JList()**
**JList(ListModel dataModel)**
Crée une liste dynamique (i.e. modifiable après création).

**public void setModel(ListModel model)**
**public ListModel getModel()**
Fixe ou récupère le modèle qui maintient le contenu de la liste.

**public DefaultListModel()**
Le constructeur de la classe javax.swing.DefaultListModel, crée une modèle de liste.

# Modification de listes dynamiques (classe DefaultListModel)

**Quelques méthodes utiles fournissant des informations sur le modèle public int size()**
Retourne la taille du modèle de liste

**public boolean isEmpty()**
Retourne si le modèle de liste est vide

**public Object get(int index)**
Retourne l'objet situé en position **index** du modèle de liste.
Génère une exception ArrayIndexOutOfBoundsException si index n'est pas valide (**index** < 0 ou **index** >= size()).

**Ajouter des éléments**
Toutes ces méthodes génèrent une exception ArrayIndexOutOfBoundsException si l'argument **index** n'est pas valide (**index** < 0 ou **index** >= size()).

**public void addElement(Object obj)**
Ajoute un élément en fin du modèle de liste.

**public void add(int index, Object element)**
Ajoute un élément dans le modèle de liste, en position **index**.

**Supprimer ou remplacer des éléments**
Toutes ces méthodes génèrent une exception ArrayIndexOutOfBoundsException si l'argument **index** n'est pas valide (**index** < 0 ou **index** >= size()).

**public Object remove(int index)**
Supprime l'objet situé en position index du modèle de liste.

**public void removeRange(int fromIndex, int toIndex)**
Supprime les éléments du modèle de liste situés entre **fromIndex** et **toIndex**.
Génère en plus une exception IllegalArgumentException si **fromIndex** > **toIndex**.

**public Object set(int index, Object element)**
Remplace l'objet du modèle de liste situé en position **index**.
Retourne l'objet remplacé, précédemment situé à cette position.

**public void clear()**
Supprime tous les éléments du modèle de liste.

# Gestion de la sélection utilisateur

**public void addListSelectionListener(ListSelectionListener listener)**
Ajoute un écouteur à la liste notifié à chaque fois que l'utilisateur modifie la sélection.

**public void setSelectedIndex(int index)**
**public void setSelectedIndices(int[] indices)**
**public void setSelectedValue(Object anObject, boolean shouldScroll)**
**public void setSelectionInterval(int anchor, int lead)**
Permet de fixer le ou les éléments sélectionnés dans une liste. L'argument **shouldScroll** précise si la liste doit être déplacée pour que les éléments sélectionnés soient visibles.

**public int getSelectedIndex()**
**public Object getSelectedValue()**
**public int[] getSelectedIndices()**
**public Object[] getSelectedValues()**
Permet de récupérer le ou les éléments (ou leurs indices dans la liste) sélectionnés.

# Gestion du mode de sélection

Par défaut il est possible de sélectionner n'importe quel ensemble d'éléments dans une liste. Il est possible de modifier ce comportement de façon à n'autoriser que la sélection d'un ensemble contigu d'éléments, voire d'un élément unique.
Cliquer dans la liste provoque la sélection d'un élément (et la désélection de ceux qui l'étaient auparavant). La combinaison SHIFT+CLIC permet d'étendre la sélection à un ensemble contigu et CTRL+CLIC permet les sélections/désélections quelconque d'éléments. Ces combinaisons de touches sont susceptibles de varier d'une plateforme à une autre.

Sélection unique Sélection contigüe Sélections quelconques

**void setSelectionMode(int selectionMode)**
**int getSelectionMode()**
Fixe ou récupère le mode de sélection autorisé des éléments de la liste. La valeur de **selectionMode** est l'une des suivante :
SINGLE_SELECTION : sélection unique
SINGLE_INTERVAL_SELECTION : sélection contigüe unique
MULTIPLE_INTERVAL_SELECTION : sélections quelconques

# Personnaliser l'apparence des listes

**void setSelectionBackground(Color selectionBackground)**
**Color getSelectionBackground()**
Fixe ou récupère la couleur de fond des sélections.

