6.6.3. Flux objets

Les deux classes suivantes

```
public class ObjectOutputStream extends OutputStream
implements ObjectOutput, ObjectStreamConstants
  public ObjectOutputStream(OutputStream out) ...
  public void writeBoolean(boolean data) throws IOE
  ...
  public final void writeObject(Object obj) throws...
```

et

```
public class ObjectInputStream extends InputStream
implements ObjectInput, ObjectStreamConstants
  public ObjectInputStream(InputStream in) throws...
  public boolean readBoolean() throws IOException
  ...
  public final Object readObject()
   throws ClassNotFoundException, IOException
```

sont à utiliser avec des objets sérialisables (objets qui savent se représenter comme un flux d'octets).

Un objet extrêmement complexe, par exemple une fenêtre d'édition graphique, peut être *sérialisé* puis *désérialisé* très simplement

```
oos = new ObjectOutputStream(ostream);
oos.writeObject(fenetre);
...
ois = new ObjectInputStream(istream);
fenetre = (Fenetre)ois.readObject();
```

7. Les interfaces fenêtrées

7.1. Introduction

7.1.1. Définition

Nous connaissons tous les IHM à base de fenêtres et d'un outil de pointage appelé "souris". Tentons une définition très générale

> une fenêtre est une partie (rectangulaire) de l'écran destinée à présenter des informations visuelles à l'utilisateur et à recevoir des informations texte ou de pointage de celui-ci.

Cette définition peut être affinée et spécialisée dans une hiérarchie sans fin. Après les collections c'est le second exemple de hiérarchie naturelle de spécifications et de classes que nous rencontrons, mais cet exemple est bien plus foisonnant que le précédent.

En réalité cet exemple a joué un rôle fondamental pour le développement des langages OO.

7.1.2. Ebauche

Sans aller dans les détails, on sait que certaines fenêtres se spécialisent dans la présentation de certain type d'information :

> texte, graphismes
>
> structure de données : liste, arbre, tableau bidimensionnel
>
> le regroupement et organisation de sous-fenêtres
>
> facilitation d'un choix par pointage : boutons, menus.

Au total une telle hiérarchie peut compter quelques centaines de classes ou interfaces auxquelles il faut ajouter d'autres encore, non fenêtres, dont la hiérarchie de fenêtres a besoin pour fonctionner.

Dans cette hiérarchie il y a deux choses à apprendre :

> la partie "facile" : l'affichage d'information vers l'utilisateur
>
> la partie "tordue" : la réception d'information de l'utilisateur

Java offre deux implémentations imbriquées de hiérarchies fenêtrées : AWT et Swing. Nous allons étudier cette dernière.

7.2. Premier exemple

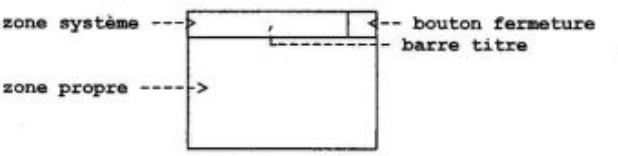
Réaliser en Java une application minimale à base de fenêtres n'est pas difficile. Voici un exemple

```
import javax.swing.*;
public class Main {
  public static void main(String[] args) {
    JFrame d = new Frame(); // création de la fenêtre
    d.setVisible(true); // rend la fenêtre visible
    d.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent we){
        System.exit(0);
      }
    });
  }
}
```

La classe Main contient un programme principal qui

> crée une fenêtre
>
> rend cette fenêtre visible à l'utilisateur (affichage d'information)
>
> et permet à l'utilisateur de la refermer (prise d'information).

Illustration



Dans une telle applications à base de fenêtres il ne se "passe" rien tant que l'utilisateur ne fournit rien (ni clic, ni frappe...). La fenêtre est suspendue à cette attente :

> on parle de programmation basée événements par opposition à une programmation procédurale.

Dans un programme procédural on peut suivre le fil de l'exécution tout au long du programme du début à la fin (ex : débogueur).

Dans un programme événementiel, suivons l'effet d'un clic sur un bouton.

l'utilisateur clique sur un bouton dans un programme

le système mémorise les coordonnées du clic dans un *événement*

et place l'événement dans une file d'attente pour traitement

le système recherche le programme et la fenêtre précise visés par le clic

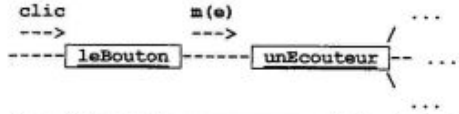
et invoque la méthode adéquate des *écouteurs* type *clic* attachés

Schéma :

Dans cette séquence nous pouvons oublier les étapes médians :

l'utilisateur clique le bouton

les écouteurs attachés au bouton sont appelés : le programmeur y aura mis le code qu'il veut faire réaliser par le clic.



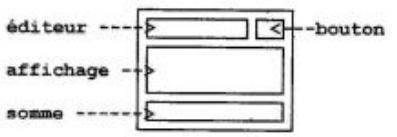
Mais c'est aussi le programmeur qui attache les écouteurs de son choix au bouton. Cela est en général fait à l'initialisation du programme.

7.3. Second exemple

Soit à réaliser un additionneur de nombres fonctionnant comme suit :

l'additionneur comporte un éditeur, un bouton, un affichage de la liste des nombres déjà édités et un présentoir de la somme des nombres édités

soit :



L'utilisateur s'en sert comme suit :

l'utilisateur édite un nombre dans un éditeur et entérine l'édition en cliquant sur le bouton. Le nombre édité est alors ajouté à la fin de la liste dans l'affichage, et la somme totale de tous ces nombres est mise à jour.

Enfin

l'utilisateur quitte l'application en cliquant le bouton adéquat de fermeture.

Cet exemple nous permettra d'illustrer les rudiments de la mise en page, de la gestion d'événements et de l'utilisation de quelques éditeurs ou afficheurs de texte.

7.3.1. Conception et réalisation

- Choix des composants

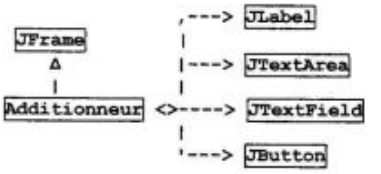
On prend

```
L'encadrement standard  lAdditionneur : JFrame
L'éditeur monoligne     lEditeur      : JTextField
Le bouton ok            ok            : JButton
L'affichage multiligne  lAffichage    : JTextArea
L'affichage de la somme laSomme       : JLabel
```

On peut coder ceci de la façon suivante

```
public class Additionneur extends JFrame {
  JTextField lEditeur = new JTextField();
  JTextArea lAffichage = new JTextArea();
  JLabel laSomme = new JLabel();
  JButton ok = new JButton("ok");
  ...
}
```

et le représenter en UML comme ceci



- Mise en page

Pour obtenir la mise en page souhaitée :

```
Additionneur : mise en page le long des bords
               (BorderLayout)
  au nord : lEditeur, ok
  au centre : lAffichage
  au sud : leLabel
```

le nord est lui-même mis en page à l'aide d'un composant "invisible", JPanel nord

```
nord : JPanel, BorderLayout
  centre : lEditeur
  est : ok
```

Donc la mise en page totale est

```
Additionneur : JFrame, BorderLayout
  nord : JPanel, BorderLayout
    centre : lEditeur
    est : ok
  centre : lAffichage
```

On peut coder ceci de la façon suivante

```
public class Additionneur extends JFrame {
  public Additionneur () {
    super("Additionneur");
    miseEnPage();
    ...
  }
  ...
}
```

Avec

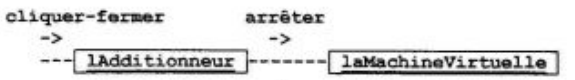
```
private void miseEnPage() {
  Container top = getContentPane();
  top.setLayout(new BorderLayout());
  top.add(northPanel(), BorderLayout.NORTH);
  top.add(lAffichage, BorderLayout.CENTER);
  top.add(laSomme, BorderLayout.SOUTH);
}
```

et

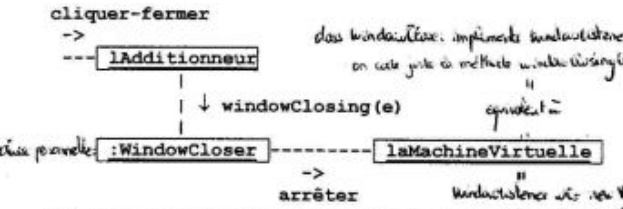
```
private JPanel northPanel() {
  JPanel panel = new JPanel();
  panel.setLayout(new BorderLayout());
  panel.add(lEditeur, BorderLayout.CENTER);
  panel.add(ok, BorderLayout.EAST);
  return panel;
}
```

- Traitement des événements

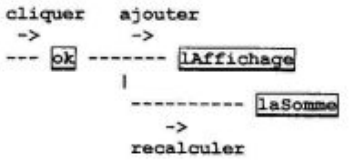
Nous savons comment coder la fermeture de l'Additionneur. L'action mise en œuvre peut être représentée par un diagramme de collaboration :



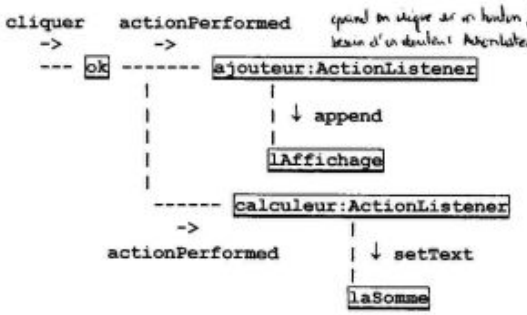
Dans ce schéma, insérons l'écouteur adéquat (et indispensable) et explicitons



Etablissons les diagrammes des autres événements.



qui devient avec les écouteurs adéquats



Ici les deux ActionListener sont obtenus ainsi

```
ActionListener ajouteur = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    lAffichage.append(lEditeur.getText() + '\n');
} };
ActionListener calculeur = new ActionListener() {
  int somme = 0;
  public void actionPerformed(ActionEvent e) {
    somme += Integer.parseInt(lEditeur.getText());
    laSomme.setText("" + somme);
} };
```

Le constructeur d'Additionneur devient

```
  public Additionneur(String nom) {
    super(nom);
    miseEnPage();
    raccordements();
    ...
  }
```

avec

```
private void raccordements () {
  ok.addActionListener(ajouteur);
  ok.addActionListener(calculeur);
  addWindowListener(new WindowCloser());
}
```

Il reste à rendre l'application visible

```
public class Additionneur extends JFrame {
  ...
  public Additionneur(String nom) {
    ...
    setSize(300, 300);
    setVisible(true);
  }
  ...
}
```

7.3.2. Compléments et exercices

1) Consultez la documentation Javadoc concernant toutes les classes Java introduites.
2) Codez Additionneur
3) Codez 3 applications Commandeur1, 2, 3 similaires à Additionneur mais sans le label. Le traitement doit être respectivement
   1- mise en majuscules
   2- calcul du carré du nombre édité
   3- liste des facteurs premiers avec répétition du nombre édité

   Définissez Commandeur (sans chiffre) comme un programme générique dont les cas particuliers précédents peuvent être déduits par héritage.

