Chapitre 10

Les principaux contrôles de Swing

Connaissances requises

- Cases à cocher (*JCheckBox*) et boutons radio (*JRadioButton*) ; construction ; événements générés : *Action* et *Item* (méthode *itemStateChanged*) ; méthodes *isSelected* et *setSelected* ; groupes de boutons radio (*ButtonGroup*)
- Étiquettes (*JLabel*) ; construction ; modification de libellé (*setText*)
- Champs de texte (*JTextField*) ; construction ; méthodes *getText* , *setEditable* et *setColumns* ; événements générés : *Action* et *Focus* (méthodes *focusGained* et *focusLost*) ; exploitation fine (interface *DocumentListener*, méthodes *insertUpdate, removeUpdate* et *changedUpdate*)
- Boîtes de liste (*JList*) ; construction et choix du type de sélection (simple, multiple, intervalle) ; méthodes *getSelectedValue, getSelectedValues*, *getSelectedIndex* et *getSelectedIndices* ; événements générés : *ListSelection* (méthodes *valueChanged* et *getValueIsAdjusting*)
- Boîte combo (*JComboBox*) ; construction ; méthodes *setEditable* et *getSelectedIndex* ; événements générés : *Action*, *Item* (méthode *itemStateChanged*), *Focus* (méthodes *focusGained* et *focusLost*) ; évolution dynamique : *addItem*, *addItemAt* et *removeItem*

Note : les boutons (*JButton*) ont fait l'objet du Chapitre 8.

112 Cases à cocher

Écrire un programme qui affiche deux boutons marqués *RAZ* et *Etat* et trois cases à cocher, de la façon suivante :

L'action sur le bouton *Etat* provoquera l'affichage en fenêtre console des cases sélectionnées. L'action sur *RAZ* remettra les trois cases à l'état non coché. Enfin, on signalera en fenêtre console les événements de type *Action* et *Item* associés à chacune des trois cases (en précisant la source concernée).

Solution

Nous placerons les trois cases dans un panneau associé à la fenêtre. Nous faisons de la fenêtre l'écouteur des boutons et des cases. Comme l'impose l'énoncé, nous redéfinissons à la fois les méthodes *actionPerformed* et *itemStateChanged*.

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

class MaFenetre extends JFrame implements ActionListener, ItemListener
{ public MaFenetre ()
  { setTitle ("Cases a cocher") ;
    setSize (300, 140) ;
    Container contenu = getContentPane () ;
       // les deux boutons
    boutRaz = new JButton ("RAZ") ;
    boutRaz.addActionListener (this) ;
    contenu.add (boutRaz, "North") ;
    boutEtat = new JButton ("Etat") ;
    boutEtat.addActionListener (this) ;
    contenu.add (boutEtat, "South") ;
       // les cases a cocher dans un panneau
    pan = new JPanel () ;
    contenu.add (pan) ;
    cercle = new JCheckBox ("Cercle") ;
    pan.add (cercle) ;
```

```
    cercle.addActionListener (this) ;
    cercle.addItemListener (this) ;

    rectangle = new JCheckBox ("Rectangle") ;
    pan.add (rectangle) ;
    rectangle.addActionListener (this) ;
    rectangle.addItemListener (this) ;
    triangle = new JCheckBox ("Triangle") ;
    pan.add (triangle) ;
    triangle.addActionListener (this) ;
    triangle.addItemListener (this) ;
  }
  public void actionPerformed (ActionEvent e)
  { Object source = e.getSource() ;
    if (source == boutRaz)
    { cercle.setSelected (false) ;
      rectangle.setSelected (false) ;
      triangle.setSelected (false) ;
    }
    if (source == boutEtat)
    { System.out.print ("Cases selectionnees : ") ;
      if (cercle.isSelected())    System.out.print (" cercle ") ;
      if (rectangle.isSelected()) System.out.print (" rectangle ") ;
      if (triangle.isSelected())  System.out.print (" triangle ") ;
      System.out.println() ;
    }
    if (source == cercle)    System.out.println ("Action case cercle") ;
    if (source == rectangle) System.out.println ("Action case rectangle") ;
    if (source == triangle)  System.out.println ("Action case triangle") ;
  }
  public void itemStateChanged (ItemEvent e)
  { Object source = e.getSource() ;
    if (source == cercle)    System.out.println ("Item case cercle") ;
    if (source == rectangle) System.out.println ("Item case rectangle") ;
    if (source == triangle)  System.out.println ("Item case triangle") ;
  }
  private JButton boutRaz, boutEtat ;
  private JPanel pan ;
  private JCheckBox cercle, rectangle, triangle ;
}

public class Coches
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

```
Item case cercle
Action case cercle
```

```
Item case rectangle
Action case rectangle
Cases selectionnees :  cercle  rectangle
Item case cercle
Item case rectangle
Item case triangle
Item case cercle
Action case cercle
Item case rectangle
Action case rectangle
Cases selectionnees :  cercle  rectangle
```

