Claude Delannoy

Exercices en JAVA

175 exercices corrigés
Couvre Java 8

4e édition

EYROLLES

Chapitre 13

Les événements de bas niveau

Connaissances requises

• Événements de type *MouseEvent* liés aux boutons de la souris (rappel) ; méthodes *mousePressed*, *mouseReleased* et *mouseClicked*

• Identification du bouton de la souris ; méthodes *getModifiers* et constantes correspondantes *InputEvent.BUTTON1_MASK*, *InputEvent.BUTTON2_MASK* et *InputEvent.BUTTON3_MASK*

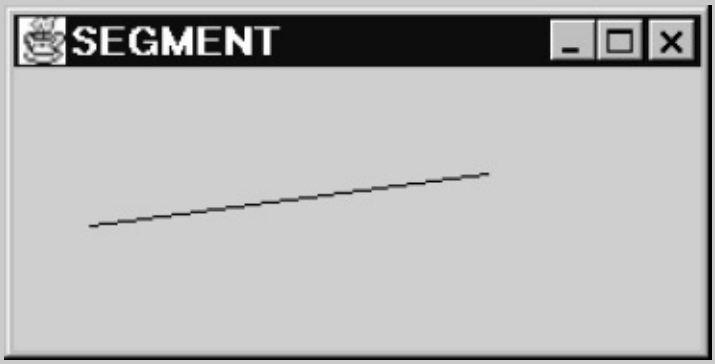
• Gestion des clics multiples ; méthode *getClickCount*

• Gestion des déplacements de la souris ; méthodes *mouseEntered*, *mouseExited*, *mouseMoved* et *mouseDragged*

• Événements de type *KeyEvent* ; méthodes *keyPressed*, *keyReleased* et *keyTyped* ; identification d'une touche par son code de touche virtuelle (méthode *getKeyCode*) ou par le caractère correspondant (méthode *getKeyChar*) ; connaissance de l'état des touches modificatrices (méthodes *isXXXDown* et *getModifiers*) ; source d'un événement clavier

134 Identification des boutons de la souris

Afficher en permanence un segment dans une fenêtre. Son origine sera définie par un clic sur le bouton de gauche de la souris et elle se modifiera à chaque nouveau clic sur ce même bouton. Son extrémité sera définie de la même manière avec le bouton de droite :



Pour obtenir la permanence du dessin, nous tracerons notre segment dans un panneau. Ici, il est plus simple d'écouter les clics (*mouseClicked*) dans le panneau lui-même. Pour identifier le bouton de la souris, nous utilisons la méthode *getModifiers* de la classe *MouseEvent*. Elle fournit un entier dans lequel un bit de rang donné, associé à chacun des boutons, prend la valeur 1. La classe *InputEvent* contient des constantes qu'on peut utiliser comme masque pour faciliter les choses ; ici, ce sont les constantes *BUTTON1_MASK* (bouton de gauche) et *BUTTON3_MASK* (bouton de droite) qui nous intéressent.

Le segment est défini par les coordonnées de son origine (*xOr* et *yOr*) et celles de son extrémité (*xExt* et *yExt*). Deux indicateurs booléens *orConnue* et *extConnue* permettent de savoir si ces informations sont disponibles (elles sont en fait placées à *false* au début du programme). Le dessin proprement dit est réalisé dans la méthode *paintComponent* qui exploite ces différentes informations. Notez qu'il est nécessaire d'appeler *repaint* après un clic gauche ou droite, afin de provoquer l'appel de *paintComponent*.

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame
{ public MaFenetre ()
 { setTitle ("SEGMENT") ;
  setSize (300, 150) ;
  pan = new Panneau () ;
  getContentPane().add (pan) ;
  pan.addMouseListener (pan) ;
 }
 private Panneau pan ;
}
class Panneau extends JPanel implements MouseListener
{ public void paintComponent (Graphics g)
 { super.paintComponent (g) ;
  if (orConnue && extConnue) g.drawLine (xOr, yOr, xExt, yExt) ;
 }
 public void mousePressed (MouseEvent e)
 { int x=e.getX(), y=e.getY() ;
  int modifieurs = e.getModifiers() ;
  if ( (modifieurs & InputEvent.BUTTON1_MASK) != 0)
  { /* clic bouton gauche */
   xOr = x ; yOr = y ;
   orConnue = true ;
   repaint() ;
  }
  if ( (modifieurs & InputEvent.BUTTON3_MASK) != 0)
  { /* clic bouton droite */
   xExt = x ; yExt = y ;
   extConnue = true ;
   repaint() ;
  }
 }
 public void mouseReleased (MouseEvent e) {}
 public void mouseClicked (MouseEvent e) {}
 public void mouseEntered (MouseEvent e) {}
 public void mouseExited (MouseEvent e) {}
 private int xOr, yOr, xExt, yExt ;
```

```
 private boolean orConnue=false, extConnue=false ;
}
public class Segments
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
 }
}
```

135 Vrais doubles-clics

Java ne dispose que d'un seul compteur de clics pour les différents boutons de la souris. Dans ces conditions, il n'est pas possible de distinguer un véritable double-clic de deux clics successifs sur deux boutons différents. Écrire un programme qui détecte les "vrais" doubles-clics sur le bouton de gauche et qui affiche alors un message en fenêtre console.

Nous ferons naturellement de la fenêtre son propre écouteur d'événements souris. Nous utiliserons :

- la méthode *getClickCount* qui fournit le nombre de clics (rapprochés) successifs,
- la méthode *getModifiers* pour identifier le bouton de la souris.

Un indicateur booléen *clicGauche* indique si le dernier clic concernait le bouton de gauche.

Il faut bien prendre garde à :

- mettre l'indicateur *clicGauche* à *false* après un double-clic gauche (vrai ou faux) ainsi qu'après tout clic sur un autre bouton,
- mettre l'indicateur *clicGauche* à *true* après un simple clic gauche.

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

class MaFenetre extends JFrame implements MouseListener
{ public MaFenetre ()
 { setTitle ("DOUBLES CLICS") ;
  setSize (300, 150) ;
  clicGauche = false ;
  addMouseListener (this) ;
 }

 public void mousePressed (MouseEvent e) {}
```

```
 public void mouseReleased (MouseEvent e) {}

 public void mouseClicked (MouseEvent e)
 { int modifieurs = e.getModifiers () ;
  if ((modifieurs & InputEvent.BUTTON1_MASK) != 0)
    /* ici, on a affaire a un clic gauche */
  { if ((e.getClickCount() == 2) && clicGauche)
    { System.out.println ("Double clic gauche") ;
     clicGauche = false ;
    }
   else clicGauche = true ;
  }
  else clicGauche = false ;
 }
 public void mouseEntered (MouseEvent e) {}
 public void mouseExited (MouseEvent e ) {}
 private boolean clicGauche ;
}

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

136 Suivi des déplacements de la souris (1)

Créer une fenêtre dotée d'un bouton. Afficher en fenêtre console des messages de suivi des déplacements de la souris comme dans cet exemple :