7.4. Mise en page

7.4.1. Généralités

Le positionnement des sous-fenêtres les unes par rapport aux autres est réalisé par les *protocoles de mise en page*.

Pour construire une interface on

dote le conteneur d'un protocole de mise en page : *layout*

ajoute des composants relatifs au protocole

On préfère en effet éviter d'utiliser des positions absolues en pixels, car le positionnement absolu

est difficilement retaillable

réserve des surprises en cas de changement de fonte ou de plate-forme

Les deux invocations suivantes sont donc fondamentales

```
telConteneur.setLayout(new TelLayout());
telConteneur.add(telleFenetre);
```

La méthode add est donc une méthode fondamentale de la classe Container et ses héritières. Elle existe dans les variantes suivantes :

```
Component add(Component comp)
Component add(Component comp, int index)
add(Component comp, Object constraints, int index)
void add(Component comp, Object constraints)
Component add(String name, Component comp)
```

La classe Container possède beaucoup d'autres méthodes. Citons

```
void remove(Component comp)
void remove(int index)
LayoutManager getLayout()
```

Nous reviendrons sur les possibilités des différents composants fenêtrés plus loin.

7.4.2. Exemples fondamentaux de politiques

Les protocoles suivants appartiennent à java.awt sauf BoxLayout :

- BorderLayout :

  place un composant au centre et les autres le long des bords.

Ses constructeurs sont

```
BorderLayout()
BorderLayout(int hgap, int vgap)
```

On spécifie l'emplacement du composant par un objet "contrainte" lorsqu'on l'ajoute au conteneur :

```
telConteneur.setLayout(new BorderLayout());
telConteneur.add(telleFenetre, BorderLayout.NORTH);
```

- FlowLayout :

  ajoute les composants par lignes de gauche à droite et de haut en bas.

```
FlowLayout()
FlowLayout(int align) // FlowLayout.CENTER, -LEFT, -RIGHT
FlowLayout(int align, int hgap, int vgap)
```

- GridLayout :

  place les composants dans une grille.

```
GridLayout()
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)
```

- BoxLayout (javax.swing) :

  place les composants verticalement ou horizontalement

```
BoxLayout(Container target, int axis) // noter target !
```

L'axe est BoxLayout.X_AXIS ou –Y_AXIS.

- GridBagLayout :

  très versatile, mais difficile à comprendre et à configurer

```
GridBagLayout()

GridBagConstraints gbc = ...
telConteneur.add(comp, gbc);
```

7.4.3. Principe de fonctionnement

L'algorithme de mise en page opère récursivement en deux temps

1) elle demande d'abord la taille préférée des composants
2) ensuite elle distribue les vraies tailles comme un compromis entre la taille préférée et la place disponible.

La taille préférée comporte deux parties distinctes : la hauteur et la largeur.

Voici le comportement de quelques politiques

- GridLayout : divise la fenêtre en mailles de même taille
  1) la taille préférée est en largeur la plus grande somme des tailles préférées des composants sur une ligne, idem hauteur - colonne
  2) ensuite elle divise la taille qui lui est finalement octroyée en cellules de même taille pour tous les composants
- FlowLayout : tente de respecter la hauteur préférée du plus haut composant et la somme des largeurs préférées des composants
  1) la taille préférée sera en hauteur la plus grande des tailles préférées de ses composants et en largeur la somme des largeurs préférées
  2) ensuite elle remplit l'espace octroyé, de gauche à droite et de haut en bas jusqu'à épuisement de la place, et en respectant les tailles préférées des composants
- BorderLayout : tente de respecter la hauteur préféré au nord et au sud, la largeur préférée à l'est et à l'ouest, et donne le reste au centre
  1) la taille préférée sera en hauteur la somme des hauteurs préférées du nord, du centre et de l'est, et similairement en largeur
  2) ensuite elle repecte si possible la hauteur du nord et du sud, puis la largeur de l'est et de l'ouest, et enfin donne ce qui rest au centre.
- BoxLayout, cas haut-bas (BoxLayout.Y_AXIS) : tente de respecter la hauteur préférée de chaque composante, et d'étendre la largeur de chaque composant à la plus grande
  1) la taille préférée en hauteur sera la somme des hauteurs préférées. En largeur ce sera par défaut la plus grande largeur préférée
  2) ajuste evt. la hauteur dans la fourchette des max et min hauteurs préférées. Si trop de place, le bas reste vide.

Des variantes importantes peuvent être obtenues en jouant sur le paramètre d'alignement d'un composant donné et/ou sur sa taille maximum préférée.

Exemple

Considérons un objet Commandeur. Avant affichage une taille de 300x300 pixels est accordée à la fenêtre totale. Comment cette taille sera-t-elle répartie entre les composants ?

1) Taille préférée. Hauteur : au nord : dépend de la hauteur de la fonte ; au centre ??? Largeur : au nord : somme de la largeur du texte des boutons et ??? Au centre ???
2) La hauteur préférée du nord sera respectée (car la fonte ne dépassera sans doute pas 300 pixels), le centre aura le reste. En largeur : respect des largeurs des 2 boutons, l'éditeur ligne au centre aura le reste.

7.4.4. Exemples

Dans tous ces exemples il est important de comprendre ce qui se passe lors d'un redimensionnement de la fenêtre.

Ex1

```
.................
......OK Abandon

BorderLayout
  sud panel FlowLayout droite
    OK
    Abandon
```

Ex2

```
...................
....**OK** Abandon

BorderLayout
  sud panel FlowLayout droite
    panel GridLayout(1,0)
      OK
      Abandon
```

Ex3

```
Nom  <TextField>
.................

BorderLayout
  nord Panel BorderLayout
    ouest Nom
    centre TextField
```

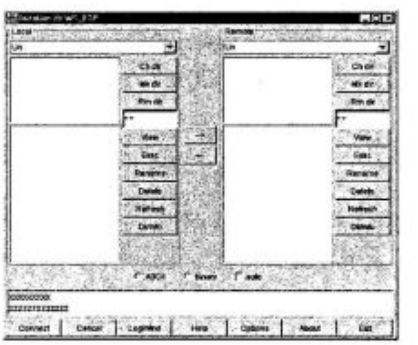
Ex4

```
Nom  <TextField>  Prénom <TextField>
Voie <TextField>  Ville  <TextField>
Tél. <TextField>

BorderLayout
  nord panel GridLayout(1,0)
    panel BorderLayout
      ouest GridLayout(3,0)   Nom, Voie, Tél.
      centre GridLayout(3,0)  3 TextField
    panel BorderLayout ...
```

Ex 5

Voici une copie réalisée en Java d'un client FTP connu



Cette fenêtre est redimensionnable tout en gardant en gros ses aspects symétrique et centré.

Voyons comment on peut la construire avec des layouts et fenêtres Java.

```
BorderLayout
  sud : BorderLayout
    nord : FlowLayout
      2 JRadioButton, 1 JCheckBox
    centre : TextArea(2,30)
    sud : GridLayout(1,0)
      7 boutons
  centre : GridBagLayout
    deux fenêtres "fichiers" et deux boutons...
```

Chaque fenêtre "fichiers" :

```
BorderLayout
  nord : Combo
  centre : JSplitPane avec 2 JList
  est : BorderLayout
    nord : GridLayout(0,1) 8 boutons et un éd.
```

Configuration du GridBagLayout

```
.1.|3|.2.
 ---------
.1.|4|.2.
```

		1	2	3	4
gridx		0	2	1	1
gridy		0	0	0	1
gridwidth		1	1	1	1
gridheight		2	2	1	1
ipadx		0	0	0	0
ipady		0	0	0	0
fill		BOTH	BOTH	NONE	NONE
anchor		CENTER	CENTER	SOUTH	NORTH
weightx		0.5	0.5	0.0	0.0
weighty		1.0	1.0	0.5	0.5
insets	top	0	0	0	2
	left	0	0	4	4
	bottom	0	0	2	0
	right	0	0	4	4

1 2 3 4

Le code de configuration du correspond à

```
filePanel = new Panel(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();

gbc.gridx      = 0;   // coord / cellules
gbc.gridy      = 0;   // défaut = RELATIVE
gbc.gridwidth  = 1;   // défaut = 1
gbc.gridheight = 2;
gbc.fill       = GridBagConstraints.BOTH; // = VERT.+HOR.
gbc.anchor     = GridBagConstraints.CENTER; // 9 pts.poss.
gbc.ipadx      = 0;   // remplissage interne
gbc.ipady      = 0;   // != insets = rempl.ext.
gbc.weightx    = 0.5; // 1 partage à égalité avec 2
gbc.weighty    = 1.0; // tout
filePanel.add(localFiles(), gbc);
gbc.gridx      = 2;
...
gbc.weighty    = 1.0;
filePanel.add(remoteFiles(), gbc);
gbc.gridx      = 1;
...
gbc.weighty    = 0.5;
gbc.insets     = new Insets(0, 4, 2, 4);
filePanel.add(leftButton(), gbc);
gbc.gridx      = 1;
...
gbc.weighty    = 0.5;
gbc.insets     = new Insets(2, 4, 0, 4);
filePanel.add(rightButton(), gbc);
```

D'autres effets peuvent être obtenus par

un choix approprié de bordures (dont une avec titre pour les fenêtres fichiers)

une fenêtre scindée horizontalement par une barre déplaçable pour "répertoires" et "fichiers".

comme nous l'avons fait.

7.4.5. Compléments et exercices

1) Consultez la documentation Java de java.awt. Inventoriez l'ensemble des politiques de mise en page existantes. Est-ce que certaines politiques sont définies dans d'autres paquetages ?
2) Codez ces différents exemples du paragraphe précédent.
3) Inventez une utilisation de BoxLayout et variez les alignements et tailles max. préférées.