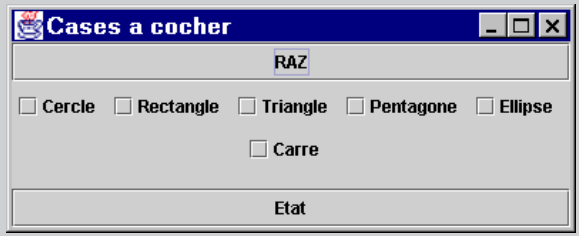
On notera qu'à chaque événement *Action* relatif à une case à cocher correspond toujours un événement *Item*. La réciproque est fausse puisqu'un événement *Item* peut être généré suite à une modification par programme de l'état d'une case ; dans ce cas, elle ne génère pas d'événement *Action*.

Remarque

Plusieurs instructions semblables doivent être écrites pour chaque case à cocher. Si le nombre de cases devenait important, cela pourrait s'avérer fastidieux. Il serait alors préférable de s'acheminer vers une solution plus concise telle que l'écriture de méthodes (statiques) d'intérêt général, par exemple pour l'ajout d'une case de référence donnée à la fenêtre. On pourrait aussi conserver dans la fenêtre un tableau des références des cases ainsi qu'un tableau de chaînes correspondant à leurs libellés...

113 Cases à cocher en nombre quelconque

Généraliser le programme de l'exercice 105, de manière que le nombre de cases à cocher puisse être quelconque et déterminé lors de l'appel du constructeur de la fenêtre, auquel on fournira un tableau de chaînes contenant les libellés à associer aux cases :



Les messages en fenêtre console continueront de repérer une case à cocher par son libellé.

Solution

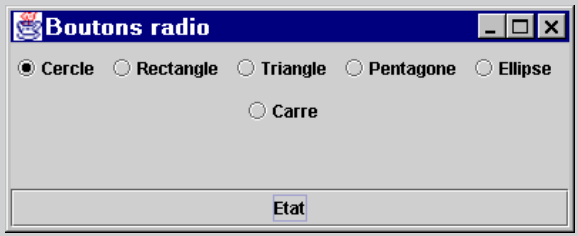
On continue naturellement à placer les cases dans un panneau. Les différents écouteurs restent les mêmes. Mais, cette fois, on va conserver les références des cases dans un tableau dont la taille est égale à celle du tableau de chaînes reçu en argument du constructeur de la fenêtre.

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame implements ActionListener, ItemListener
{ public MaFenetre (String libelles[])
  { setTitle ("Cases a cocher") ; setSize (400, 160) ;
    Container contenu = getContentPane () ;
       // les deux boutons
    boutRaz = new JButton ("RAZ") ;
    boutRaz.addActionListener (this) ;
    contenu.add (boutRaz, "North") ;
    boutEtat = new JButton ("Etat") ;
    boutEtat.addActionListener (this) ;
    contenu.add (boutEtat, "South") ;
       // les cases a cocher dans un panneau
    pan = new JPanel () ;  contenu.add (pan) ;
    this.libelles = libelles ;
    nbCases = libelles.length ;
    cases = new JCheckBox [nbCases] ;
    for (int i=0 ; i<nbCases ; i++)
    { cases[i] = new JCheckBox (libelles[i]) ;
      pan.add (cases[i]) ;
      cases[i].addActionListener (this) ;
      cases[i].addItemListener (this) ;
    }
  }
  public void actionPerformed (ActionEvent e)
  { Object source = e.getSource() ;
    if (source == boutRaz)
      for (int i=0 ; i<nbCases ; i++)
        cases[i].setSelected (false) ;
    if (source == boutEtat)
    { System.out.print ("Cases selectionnees : ") ;
      for (int i=0 ; i<nbCases ; i++)
        if (cases[i].isSelected()) System.out.print (libelles[i]+ " ") ;
      System.out.println() ;
    }
    for (int i=0 ; i<nbCases ; i++)
      if (source == cases[i]) System.out.println ("Action case " + libelles[i]) ;
  }
  public void itemStateChanged (ItemEvent e)
  { Object source = e.getSource() ;
    for (int i=0 ; i<nbCases ; i++)
      if (source == cases[i])  System.out.println ("Item case " + libelles[i]) ;
  }
```