```
la souris entre dans la fenetre
la souris quitte la fenetre
la souris entre dans le bouton
la souris quitte le bouton
la souris entre dans la fenetre
la souris quitte la fenetre
la souris entre dans la fenetre
la souris quitte la fenetre
la souris entre dans le bouton
la souris quitte le bouton
la souris entre dans la fenetre
la souris quitte la fenetre
```

Il nous suffit de suivre les événements *mouseEntered* et *mouseExited* ayant pour source le bouton ou la fenêtre.

Ici, nous utilisons pour les deux un même écouteur, à savoir la fenêtre elle-même. Nous y redéfinissons les six méthodes prévues par l'interface *MouseListener* ; ici ce sont *MouseEntered* et *MouseExited* qui nous intéressent. Dans chacune de ces deux méthodes, *getSource* nous permet d'identifier la source de l'événement.

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

class MaFenetre extends JFrame implements MouseListener
{ public MaFenetre ()
 { setTitle ("Evenements souris") ;
  setSize (300, 150) ;
```

```
  contenu = getContentPane() ;
  contenu.setLayout (new FlowLayout()) ;
  addMouseListener (this) ;
  bouton = new JButton ("A") ;
  contenu.add (bouton) ;
  bouton.addMouseListener (this) ;
 }

 public void mousePressed (MouseEvent e) {}
 public void mouseReleased (MouseEvent e) {}
 public void mouseClicked (MouseEvent e) {}

 public void mouseEntered (MouseEvent e)
 { if (e.getSource() == this)
   System.out.println ("la souris entre dans la fenetre") ;
  if (e.getSource() == bouton)
   System.out.println ("la souris entre dans le bouton") ;
 }
 public void mouseExited (MouseEvent e)
 { if (e.getSource() == this)
   System.out.println ("la souris quitte la fenetre") ;
  if (e.getSource() == bouton)
   System.out.println ("la souris quitte le bouton") ;
 }
 private JButton bouton ;
 private Container contenu ;
}
public class DpSour
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
 }
```

L’exemple d’exécution de l’énoncé montre bien que lorsque la souris entre dans le bouton, elle sort de la fenêtre.

Voici une autre solution dans laquelle la fenêtre et le bouton ont été chacun doté d'un écouteur objet d'une classe anonyme (implémentant l'interface *MouseAdapter*). Ici, il n'est plus nécessaire de tester la source d'un événement.

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame
{ public MaFenetre ()
 { setTitle ("Evenements souris") ;
  setSize (300, 150) ;
  contenu = getContentPane() ;
  contenu.setLayout (new FlowLayout()) ;
  addMouseListener (new MouseAdapter()
    { public void mouseEntered (MouseEvent e)
     { System.out.println ("la souris entre dans la fenetre") ;
     }
     public void mouseExited (MouseEvent e)
     { System.out.println ("la souris quitte la fenetre") ;
     }
    }) ;
  bouton = new JButton ("A") ;
  contenu.add (bouton) ;
  bouton.addMouseListener (new MouseAdapter()
    { public void mouseEntered (MouseEvent e)
     { System.out.println ("la souris entre dans le bouton") ;
     }
     public void mouseExited (MouseEvent e)
     { System.out.println ("la souris quitte le bouton") ;
     }
    }) ;
 }
 private JButton bouton ;
 private Container contenu ;
}
public class DpSourb
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
```

```
 }
}
```

Voici une troisième solution dans laquelle la fenêtre et le bouton partagent le même écouteur, là encore objet d'une classe anonyme implémentant l'interface *MouseAdapter*.

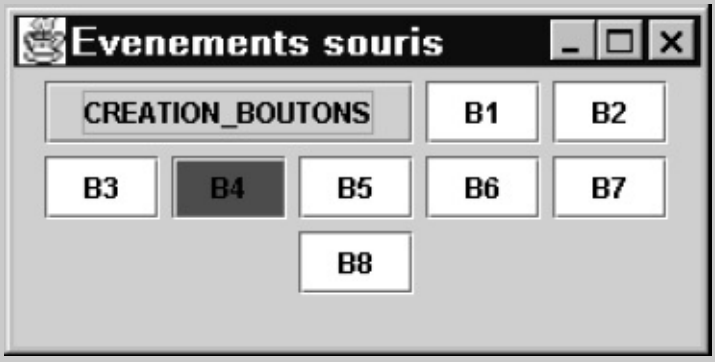
```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;

class MaFenetre extends JFrame
{ public MaFenetre ()
 { setTitle ("Evenements souris") ;
  setSize (300, 150) ;
  contenu = getContentPane() ;
  contenu.setLayout (new FlowLayout()) ;
  MouseAdapter ecout = new MouseAdapter()
    { public void mouseEntered (MouseEvent e)
     { if (e.getSource() == contenu)
       System.out.println ("la souris entre dans la fenetre") ;
        if (e.getSource() == bouton)
       System.out.println ("la souris entre dans le bouton") ;
     }
     public void mouseExited (MouseEvent e)
     { if (e.getSource() == contenu)
       System.out.println ("la souris quitte la fenetre") ;
        if (e.getSource() == bouton)
       System.out.println ("la souris quitte le bouton") ;
     }
    } ;
  contenu.addMouseListener (ecout) ;
  bouton = new JButton ("A") ;
  contenu.add (bouton) ;
  bouton.addMouseListener (ecout) ;
 }
 private JButton bouton ;
 private Container contenu ;
```

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

Ici, par souci de simplicité, nous avons intercepté les événements ayant pour source non plus la fenêtre elle-même, mais son contenu. En effet, dans la classe anonyme de l'écouteur, on ne peut plus identifier la fenêtre par *this*. On pourrait le faire en conservant la référence de la fenêtre dans un champ.

137 Suivi des déplacements de la souris (2)

Réaliser une fenêtre disposant d'un bouton marqué *CREATION_BOUTONS* permettant de créer dynamiquement des boutons marqués *B1*, *B2*, *B3*…



Lorsque la souris "passe" sur l'un de ces boutons, il se colore en fonction de son numéro (par exemple, le premier en rouge, le second en jaune, le troisième en vert, le quatrième en bleu, le cinquième à nouveau en rouge…) ; le bouton se colore en blanc lorsque la souris en sort

Ici, nous faisons de la fenêtre l'unique écouteur des différentes événements :

- *Action* pour le bouton de création,
- *Mouse* pour les boutons dynamiques.

Nous nous contentons du gestionnaire par défaut de la fenêtre.

Pour identifier le bouton concerné par un événement souris, nous aurions pu utiliser la référence à la source fournie par *getSource*. Cela aurait toutefois nécessité de conserver les références de tous les boutons créés dynamiquement. Ici, nous avons choisi d'exploiter la chaîne de commande de la source ; elle s'obtient à l'aide de la méthode *getActionCommand* qui figure dans toutes les classes dérivées de *AbstractButton*, donc en particulier dans *JButton*[1].