7.4.6. La mise en forme des menus

La construction d'un système de menus se fait aussi par emboîtements, mais sans la notion de mise en page, puisque l'ordre est linéaire et non planaire.

JMenuBar (hérite de JComponent) permet de créer une barre de menus formant la racine du système de meus d'une fenêtre JFrame. L'arborescence du menu est réalisé à l'aide de JMenu, et les feuilles à l'aide de JMenuItem (hérite de AbstractButton)

Les principales méthodes sont

```
JFrame :    setJMenuBar(JMenuBar jmb);
JMenuBar :  JMenuBar()
            JMenu add(JMenu deroulant);
JMenu :     JMenu(String s)
            JMenuItem add(Action a)
            JMenuItem add(JMenuItem menuItem)
            JMenuItem add(String s)
            addSeparator()
JMenuItem :JMenuItem()
            JMenuItem(Icon icon)
            JMenuItem(String text)
            JMenuItem(String text, Icon icon)
            void addActionListener(ActionListener l)
```

7.5. La hiérarchie des composants

7.5.1. Héritages

Les classes Swing principales (en italique) sont organisées par héritage comme suit

```
Component
  Container
    Window
      Frame
        JFrame
      JWindow
    Panel
      Applet
        JApplet
    JComponent
      AbstractButton (JButton, JMenuItem, JMenu)
      JMenuBar
      JTextComponent (JEditorPane, JTextField, JTextArea
      JLabel
      JList
      JTree
      JTable
      ...
```

7.5.2. Les classes Component et Container

C'est la base de la hiérarchie de fenêtres Java.

Le rôle de Component est de mettre en place toutes les possibilités de base d'une fenêtre Java, qu'elle soit native (AWT) ou non (Swing) . Beaucoup de ces possibilités sont implémentées comme des propriétés

Type getProp() – void setProp(Type)

boolean isProp() – void setProp(boolean)

par exemple

nom, visibilité, position, taille

couleur, curseur, fonte, surface graphique

gestion du focus

menus popup

affichage d'image, gestion parentèle, localisation et accessibilité

gestion des listeners de base (dont property change, component, focus, key, mouse et mouse motion...)

auxquels Container ajoute

gestion des fenêtres contenues (add, remove, mise ne page)

gestion des listeners d'événements conteneurs

7.5.3. La classe (abstraite) JComponent

La classe JComponent concentre en elle beaucoup de nouvelles possibilités :

aspect et toucher échangeable

gestion des touches clavier

objets actions

bordures

tooltips

double buffering

slow rendering (débogage)

taille max, min et préférée

7.5.4. Un aperçu des principales classes

Swing offre des fenêtres spécialisées pour réaliser pratiquement tout ce qu'une IHM moderne demande :

- boutons JButton, JCheckBox, JRadioButton
- éditeurs, labels JTextField, JTextArea, JTextPane, JLabel
- listes, tables, arbres JList JComboBox, JTable, JTree
- dialogues simples JDialog, JOptionPane, JColorChooser
- outils divers JToolBar, JProgressBar, JScrollBar, JSlider
- menus JMenuBar, JMenu, JMenuItem, JSeparator, JPopupMenu, JCheckBoxMenuItem
- conteneurs généralistes JFrame, JWindow, JPanel, JApplet
- conteneurs spécialisées JScrollPane, JSplitPane, JTabbedPane
- conteneurs très spécialisées JInternalFrame, JLayeredPane, JRootPane

A côté de ces classes fenêtres existent de nombreuses autres classes qui participent à gérer l'état ou l'affichage des fenêtres, par exemple

ButtonGroup pour coordonner des JRadioButton

JToolTip pour une information "à la volée"

Border et ses descendants pour doter les fenêtres de bordures

ImageIcon pour gérer l'affichage d'images

etc.

7.5.5. Organisation

Les classes Swing sont réparties au total sur plusieurs paquetages

```
javax.swing           // composants de base : J...
javax.swing.border
javax.swing.colorchooser
javax.swing.event     // événements et listeners swing
javax.swing.filechooser
javax.swing.plaf      // plugable look and feel
javax.swing.table     // présentation données tabulaires
javax.swing.text      // édition avancée de texte
javax.swing.tree      // présentation données arborescentes
javax.swing.undo      // transactions réversibles
```

Le "x" de javax vient de "extension". En effet, dans les premières versions de Java une autre API, appelée AWT – plus rudimentaire que Swing – était la seule permettant de créer des fenêtres.

AWT reste importante, surtout par ses classes non-fenêtres réparties sur d'autres paquetages

```
java.awt              // classes de base AWT
java.awt.color
java.awt.datatransfer // presse-papier
java.awt.dnd          // glisser-poser
java.awt.event
java.awt.font
java.awt.geom         // géométrie 2D
java.awt.image
java.awt.image.renderable
java.awt.print        // gestion imprimante
```

7.5.6. Compléments et exercices

Il est hors de question de passer en revue toutes les méthodes de ces classes. Déjà la classe de base java.awt.Component en possède près de 150.

D'ailleurs une telle étude serait probablement inutile, car

un tout petit nombre constructeurs et de méthodes par classe suffit à se servir de ses instances

toutes les méthodes sont groupées autour de certaines propriétés qu'on retient assez facilement

Tous les compléments, on les apprend peu à peu dans la documentation en ligne ou ailleurs.

1) Pour commencer cet apprentissage, consultez la documentation de l'API Java. Vous pouvez commencer par regarder la documentation du paquetage javax.swing.
2) Réalisez l'interface de l'application FTP vue à propos des mises en page. Pour simplifier, remplacez les listes par des éditeurs JTextArea. Pour obtenir des effets de scrolling, utilisez des JScrollPane.
3) Ajoutez un même traitement à chaque bouton consistant à afficher le texte du bouton dans un dialogue fourni par JOptionPane.showMessageDialog.
4) Pour chaque composant, consultez la documentation Java en ligne.

7.6. Evénements et raccordements

7.6.1. Sources, événements, écouteurs

Nous avons vu qu'une application fenêtrée fonctionne comme suit

l'utilisateur clique un bouton ou entérine une édition

le bouton ou l'éditeur émet un objet événement d'un certain type

des objets d'un type adapté, inscrits comme écouteurs, sont avertis de l'événement.

Dans ces cas la fenêtre du clic ou de la frappe du caractère est considérée comme étant la source Java de l'événement, et les écouteurs sont les destinataires. Il est à noter que dans ce modèle source-événement-écouteur l'écouteur est passif, c'est la source qui prend l'initiative de prévenir.

Ce mécanisme est fondamental : on dit que l'application est mue par événement.

Selon son type, toute fenêtre peut être la source d'une foule de types d'événements, mais ces événements ont en commun un certain nombre de facteurs :

l'événement admet pour ancêtre la classe `java.util.EventObject` dotée des méthodes

```
EventObject(Object source)
Object getSource()
```

la source offre les deux méthodes

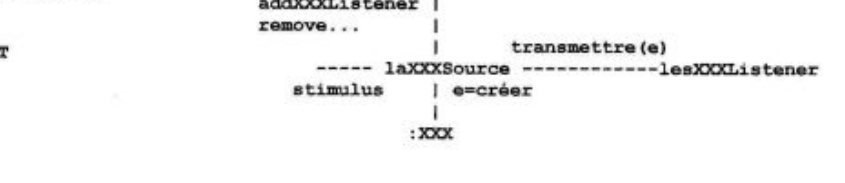
```
public void addXXXListener(XXXListener l)
public void removeXXXListener(XXXListener l)
```

l'interface `XXXListener` offre une ou plusieurs méthodes sur le style

```
public void transmettre(XXX event)
```

que la source peut appeler lorsque l'événement doit être distribué.

Schéma



7.6.2. Genres d'événements

Regardons de quels événements telle fenêtre peut être la source

```
Component : Component, Focus, Hierachy, -Bounds,
InputMethod,
            Key, Mouse, MouseMotion, PropertyChange
Container : Container
Window :    Window
JComponent: Ancestor*, VetoableChange [java.bean]
AbstractButton :   Action, Change, Item (sélection)
JList :     ListSelection*
JTree :     TreeExpansion*, TreeSelection*,
TreeWillExpand*
JTextField: Action, Caret*
```

La règle est que les événements et interfaces écouteurs déjà existants pour la version Java 1.0 sont définies dans java.awt.event. Les autres sont marqués d'un * et sont définies dans javax.swing.event.

Chaque interface écouteur offre une ou plusieurs méthodes. Ces méthodes ne suivent pas de règle de nommage particulier, seulement une règle pour le nom de paramètre. Il faut donc un minimum les connaître "par cœur". De même il faut connaître les noms des méthodes des événements.

Par exemple

événement et ses méthodes	interface et ses méthodes
ActionEvent	ActionListener
String getActionCommand	ActionPerformed
WindowEvent	WindowListener
Window getWindow	windowActivated,
windowClosed	
	windowClosing,
windowDeactivated	
	windowDeiconified,
windowIconified	
	windowOpened
KeyEvent	KeyListener
int getKeyCode	keyTyped
int getKeyChar	keyReleased
boolean isActionKey	keyPressed
MouseEvent	MouseListener
int getX	mouseClicked, mousePressed
int getY	mouseReleased
int getClickCount	mouseEntered, mouseExited
	MouseMotionListener
	mouseDragged, mouseMoved
PropertyChangeEvent	PropertyChangeListener
Object getNewValue	propertyChange
Object getOldValue	
String getPropertyName	

et encore

```
Change :    stateChanged
Component : componentHidden, -Moved, -Resized, -Shown
Container : componentAdded, -Removed
Focus :     focusGained, -Lost
Item :      itemStateChanged
```

Nous verrons plus loin que d'autres objets que les fenêtres peuvent être sources d'événements. L'ensemble des événements et des écouteurs est donc également plus grand que ce que nous avons montré ici. Notamment les modèles de composants Swing seront de telles sources.