```
  private JButton boutRaz, boutEtat ;
  private JPanel pan ;
  private JCheckBox cases[] ;
  private String libelles[] ;
  private int nbCases ;
}

public class Cochesb
{ public static void main (String args[])
  { String libelles[] = {"Cercle", "Rectangle", "Triangle", "Pentagone",
                         "Ellipse", "Carre"} ;
    MaFenetre fen = new MaFenetre(libelles) ;
    fen.setVisible(true) ;
  }
}
```

114 Boutons radio en nombre quelconque

Écrire un programme qui affiche un bouton marqués *Etat* et un (seul) groupe de boutons radio de la façon suivante :



Les libellés des boutons radio seront fournis en argument du constructeur de la fenêtre.

L'action sur le bouton *Etat* provoquera l'affichage en fenêtre console du libellé associé au bouton radio sélectionné. On signalera en fenêtre console les événements de type *Action* associés.

Solution

On peut facilement adapter le programme de l'exercice 106, en remplaçant les objets de type *JCheckBox* par des objets de type *JRadioButton* et en supprimant l'écoute des événements *Item*. Il faut simplement prendre soin de rattacher les différents boutons radio à un groupe (objet de type *ButtonGroup*), afin d'obtenir le comportement attendu d'un groupe : la sélection d'un des boutons du groupe désactive tous les autres.

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame implements ActionListener
{ public MaFenetre (String[] libelles)
  { setTitle ("Boutons radio") ;
    setSize (400, 160) ;
    Container contenu = getContentPane () ;
    boutEtat = new JButton ("Etat") ;
    boutEtat.addActionListener (this) ;
    contenu.add (boutEtat, "South") ;
       // les boutons radio dans un panneau
    pan = new JPanel () ;
    contenu.add (pan) ;
    this.libelles = libelles ;
    nbBoutons = libelles.length ;
    ButtonGroup groupe = new ButtonGroup() ;
    boutons = new JRadioButton [nbBoutons] ;
    for (int i=0 ; i<nbBoutons ; i++)
    { boutons[i] = new JRadioButton (libelles[i]) ;
      pan.add (boutons[i]) ;
      groupe.add (boutons[i]) ;
      boutons[i].addActionListener (this) ;
    }
    if (nbBoutons > 0) boutons[0].setSelected(true) ;
  }
  public void actionPerformed (ActionEvent e)
  { Object source = e.getSource() ;
    if (source == boutEtat)
    { System.out.print ("Bouton selectionne = ") ;
      for (int i=0 ; i<nbBoutons ; i++)
        if (boutons[i].isSelected()) System.out.print (libelles[i]+ " ") ;
      System.out.println() ;
    }
    for (int i=0 ; i<nbBoutons ; i++)
      if (source == boutons[i])
         System.out.println ("Action bouton " + libelles[i]) ;
  }
  private JButton boutDef, boutEtat ;
  private JPanel pan ;
  private JRadioButton boutons[] ;
  private String libelles[] ;
  private int nbBoutons ;
}
public class Radios
{ public static void main (String args[])
  { String libelles[] = {"Cercle", "Rectangle", "Triangle", "Pentagone",
                         "Ellipse", "Carre"} ;
    MaFenetre fen = new MaFenetre(libelles) ;
    fen.setVisible(true) ;
  }
}
```

```
Action bouton Triangle
Action bouton Carre
Bouton selectionne = Carre
Action bouton Pentagone
Action bouton Rectangle
Bouton selectionne = Rectangle
Action bouton Cercle
```

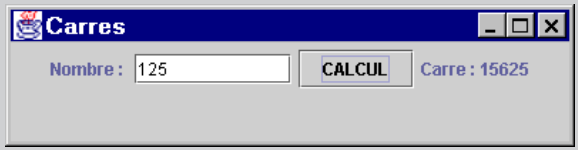
Notez que nous avons pris soin de sélectionner initialement le premier bouton du groupe (en nous assurant que la dimension du tableau de libellés était non nulle).

Remarque

Ici, rien ne montre à l'utilisateur que nos boutons radio font partie d'un même groupe. Dans un programme réel, on sera souvent amené à mettre en évidence un groupe en le plaçant dans un panneau qu'on pourra colorer différemment du reste de la fenêtre ou encore doter d'une "bordure" à l'aide de la méthode *setBorder*.

115 Champs de texte

Écrire un programme qui permet à l'utilisateur de saisir un nombre entier dans un champ texte et qui en affiche le carré lorsqu'il agit sur un bouton marqué CALCUL :



Le programme devra gérer convenablement le cas où l'utilisateur entre autre chose qu'un nombre dans le champ texte ; il pourra par exemple remettre ce champ à blanc.