**void setSelectionForeground(Color selectionForeground)**

**Color getSelectionForeground()**
Fixe ou retourne la couleur d'avant plan des sélections.

# Listes de choix (JComboBox)

Les listes de choix existent en version non éditable (valeur par défaut) et éditables. Une liste de choix non éditable ressemble à un bouton qui affiche un menu de choix lorsque l'utilisateur clique dessus. Dans sa version éditable, le bouton prend l'apparence d'un champ de texte éditable, qui comporte un petit bouton permettant de faire apparaître le menu.

## Création d'une liste de choix

**JComboBox()**
**JComboBox(Object[] items)**
**JComboBox(Vector items)**
Crée une liste de choix comportant les items spécifiés.

**void addItem(Object anObject)**
**void insertItemAt(Object anObject, int index)**
Permet d'ajouter un objet à une liste de choix, en fin de liste, ou dans la position spécifiée (**index**).
**Object getItemAt(int index)**
**Object getSelectedItem()**
Récupère l'item positionné à l'index spécifié, ou l'item couramment sélectionné.

**int getItemCount()**
Récupère le nombre d'éléments dans la liste de choix.

**void removeAllItems()**
**void removeItemAt(int anIndex)**
**void removeItem(Object anObject)**
Permet de supprimer des items de la liste de choix.

## Personnaliser l'apparence de la liste de choix

**void addActionListener(ActionListener l)**
**void removeActionListener(ActionListener l)**
Ajoute ou retire l'écouteur d'action **l** à la liste de choix. La méthode **actionPerformed** de l'écouteur est invoquée lorsque l'utilisateur sélectionne un item dans la liste, ou, dans une liste éditable, lorsque l'utilisateur presse la touche retour.

**void addItemListener(ItemListener aListener)**
**void removeItemListener(ItemListener aListener)**
Ajoute ou retire l'écouteur dévénement d'item **aListener** de la liste. La méthode **itemStateChanged** est infoquée lorsque l'état d'un des items de la liste change.

**void setEditable(boolean aFlag)**
**boolean isEditable()**
Fixe ou permet de savoir si la liste de choix est éditable.

**void setRenderer(ListCellRenderer aRenderer)**
**ListCellRenderer getRenderer()**
Fixe ou récupère l'objet chargé d'effectuer l'affichage personnalisé des items de la liste.

**void setEditor(ComboBoxEditor anEditor)**
**ComboBoxEditor getEditor()**

Fixe ou récupère l'objet chargé d'assurer la saisie et l'affichage personnalisés dans une liste de choix éditable.

**void setMaximumRowCount(int count)**
**int getMaximumRowCount()**
Fixe ou retourne le nombre maximal d'items que la liste affiche simultanément. Si le nombre total d'items est supérieur, la liste est automatiquement munie d'un ascenseur.

# Champs de texte et de mots de passe (JTextField, JPasswordField)

Les champs de texte et de mots de passe sont des composants qui permettent la saisie d'une ligne unique de texte. La fin de la saisie est signifiée lorsque l'utilisateur presse la touche retour.
**Note** : le caractère correspondant à la touche retour n'est pas intégré dans le texte retourné par la saisie.

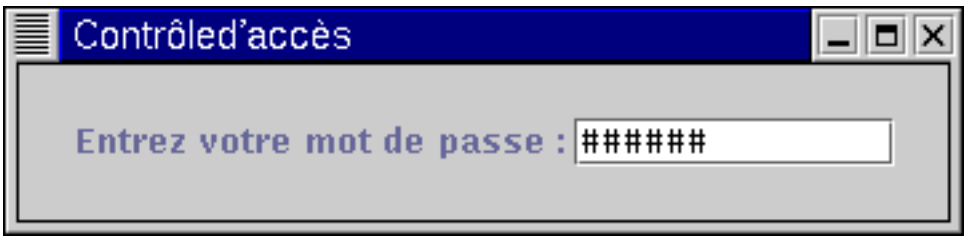
## Création de champs de texte, récupération du texte saisi (JTextField)

**JTextField()**
**JTextField(String text)**
**JTextField(int columns)**
**JTextField(String text,int columns)**
**JTextField(Document doc, String text, int columns)**
Crée un champ de saisie de texte de taille spécifiée par **columns**, comportant initialement le texte spécifié par l'argument **text.** L'argument **doc** fournit un document personnalisé pour le champ de saisie.