7.6.3. Exemples

Suivre la souris à la trace

```
w.addMouseMotionListener(new MouseMotionListener() {
  public void mouseMoved(MouseEvent me) {
    System.out.print("Moved ");
    System.out.println("x="+me.getX()+", y="+me.getY());
  }
  public void mouseDragged(MouseEvent me) {
    System.out.print("Dragged ");
    System.out.println("x="+me.getX()+", y="+me.getY());
} }
```

Surveiller l'état d'un JCheckBox

```
b.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent ce) {
    System.out.println(
      +((JCheckBox)b.getSource()).isSelected();
} }
```

Surveiller la sélection dans une liste

```
l.addListSelectionListener(new ListSelectionListener() {
  public void valueChanged(ListSelectionEvent les) {
    JList list = (JList)les.getSource();
    System.out.println(
      "Sélection = "+list.getSelectedValue());
} }
```

Surveiller les multiples clics

```
w.addMouseListener(new MouseInputAdapter() {
  public void mouseClicked(MouseEvent me) {
    System.out.println(me.getClickCount());
} }
```

7.6.4. Actions

Action, que nous avons cité a propos des menus, est une interface qui hérite d'ActionListener. Le rôle d'une action est de mémoriser les éléments communs à plusieurs commandes. Par exemple une commande donné peut être installée à la fois comme un item de menu et comme un bouton sur une barre à outils. L'item et le bouton doivent avoir en commun notamment

- le texte
- l'icône
- le texte tooltip
- le raccourci clavier
- l'armement (actif, inactif)

L'action permet de gérer ces éléments. Par exemple

```
Action chargerFichier = new AbstractAction() {
  public void actionPerformed(ActionEvent ae) {
    JFileChooser fc = new JFileChooser();
    int val = fc.showOpenDialog(this);
      if (val == JFileChooser.APPROVE_OPTION) {
      ...
} } }
chargerFichier.putValue(Action.NAME, "Charger fichier");
chargerFichier.putValue(Action.SMALL_ICON, new
ImageIcon("open.gif"));
...// configuration de base de chargerFichier
```

puis

```
JMenuItem mi = telMenu.add(chargerFichier);
mi.setIcon(null); // adapter la configuration
...

JButton bt = telJMenuBar.add(chargerFichier);
bt.setToolTipText((String)action.getValue(Action.SHORT_DI
CRIPTION));
...// adapter la configuration
```

7.6.5. Compléments et exercices

1) Explorez les hiérarchies d'événements et listeners dans la documentation Java. Partez de java.util.EventObject et java.util.EventListener.
2) Réalisez une petite application Swing mettant en œuvre les écoutes des exemples ci-dessus.
3) Dotez Commandeur d'un menu et d'une barre d'outils comportant au moins un item/bouton permettant de quitter l'application. utilisez une Action. Dotez l'action d'un texte tooltip et d'une petite icône.

8. Modèle, vue, contrôle

8.1. Représentation et modèle

Tous les composants concrets Swing sont divisés en deux parties,

une partie vue (la "fenêtre", la représentation) et

une partie gestion de ce qui est représenté (la donnée), appelé modèle en Swing.

En fait la partie vue contient deux parties distinctes

le graphisme sur l'écran (le "look", sortie des informations vers l'utilisateur)

l'ensemble des clics et mouvements de souris et frappes de clavier ("le feel") l'interprétables en termes de commande de ces actions (appelé aussi le contrôle).

On peut donc parler de tripartition Vue / Contrôle / Modèle où vue - contrôle prend le sens "look and feel". Cependant un programmeur a beaucoup plus souvent à coder un modèle qu'un "look and feel", si bien que la plupart du temps il raisonnera en bipartition Modèle / Vue.

Rappelons que Swing offre en standard quelques "look and feel". Voici comment on peut les utiliser tous :

```
import javax.swing.*;
public class LAndF extends ... { // choisir une appli
Swing
  public static void main(String[] args) {
    UIManager.LookAndFeelInfo[] lfi =
UIManager.getInstalledLookAndFeels();
    try {
      for (int i = 0;  ; i++) {

UIManager.setLookAndFeel(lfi[i].getClassName());
        new LAndF(lfi[i].getName());
      }
    } catch (Exception e) {
  } }
  public LAndF(String name) {
    super(name);   // adapter
} }
```

Dans les cas simples comme celui d'un AbstractButton la distinction entre vue et modèle est peu importante, car

on est rarement amené à recoder un modèle de bouton

toutes les méthodes importantes de l'interface ButtonModel sont accessibles depuis JButton.

Par exemple on peut écrire

```
JButton ok = new JButton("OK");
ok.addActionListener(al);
```

au lieu de

```
ok.getModel().addActionListener(al);
```

Dans les composants plus complexes, on ne retrouve pas cette simplification, car il faudrait dédoubler trop de méthodes, et le modèle est trop sujet à modifications de la part de l'utilisateur.

8.2. JList

8.2.1. Généralités

JList est un composant offrant une vue sur une liste d'items. Ce composant permet notamment de

connaître la longueur de la liste

sélectionner un élément dans la liste et obtenir l'élément sélectionné

gérer les écouteurs de sélection

Pour gérer le contenu de la liste on s'attendrait à trouver des méthodes

modifier, ajouter, supprimer un élément à la liste

Cependant, JList ne possède aucune méthode permettant de modifier, d'ajouter ou de supprimer un item. Ce constat s'explique par le fait que JList est implémentée selon le paradigme de la séparation vue, contrôle, données :

la vue et le contrôle sont regroupés dans JList,

les données d'une JList sont stockées dans une structure séparé appelée modèle implémentation de ListModel.

8.2.2. Séparation des responsabilités

La séparation nette en données – vue correspond à une séparation nette des responsabilités des deux parties. Voyons un extrait des méthodes :

- JList est un JComponent, il gère
  - l'attachement (éventuellement création) d'un modèle de données
  - la représentation visuelle des éléments installés dans la liste
  - les sélections d'éléments à l'aide du clavier et de la souris
  - les écouteurs d'événements de sélection
  - la correspondance coordonnées de la souris <--> item représenté.

- Le modèle gère
  - les objets réels contenus dans la liste et leur nature
  - les ajouts, modifications et suppressions d'éléments
  - les écouteurs d'événements de modifications du contenu

Le fonctionnement typique d'une liste est la suivante :

```
une JList affiche une représentation visuelle des items
la liste
un utilisateur sélectionne un item visualisé
le programme récupère l'objet correspondant

le programme ajoute (enlève, modifie) des objets à la
liste
la représentation visuelle est mise à jour
```

Pour que cela fonctionne il existe un double lien

```
la liste a connaissance de son modèle et peut s'adresser
lui
le modèle a accès à la JList par un ListDataListener
```

8.2.3. Méthodes de JList

Création et attachement de modèle

```
JList(ListModel dataModel)
JList()
JList(Object[] listData)     // DefaultListModel
JList(Vector listData)       // DefaultListModel
void setModel(ListModel lm) // get... : c'est une
propriété
void setListeData(Object[] listData)
void setListeData(Vector listData)
```

Représentation visuelle, propriétés (get / set)

```
FixedCellWidth/Height : int
SelectionForeground/Background : Color
VisibleRowCount
First(Last)VisibleIndex : int, en lecture
void ensureIndexIsVisible(int index)
CellRenderer : ListCellRenderer
```

Sélection

```
SelectedIndex : int
SelectedIndices : int[]
SelectedValue : Object
SelectedValues : Object[], en lecture
SelectionMode : int
(SINGLE_ SINGLE_INTERVAL_ MULTIPLE_INTERVAL_SELECTION)
...
void setSelectionInterval(int anchor, int lead)
void addSelectionInterval(int anchor, int lead)
int getAnchorSelectionIndex()
int getLeadSelectionIndex()
int maxSelectionIndex()
int minSelectionIndex()
SelectioModel : ListSelectioModel
```

Ecouteurs

```
add...
removeListSelectionListener
```

Correspondance

```
int locationToIndex(Point location)
Point indexToLocation(int index)
```

Nous voyons que la classe JList est liée à quelques autres classes (en fait des interfaces)

```
JList -------> ListModel
          |--> ListSelectionListener
          |--> ListCellRenderer
          '--> ListSelectionModel
```

8.2.4. ListModel

Il s'agit d'une interface assez simple

```
public interface ListModel {
  public Object getElementAt(int index)
  public int getSize()
  public void addListDataListener(ListDataListener l)
  public void removeListDataListener(ListDataListener l)
}
```

qui représente le minimum de ce dont JList a besoin pour interagir avec le modèle. Dans la pratique on se codera d'une dérivation de la classe abstraite

```
AbstractListModel
  void fireContentsChanged(Object source, int index0, int
index1)
  void fireIntervalAdded(Object source, int index0, int
index1)
  void fireIntervalRemoved(Object source, int index0, int
index1)
```

cette classe abstraite implémente ce qu'il faut pour générer les événements et gérer les écouteurs. On peut notamment utiliser DefaultListModel qui en est une implémentation très complète.

8.2.5. ListSelectionListener

L'interface n'a qu'une seule méthode

```
public interface ListSelectionListener {
  public void valueChanged(ListSelectionEvent e)
}
```

L'événement offre notamment

```
int getFirstIndex()
int getLastIndex()
```

8.2.6. ListCellRenderer

Le tracé de chaque item est confié à un Component implémentation de l'interface ListCellRenderer avec l'unique méthode

```
  public Component getListCellRendererComponent(
    JList list, Object value, int index,
    boolean isSelected, boolean cellHasFocus)
```

Par défaut le rendu se contente d'invoquer toString sur l'objet à représenter, mais il est donc facile de modifier ce fait.

8.2.7. ListSelectionModel

Il s'agit d'une grosse interface. La JListe délègue à l'objet correspondant toute la gestion de la sélection. Il existe une implémentation par défaut utilisée par défaut par JList.

8.2.8. Exemple

On voit que, à condition d'implémenter correctement ListModel et éventuellement ListCellRenderer on peut représenter visuellement toute sorte de données dans une JList.

Nous allons adapter Additionneur pour utiliser une JList pour montrer les nombres édités.

Nous devons pour cela installer les nombres dans un modèle adéquat. Nous pourrions utiliser DefaultListModel, mais pour l'exemple nous allons créer un modèle à nous. Comme base, nous prendrons AbstractListModel. De plus nous changeons légèrement le rendu pour illustrer l'utilisation du ListCellRenderer.