```
import javax.swing.* ;
```

```
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame implements MouseListener,
ActionListener
{ static final Color couleurs[] = {Color.red, Color.yellow,
Color.green,
               Color.blue} ;
 public MaFenetre ()
 { setTitle ("Evenements souris") ; setSize (300, 150) ;
  contenu = getContentPane() ;
  contenu.setLayout (new FlowLayout()) ;
  boutonCreation = new JButton ("CREATION_BOUTONS") ;
  contenu.add (boutonCreation) ;
  boutonCreation.addActionListener (this) ;
 }
 public void actionPerformed (ActionEvent e)
 { if (e.getSource() == boutonCreation)
   { numBouton++ ;
    JButton b = new JButton ("B"+numBouton) ;
    contenu.add (b) ;
    b.addMouseListener (this) ;
   }
 }
 public void mousePressed (MouseEvent e) {}
 public void mouseReleased (MouseEvent e) {}
 public void mouseClicked (MouseEvent e) {}
 public void mouseEntered (MouseEvent e)
 { Object source = e.getSource () ;
  JButton bSource ;
  if (source instanceof JButton)    // par precaution
  { bSource = (JButton)source ;
   String ch = bSource.getActionCommand() ;
   if (ch.charAt(0) == 'B')
    { int n = Integer.parseInt (ch.substring(1)) ;
     int numCoul = n % couleurs.length ;
     bSource.setBackground (couleurs[numCoul]) ;
    }
  }
 }
 public void mouseExited (MouseEvent e)
```

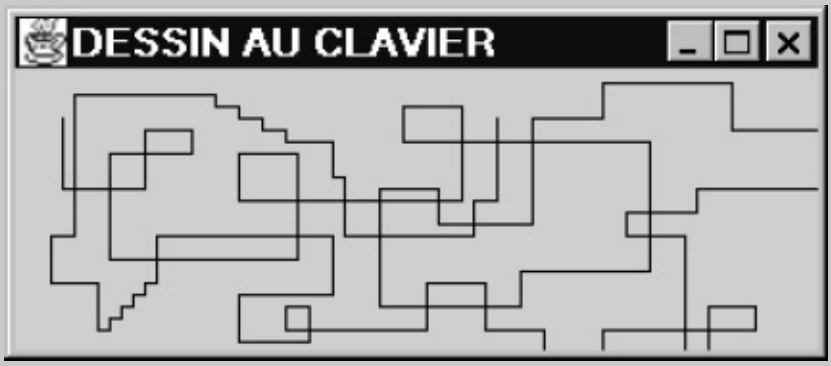
```
 { Object source = e.getSource () ;
  JButton bSource ;
  if (source instanceof JButton)          // par precaution
  { bSource = (JButton)source ;
   String ch = bSource.getActionCommand() ;
   if (ch.charAt(0) == 'B')
    bSource.setBackground (Color.white) ;
  }
 }
 private Container contenu ;
 private JButton boutonCreation ;
 private int numBouton = 0 ;
}
public class BtDynCol
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
 }
}
```

Dans les méthodes *mouseEntered* et *mouseExited*, nous nous sommes assurés que la source était bien d'un type *JButton* (opérateur *instanceof*) avant de lui appliquer la méthode *getActionCommand*. Ce n'était pas indispensable ici, mais cela pourrait le devenir si l'on modifiait le programme en écoutant les événements souris générés par d'autres composants.

---

138 Dessin par le clavier (1)

Écrire un programme permettant de dessiner à la volée dans une fenêtre en utilisant les touches fléchées du clavier :



Le dessin commencera en un point donné de la fenêtre (ici 20 × 20). On pourra fixer un incrément de plusieurs pixels (ici 5) pour chaque appui sur une touche.

Comme il s'agit ici de dessin à la volée, nous aurions pu opérer directement sur la fenêtre (ou plutôt sur son contenu). Mais, pour conserver au programme un caractère plus général, nous avons préféré dessiner sur un panneau.

La position de début du dessin est fixée par les valeurs initiales des variables *x* et *y* qui désignent ensuite la position courante de fin de dessin. L'incrément du déplacement est fixé par les constantes *incx* et *incy*.

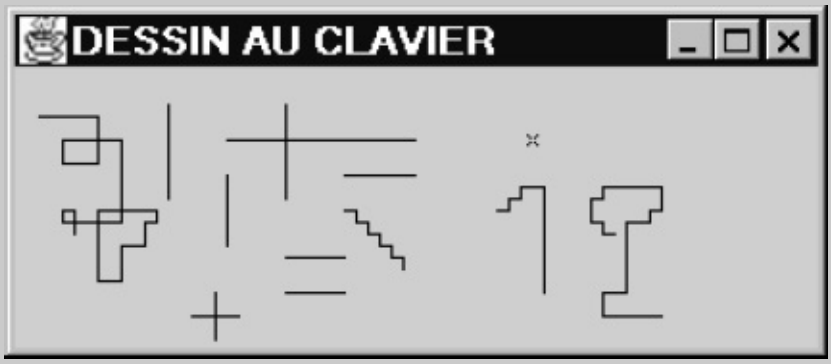
Nous pouvons faire de la fenêtre l'écouteur des événements clavier. En effet, ceux-ci seront transmis à la fois au panneau et à son conteneur, c'est-à-dire la fenêtre. Ici, nous redéfinissons la méthode *keyPressed*, ce qui revient à dire que nous décidons que les déplacements seront effectués lors de l'appui des touches. La méthode *getKeyCode* de la classe *KeyEvent* nous permet de connaître le code de touche virtuelle concerné. Nous utilisons les constantes telles que *KeyEvent.KEY_UP* pour identifier les touches fléchées.

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