**void setText(String t)**
**String getText()**
Fixe le texte affiché dans le champ de saisie, ou retourne le texte entré par l'utilisateur.

## Création de champs de saisie de mots de passe, récupération du mot de passe saisi (JPasswordField)

Les champs de saisie de mots de passe sont similaires aux champs de texte simples, mais chaque caractère entrés au clavier est remplacé à l'affichage par un caractère spécial personnalisable et qui est par défaut '*'.




**JPasswordField()**
**JPasswordField(String text)**
**JPasswordField(int columns)**
**JPasswordField(String text, int columns)**
**JPasswordField(Document doc, String txt, int columns)**
Crée un champ de saisie de mot de passe de taille spécifiée par **columns**, comportant initialement le texte spécifié par l'argument **text.** L'argument **doc** fournit un document personnalisé pour le champ de

saisie.

**char[] getPassword()**
Retourne le mot de passe entré par l'utilisateur.

## Personnaliser l'apparence du champ de texte ou de mot de passe

**char getEchoChar(),** dans JPasswordField
**void setEchoChar(char c)**
Fixe ou récupère le caractère affiché à la place de ceux entrés par l'utilisateur.

**void setEditable(boolean b)**
**boolean isEditable()**
Fixe ou retourne si le champ de texte/mot de passe est éditable ou non.

**void setFont(Font f)**
Fixe la police employée.

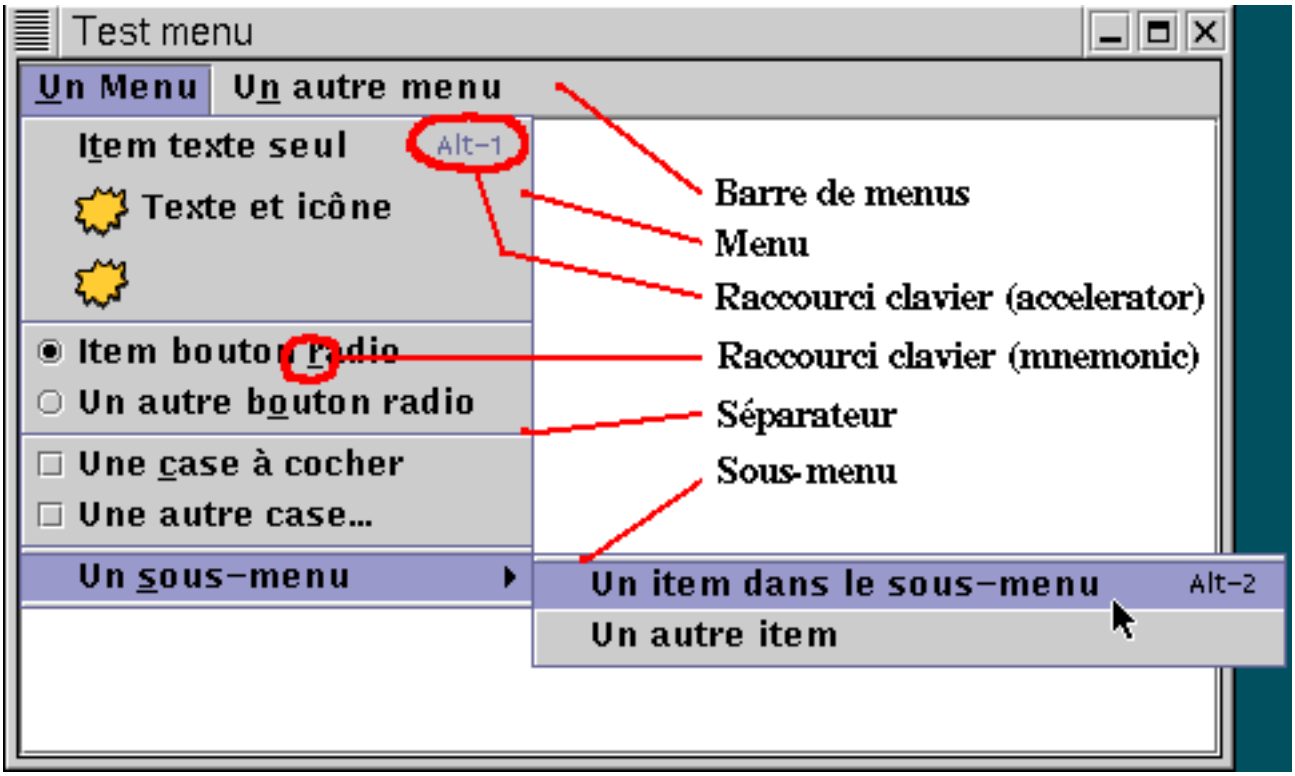
**void setHorizontalAlignment(int alignment)**
**int getHorizontalAlignment()**
Fixe ou retourne la façon dont le texte est aligné dans le champ d'édition. Les valeurs possibles sont JTextField.LEFT, JTextField.CENTER, et JTextField.RIGHT.