Solution

Ici, nous pouvons nous permettre d'introduire directement dans la fenêtre les différents contrôles dont nous avons besoin. Nous remplaçons simplement le gestionnaire par défaut par un gestionnaire de type *FlowLayout*.

Nous utilisons des objets de type *JLabel* pour les libellés, ainsi que pour la valeur du carré. La saisie du nombre se fait dans un objet nommé *nombre* de type *JTextField*.

Ici, nous n'avons pas à nous préoccuper des événements générés par *nombre* puisque le calcul proprement dit est déclenché par une action extérieure à l'objet. En revanche, nous devons traiter les événements de type *Action* déclenchés par le bouton. Nous y récupérons le contenu du champ texte que nous convertissons en entier avec la méthode *Integer.parseInt*. Celle-ci déclenche une exception *NumberFormatException* lorsque la chaîne ne correspond pas à un nombre entier (y compris lorsqu'elle contient trop de chiffres). Dans le gestionnaire d'exception correspondant, nous nous contentons de remettre à blanc le contenu du champ texte.

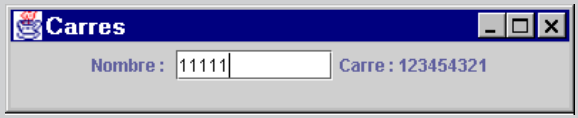
Ici, nous calculons le carré du nombre dans le type *long*, ce qui évite tout problème de dépassement de capacité.

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
class MaFenetre extends JFrame implements ActionListener
{ public MaFenetre ()
  { setTitle ("Carres") ;
    setSize (400, 100) ;
    Container contenu = getContentPane() ;
    contenu.setLayout (new FlowLayout() ) ;
    labNombre = new JLabel (etiqNombre) ;
    contenu.add(labNombre) ;
    nombre = new JTextField (10) ;
    contenu.add(nombre) ;
    boutonCalcul = new JButton ("CALCUL") ;
    contenu.add(boutonCalcul) ;
    boutonCalcul.addActionListener(this) ;
    labCarre = new JLabel (etiqCarre) ;
    contenu.add(labCarre) ;
  }
  public void actionPerformed (ActionEvent e)
  { if (e.getSource() == boutonCalcul)
    try
    { String texte = nombre.getText() ;
      int n = Integer.parseInt(texte) ;
      long carre = (long)n*(long)n ;
      labCarre.setText (etiqCarre + carre) ;
    }
    catch (NumberFormatException ex)
    { nombre.setText ("") ;
      labCarre.setText (etiqCarre) ;
    }
  }
  private JLabel labNombre, labCarre ;
  private JTextField nombre ;
  static private String etiqNombre = "Nombre : ", etiqCarre = "Carre : " ;
  private JButton boutonCalcul ;
}
```

```
public class Carre
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

116 Champ de texte et événements Action et Focus

Adapter le programme de l'exercice 108 en supprimant le bouton CALCUL et de manière que le carré du nombre s'affiche lorsque l'utilisateur valide l'information saisie ou lorsque le champ de texte perd le focus :



Solution

Il suffit que les actions précédemment réalisées dans l'écouteur du bouton soient transposées :

- dans l'écouteur de l'événement *focusLost* associé au champ de texte,
- dans l'écouteur de l'événement *Action* associé à ce même champ de texte.

Pour éviter de dupliquer les instructions correspondantes, nous prévoyons une méthode de service nommée *actualise*.

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

class MaFenetre extends JFrame implements ActionListener, FocusListener
{ public MaFenetre ()
  { setTitle ("Carres") ;
    setSize (400, 100) ;
    Container contenu = getContentPane() ;
    contenu.setLayout (new FlowLayout() ) ;

    labNombre = new JLabel (etiqNombre) ;
    contenu.add(labNombre) ;
    nombre = new JTextField (10) ;
    contenu.add(nombre) ;
```

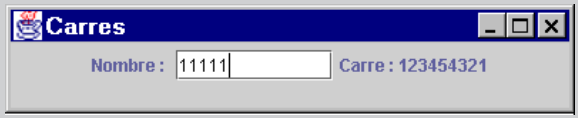
```
    nombre.addFocusListener (this) ;    // pour la perte de focus
    nombre.addActionListener (this) ;   // pour la validation
    labCarre = new JLabel (etiqCarre) ;
    contenu.add(labCarre) ;
  }
  public void actionPerformed (ActionEvent e)
  { actualise () ;
  }

  public void focusLost (FocusEvent e)
  { actualise () ;
  }
  public void focusGained (FocusEvent e)
  {
  }

  public void actualise()
  { try
    { String texte = nombre.getText() ;
      int n = Integer.parseInt(texte) ;
      long carre = (long)n*(long)n ;
      labCarre.setText (etiqCarre + carre) ;
    }
    catch (NumberFormatException ex)
    { nombre.setText ("") ;
      labCarre.setText (etiqCarre) ;
    }
  }

  private JLabel labNombre, labCarre ;
  private JTextField nombre ;
  static private String etiqNombre = "Nombre : ", etiqCarre = "Carre : " ;
  private JButton boutonCalcul ;
}

public class Carre1
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

Remarque

Comme notre fenêtre ne comporte qu'un seul composant susceptible de recevoir le focus, le seul moyen de faire perdre le focus au champ de texte consiste ici à rendre active une autre fenêtre.