```
class MaFenetre extends JFrame implements KeyListener
{ static int incx=5, incy=5 ;
 public MaFenetre ()
 { setTitle ("DESSIN AU CLAVIER") ; setSize (350, 150) ;
  addKeyListener (this) ;
  pan = new JPanel () ;
  getContentPane().add (pan) ;
 }
 public void keyPressed (KeyEvent e)
 { int code = e.getKeyCode () ;
  switch (code)
  { case KeyEvent.VK_UP : dx = 0 ; dy = -incy ; bouge = true ;
  break ;
   case KeyEvent.VK_DOWN : dx = 0 ; dy = incy ; bouge = true ;
   break ;
   case KeyEvent.VK_LEFT : dx = -incx ; dy = 0 ; bouge = true ;
   break ;
   case KeyEvent.VK_RIGHT : dx = incx ; dy = 0 ; bouge = true ;
   break ;
  }
  if (bouge)
  { Graphics g = pan.getGraphics() ;
   g.drawLine (x, y, x+dx, y+dy) ;
   g.dispose() ;
   x += dx ; y += dy ;
  }
 }
 public void keyReleased (KeyEvent e) {}
 public void keyTyped (KeyEvent e) {}
 private JPanel pan ;
 private int x=20, y=20 ;
 private int dx, dy ;
 private boolean bouge ;
}
public class DesClav
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
 }
}
```

139 Synthèse : dessin par le clavier (2)

Modifier le programme de l'exercice, de manière qu'on puisse interrompre le dessin et le reprendre en un autre point :



Un motif (en forme de x) permettra de visualiser la position courante du "curseur". Les touches fléchées agiront toujours sur la position du curseur ; en revanche, le dessin n'aura lieu que si la touche *Shift* est enfoncée.

Note : cet exercice nécessite (en plus des prérequis mentionnés en début de ce chapitre et du précédent) de savoir ce qu'est un "mode de dessin" et comment le modifier.

Les touches fléchées provoqueront donc toujours le déplacement du curseur. Pour ce faire, il est nécessaire de pouvoir effacer le curseur de son ancienne position et de le tracer dans sa nouvelle position. Pour y parvenir, le plus simple consiste à utiliser le mode de dessin dit *XOR*, en le paramétrant par la couleur de fond du panneau. Dans ce cas, en effet :

- le dessin sur une zone ayant la couleur de fond est fait avec la couleur courante,
- le même dessin effectué deux fois de suite efface le premier.

En ce qui concerne l'éventuel tracé du trait, il faut cette fois tenir compte de l'état de la touche *Shift*. On l'obtient avec la méthode *getModifiers* qui fournit un entier dans lequel un bit de rang *InputEvent.SHIFT_MASK* correspond à la touche *Shift*.

On notera que si l'on traçait ce trait dans le mode *XOR*, on effacerait le point situé à

l'intersection des deux segments représentant le curseur. On pourrait éventuellement prévoir d'afficher à nouveau ce point mais cette démarche serait dépendante du motif utilisé pour le curseur. Le plus raisonnable consiste à afficher le trait dans le mode de dessin normal qu'on obtient par appel de *setPaintMode*.

Initialement, aucun curseur ne s'affiche dans la fenêtre. En effet, nous ne pouvons pas effectuer ce tracé dans le constructeur de la fenêtre car aucun contexte graphique ne serait encore disponible pour le panneau (la méthode *getGraphics* fournirait la valeur *null*). Par souci de simplicité, nous nous sommes donc contentés d'afficher ce curseur après la première action sur une touche fléchée (nous recourons à un indicateur booléen nommé *debut*).

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
class MaFenetre extends JFrame implements KeyListener
{ static int incx=5, incy=5 ;
 public MaFenetre ()
 { setTitle ("DESSIN AU CLAVIER") ;
  setSize (350, 150) ;
  addKeyListener (this) ;
  pan = new JPanel () ;
  getContentPane().add (pan) ;
 }
 public void keyPressed (KeyEvent e)
 { int code = e.getKeyCode () ;
  bouge = false ;
  switch (code)
  { case KeyEvent.VK_UP : dx = 0 ; dy = -incy ; bouge = true ;
  break ;
   case KeyEvent.VK_DOWN : dx = 0 ; dy = incy ; bouge = true ;
   break ;
   case KeyEvent.VK_LEFT : dx = -incx ; dy = 0 ; bouge = true ;
   break ;
   case KeyEvent.VK_RIGHT : dx = incx ; dy = 0 ; bouge = true ;
   break ;
  }
  if (bouge)
  { Graphics g = pan.getGraphics() ;
   g.setXORMode (pan.getBackground()) ;
     /* efface l'ancien curseur (s'il existe) et affiche le nouveau
    */
```

```
   if (debut) debut = false ;
     else afficheCurseur (g, x, y) ;
   afficheCurseur (g, x+dx, y+dy) ;
   g.setPaintMode() ;
    /* on ne trace que si la touche Shift est enfoncee */
   if ( (e.getModifiers() & InputEvent.SHIFT_MASK) != 0)
        g.drawLine (x, y, x+dx, y+dy) ;
   x += dx ;
   y += dy ;
   g.dispose() ;
  }
 }
 private void afficheCurseur (Graphics g, int x, int y)
 { int dx=2, dy=2 ;
  g.drawLine (x-dx, y-dy, x+dx, y+dy) ;
  g.drawLine (x-dx, y+dy, x+dx, y-dy) ;
 }
 public void keyReleased (KeyEvent e) {}
 public void keyTyped (KeyEvent e) {}
 private JPanel pan ;
 private int x=20, y=20 ;
 private int dx, dy ;
 private boolean bouge ;
 private boolean debut = true ;
}

public class DesClav2
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible (true) ;
 }
}
```

140 Sélection d'un composant par le clavier

Afficher dans une fenêtre *n* boutons (*n*<=9) étiquettés de 1 à *n*. Faire en sorte que la frappe de l'une des touches 1 à *n* sélectionne le bouton de numéro *n* (lui donne le focus).

Nous introduisons classiquement les boutons dans la fenêtre, en conservant leurs références dans un tableau *boutons*. Nous faisons de la fenêtre son propre écouteur d'événements clavier et nous redéfinissons les méthodes *keyPressed*, *keyReleased* et *keyTyped* (seule la dernière nous intéresse ici).

Pour forcer le focus sur un bouton, nous utilisons la méthode *requestFocus*.

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

class MaFenetre extends JFrame implements KeyListener
{ private static int nBoutons = 7 ;
 public MaFenetre ()
 { setTitle ("SELECTIONS PAR CLAVIER") ;
  setSize (350, 150) ;
  Container contenu = getContentPane() ;
```

```
  contenu.setLayout (new FlowLayout()) ;
  addKeyListener (this) ; // attention : ajouter a la fenetre, pas
  au contenu
  boutons = new JButton [nBoutons] ;
  for (int i=0 ; i<nBoutons ; i++)
  { boutons[i] = new JButton ("BOUTON "+(i+1)) ;
   contenu.add(boutons[i]) ;
  }
 }
 public void keyPressed (KeyEvent e) {}
 public void keyReleased (KeyEvent e) {}
 public void keyTyped (KeyEvent e)
 { char c = e.getKeyChar() ;
  int num = c -'0' ;
  if ( (num>0) && (num<=nBoutons))
   boutons[num-1].requestFocus() ;
 }
 private JButton boutons[] ;
}

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

141 Mise en évidence d'un composant sélectionné

Afficher dans une fenêtre un certain nombre de boutons de couleur jaune. Faire en sorte que lorsqu'un bouton prend le focus, il se colore en rouge.

Ici, nous faisons de la fenêtre l'écouteur des événements *Focus* générés par les différents boutons. Nous redéfinissons les méthodes *focusGained* et *focusLost* de manière à modifier comme voulu la couleur du bouton.

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

class MaFenetre extends JFrame implements FocusListener
{ private static int nBoutons = 8 ;
 private static Color coulRepos = Color.yellow, coulSelec = Color.red ;
 public MaFenetre ()
 { setTitle ("SELECTIONS COLOREES") ;
  setSize (350, 150) ;
  Container contenu = getContentPane() ;
  contenu.setLayout (new FlowLayout()) ;
  for (int i=0 ; i<nBoutons ; i++)
```

```
  { bouton = new JButton ("BOUTON "+(i+1)) ;
    contenu.add(bouton) ;
    bouton.addFocusListener (this) ;
    bouton.setBackground (coulRepos) ;
  }
 }
 public void focusGained (FocusEvent e)
 { Object source = e.getSource() ;
  if (source instanceof JButton )
  { JButton bSource = (JButton) source ;
    bSource.setBackground (coulSelec) ;
  }
 }

 public void focusLost (FocusEvent e)
 { Object source = e.getSource() ;
  if (source instanceof JButton )
  { JButton bSource = (JButton) source ;
    bSource.setBackground (coulRepos) ;
  }
 }
 private JButton bouton ;
}

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

1. Bien qu'elle fournisse le même résultat, il s'agit bien d'une méthode différente de *getActionCommand* de la classe *ActionEvent*.

Chapitre 14

Les applets

Connaissances requises

- La classe *JApplet* ; les méthodes *init*, *start*, *stop* et *destroy* ; le gestionnaire par défaut
- Écriture d'un fichier HTML permettant de lancer une applet ; informations *code*, *width* et *height*
- Transmission d'informations à une applet par le fichier HTML et récupération par la méthode *getParameter*
- Transformation d'une application en une applet

142 Comptage des arrêts d'une applet

Réaliser une applet affichant en permanence le nombre de fois où elle a été interrompue.