Nous améliorons en outre le comportement du programme de la façon suivante : si l'on sélectionne un nombre déjà présent dans la liste, ce nombre est affiché dans l'éditeur pour modification, et lorsqu'on entérine pendant qu'un item est sélectionné, on remplace en fait le nombre sélectionné.

```
class LocalListModel extends AbstractListModel {
  ArrayList liste = new ArrayList();
  public void add(Object o) {
    int index = liste.size();
    liste.add(o);
    fireIntervalAdded(this, index, index);
  }
  public Object getElementAt(int index) {
    return liste.get(index);
  }
  public int getSize() {
    return liste.size();
} }
```

Le traitement des événements est :

```
ActionListener al = new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    localListModel.add(editeur.getText());
  }                    // la liste contient des String !
};
lEditeur.addActionListener(al);
ok.addActionListener(al);
lAffichage.addListSelectionListener(new
ListSelectionListener() {
  public void valueChanged(ListSelectionEvent lse) {
    int i = lse.getFirstIndex();
    lEditeur.setText(localListModel.getElementAt(i));
  }
} );
localListModel.addListDataListener(new ListDataListener(
{
  public void contentsChanged(ListDataEvent e)
{recalculer();}
  public void intervalAdded(ListDataEvent e)
{recalculer();}
  public void intervalRemoved(ListDataEvent e)
{recalculer();}
  private void recalculer() {
    int s = 0;
    for (int i = 0; i < localListModel.getSize(); i++) {
      s +=
Integer.parseInt(localListModel.getElementAt(i));
    }                        // la liste contient des
String !
    somme.setText("" + s);
  }
} );
```

Le rendu est modifié ainsi

```
class LocalCellRenderer extends JLabel
implements ListCellRenderer
{
  static Icon iconArrow = new ImageIcon("Arrow.gif");
  static Icon iconVoid  = new ImageIcon("Void.gif");
  public LocalCellRenderer() {
    setOpaque(true); // pour le fond
  }
  public Component getListCellRendererComponent(
    JList list, Object value, int index,
    boolean isSelected, boolean cellHasFocus)
  {
    setText(value.toString());
    setIcon(isSelected ? iconArrow : iconVoid);
    setBackground(isSelected ? Color.red : Color.white)
    setForeground(isSelected ? Color.white :
Color.black);
    return this;
  }
}
```

8.2.9. Conclusion sur JList

Essayons de prendre le point de vue du concepteur de JList. On peut prétendre qu'il a prévu trois sortes d'utilisations (acteurs) de la classe

Affichage : la sélection d'un ou plusieurs items dans une liste d'objets

Gestion : ajoute à Affichage la modification du contenu de la liste

Pro : ajoute à Gestion l'adaptation de JList à des besoins spécifiques.

Dans le premier cas une simple instance de JList créé par un constructeur adéquat permet de répondre au besoin : affichage et installation d'écouteurs de sélection.

Dans le second cas il faut aussi accéder au modèle des données et lui ajouter des ListDataListener.

Dans le troisième cas on peut être amené à redéfinir presque tous les composants, notamment ListCellRenderer et ListSelectionModel.

On remarquera aussi que, dans une description comme la suivante, le concepteur a incarné tout substantif en une classe :

une Liste sert à présenter un Rendu des Données de la liste pour autoriser des Sélections.

On notera que la JList ne permet pas qu'on édite, par l'intermédiaire du rendu, une donnée individuelle de la liste.

8.2.10. Compléments et exercices

1) Consultez la documentation de JList et celle des autres interfaces et classes citées.
2) Réalisez l'exemple esquissé au paragraphe précédent.
3) Dans une seconde version, installez dans la liste des objet Integer ou Double.

8.3. JTree

JTree permet de visualiser une structure de données arborescente.

8.3.1. Classes d'appui

Comme dans le cas de la liste, la classe JTree partage ses responsabilités avec d'autres classes, notamment un TreeModel. Tentons la description suivante

un Arbre présente un Rendu de Données organisées en Aborescence pour des expansions – contractions, des Sélections et éventuellement des Editions.

Ici on a

```
Arbre = JTree (expansion - contractions :
TreeExpansionListener,
  TreeWillExpandListener)
Sélection = TreeSelectionModel (modifs :
TreeSelectionListener)
Données = TreeModel (modifs : TreeModelListener)
Arborescence = TreeNode
Rendu = TreeCellRenderer
Edition = TreeCellEditor
```

Nous avons donc le schéma

```
JTree -------> TreeModel
          |--> TreeSelectionListener
          |--> TreeCellRenderer
          |--> TreeCellEditor
          '--> TreeSelectionModel
```

8.3.2. Création

Cependant on se rend compte qu'il est moins facile de construire des arbres à plusieurs branches que des listes, si bien que dans le cas d'un arbre on est pratiquement toujours obligé de construire explicitement son contenu arborescent.

Pour cet arborescence on utilise DefaultMutableTreeNode, impémentation de l'interface TreeNode.

Voici un exemple de création d'arbre :

```
DefaultMutableTreeNode pepe = new
DefaultMutableTreeNode("pepe");
DefaultMutableTreeNode pere = new
DefaultMutableTreeNode("père");
DefaultMutableTreeNode tante = new
DefaultMutableTreeNode("tante");
DefaultMutableTreeNode moi = new
DefaultMutableTreeNode("moi");
DefaultMutableTreeNode soeur = new
DefaultMutableTreeNode("soeur");
DefaultMutableTreeNode cousin = new
DefaultMutableTreeNode("cousin");
pepe.add(pere);
pepe.add(tante);
pere.add(moi);
pere.add(soeur);
tante.add(cousin);
```

Nous pouvons maintenant créer un modèle et un JTree

```
DefautTreeModel treeModel = new DefautTreeModel(pepe);
JTree tree = new JTree(treeModel);
```

8.3.3. Sélection

La sélection dans un arbre est aussi une notion plus complexe que dans une liste. En effet, dans une liste on peut utiliser un index (entier). Dans un arbre la sélection implique le chemin de la racine jusqu'à l'item sélectionné : le TreePath

```
tree.addTreeSelectionListener(new TreeSelectionListener(
{
  public void valueChanged(TreeSelectionEvent tse) {
    TreePath selection = tree.getSelectionPath();
    TreePath chemin = selection.getParentPath();
    Object dernier = selection.getLastPathComponent();
    System.out.println("Sélection = " + chemin + " " +
dernier);
  }
} );
```

8.3.4. Expansion – contraction

de la représentation d'un nœud.

```
tree.addTreeExpansionListener(new TreeExpansionListener(
{
  public void treeExpanded(TreeExpansionEvent tee) {
    TreePath chemin = tee.getPath();
    System.out.println("Expansion de :" + chemin);
  }
  public void treeCollapsed(TreeExpansionEvent tee) {
    TreePath chemin = tee.getPath();
    System.out.println("Contraction de :" + chemin);
  }
} );
```

8.3.5. Modification de l'arbre

Nous pouvons procéder de deux façons pour modifier l'arbre :

éditer une cellule du rendu de l'arbre (supposé éditable)

utiliser les méthodes du DefaultTreeModel

modifier directement l'arborescence des DefaultMutableTreeNode.

Le premier cas est "automatique" (appelle `valueForPathChanged(TreePath p, Object new)`

Dans le second cas les méthodes invoquées sur le DefaultTreeModel prennent soin d'avertir tous les écouteurs inscrits auprès du modèle :

```
void insertNodeInto(MutableTreeNode node, MutableTreeNod
parent, int i);
void removeNodeFromParent(MutableTreeNode node);
```

Dans le troisième cas c'est à nous de nous assurer que le TreeModel prenne connaissance du changement que nous avons opéré, en utilisant une des méthodes

```
void nodeChanged(TreeNode node)
void nodesChanged(TreeNode node, int[] childIndices)
void nodeStructureChanged(TreeNode node)
void nodesWereInserted(TreeNode node, int[] childIndices
void nodesWereRemoved(TreeNode node,
                      int[] childIndices, Object[]
removedChildren)
```

Le TreeModel peut alors avertir les écouteurs :

```
treeModel.addTreeModelListener(new TreeModelListener() {
  public void treeNodesChanged(TreeModelEvent e) {...}
  public void treeNodesInserted(TreeModelEvent e) {...}
  public void treeNodesRemoved(TreeModelEvent e) {...}
  public void treeStructureChanged(TreeModelEvent e) {..
}
```

8.4. JTable

JTable permet de visualiser une structure de données tabulaire

8.4.1. Classes d'appui

Tentons une description de JTable comme nous l'avons fait pour JTree et JList :

> une Table présente un Rendu de Données d'organisation tabulaire pour des Sélections et éventuellement des Editions ; les Colonnes peuvent être déplacées et possèdent des Noms.

et

```
JTable ------> TableModel
          |--> TableColumnModel
          |--> TableSelectionListener
          |--> ListCellRenderer   //
          |--> ListSelectionModel // comme pour une liste
          |--> TableCellEditor
          '--> JTableHeader
```

Des données tabulaires peuvent être créées de différentes façons :

> tableau à deux dimensions
>
> vecteur de tableaux
>
> vecteur de vecteurs

etc. Inutile donc d'introduire un type similaire à TreeNode.

8.4.2. Exemple

Supposons que nous disposions d'un Vector vect représentant une liste de Personnes

```
class Personne {
  String nom;
  int age;
  boolean masc;
}
```

et que notre problème soit d'afficher le contenu du Vector sous la forme d'une table avec une personne par ligne. Nous voulons à la fois visualiser les données et pouvoir les modifier.

Pour cela nous dérivons un TabelModel adapté à nos données à partir d'une AbstractTableModel. Comme dans le cas de AbstractListModel, cette classe abstraite prépare presque tout pour nous.