117 Écoute permanente d'un champ de texte

Adapter le programme de l'exercice 108 en supprimant le bouton CALCUL et de manière que le carré du nombre s'affiche en permanence, indépendamment de toute validation ou de transfert de focus :



Ainsi, ici (si l'utilisateur n'a pas fait de corrections au cours de la frappe), on verra s'afficher successivement les carrés de 1, de 11, de 111... et enfin de 11111.

Solution

Cette fois, il faut savoir que pour implémenter un objet de type *JTextField*, Java utilise à la fois un objet dit "document" (de type *Document*) pour y conserver l'information et un objet dit "vue" pour en fournir la représentation visuelle. Toute modification d'un objet de type *Document* génére un des événements de la catégorie *Document* qu'on traite à l'aide d'un écouteur implémentant l'interface *DocumentListener.* Celle-ci comporte trois méthodes *insertUpdate*, *removeUpdate* et *changedUpdate*. Seules les deux premières sont concernées par un champ de texte. L'objet document associé à un composant s'obtient par la méthode *getDocument*.

Il nous faut donc transposer dans ces deux méthodes les actions précédemment réalisées dans l'écouteur du bouton de l'exercice . Pour éviter de dupliquer les instructions correspondantes, nous prévoyons une méthode de service nommée *actualise*.

Cette fois, cependant, en cas d'exception, nous évitons de remettre à blanc le contenu du champ de texte. En effet, une telle modification risquerait de provoquer une boucle infinie et elle est interdite par Java (elle provoque une exception). Nous nous contentons d'effacer la valeur affichée comme carré.

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
import javax.swing.event.* ;  //  utile pour DocumentListener

class MaFenetre extends JFrame implements DocumentListener
{ public MaFenetre ()
  { setTitle ("Carres") ;
    setSize (400, 100) ;
```

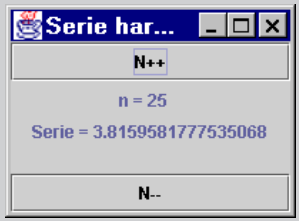
```
    Container contenu = getContentPane() ;
    contenu.setLayout (new FlowLayout() ) ;

    labNombre = new JLabel (etiqNombre) ;
    contenu.add(labNombre) ;
    nombre = new JTextField (10) ;
    contenu.add(nombre) ;
    nombre.getDocument().addDocumentListener (this) ;
    labCarre = new JLabel (etiqCarre) ;
    contenu.add(labCarre) ;
  }

  public void insertUpdate (DocumentEvent e)
  { actualise () ;
  }

  public void removeUpdate (DocumentEvent e)
  { actualise () ;
  }

  public void changedUpdate (DocumentEvent e)
  {
  }

  public void actualise()
  { try
    { String texte = nombre.getText() ;
      int n = Integer.parseInt(texte) ;
      long carre = (long)n*(long)n ;
      labCarre.setText (etiqCarre + carre) ;
    }
    catch (NumberFormatException ex)
    { //nombre.setText ("") ;  generait une exception
      labCarre.setText (etiqCarre) ;
    }
  }

  private JLabel labNombre, labCarre ;
  private JTextField nombre ;
  static private String etiqNombre = "Nombre : ", etiqCarre = "Carre : " ;
  private JButton boutonCalcul ;
}

public class Carre2
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

118 Synthèse : série harmonique

Écrire un programme permettant d'afficher la somme partielle de la série harmonique :

s = 1 + 1/2 +1/3 +1/4 + ... + 1/n

La valeur de *n* sera initialisée à 0 (on conviendra alors que *s* vaut 0) et deux boutons marqués N++ et N-- permettront de la faire évoluer :



Solution

Nous conservons le gestionnaire par défaut de la fenêtre, ce qui nous permettra de disposer le bouton N++ avec l'option "North" et le bouton N-- avec l'option "South". Au centre de la fenêtre, nous plaçons un panneau dans lequel nous disposons deux étiquettes (*JLabel*) qui serviront à afficher les informations voulues. Les actions sur les boutons sont gérées dans la fenêtre et elles conduisent à l'actualisation des valeurs de *n* et de la somme correspondante.