Le nombre de fois où l'applet a été interrompue peut s'obtenir en comptant le nombre de fois où sa méthode *stop* a été appelée (avec la méthode *start*, on obtiendrait la même chose à une unité près). Un compteur est initialisé à 0 dans sa méthode *init* et incrémenté de 1 à chaque appel de *stop*. D'autre part, à chaque appel de *stop*, il faut actualiser le contenu d'un objet étiquette (*JLabel*) indiquant la valeur de ce compteur. Cet objet est ajouté par *add* au contenu de l'applet, sachant que son gestionnaire par défaut (*BorderLayout*) nous convient ici.

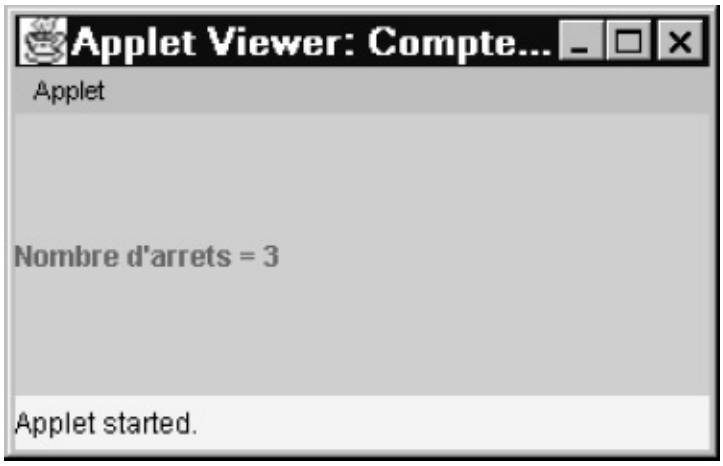
```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
public class Compteur extends JApplet  // ne pas oublier public
{ public void init ()
 { valeurCompteur = new JLabel (texte + compteur) ;
  getContentPane().add(valeurCompteur) ;
 }
 public void stop ()
 { compteur++ ;
  valeurCompteur.setText (texte + compteur) ;
 }
 private JLabel valeurCompteur ;
 private int compteur = 0 ;
 private String texte = "Nombre d'arrets = " ;
}
```

Voici un exemple de fichier HTML permettant de lancer cette applet[1] soit au sein d'un navigateur, soit dans une page Web (limitée alors ici à l'applet et ne disposant pas de

titre) :

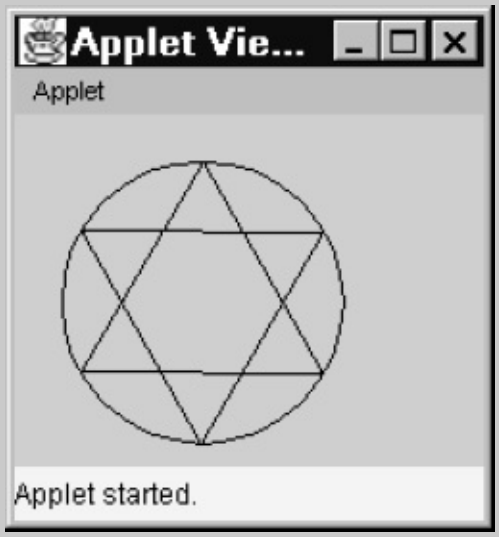
```
HTML>
 <BODY>
  <APPLET
   CODE = "Compteur.class"
   WIDTH = 250
   HEIGHT = 120
  >
  </APPLET>
</BODY>
</HTML>
```

Voici un exemple d'exécution dans un visualisateur d'applet :



143 Dessin dans une applet

Réaliser une applet qui affiche en permanence le dessin suivant (étoile dans un cercle) de taille fixe :



Ecrire un exemple de fichier HTML de lancement de cette applet.

Afin d'en assurer la permanence, le dessin est réalisé dans un panneau dont on redéfinit la méthode *paintComponent*.

Dans la méthode *init* de l'applet, on crée le panneau et on le rattache par *add* au contenu de l'applet fourni par la méthode *getContentPane* (on procède exactement comme dans le constructeur d'une fenêtre).

Les dimensions du dessin sont définies par les constantes *xc*, *yc* (coordonnées du centre du cercle) et *rayon* de la classe *Panneau*.

Le tracé de l'étoile est réalisé par une boucle dessinant successivement chacun de ses 6 segments. La variable *angle* correspond à l'angle que forme avec l'axe des abcisses le rayon passant par l'origine de chacun des segments.

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

```
public class AppEtoil extends JApplet  // ne pas oublier public
{ public void init ()
 { Container contenu = getContentPane() ;
  pan = new Panneau () ;
  contenu.add (pan) ;
 }
 private Panneau pan ;
}

class Panneau extends JPanel
{ private static int xc = 80, yc = 80, rayon =60 ;
 public void paintComponent (Graphics g)
 { super.paintComponent (g) ;
   /* trace du cercle */
  g.drawOval (xc-rayon, yc-rayon, 2*rayon, 2*rayon) ;
   /* trace des 6 segments de l'etoile */
  double angle, xd, xf, yd, yf ;
  int i ;
  { for (i=0, angle=Math.PI/6. ; i<6 ; i++, angle+= Math.PI/3)
   { xd = xc + rayon*Math.cos(angle) ;
    yd = yc - rayon*Math.sin(angle) ;
    xf = xc + rayon*Math.cos(angle+2*Math.PI/3) ;
    yf = yc - rayon*Math.sin(angle+2*Math.PI/3) ;
    g.drawLine ((int)xd, (int)yd, (int)xf, (int)yf) ;
   }
  }
 }
}
```

Voici un exemple de fichier HTML de lancement de l'applet[2] :

```
<HTML>
 <BODY>
  <APPLET CODE = "AppEtoil.class" WIDTH = 200 HEIGHT = 150>
  </APPLET>
</BODY>
</HTML>
```

Dans notre précédente solution, les coordonnées des segments sont recalculées à

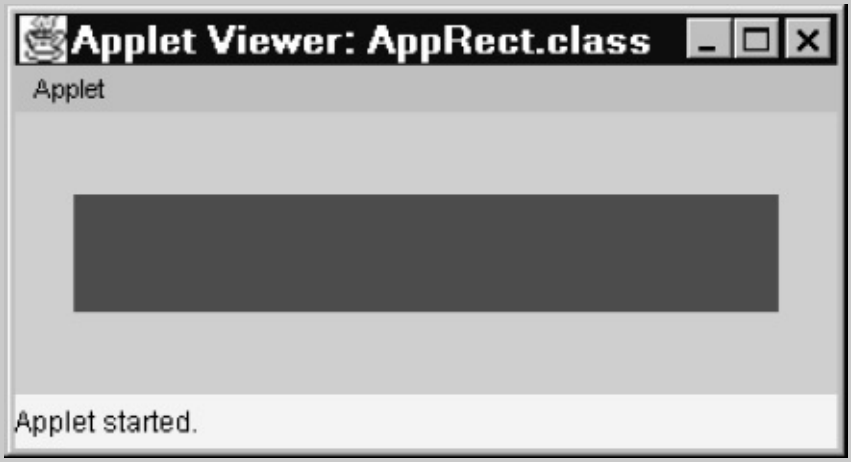
chaque appel de *paintComponent*. On peut en réalité profiter du fait que l'image est de taille fixe pour n'effectuer qu'une seule fois l'essentiel des calculs, par exemple dans le constructeur du panneau :