## Implémenter la fonctionnalité du champ de texte/mot de passe

**void addActionListener(ActionListener l)**
**void removeActionListener(ActionListener l)**
Ajoute ou retire l'écouteur d'événement d'action, généré lorsque l'utilisateur presse la touche retour.

# Menus (JMenuBar, JMenu, JPopupMenu, JMenuItem)

Les menus permettent de placer des outils à l'écran en économisant l'espace utilisé, puisqu'ils se referment automatiquement après utilisation. Ils permettent également de guider l'utilisateur en fournissant un aperçu des actions possibles dans l'application.
La barre de menu peut être disposée différemment suivant la plateforme utilisée.
En plus de texte, les items de menu peuvent comporter des icônes, être composés de boutons radios ou de cases à cocher.




# Créer une barre de menu (JMenuBar)

**JMenuBar()**
Crée une nouvelle barre de menus.

**JMenu add(JMenu c)**
Ajoute le menu **c** en fin de la barre.

**void setJMenuBar(JMenuBar menubar)**
**JMenuBar getJMenuBar()**
Dans JApplet, JDialog, JFrame, JInternalFrame, JRootPane.
Fixe ou retourne la barre de menu attachée à l'applet, le dialogue, le cadre, etc...

# Créer un menu (JMenu)

**JMenu()**
**JMenu(String s)**
Crée un menu, comportant éventuellement le texte spécifié.

**JMenuItem add(JMenuItem menuItem)**
**JMenuItem add(String s)**
**JMenuItem add(Action a)**
Ajoute un l'item passé en paramètre dans le menu.

**void addSeparator()**
Ajoute un séparateur en fin de menu.

**void insert(String s, int pos)**
**JMenuItem insert(JMenuItem mi, int pos)**
**JMenuItem insert(Action a, int pos)**
**void insertSeparator(int index)**
Insère un item ou un séparateur dans le menu à la position spécifiée. La première position possède l'indice 0.

**void remove(JMenuItem item)**
**void remove(int pos)**
**void removeAll()**
Supprime un ou tous les éléments du menu.

# Gestion des menus contextuels (JPopupMenu)

Les menus contextuels sont des menus qui apparaîssent sous le pointeur de la souris généralement lorsque l'utilisateur presse un bouton particulier de la souris, ou une combinaison touche/bouton de souris.
C'est une façon pratique d'avoir toujours des outils "sous la main".

**JPopupMenu()**
**JPopupMenu(String label)**
Crée un menu contextuel, comportant éventuellement le titre spécifié.

**JMenuItem add(JMenuItem menuItem)**
**JMenuItem add(String s)**
**JMenuItem add(Action a)**
Ajoute un item au menu contextuel.

**void addSeparator()**
Ajoute un séparateur au menu contextuel.

**void insert(Action a, int index)**
**void insert(Component component, int index)**
Insère le composant ou l'action indiqué à l'indice **index** (le premier indice possède le numéro 0).

**void remove(Component comp)**
**void remove(int index)**
**void removeAll()**
Supprime un ou tous les items contenus dans le menu contextuel.