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

class MaFenetre extends JFrame implements ActionListener
{ private static String texteN = "n = " ;
  private static String texteSomme = "Serie = " ;

  public MaFenetre ()
  { setTitle ("Serie harmonique") ;
    setSize (200, 150) ;
    Container contenu = getContentPane() ;
    pan = new JPanel () ;
    contenu.add(pan) ;
    boutPlus = new JButton ("N++") ;
    boutPlus.addActionListener (this) ;
    contenu.add (boutPlus, "North") ;
    boutMoins = new JButton ("N--") ;
```

```
    boutMoins.addActionListener (this) ;
    contenu.add (boutMoins, "South") ;

    n = 0 ;
    somme = 0. ;
    valeurN = new JLabel (texteN + n + "  ") ;
    pan.add (valeurN) ;
    valeurSomme = new JLabel (texteSomme + somme) ;
    pan.add (valeurSomme) ;
  }
  public void actionPerformed (ActionEvent e)
  { Object source = e.getSource() ;
    if (source == boutPlus)         { n++ ;
                                      somme += 1./n ;
                                    }
    if (source == boutMoins && n>0) { somme -= 1./n ;
                                      n-- ;
                                    }
    valeurN.setText (texteN + n + "   ") ;
    valeurSomme.setText (texteSomme + somme) ;
  }

  private JPanel pan ;
  private JButton boutPlus, boutMoins ;
  private JLabel valeurN, valeurSomme ;
  private int n ;
  private double somme ;
}

public class Serie
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```

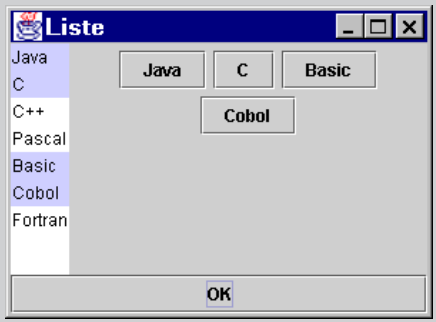
Remarque

Ici, nous avons actualisé la valeur de la somme en lui ajoutant ou en lui soustrayant la valeur 1/*n*. Dès lors que l'utilisateur incrémente et décrémente la valeur de *n* à diverses reprises, ce mode de calcul conduit à un cumul des erreurs. Pour l'éviter, on pourrait recalculer entièrement la valeur de la somme à chaque action sur l'un des boutons.

119 Gestion d'une boîte de liste

Écrire un programme affichant dans une fenêtre des boutons dont les étiquettes sont des noms de langage sélectionnés dans une boîte de liste. La liste permettra de sélectionner un nombre quelconque de plages de valeurs. Les noms des langages seront fixés dans la méthode *main* (et non dans la fenêtre). On proposera deux solutions :

- une où la sélection sera validée par l'action sur un bouton OK :



- une où les boutons affichés dans la fenêtre seront actualisés à chaque modification de la sélection dans la liste (il n'y aura plus de bouton OK).

Solution 1

Les noms de langages sont définis par un tableau de chaînes de la méthode *main* qu'on fournit en argument au constructeur de la fenêtre. La boîte de liste est ajoutée à la fenêtre elle-même avec l'option "West". Un panneau est ajouté au centre de la fenêtre, en vue d'y afficher les boutons voulus.

Le bouton OK est ajouté avec l'option "South" et on gère ses événements de type *Action*. La méthode *actionPerformed* réalise les actions suivantes :

- suppression des boutons du panneau par la méthode *removeAll* (qui supprime tous les composants d'un conteneur) ;
- récupération des valeurs sélectionnées dans la boîte de liste à l'aide de *getSelectedValues*. Elle fournit un tableau d'éléments de type *Object* qui seront convertis en *String*, avant d'être transmis au constructeur de chacun des boutons ;
- appel de la méthode *validate* du panneau pour forcer le recalcul par le gestionnaire de mise en forme.