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

public class AppEtoib extends JApplet  // ne pas oublier public
{ public void init ()
 { Container contenu = getContentPane() ;
  pan = new Panneau () ;
  contenu.add (pan) ;
 }
 private Panneau pan ;
}

class Panneau extends JPanel
{ private static int xc = 80, yc = 80, rayon =60 ;
 public Panneau ()
 { xd = new int[6] ;
  yd = new int[6] ;
  xf = new int[6] ;
  yf = new int[6] ;
   /*  calculs  des  coordonnes  des  origines  et  extremites  des  6
   segments */
  double angle ;
  int i ;
  for (i=0, angle=Math.PI/6. ; i<6 ; i++, angle+= Math.PI/3)
  { xd[i] = (int) (xc + rayon*Math.cos(angle)) ;
   yd[i] = (int) (yc - rayon*Math.sin(angle)) ;
   xf[i] = (int) (xc + rayon*Math.cos(angle+2*Math.PI/3)) ;
   yf[i] = (int) (yc - rayon*Math.sin(angle+2*Math.PI/3)) ;
  }
 }
 public void paintComponent (Graphics g)
 { super.paintComponent (g) ;
   /* trace du cercle */
  g.drawOval (xc-rayon, yc-rayon, 2*rayon, 2*rayon) ;
   /* trace des 6 segments de l'etoile */
  for (int i=0 ; i<6 ; i++)
```

```
    g.drawLine (xd[i], yd[i], xf[i], yf[i]) ;
  }
  private int[] xd, yd, xf, yf ;
}
```

144 Synthèse : dessin paramétré dans une applet

Réaliser une applet qui affiche en permanence un rectangle coloré dont les dimensions et la couleur sont fournies par des paramètres figurant dans le fichier HTML de lancement :



Le rectangle sera placé au centre de l'applet dont on supposera que la taille n'évolue pas[a]. Donner un exemple de fichier HTML permettant de lancer cette applet.

a. Les visualisateurs d'applet autorisent cette modification de taille, mais pas les navigateurs.

Le rectangle est dessiné dans un panneau dont on redéfinit la méthode *paintComponent* pour assurer la permanence du dessin. Dans l'objet applet, on récupère les valeurs des paramètres figurant dans le fichier HTML. Rappelons que ces derniers sont identifiés par un nom (chaîne dans laquelle la casse n'est pas significative) et une valeur (chaîne également). On récupère la valeur d'un paramètre à l'aide de la méthode *getParameter* à laquelle on fournit en argument le nom du paramètre voulu.

En cas de besoin, les dimensions de l'applet (de nom *width* et *height*) peuvent également être récupérés de cette manière. C'est ce qui nous permet ici de calculer la position du rectangle dans la fenêtre de l'applet.

Si les valeurs de ces paramètres ne sont pas présentes dans le fichier HTML ou si elles ne sont pas convertibles en un entier, nous attribuons au rectangle des dimensions par défaut (celles de l'applet ne peuvent pas être incorrectes, sinon l'applet ne s'exécuterait pas).

La "valeur" d'une couleur est définie par une chaîne représentant son nom (*rouge*, *vert*…). On lui fait correspondre un objet de type *Color* à l'aide de deux tableaux, l'un de type *String* contenant les noms de couleur, l'autre de type *Color* contenant les couleurs associées.

La communication entre l'applet et le panneau se fait par des méthodes d'accès de l'applet.

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
public class AppRect extends JApplet  // ne pas oublier public
{ String nomsCouleurs[] = {"rouge", "vert", "bleu", "jaune" } ;
 Color   couleurs[]   =   {Color.red,   Color.green,   Color.blue,
 Color.yellow} ;
 public void init ()
 { Container contenu = getContentPane () ;
  pan = new Panneau (this) ;
  contenu.add (pan) ;    // avec le gestionnaire BorderLayout, le
  panneau
            // occupe toute la fenetre
   /* recuperation parametres dimension applet, dimension rectangle,
  couleur */
  String chLargeurApplet = getParameter ("width") ;
  String chHauteurApplet = getParameter ("height") ;
  String chLargeurRect = getParameter ("Largeur") ;
  String chHauteurRect = getParameter ("Hauteur") ;
  try
  { largeurApplet = Integer.parseInt (chLargeurApplet) ;
   hauteurApplet = Integer.parseInt (chHauteurApplet) ;
   largeurRect = Integer.parseInt (chLargeurRect) ;
   hauteurRect = Integer.parseInt (chHauteurRect) ;
  }
  catch (NumberFormatException ex)
  { /* on attribue des dimensions par defaut pour le rectangle */
      /* (celles de l'applet sont toujours bonnes) */
   largeurRect = 80 ; hauteurRect = 50 ;
```

```
   }
   nomCouleur = getParameter ("Couleur") ;
   couleur = Color.black ;   // couleur par defaut
   for (int i=0 ; i<nomsCouleurs.length ; i++)
   { if (nomCouleur.equals(nomsCouleurs[i])) couleur = couleurs[i] ;
   }
  }
  public int getLargeurApplet () { return largeurApplet ; }
  public int getHauteurApplet () { return hauteurApplet ; }
  public int getLargeurRect () { return largeurRect ; }
  public int getHauteurRect () { return hauteurRect ; }
  public Color getCouleur () { return couleur ; }
  private Panneau pan ;
  private    int    largeurApplet,    hauteurApplet,    largeurRect,
  hauteurRect ;
  private String nomCouleur ;
  private Color couleur ;
}
class Panneau extends JPanel
{ public Panneau (AppRect ap)
  { this.ap = ap ;
  }
  public void paintComponent (Graphics g)
  { super.paintComponent (g) ;
   int x = (ap.getLargeurApplet() - ap.getLargeurRect())/2 ;
   int y = (ap.getHauteurApplet() - ap.getHauteurRect())/2 ;
   g.setColor(ap.getCouleur()) ;
   g.fillRect(x, y, ap.getLargeurRect(), ap.getHauteurRect()) ;
  }
  AppRect ap ;
}
```

Voici un fichier HTML de lancement de cette applet[3] dans une fenêtre de dimensions 350 × 120 avec un rectangle de dimensions 300 × 50 et de couleur rouge :

```
<HTML>
 <BODY>
  <APPLET CODE = "AppRect.class" WIDTH = 350 HEIGHT = 120 >
   <PARAM NAME = "Largeur" VALUE = "300">
   <PARAM NAME = "Hauteur" VALUE = "50">
   <PARAM NAME = "Couleur" VALUE = "rouge">
```

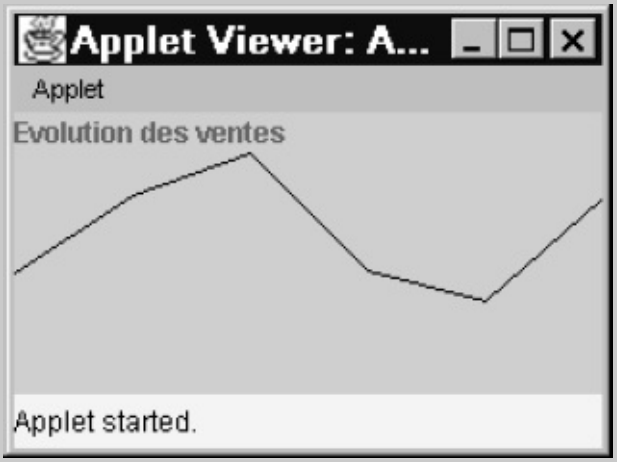
```
  </APPLET>
</BODY>
</HTML>
```

Ici, les dimensions du rectangle sont recalculées à chaque appel de *paintComponent*. Ce calcul pourrait être fait une fois pour toutes, par exemple dans la méthode *init*, à condition toutefois que ce soit avant le premier affichage du panneau.