**void setLightWeightPopupEnabled(boolean aFlag)**
Fixe le mode d'affichage du menu contextuel. Dans certaines configurations (lorsque l'on combine des composants Swing et AWT, ce qui n'est toutefois pas recommandé), il peut apparaître des problèmes d'affichage du menu. Dans ce cas invoquez la présente méthode avec **aFlag**=false.

**void show(Component invoker, int x, int y)**
Affiche le menu contextuel aux coordonnées **(x,y)** dans le repère local du composant **invoker**.

# Gestion des items des menus (JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem)

Les items de menus comportent des raccourcis claviers de deux types, appelés *Mnemonic* et *Accelerator*. Le premier permet de naviguer dans le menu à l'aide de commandes au clavier, tandis que le second permet de déclencher l'action associée à l'item directement sans l'afficher.

**JMenuItem()**
**JMenuItem(Icon icon)**
**JMenuItem(String text)**
**JMenuItem(String text, Icon icon)**
**JMenuItem(String text, int mnemonic)**
Crée un item de menu comportant le texte, l'icône et le raccourci clavier éventuellement spécifiés.

**JCheckBoxMenuItem()**

**JCheckBoxMenuItem(Icon icon)**
**JCheckBoxMenuItem(String text)**
**JCheckBoxMenuItem(String text, Icon icon)**
**JCheckBoxMenuItem(String text, boolean b)**
**JCheckBox(Icon icon, boolean selected)**
**JCheckBoxMenuItem(String text, Icon icon, boolean b)**
Crée un item de menu à l'apparence et à la fonctionnalité d'une case à cocher. L'argument **b** permet de spécifier si la case est initialement cochée.

**JRadioButtonMenuItem()**
**JRadioButtonMenuItem(Icon icon)**
**JRadioButtonMenuItem(String text)**
**JRadioButtonMenuItem(String text, Icon icon)**
**JRadioButtonMenuItem(String text, boolean b)**
**JRadioButtonMenuItem(Icon icon, boolean selected)**
**JRadioButtonMenuItem(String text, Icon icon, boolean selected)**
Crée un item de menu à l'apparence et à la fonctionnalité d'un bouton radio. L'argument **b** permet de spécifier si le bouton est initialement coché.

**void setState(boolean b),** dans JCheckBoxMenuItem
**boolean getState()**
Fixe ou retourne l'état (coché/pas coché) de la case.

**void setEnabled(boolean b)**
Fixe ou retourne si l'item est activé ou désactivé.

**void setMnemonic(char mnemonic)**
Fixe le raccourci clavier qui permet de naviguer jusqu'à cet item.

**void setAccelerator(KeyStroke keyStroke)**
Fixe le raccourci clavier qui permet d'activer cet item.

**void setActionCommand(String actionCommand)**
Fixe le nom de l'action executée par cet item.

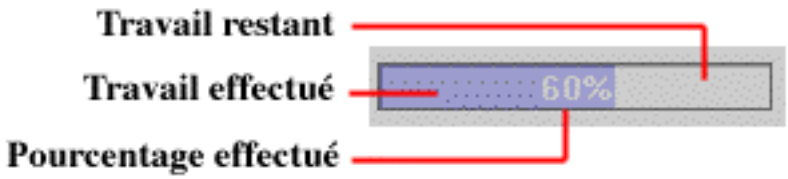
**void addActionListener(ActionListener l)**
**void addItemListener(ItemListener l)**
Ajoute un écouteur d'action ou d'item à l'item.

# Barres de Progression (JProgressBar)

Les barres de progression sont des composants atomiques, et peuvent donc se placer dans des conteneurs, voire à plusieurs. Elles permettent d'implémenter un aspect primordial d'une interface graphique : le retour d'information. Ces barres permettent en effet de donner à l'utilisateur une information sur le déroulement d'une tâche en cours, elles donnent trois informations :

la proportion de travail qui a été effectuée,
la proportion de travail restant/la quantité totale de travail,
la vitesse relative d'execution de la tâche.

Ces informations sont particulièrement appréciables lorsque l'opération en question est longue...