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

class MaFenetre extends JFrame implements ActionListener
{ public MaFenetre (String noms[])
  { setTitle ("Liste") ;
    setSize (300, 220) ;
    Container contenu = getContentPane() ;
    liste = new JList (noms) ;
    contenu.add (liste, "West") ;
    ok = new JButton ("OK") ;
    contenu.add (ok, "South") ;
    ok.addActionListener (this) ;
    pan = new JPanel () ;
    contenu.add (pan) ;
  }
  public void actionPerformed (ActionEvent e)
  { if (e.getSource() == ok)
    { pan.removeAll () ;  // supprime tous les composants de pan
      Object noms[] = liste.getSelectedValues() ;
      for (int i=0 ; i<noms.length ; i++)
      { JButton bouton = new JButton ((String)noms[i]) ;
        pan.add (bouton) ;
      }
      pan.validate() ;
    }
  }
  private JList liste ;
  private JButton ok ;
  private JPanel pan ;
}
public class Liste
{ public static void main (String args[])
  { String [] nomsLangages = {"Java", "C", "C++", "Pascal", "Basic", "Cobol",
                              "Fortran"} ;
    MaFenetre fen = new MaFenetre(nomsLangages) ;
    fen.setVisible(true) ;
  }
}
```

Solution 2

On supprime le bouton OK et on associe à la boîte de liste un écouteur (ici la fenêtre) implémentant l'interface *ListSelectionListener* :

```
liste.addListSelectionListener (this) ;
```

L'interface *ListSelectionListener* comporte une seule méthode *valueChanged*. Les événements correspondants sont générés plus souvent qu'il n'est nécessaire pour une gestion usuelle de la boîte. Il est préférable de faire appel à la méthode *getValueIsAdjusting* de la classe *ListSelectionEvent*, afin d'éviter les événements de transition. Voici comment pourrait se présenter la méthode *valueChanged* :

```
public void valueChanged (ListSelectionEvent e)
  { if ((e.getSource() == liste) && (!e.getValueIsAdjusting()))
    { pan.removeAll () ;  // supprime tous les composants de pan
      Object noms[] = liste.getSelectedValues() ;
      for (int i=0 ; i<noms.length ; i++)
      { JButton bouton = new JButton ((String)noms[i]) ;
        pan.add (bouton) ;
      }
      pan.validate() ;
    }
  }
```

Le programme complet ainsi adapté figure sur le site Web d'accompagnement sous le nom *Liste1.java*.

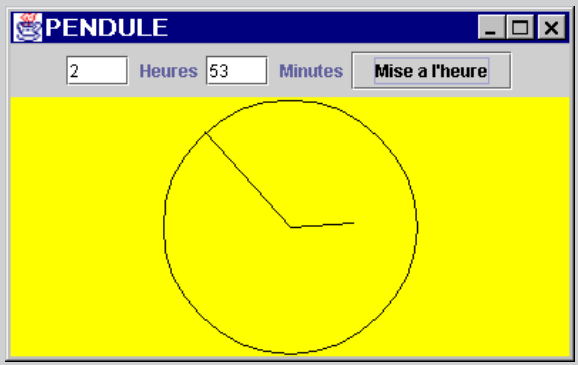
Remarque

Rappelons que, par défaut, un boîte de liste autorise la sélection de plusieurs plages de valeurs. On a affaire au type *MULTIPLE_INTERVAL_SELECTION*. On peut imposer un autre type à l'aide de la méthode *setSelectionMode*.

D'autre part, une boîte de liste ne dispose pas de barre de défilement. Si celle-ci s'avère nécessaire, il faut alors introduire la boîte de liste dans un "panneau de défilement" (*JScrollPanel*) et définir le nombre de valeurs visibles à un moment donné par *setVisibleRowCount*.

120 Synthèse : pendule

Afficher une pendule indiquant l'heure fournie par le biais de deux champs de texte (et validée par un bouton "Mise à l'heure") :



La pendule sera dessinée sur un fond de couleur jaune et on s'arrangera pour qu'elle soit la plus grande possible (tout en étant entièrement visible) en tenant compte du fait que l'utilisateur peut modifier les dimensions de la fenêtre. On utilisera la méthode *drawOval (int abscisse, int ordonnee, int largeur, int hauteur)* pour dessiner un cercle.

Solution

On utilise deux panneaux : un pour les champs de texte et le bouton, un pour le dessin de la pendule. Ils sont disposés dans la fenêtre en conservant le gestionnaire par défaut. Les trois contrôles sont écoutés par la fenêtre elle-même.

Pour que la pendule s'ajuste à une éventuelle modification de la fenêtre, il est préférable de la dessiner dans la méthode *paintComponent* du panneau dans lequel elle se trouve. Il faut donc créer une classe spécialisée (nommée ici *PanPendule*) dérivée de *JPanel*. Il apparaît alors un besoin de communication des valeurs saisies entre la fenêtre et le panneau. Pour le régler, les valeurs saisies sont conservées dans la fenêtre qu'on dote de deux méthodes d'accès *getHeures* et *getMinutes*.

Nous prévoyons par défaut des valeurs nulles pour l'heure (heures et minutes). D'autre part, nous gérons les éventuelles erreurs de saisie de l'utilisateur (valeurs non numériques ou simplement incompatibles). Dans ce cas, nous avons prévu de redonner son ancienne valeur au champ de texte correspondant.

L'actualisation de la pendule est tout simplement déclenchée par l'appel de la méhode *repaint* du panneau, en réponse à une action sur le bouton.

Nous dessinons une grande aiguille ayant une taille égale au rayon de la pendule et une petite aiguille ayant la moité de cette taille. Pour dessiner la petite aiguille, nous tenons compte du fait qu'elle se déplace non seulement en fonction du nombre d'heures, mais aussi en fonction du nombre de minutes.