---

145 Synthèse : tracé de courbe dans une applet

Réaliser une applet permettant de représenter sous forme d'une courbe une suite de valeurs entières positives ou nulles figurant en paramètres dans le fichier HTML correspondant, comme dans cet exemple :



Le titre sera fourni en paramètre. Le nombre de valeurs devra pouvoir être quelconque et sera également fourni en paramètre. L'applet affichera la valeur maximale.

Donner un exemple de fichier HTML permettant de lancer cette applet.

Nous introduisons dans la fenêtre de notre applet un panneau pour la courbe et un champ de texte pour le titre. Nous conservons le gestionnaire par défaut (*BorderLayout*) en plaçant le titre en haut (*"North"*) et le panneau au centre.

Les paramètres du fichier HTML sont récupérés classiquement dans la méthode *init* en utilisant *getParameter*. Nous supposons ici que les noms de ces paramètres sont *TITRE*, *NB_VALEURS*, *VALEUR1*, *VALEUR2*, *VALEUR3*… Notez que les noms des différentes valeurs sont formés d'un même préfixe (ici *VALEUR*) suivi du "numéro" de valeur. Ceci nous permet de traiter un nombre quelconque de valeurs. Ici, nous ne traitons pas les éventuelles exceptions que pourraient déclencher des valeurs

incorrectes ou manquantes ; l'applet se terminerait alors simplement avec un message d'erreur.

La méthode *paintComponent* du panneau en détermine la taille à l'aide de la méthode *getSize*. Les valeurs à tracer sont obtenues par la méthode d'accès *getValeurs* de l'applet. Les coordonnées des différents points sont alors calculées en tenant compte d'un facteur d'échelle (*echelle*) déterminé de manière que :

- le point correspondant à la plus grande valeur s'affiche tout en haut du panneau,
- le premier point s'affiche à l'extrémité gauche du panneau, le dernier à l'extrémité droite.

Notez que nous employons des variables de type *double* pour éviter une imprécision résultant de division d'entiers.

Notez également qu'il est nécessaire d'inverser les ordonnées afin d'obtenir un axe des y dirigé vers le haut.

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

public class AppCourb extends JApplet  // ne pas oublier public
{ public void init ()
 { /* les deux composants de l'applet : champ texte et panneau */
  Container contenu = getContentPane () ;
  JLabel champTitre = new JLabel (getParameter ("TITRE")) ;
  contenu.add (champTitre, "North") ;   // titre en haut
  pan = new Panneau (this) ;
  contenu.add (pan) ;                   // panneau pour la courbe au
  centre
    /* recuperation des parametres HTML : nombre de valeurs et
  valeurs */
  nValeurs = Integer.parseInt (getParameter ("NB_VALEURS")) ;
  if (nValeurs <= 1) System.exit (-1) ; // au moins 2 valeurs pour
  une courbe
  valeurs = new int [nValeurs] ;
  for (int i=0 ; i<nValeurs ; i++)
   valeurs[i] = Integer.parseInt(getParameter ("VALEUR"+(i+1))) ;
 }
 public int[] getValeurs ()
 { return valeurs ;
 }
 private Panneau pan ;
```

```
 private int nValeurs ;
 private int valeurs[] ;
 }

class Panneau extends JPanel
{ public Panneau (AppCourb ap)
 { this.ap = ap ;
 }
 public void paintComponent (Graphics g)
 { super.paintComponent (g) ;
   /* determination de la dimension du panneau */
  Dimension dimPanneau = getSize () ;
  int hauteur = dimPanneau.height ;
  int largeur = dimPanneau.width ;
   /* recuperation des valeurs */
  int[] valeurs = ap.getValeurs() ;
  int nValeurs = valeurs.length ;
   /* recherche de la valeur maximale */
  int valMax = valeurs [0] ;
  for (int i=1 ; i<nValeurs ; i++)
   if (valeurs[i] > valMax) valMax = valeurs [i] ;
   /* trace de la courbe point par point */
  double ecart = (double)largeur/(nValeurs-1) ;  // on a nValeurs >1
  double echelle = (double)hauteur/valMax ;
  double xDeb = 0, yDeb = hauteur - valeurs[0] * echelle ;
  double xFin, yFin ;
  for (int i=1 ; i<nValeurs ; i++)
  { xFin = xDeb + ecart ;
   yFin = hauteur - valeurs[i] * echelle ;
   g.drawLine ((int)xDeb, (int)yDeb, (int)xFin, (int)yFin) ;
   xDeb = xFin ;
   yDeb = yFin ;
  }
 }
 AppCourb ap ;
}
```

Voici un exemple de fichier HTML permettant d'exploiter ce programme[4] (il fournit la courbe présentée dans l'énoncé) :

```
<HTML>
```

```
 <BODY>
  <APPLET CODE = "AppCourb.class" WIDTH = 250 HEIGHT = 120>
   <PARAM NAME = "TITRE" VALUE = "Evolution des ventes">
   <PARAM NAME = "NB_VALEURS" VALUE = "6">
   <PARAM NAME = "VALEUR1" VALUE = "175">
   <PARAM NAME = "VALEUR2" VALUE = "288">
   <PARAM NAME = "VALEUR3" VALUE = "352">
   <PARAM NAME = "VALEUR4" VALUE = "181">
   <PARAM NAME = "VALEUR5" VALUE = "135">
   <PARAM NAME = "VALEUR6" VALUE = "285">
  </APPLET>
</BODY>
</HTML>
```

Ici, les coordonnées du tracé sont calculées à chaque appel de *paintComponent*. Si l'on utilise un visualisateur qui autorise le redimensionnement de l'applet, on pourra ainsi voir le tracé s'adapter à la taille courante de la fenêtre. Il n'en irait pas ainsi si l'on déterminait ces dimensions dans la méthode *init*.

---

146 Différences entre applet et application

Adapter l'exercice 103 du chapitre 8 de manière que l'utilisateur puisse dessiner dans une applet et non plus dans une fenêtre.

Il suffit d'adapter le code en tenant compte des quelques remarques suivantes :

- supprimer la méthode *main* (si on la conservait, elle ne serait pas appelée lors du lancement du code depuis un fichier HTML) ;
- transformer la classe fenêtre (*MaFenetre*) en une classe (ici *DesVol*) dérivée de *JApplet* ;
- transposer dans la méthode *init* de la classe *DesVol* les actions réalisées dans le constructeur de la fenêtre *MaFenetre* ;
- supprimer les appels à *setTitle* et *setSize* qui n'ont plus de raison d'être pour une applet (pas de titre, dimensions définies par les paramètres *WIDTH* et *HEIGHT* du fichier HTML de lancement).