# Création d'une barre de progression

**JProgressBar()**
**JProgressBar(int orient)**
**JProgressBar(int min, int max)**
**JProgressBar(int orient,  int min, int max)**
**JProgressBar(BoundedRangeModel newModel)**
Crée une barre de progression, par défaut horizontale, mais il est possible de spécifier son orientation **orient** (JProgressBar.HORIZONTAL ou JProgressBar.VERTICAL). Les valeurs **min** (pas de travail effectué) et **max** (tâche complétée) définissent les bornes inférieure et supérieure de la barre de progression.

# Gérer la valeur et les contraintes de la barre de progression

**void setValue(int n)**
**int getValue()**
Fixe ou récupère la valeur courante de la barre, qui doit être située entre les bornes min et max de la barre.

**double getPercentComplete()**
Retourne le pourcentage de tâche complétée.

**void setMinimum(int n)**
**int getMinimum()**
**void setMaximum(int n)**
**int getMaximum()**
Fixe  ou récupère la borne min ou max de la barre.

# Régler l'apparence de la barre de progression

**void setOrientation(int newOrientation)**
**int getOrientation()**
Fixe ou récupère l'orientation de la barre. Les valeurs possibles sont JProgressBar.HORIZONTAL et JProgressBar.VERTICAL.

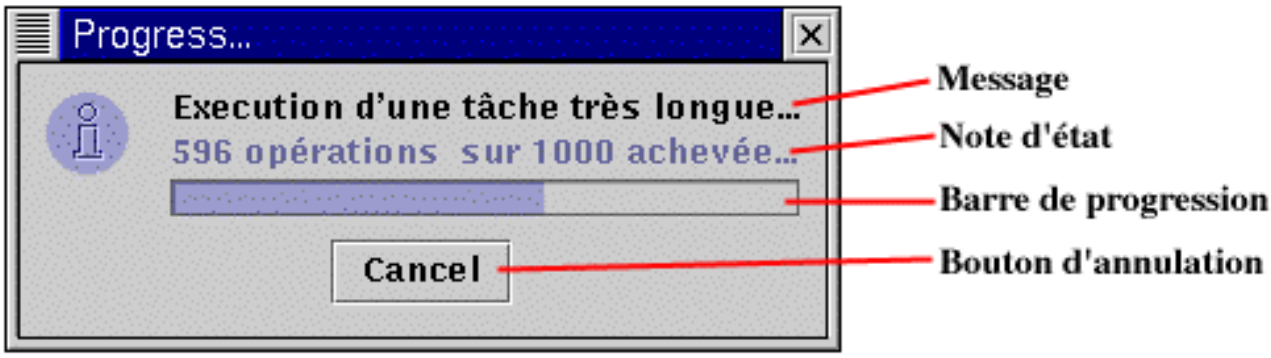
**void setBorderPainted(boolean b)**
**boolean isBorderPainted()**
Fixe ou retourne si la barre de progression possède un bord.

**boolean isStringPainted()**
**void setStringPainted(boolean b)**
Fixe ou retourne si la barre affiche un pourcentage de progression.

**void setString(String s)**
**String getString()**
Sixe ou récupère la chaîne qui personnalise l'affichage du pourcentage de progression.

# Création de dialogues de progression

Un dialogue de progression affiche une barre de progression dans une fenêtre de dialogue séparée. Il donne la possibilité à l'utilisateur d'interrompre l'execution de la tâche en cours.
Contrairement à une barre de progression simple, un dialogue de progression ne peut pas être réutilisé, il faut donc en instancier un nouveau à chaque utilisation.




**ProgressMonitor(Component parentComponent, Object message, String note, int min, int max)**
Crée un dialogue de progression rattaché au composant **parentComponent**, et qui comporte le **message**, la **note** d'état et les bornes **min** et **max** spécifiés.

**ProgressMonitorInputStream(Component parentComponent, Object message, InputStream in)**
Version particulière de dialogue de progression relié à un flux d'entrée.

## Configuration du dialogue de progression

**void setMinimum(int m)**
**int getMinimum()**
**void setMaximum(int m)**
**int getMaximum()**
Fixe ou retourne les bornes min et max de la barre de progression du dialogue.