```
import java.awt.* ;
import java.awt.event.* ;
import javax.swing.* ;
class MaFenetre extends JFrame implements ActionListener
{ public MaFenetre ()
  { setTitle ("PENDULE") ;
    setSize (400, 250) ;
    Container contenu = getContentPane() ;
    panControles = new JPanel() ;
    contenu.add (panControles, "North") ;
    saisieHeures = new JTextField (4) ;
    panControles.add (saisieHeures) ;
    etiqHeures = new JLabel (" Heures") ;
    panControles.add (etiqHeures) ;
    saisieMinutes = new JTextField (4) ;
    panControles.add (saisieMinutes) ;
    etiqMinutes = new JLabel (" Minutes") ;
    panControles.add (etiqMinutes) ;
```

```
  ok = new JButton ("Mise a l'heure") ;
  panControles.add (ok) ;
  ok.addActionListener (this) ;
  panPendule = new PanPendule(this) ;
  contenu.add (panPendule) ;
  panPendule.setBackground (Color.yellow) ;
}
public int getMinutes ()
{ return minutes ;
}
public int getHeures ()
{ return heures ;
}
public void actionPerformed (ActionEvent e)
{ int h, m ;   // pour les valeurs saisies
  if (e.getSource() == ok)
  { try
    { String chHeures = saisieHeures.getText() ;
      h = Integer.parseInt (chHeures) ;
    }
    catch (NumberFormatException ex)
    { h = -1 ;  // on force une valeur invalide
      saisieHeures.setText ("") ;
    }
    try
    { String chMinutes = saisieMinutes.getText() ;
      m = Integer.parseInt (chMinutes) ;
    }
    catch (NumberFormatException ex)
    { m = -1 ;  // on force une valeur invalide
      saisieMinutes.setText ("") ;
    }
      // si les valeurs obtenues sont valides, on les place dans
      // les champs heures et minutes et on force le dessin
      // sinon, on replace les anciennes valeurs dans les champs texte
    if ((h>=0) && (h<24) && (m>=0) && (m<60))
      { heures = h ; minutes = m ;
        repaint() ;
      }
     else
      { saisieMinutes.setText (""+minutes) ;
        saisieHeures.setText (""+heures) ;
      }
  }
}
private JPanel panControles ;
private PanPendule panPendule ;
private JTextField saisieHeures, saisieMinutes ;
private JLabel etiqHeures , etiqMinutes ;
```

```
  private JButton ok ;
  private int minutes=0, heures=0 ;
}

class PanPendule extends JPanel
{ public PanPendule (MaFenetre fen)
  { this.fen = fen ;
  }
  public void paintComponent (Graphics g)
  { super.paintComponent(g) ;
     // dessin du cercle
    Dimension dim = getSize() ;
    int largeur = dim.width, hauteur = dim.height ;
    boolean panTropLarge = (largeur>hauteur) ;
    int xCentre = largeur/2, yCentre = hauteur/2 ;
    int rayon ;
    if (panTropLarge) rayon = hauteur/2 - 2 ; else rayon = largeur/2 - 2 ;
    g.drawOval (xCentre-rayon, yCentre-rayon, 2*rayon, 2*rayon) ;
     // dessin grande aiguille
    int minutes = fen.getMinutes() ;
    double angle = Math.PI/2 * (1. - minutes/15.) ;
    g.drawLine (xCentre, yCentre,
                (int)(xCentre+rayon*Math.cos(angle)),
                (int)(yCentre-rayon*Math.sin(angle))) ;
     // dessin petite aiguille
    int heures = fen.getHeures() ;
    angle = Math.PI/2 * (1. - heures/3. - minutes/180.) ;
    g.drawLine (xCentre, yCentre,
                (int)(xCentre+rayon/2.*Math.cos(angle)),
                (int)(yCentre-rayon/2.*Math.sin(angle))) ;
  }
  private MaFenetre fen ;
}

public class Pendule
{ public static void main (String args[])
  { MaFenetre fen = new MaFenetre() ;
    fen.setVisible(true) ;
  }
}
```