Notre implémentation devient

```
  class PersoTableModel extends AbstractTableModel {
    Vector data;
    Object[] colNames;
    public PersoTableModel(Vector data, Object[] colNames)
{
      this.data  = data;
      this.colNames = colNames;
    }
    public int getRowCount() {return data.size();}
//
    public int getColumnCount() {return
colNames.length;}// indisp.
    public Object getValueAt(int row, int col) {
//
      Personne p = (Personne)data.elementAt(row);
      Object v = null;
      switch (col) {
        case 0 : v = p.nom; break;
        case 1 : v = new Integer(p.age); break;
        default: v = new Boolean(p.masc);
      }
      return v;
    }
    public String getColumnName(int col) {
      return colNames[col];
    }
    public Class getColumnClass(int index) {
      return getValueAt(0, index).getClass();
    }
    public void removeRow(int row) {
      fireTableRowsDeleted(row, row);
      data.removeElementAt(row);
    }
    public void addRow(Object o) {
      int l = data.size();
      data.add(o);
      fireTableRowsInserted(l, l);
    }
    public boolean isCellEditable(int row, int col) {
      return true;
    }
    public void setValueAt(Object value, int row, int co
{
      Personne p = (Personne)data.elementAt(row);
      switch (col) {
        case 0 : p.nom = (String)value; break;
        case 1 : p.age = ((Integer)value.intValue();
break;
        default: p.masc = ((Boolean)value).booleanValue(
      }
      fireTableCellUpdated(row, col);
  } }
```

Nous pouvons maintenant créer une JTable acceptant l'édition de ses valeurs :

```
String[] cols = new String[]{"Nom", "Age", "Masc"};
TableModel model = new PersoTableModel(vect, cols)
JTable table = new JTable(model);
table.setEditable(true);
```

8.4.3. Sélection

Installons un écouteur de sélections. :

```
table.addListSelectionListener(new ListSelectionListener
{
  public void valueChanged(ListSelectionEvent lse) {
    int i = table.getSelectedRow();
    int j = table.getSelectedColumn();
    Object o = model.getValueAT(i, j);
    System.out.println("Sélection ("
                       + i + ", " + j + ") = " + o);
  }
} );
```

8.4.4. Modification de la table

Ici aussi on a le choix :

> éditer une cellule du rendu de la table (supposée éditable)
>
> modifier la donnée tabulaire en utilisant les méthodes du modèle
>
> modifier directement la donnée en court-circuitant le modèle.

Pour ce dernier cas AbstractTableModel offre une série de méthodes "fire"

```
void fireTableCellUpdated(int row, int column)
void fireTableChanged(TableModelEvent e)
void fireTableDataChanged()
void fireTableRowsDeleted(int firstRow, int lastRow)
void fireTableRowsInserted(int firstRow, int lastRow)
void fireTableRowsUpdated(int firstRow, int lastRow)
void fireTableStructureChanged()
```

8.4.5. Exercices et compléments

1) Codez l'exemple. Dotez le programme d'un moyen pour lire et sauvegarder les données.
2) Réalisez un afficheur de contenu de répertoire : les différentes informations relatifs aux fichiers sont présentées par colonnes, une ligne par fichier.
3) Réalisez un afficheur d'arborescence de répertoires
4) Réunissez les deux exercices précédents dans un explorateur de répertoires et fichiers.
5) Réalisez un gestionnaire de relations de filiation descendantes. Le programme devra permettre

- d'afficher les données personnelles dans un JTable, l'ordre ici est sans importance
- d'afficher les relations de filiations dans un JTree, l'arbre n'affiche que les noms
- la sélection dans l'arbre provoque la sélection de la même personne dans la table
- la table est éditable, l'arbre non. Lorsque le nom d'une personne est modifiée dans la table, le nom dans l'arbre est mis à jour.
- on peut ajouter et supprimer des personnes par rapport à la sélection dans l'arbre

8.5. Les composants texte

8.5.1. Les classes

La hiérarchie des composants textuels comporte une classe abstraite et cinq classes

```
                    JTextComponent
   JTextField       JTextArea          JEditorPane
JPassWordField                         JTextPane
```

- JTextField est un éditeur mono-ligne

  pouvant générer un ActionEvent par retour chariot à la fin de l'édition

  => c'est un contrôle.

```
textField = new JTextField(20);
textField.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    String text = textField.getText(); // - retour ch.
    ...
} }
```

- JTextArea est un éditeur multi-lignes. Il peut

  afficher et éditer du texte dans une fonte et couleur quelconque

  tout le texte est uniformément de cette couleur et cette fonte

  => convient pour un texte non formaté de n'importe quelle longueur

```
JTextArea textArea = new JTextArea(unTexte);
textArea.setFont(new Font("Serif", Font.ITALIC, 16));
textArea.setLineWrap(true);
textArea.setWrapStyleWord(true);
void setColumns(30)
void setRows(10)
void setTabSize(4)
```

- JEditorPane est un éditeur multi-ligne multi-fontes. Lui (et ses descendants)

  est largement configurable

  peut être adapté à des besoins d'édition professionnelle de texte

  avec par exemple des images et objets inclus

  sait en outre afficher du texte HTML (et un jour du RTF ?).

```
JEditorPane editorPane = new JEditorPane();
editorPane.setEditable(true);
editorPane.setText(unString);    // DefaultEditorKit
```

ou

```
editorPane.read(unFichier.html, new HTMLDocument()); //
```

HTMLEditorKit

ou encore

```
editorPane.setEditable(false); // pour les HyperlikEvent
editorPane.setPage(unUrl);     // HTMLEditorKit
```

Dans ce dernier cas l'editorPane peut se transformer en un navigateur sommaire :

```
editorPane.addHyperlinkListener(new HyperlinkListener());
```

avec

```
public interface HyperlinkListener {
  public void hyperlinkUpdate(HyperlinkEvent e);
}
```

où l'événement hyperlink offre les méthodes

```
String getDescription()
HyperlinkEvent.EventType getEventType()
  (ACTIVATED, ENTERED, EXCITED)
URL getURL()
```

- Ajoutons JLabel : il sait afficher image et texte et reconnaît le HTML !

```
label1 = new JLabel("Image et Texte", icone,
JLabel.CENTER);
label2 = new JLabel("Seulement texte");
label3 = new JLabel(icone);
label4 = new JLabel(unTexteHTML);
```

8.5.2. Document et Caret

Un JTextComponent est composé du texte à éditer, le Document – qui est son modèle de données – et de la position d'édition, le caret.

```
Document d = unEditeur.getDocument();
```

Document est une interface qui représente le texte édité :

```
int getLength()
String getText(int offset, int length)
void getText(int offset, int length, Segment txt)
void insertString(int offset, String str, AttributeSet a
void remove(int offs, int len)
```

Les deux dernières méthodes sont celles qu'on est le plus souvent amené à redéfinir.

Il supporte des écouteurs

```
DocumentListener
UndoableEditListener
```

avec

```
public interface DocumentListener {
  void changedUpdate(DocumentEvent e) // element
attributes changed.
  void insertUpdate(DocumentEvent e)  // insert into the
document.
  void removeUpdate(DocumentEvent e)  // remove from
}
```

et DocumentEvent offre

```
DocumentEvent.ElementChange getChange(Element elem)
Document getDocument()
int getLength()    Returns the length of the change.
int getOffset()    Returns the offset of the start of the
change.
DocumentEvent.EventType getType()  // CHANGE, INSERT,
REMOVE
```

Le caret aussi supporte des écouteurs

```
Cartet c = unEditeur.getCaret();
c.addChangeListener(cl);
```

Les différentes implémentations de Document sont

```
                AbstractDocument
  PlainDocument          DefaultStyledDocument
                         HTMLDocument
(JTextField -Area)     (JTextPane,
                         JEditorPane)
```

8.5.3. Un éditeur d'entiers

Soit à créer un éditeur monoligne n'acceptant que l'édition d'entiers. L'éditeur doit veiller à ce que

tous les caractères entrés ne soient que des chiffres.

Pour y parvenir nous choisissons de

dériver une classe IntEd à cette fin

créer un modèle Document qui n'accepte que l'insertion de chiffres

doter IntEd de ce modèle en modifiant la méthode createDefaultModel().

Il suffit donc pour nous de modifier la méthode

```
public void insertString(
  int offset, String str, AttributeSet a
) throws BadLocationException
```

- Voici une solution possible

```
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
public class IntEd extends JTextField {
  public IntEd(int columns) {
    super(columns);
  }
  public int getValue() {
    return Integer.parseInt(getText());
  }
  public void setValue(int value) {
    setText("" + value);
  }
  protected Document createDefaultModel() {
    return new IntDocument();
  }
  protected class IntDocument extends PlainDocument {
    public void insertString(int offs, String str,
AttributeSet a)
    throws BadLocationException
    {
      for (int i = 0; i < str.length(); i++) {
        if (!Character.isDigit(str.charAt(i))) {
          Toolkit.getDefaultToolkit.beep();
          return;
        }
      }
      super.insertString(offs, str, a);
} } }
```

8.5.4. Commandes

L'API Java offre dans la classe DefaultEditorKit un certain nombre de fonctionnalités d'édition, implémentées comme des d'Actions comme par exemple

couper, copier, coller, aller au début, - à la fin...

Rappelons qu'une Action est une interface extension de ActionListener qui est utile lorsqu'une même fonctionnalité doit être accédée à partir de diverses commandes. Une Action est notamment un dépôt pour des propriétés qui doivent être communes à ces fonctionnalités :

nom, icône, description courte et longue, raccourci...

L'action nous laisse gérer ces propriétés par

```
void setValue(String key, Object value)
Object getValue(String key)
```

Ceci étant dit, un JTextField par exemple n'offre aucun moyen par défaut pour utiliser ces commandes, ni par menu, ni par raccourcis clavier. C'est à la charge du programmeur de rendre utilisables ces fonctionnalités.

Une façon simple d'en rendre certaines commandes utilisables dans tous les éditeurs d'une application donnée est d'installer des raccourcis clavier. Cela est facilité par l'existence de KeyMap configurables dans les éditeurs.

Par exemple, les instructions suivantes

```
  KeyMap km;
  km =
JTextComponent.getKeymap(JTextComponent.DEFAULT_KEYMAP);
  key = KeyStroke.getKeyStroke(KeyEvent.VK_INSERT,
Event.SHIFT_MASK);
  km.addActionForKeyStroke(key, ...); // Action "coller"
```

ont pour effet qu'un SHIFT-INSERT provoquera la copie dans l'éditeur du contenu de presse-papier.

Petit problème : comment désigner l'action de collage ? Le plus simple est finalement de récupérer toutes les actions, et de les installer dans un HashMap pour ensuite pouvoir les retrouver par un nom explicite :

```
HashMap actions = null;
Action[] actionsArray = new
DefaultEditorKit().getActions();
for (int i = 0; i < actionsArray.length; i++) {
  Action a = actionsArray[i];
  actions.put(a.getValue(Action.NAME), a);
}
```

Notre ligne inachevée devient maintenant

```
  km.addActionForKeyStroke(key,
actions.get(DefaultEditorKit.pasteAction));
```

Il est évidemment tout à fait possible de définir soi-même des commandes. Voici par exemple une commande pour la recherche d'un mot dans le texte d'un éditeur

```
class SearchAction extends AbstractAction {
  JTextComponent modele;
  JTextComponent cible;
  public SearchAction(JTextComponent modele,
JTextComponent cible) {
    super("chercher");
    this.modele = modele;
    this.texte = cible;
  }
  public void actionPerformed(ActionEvent e) {
    String str = modele.getText();
    if ((str == null) || (str.length() == 0)) return;
    try {
      Document doc = cible.getDocument();
      String texte = doc.getText(0, doc.getLength());
      int index = texte.toString().indexOf(str);
      if (index >=0) {
        cible.setCaretPosition(index); //force scroll
        cible.setSelectionStart(index);
        cible.setSelectionEnd(index + str.length());
      }
    } catch (BadLocationException ble) {
} } }
```

Ce code ne trouve que la première occurrence du mot dans le texte.