**void setProgress(int nv)**
Met à jour la valeur courante d'accomplissement de la tâche (qui doit être située entre les bornes min et max). La mise à jour de l'affichage est assuré automatiquement si la différence avec la valeur précédente est visuellement significative.

**void setNote(String note)**
**String getNote()**
Fixe ou retourne la note d'état de progression. Permet d'afficher régulièrement des informations supplémentaires.

**void setMillisToDecideToPopup(int millisToDecideToPopup)**
**int getMillisToDecideToPopup()**
Fixe ou retourne le nombre de millisecondes entre la création du dialogue et son apparition. Le dialogue peut ainsi s'afficher un peu après le départ de la tâche, ou ne pas s'afficher du tout si elle est déjà terminée dans ce laps de temps.

## Fermer le dialogue de progression

**void close()**

Ferme le dialogue, ce qui est automatiquement effectué lorsque la valeur courante de progression est supérieure ou égale à la borne max.

**boolean isCanceled()**
Indique si l'utilisateur a cliqué sur le bouton Annuler, dans le but d'interrompre la tâche.

# Bulles d'aide (JToolTip)

Les bulles d'aide sont de petits messages qui s'affichent lorsque le pointeur de souris passe sur un composant. Ils permettent de fournir une information supplémentaire, et plus complète afin de guider l'utilisateur, en l'informant sur la fonction d'un composant ou en fournissant un commentaire pertinent sur une image par exemple...


## Création des bulles d'aide associées aux composants (dans JComponent)

La classe JToolTip est rarement employée directement, les composants héritent de la classe JComponent qui fournit les méthodes appropriées pour leur associer des bulles d'aide.

**void setToolTipText(String text)**
**String getToolTipText()**
Fixe ou retourne l'aide affichée lorsque le pointeur de souris passe sur un composant. La bulle d'aide est associée au composant. Pour désactiver l'affichage de l'aide sur un composant il suffit d'invoquer **setToolTipText(null)**.

# ICÔNES (Interface Icon et classe ImageIcon)

Certains composants tels JButton ou JLabel peuvent être personnalisés et afficher des icônes telles qu'elles sont définies dans l'interface Icon. La classe ImageIcon de Swing implémente cette interface et propose en outre des méthodes qui permettent de dessiner des icônes à partir d'un fichier image au format GIF ou JPEG, qu'il soit local ou situé sur un autre ordinateur.

# Créer une icône (ImageIcon), fixer son contenu

**ImageIcon()**
**ImageIcon(byte[] imageData)**
**ImageIcon(byte[] imageData, String description)**
**ImageIcon(Image image)**
**ImageIcon(Image image, String description)**
**ImageIcon(String filename)**
**ImageIcon(String filename, String description)**
**ImageIcon(URL location)**
**ImageIcon(URL location, String description)**
Crée une icône dont l'image est fournie dans un fichier local (**filename**), un fichier distant (**location**), une image (**image**) de la classe java.awt.Image, ou sous forme d'un tableau d'octets (**imageData**). Le paramètre **description** permet de spécifier un texte descriptif qui peut servir d'alternative à l'affichage de l'image.

**void paintIcon(Component c, Graphics g, int x, int y)**
Dessine le contenu de l'icône dans le contexte graphique spécifié **g.** Utilisé pour implémenter une classe d'icône personnalisée, dans ce cas cette méthode doit effectuer le tracé du contenu de l'icône. Le composant **c** fournit des informations supplémentaires pour le tracé, i.e. le composant est-il activé, sélectionné, etc. Le couple (**x**,**y**) fournit les coordonnées dans le composant où doit avoir lieu le tracé.

# Obtenir des informations sur le chargement de l'image de l'icône

**void setImageObserver(ImageObserver observer)**
**ImageObserver getImageObserver()**
Fixe ou retourne l'observateur d'image de l'icône.

**int getImageLoadStatus()**
Recupère l'information sur l'état de chargement de l'image. Les valeurs retournées sont définies dans java.awt.MediaTracker : ABORTED, LOADING, ERRORED, COMPLETE.