8.5.5. Exercices

1) Créez une classe JTextActions permettant de faciliter la gestion des Actions d'édition dans l'ensemble des éditeurs d'une application. Elle doit notamment installer des raccourcis claviers pour couper, copier, coller.
2) Codez une application éditeur de texte qui offre un menu et une barre d'outils avec les facilités habituelles que l'on trouve dans un éditeur basique.
3) Etudiez le problème d'un éditeur de code source Java basé sur un JEditorPane et capable de réaliser le coloriage syntaxique du texte.

9. Processus

9.1. Approche

Vous connaissez tous la séquence code source, code compilé, code exécuté. Vous savez peut-être aussi simuler l'exécution d'un programme. Pour la simuler il faut

le code source du programme

un pointeur d'instruction

une pile d'exécution

une mémoire dynamique pour les variables allouées sur le tas (opt.).

Une exécution réelle exige

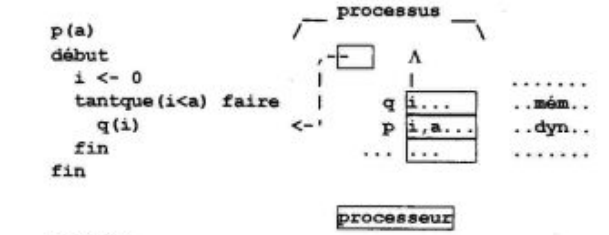
un processeur

le code compilé à la place du code source.

Lorsqu'un (sous-)programme doit être exécuté, ses paramètres et variables locales sous-programme sont alloués au sommet de la pile d'exécution. Lorsque son exécution est terminé, ce sommet est dépilé.

Le pointeur d'instruction désigne l'instruction actuellement à exécuter dans ce sous-programme.

Schéma



Définition

on appelle processus le couple pointeur et pile d'exécution.

Le rôle du processeur est de mettre en œuvre le processus d'exécution réel.

Certaines machines peuvent comporter plusieurs processeurs.

Avec plusieurs processeurs la machine peut faire fonctionner en parallèle plusieurs processus. Ces processus peuvent éventuellement avoir le même programme et/ou la même mémoire dynamique.

Mais un seul processeur peut aussi répartir son travail entre plusieurs processus d'exécution pour donner l'illusion du parallélisme.

Définition.

on appelle processus léger (tâche, thread) un objet permettant de simuler un processus réel et donc de donner l'illusion du parallélisme.

Pourquoi simuler le parallélisme ? Les exemples d'application que nous donnerons sont les suivants :

animations

invocations non synchrones

lectures non bloquantes

buffer à sémantique modifiée

parallélisme logique

L'utilisation de processus légers est très fréquente. En fait, souvent un programme comporte plus de processus que ce qu'imagine l'utilisateur. Ainsi, dans un programme Java utilisant des fenêtres il y a au moins

le processus du programme principal

le processus de la boucle des événements

le processus du ramasse-miettes

et probablement bien d'autres.

9.2. Animation

La plupart des jeux d'ordinateur comportent de multiples animations. Prenons un exemple simple : le serpent et les souris.

un serpent, des souris et des hérissons sont représentés sur l'écran. Le serpent avance constamment, le joueur peut le diriger, mais non pas l'arrêter. Le but du jeu est de faire manger toutes les souris par le serpent, mais si le serpent touche un hérisson, il meurt.

Ici le serpent avance indépendamment de l'action du joueur. Dans un tel cas d'animation le programmeur doit prévoir deux processus distincts s'il ne veut pas que le serpent soit ralenti ou même bloqué par les attentes clavier.

un processus s'occupe de faire avancer le serpent

un processus lit les commandes de l'utilisateur.

Sans réaliser un programme aussi complexe, regardons le suivant :

le programme affiche chaque seconde un caractère sur la sortie standard

l'utilisateur peut à tout moment changer le caractère en frappant un nouveau au clavier.

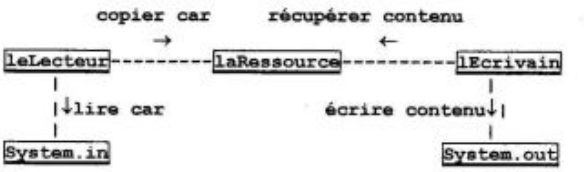
Pour réaliser ce programme, imaginons deux personnes indépendantes munies d'une ressource commune :

le Lecteur attend que l'utilisateur frappe une phrase ; dès que la phrase a été frappé, le Lecteur la lit sur System.in et la copie dans la Ressource.

l'Ecrivain récupère le contenu de la Ressource une fois par seconde et l'écrit sur System.out.

Dans la réalisation suivante la phrase est remplacée par un seul caractère, la ressource se chargeant de le dupliquer 70 fois.

Diagramme de collaboration UML



Les deux personnes sont entièrement désynchronisées et indépendantes.

Nous les coderons donc à l'aide de deux tâches (le programme utilise en réalité un 3e processus, celui du programme principal) :

```
import java.io.*;
class Lecteur extends Thread {
  Ressource laRessource;
  BufferedReader reader;
  public Lecteur(Ressource r) {
    laRessource = r;
    reader = new BufferedReader(
      new InputStreamReader(System.in));
  }
  public void run() {
    while (true) {
      try {
        String car = reader.readLine();
        laRessource.copier(car.charAt(0));
      } catch (IOException ioe) {
      }
    }
  }
}

class Ressource {
  StringBuffer buffer;
  public Ressource() {
    buffer = new StringBuffer();
    for (int i = 0; i < 70; i++)
      buffer.append('a');
  }
  public void copier(char c) {
    for (int i = 0; i < 70; i++)
      buffer.setCharAt(i, c);
  }
  public String recuperer() {
    return buffer.toString();
  }
}


class Ecrivain extends Thread {
  Ressource laRessource;
  public Ecrivain(Ressource r) {
    laRessource = r;
  }
  public void run() {
    while (true) {
      try {
```

```
        String s = laRessource.recuperer();
        System.out.println(s);
        Thread.sleep(1000);
      } catch (Exception ioe) {
      }
    }
  }
}

public class ThreadDemo {
  public static void main(String[] args) {
    Ressource r = new Ressource();
    Ecrivain lEcrivain = new Ecrivain(r);
    Lecteur leLecteur = new Lecteur(r);
    lEcrivain.start();
    leLecteur.start();
  }
}
```

9.3. Classe Thread

La classe Thread comporte quelques méthodes importantes :

```
public class Thread implements Runnable {
  public Thread();
  public Thread(Runnable target);
  public void start();
  public void run();
  public static void sleep(long millis);
  public static void yield();
  public final void join()
  ...
}
```

Un Tread qui tourne (run) peut interrompre son exécution pendant quelques temps par sleep, et peut passer le processeur à un autre processus par yield

Le cycle de vie d'un Thread est :

> création, start (démarre run) une suite de sleep, de yield, éventuellement fin de l'exécution

L'interface Runnable est encore plus simple

```
public interface Runnable {
  public void run();
}
```

Le constructeur

```
public Thread(Runnable target);
```

permet de créer une tâche dont la méthode run() est codée dans une autre classe que ce thread même.

Le thread "voit" alors l'autre classe "de l'intérieur" par l'intermédiaire de run : run peut se servir de code local à cette classe.

On obtient cependant un effet similaire en déclarant tout le thread comme classe locale dans l'autre classe, sans utilisation de l'interface Runnable.

9.4. Appels non synchrones

Soit l'appel

```
liste.trier(ordreCroissant);
faireAutreChose();
```

Si l'exécution est mono-processus, pendant que le code du tri est exécuté, faireAutreChose ne peut être exécuté, car le compteur de programme ne peut à la fois pointer une instruction à l'intérieur de trier et sur faireAutreChose.

Par contre dans notre programme démo - qui n'est pas séquentiel - nous avons les deux instructions

```
ecrivain.start();
lecteur.start();
```

La première met en route un thread qui possède son propre pointeur de programme, si bien que celui du programme main passe immédiatement à lecteur.start. Dans notre cas, sans thread notre programme serait piégé dans la boucle infinie de l'écrivain, et ne serait jamais arrivé à la seconde instruction start !

Définition

> un appel (invocation) est dit asynchrone si l'appelant n'est pas obligé d'attendre pour poursuivre que l'exécution de l'appel soit terminé
>
> il est dit synchrone sinon.

Les tâches peuvent servir à réaliser des appels asynchrones. Inversement, un appel qui n'implique pas une autre tâche est probablement synchrone.

9.5. Lecture non bloquante

Une lecture dans un flux d'entrée est une opération qui peut éventuellement bloquer le processus jusqu'à ce que ce qui doit être lu soit disponible dans le flux. Ce blocage peut être fâcheux.

Soit par exemple un programme de jeu d'échecs fonctionnant en réseau. Lorsqu'un joueur a joué, le programme écrit le mouvement dans un flux sortie réseau et passe donc la main à l'adversaire. Puis il doit recevoir en retour le mouvement de l'adversaire par une lecture dans un flux entrée réseau.

Une réalisation maladroite de cette prise d'information risque de bloquer tout le programme dans l'attente de la réponse.

```
sortieReseau.ecrire(notreMouvement);
entreeReseau.lire(sonMouvement);
```

Notamment, si le programme comporte une IHM fenêtrée, la fenêtre risque d'être "figée" : les commandes souris et clavier ne fonctionnent plus.

La solution consiste à utiliser une tache dédiée à la lecture. Seule cette tâche sera bloquée. Lorsque la lecture est terminée, elle avertit le programme qui aura continué à fonctionner normalement entre-temps.

```
void run() {
  while (true) {
    entreeReseau.lire(sonMouvement);
    leProgramme.avertir(sonMouvement);
  }
}
```

En fait, c'est exactement ce que nous avons fait dans la démo : la tâche Ecrivain a besoin de "lire" les valeurs du clavier, mais aussi de continuer à afficher le caractère courant. Ici c'est le Lecteur qui représente la tâche dédiée à la lecture.

9.6. Synchronisation

S'il est vrai qu'un seul processus ne peut pas exécuter à la fois deux instructions différentes, deux processus peuvent par contre très bien exécuter le même code en même temps.

Deux processus peuvent donc notamment en même temps accéder à une même ressource, par exemple une imprimante ou une donnée.

Ceci peut causer de sérieux problèmes si au moins un des processus modifie la valeur de la donnée. Imaginez le résultat à l'impression si deux processus pouvaient avoir accès à l'imprimante en même temps sans le passage par le spooler.

Pour illustrer ceci, modifions notre programme :

```
class Ressource {
  ...
  public void copier(char c) {
    try {
      for (int i = 0; i < 70; i++) {
        buffer.setCharAt(i, c);
        Thread.sleep(10);          // prend du temps
      }
    } catch (Exception ioe) {
    }
  }
}
```

Copier le caractère dans la ressource prend maintenant environ 0,7 sec. Comme l'écrivain récupère la chaîne toutes les secondes, il est assez probable qu'il trouvera une chaîne non entièrement installée par le Lecteur.

Nous avons un problème similaire à celui de l'accès concurrent à l'imprimante. Comment certifier qu'un seul des processus ait accès à la fois à la ressource à ?

Tout langage offrant des processus, offre des primitives assurant l'exclusion mutuelle aux ressources.

En java les primitives sont

synchronized

notify - wait

Voyons cela :

```
class Ressource {
  ...
  public void copier(int c) {
    try {
      synchronized(this) {
        for (int i = 0; i < 70; i++) {
          buffer.setCharAt(i, (char)c);
          Thread.sleep(10);
        }
      }
    } catch (Exception ioe) {
    }
  }
  public String recuperer() {
    synchronized(this) {
      return buffer.toString();
    }
  }
}
```

Maintenant tous les accès (sauf ici constructeur) sont synchronisées (à accès exclusifs serait un mot plus approprié) par rapport au même objet (nous avons utilisé this, mais tout autre objet aurait fait l'affaire).

Une syntaxe différente avec le même effet est

```
class Ressource {
  ...
  public synchronized void copier(char c) {
    try {
      for (int i = 0; i < 70; i++) {
        buffer.setCharAt(i, (char)c);
        Thread.sleep(10);
      }
    } catch (Exception ioe) {
    }
  }
  public synchronized String recuperer() {
    return buffer.toString();
  }
}
```

9.7. Notify - wait

Nous constatons qu'un processus qui cherche à lire dans un flux sera bloqué à l'attente d'un caractère. En fait il est mis en attente - endormi - et réveillé par l'événement "caractère disponible".

Est-il possible plus généralement d'endormir un processus à l'attente d'un événement ?

La réponse est : oui, le langage offre une construction syntaxique pour cela, notify - wait, mais l'événement n'est pas de haut niveau au sens EventObject.

Exemple.

Un Buffer ou file d'attente "premier entré - premier sorti" offre au moins deux méthodes

```
void deposer(Object o)
Object prendre()
```

Le buffer peut être borné ou infini, mais un problème constant est : comment spécifier "prendre" dans le cas où le buffer est vide ?

lancer une exception

est la seule spécification raisonnable dans un contexte mono-processus. Mais dans contexte multi-processus la spécification suivante est possible

tout processus qui veut prendre dans un buffer vide sera obligé d'attendre l'événement "le buffer n'est plus vide" (un autre processus vient de déposer un objet).

Voici comment on peut coder en Java un tel buffer

```
class Buffer {
  Vector fileDAttente = new Vector();
  public synchronized void deposer(Object o) {
    fileDAttente.add(o);
    notify();
  }
  public synchronized Object prendre() {
    try {
      while (fileDAttente.isEmpty()) wait();
    } catch (InterruptedException ie) {
      return null;
    }
    Object o = fileDAttente.remove(0);
    return o;
  }
}
```

Il faut considérer que la construction suivante est un tout syntaxique

```
synchronized faireSiCondition() {
  try {while (! condition) wait();} catch ...
    ... ce qui doit être fait
}
```

En outre il faut qu'un autre thread puisse changer la condition

```
synchronized changerCondition() {
  ... ce code peut changer la cond.
  notify();       // ou notyfyAll()
}
```

9.8. Parallélisme

Soit à multiplier une matrice rectangulaire n, m par une matrice colonne. Les n produits de lignes par la matrice colonne sont indépendants et peuvent éventuellement être faits en parallèle.

Voici une façon de simuler cela avec des tâches en Java.

Le programme suivant utilise un tableau de processus, chaque processus étant un objet ProdLigne capable de multiplier une ligne par la colonne.

```
class Matrices {
  static double[][] mat;
  static double[] col;
  public static double[] multiplier(double[][] matrice,
                                    double[] colonne)
  {
    mat = matrice;
    col = colonne;
    double[] resultat = new double[mat.length];
    ProdLigne[] pl = new ProdLigne[mat.length];
    for (int i = 0; i < mat.length; i++) {
      pl[i] = new ProdLigne(i);
      pl[i].start();
    }
    for (int i = 0; i < mat.length; i++) {
      try {
        pl[i].join();
        resultat[i] = pl[i].resultat();
      } catch (InterruptedException ie) {
      }
    }
    return resultat;
  }
  ... // déclaration de ProdLigne
}
```

La classe ProdLigne est déclarée localement à la classe Matrices, de cette façon elle a accès aux champs *mat* et *col* de celle-ci.

```
  static class ProdLigne extends Thread {
    double result;
    int index;
    public ProdLigne(int i) {
      index = i;
    }
    public void run() {
      result = 0;
      for (int j = 0; j < mat[index].length; j++) {
        result += mat[index][j] * col[j];
    } }
    public double resultat() {
      return result;
    }
  }
```

Nous pouvons utiliser Produit ainsi :

```
double[] resultat = Matrices.multiplier(matrice, colonne);
```

Cette forme ne révèle pas que le calcul est en réalité fait en parallèle.

9.9. Blocage

Il est facile de créer un code qui peut provoquer un interblocage entre deux ou plusieurs threads :

```
class Bloqueur {
  private Object sema1 = new int[1];
  private Object sema2 = new int[1];
  public void pieger() {
    synchronized(sema1) {
      synchronized(sema2) {
        ...
      }
    }
  }
  public void attraper() {
    synchronized(sema2) {
      synchronized(sema1) {
        ...
      }
    }
  }
```

Il sera toujours hasardeux pour un processus d'appeler pieger ou attraper, car si un autre processus appelle l'autre méthode, les deux risquent de rester bloqués.

Il peut être très difficile de prouver ou de s'assurer par des tests qu'un programme est exempt de problèmes d'interblocage.

9.10. Atomicité

Une opération est dite sûre (dans un environnement multi-thread) si son résultat ne peut pas être corrompu par un autre processus.

Une opération est dite atomique s'il est garanti qu'un seul thread peut l'exécuter à la fois.

- toute opération atomique est sûre
- un code synchronized est atomique
- la consultation et l'affectaion d'un int (ou un type moins volumineux) sont atomiques
- la consultation et l'affectaion d'un type plus volumineux qu'un int (long, double) ne sont pas garantis atomiques.

Toute autre opération est probablement non atomique, et devra donc être protégée par un synchronized si elle n'est pas sûre !

9.11. Processus et composants Swing

9.11.1. Le problème

Pour une question d'efficacité à l'exécution la plupart des méthodes des composants Swing ne sont pas sûres dans un environnement multi-processus.

La conséquence en est la règle suivante :

- toutes les opérations qui manipulent ou dépendent de l'état d'un composant Swing devraient être exécutées par le processus qui gère le cycle des événements, du moins à partir du moment où le composant a été réalisé (affiché).

Cette règle possède quelques exceptions

- quelques méthodes sont sûres (signalées dans la documentation par **"This method is thread safe, although most Swing methods are not"**)
- repaint(), revalidate(), and invalidate() sont sûres
- add et removeXXXListener sont sûres.

Pour surmonter ce problème et autoriser l'intervention sûre d'un autre processus, Swing offre quelques outils. Nous allons citer deux :

- la méthode SwingUtilities.invokeLater
- la classe Timer.

9.11.2. invokeLater

Cette méthode permet à un Thread quelconque de passer le travail à faire au Thread des événements, comme cela aucun conflit entre les deux Threads ne peut survenir.

Au lieu d'invoquer directement `travailAFaire()` le Thread invoque

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    travailAFaire();
  }
}
```

9.11.3. Timer

La classe Timer est appropriée pour déclencher une action à une certaine distance dans le temps, ou de façon répétitive.

Au lieu de créer un Thread avec une méthode run du genre

```
public void run() {
  while(true) {
    faireQqChose();
    repaint();
    try {
      Thread.sleep(delai);
    } catch(InterruptedException ie) {
    }
  }
}
```

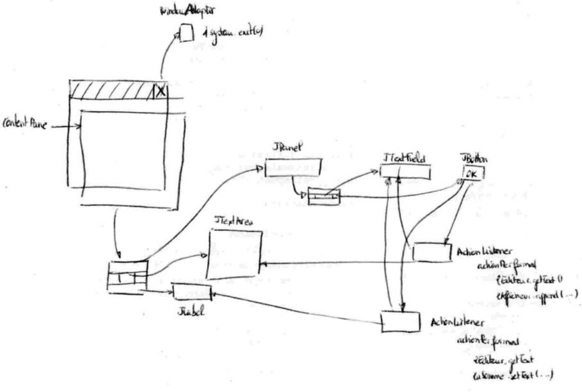
on crée un Timer qui laisse le Thread des événements réaliser ce qui est à faire par l'intermédiaire d'un ActionListener :

```
Timer timer = new Timer(delai, new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    faireQqChose();
    repaint();
  }
} );
timer.setInitialDelay(0);
timer.setCoalesce(true);
timer.setRepeats(true);
...
timer.start();
```

9.12. Conclusion

L'utilisation de Threads ouvre de nombreuses possibilités et peuvent radicalement changer la sémantique comportementale d'une classe.

Cependant les pièges sont nombreux, et dans certains cas les programmes multi-processus peuvent cacher des défauts très difficiles à déceler et à éliminer.


WindowAdapter
ContentPane
JPanel
JTextField
JButton
OK
JTextArea
JLabel
ActionListener
ActionListener