```
import javax.swing.* ;
import java.awt.* ;
import java.awt.event.* ;
public class DesVol extends JApplet   // ne pas oublier public
{ public void init ()
 { pan = new Panneau () ;
  pan.addMouseListener (pan) ;
  getContentPane().add(pan) ;
 }
 private Panneau pan ;
}
class Panneau extends JPanel implements MouseListener
{ public void paintComponent (Graphics g)
 { super.paintComponent(g) ;
```

```
   enCours = false ;
  }
  public void mouseClicked (MouseEvent e)
  { int xFin = e.getX() ; yFin = e.getY() ;
   if (enCours) { Graphics g = getGraphics() ;
           g.drawLine (xDeb, yDeb, xFin, yFin) ;
           g.dispose() ;
          }
   xDeb = xFin ; yDeb = yFin ;
   enCours = true ;
  }
  public void mousePressed (MouseEvent e) {}
  public void mouseReleased (MouseEvent e) {}
  public void mouseEntered (MouseEvent e) {}
  public void mouseExited (MouseEvent e) {}
  private boolean enCours = false ;
  private int xDeb, yDeb, xFin, yFin ;
 }
```

À titre indicatif, voici un fichier HTML très simple (*DesVol.html*) permettant de lancer cette applet :

```
<HTML>
 <BODY>
  <APPLET
   CODE = "DesVol.class"
   WIDTH = 350
   HEIGHT = 100
  >
  </APPLET>
</BODY>
</HTML>
```

1. Certains navigateurs emploient la balise *OBJECT* ou *EMBED* à la place de la balise *APPLET*.
2. Certains navigateurs emploient la balise *OBJECT* ou *EMBED* à la place de la balise *APPLET*.
3. Certains navigateurs emploient la balise *OBJECT* ou *EMBED* à la place de la balise *APPLET*.
4. Certains navigateurs emploient la balise *OBJECT* ou *EMBED* à la place de la balise *APPLET*.

Chapitre 15

Les flux et les fichiers

Connaissances requises

- Notion de flux ; flux d'entrée, flux de sortie ; flux binaire, flux texte
- Création séquentielle d'un fichier binaire ; classes *OutputStream*, *FileOutputStream* et *DataOutputStream*
- Liste séquentielle d'un fichier binaire ; classes *InputStream*, *FileInputStream* et *DataInputStream*
- Accès direct à un fichier binaire ; classes *RandomAccessFile* ; action sur le pointeur de fichier
- Création d'un fichier texte ; classe *PrintWriter*
- Lecture d'un fichier texte ; classes *FileReader, BufferedReader* et *StringTokenizer*
- Gestion des fichiers avec la classe *File*

147 Création séquentielle d'un fichier binaire

Écrire un programme permettant de créer séquentiellement un fichier binaire comportant pour différentes personnes les informations suivantes : nom, prénom et année de naissance.

Le dialogue de saisie de l'information s'effectuera en fenêtre console comme dans cet exemple :

```
Nom du fichier a creer :
e:\repert
nom 1 : Carre
Prenom : Thibault
annee naissance : 1997
 .....
nom 5 : Mitenne
Prenom : Thomas
annee naissance : 2001
nom 6 :
**** fin creation fichier ****
```

On proposera deux solutions :

1. Les informations relatives au nom et au prénom seront conservées dans le fichier sous la forme d'une suite de 20 caractères (comportant d'éventuels espaces à la fin).

2. Ces mêmes informations seront conservées sous la forme d'une chaîne codée dans le format UTF[a] ; aucune contrainte ne portera sur leur longueur.

a. Ce format (*Unicode Text Format*) permet de coder une chaîne sous forme d'une suite d'octets en nombre variable (chaque caractère étant codé sur un à trois octets). La méthode *writeUTF* de la classe *DataOutputStream* réalise cette transformation d'une chaîne en une suite de caractères UTF.

Nous utiliserons la démarche la plus classique qui consiste à exploiter les méthodes de la classe flux *DataOutputStream*. Pour ce faire, nous associerons un objet de ce type (nommé *sortie*) à un fichier dont le nom est fourni par l'utilisateur dans la chaîne

nomFichier :

```
DataOutputStream sortie = new DataOutputStream
             (new FileOutputStream (nomFichier)) ;
```

Les variables *chNom* et *chPrenom* servent à lire les informations nom et prénom sous forme de chaînes de caractères. Nous en transférons ensuite chacun des caractères (à concurrence de 20) dans des tableaux de 20 caractères *nom* et *prenom*, préalablement remplis avec des espaces.

L'écriture dans le fichier est réalisée à l'aide des méthodes *writeChar* (écriture d'un caractère) et *writeInt* (écriture d'un entier) de la classe *DataOutputStream*.

```
import java.io.* ;
public class CrFich
{ public static void main (String args[]) throws IOException
 { final int longMaxNom = 20 ;
  final int longMaxPrenom = 20 ;
  String chNom, chPrenom ;
  char[] nom = new char [longMaxNom] ;
  char[] prenom = new char [longMaxPrenom] ;
  int annee ;

  String nomFichier ;
  System.out.println ("Nom du fichier a creer : ") ;
  nomFichier = Clavier.lireString() ;
  DataOutputStream sortie = new DataOutputStream
              (new FileOutputStream (nomFichier)) ;
  int i ;
  int num = 0 ;    // pour compter les differents enregistrements

  while (true)     // on s'arretera sur nom vide
  { /* lecture infos */
   num++ ;
   System.out.print ("nom " + num + " : ") ;
   chNom = Clavier.lireString() ;
   if (chNom.length() == 0) break ;
   System.out.print ("Prenom : ") ;
   chPrenom = Clavier.lireString() ;
   System.out.print ("annee naissance : ") ;
   annee = Clavier.lireInt() ;
    /* transfert nom et prenom dans tab de char termines par des
    espaces */
   for (i=0 ; i<longMaxNom ; i++) nom[i] = ' ';
```

```
    for (i=0 ; i<longMaxPrenom ; i++) prenom[i] = ' ' ;
    for (i = 0 ; (i < chNom.length())&&(i<longMaxNom) ; i++)
     nom[i] = chNom.charAt(i) ;
    for (i = 0 ; (i < chPrenom.length())&&(i<longMaxPrenom) ; i++)
     prenom[i] = chPrenom.charAt(i) ;
     /* ecriture fichier */
    for (i=0 ; i<longMaxNom ; i++) sortie.writeChar (nom[i]) ;
    for (i=0 ; i<longMaxPrenom ; i++) sortie.writeChar (prenom[i]) ;
    sortie.writeInt(annee) ;
   }
   sortie.close() ;
   System.out.println ("**** fin creation fichier ****") ;
  }
 }
```

---

1. La clause *throws IOException* figurant dans la méthode *main* est nécessaire, dès lors qu'on n'y traite pas les exceptions susceptibles d'être déclenchées par les méthodes de la classe *DataOutputstream.*

2. Plutôt que d'écrire un à un chacun des caractères de *nom* et de *prenom*, on aurait pu espérer appliquer directement à *chNom* et *chPrenom* la méthode *writeChars* qui écrit tous les caractères d'une chaîne. Cependant, cette démarche ne correpond pas à la demande de l'énoncé (informations de taille fixe dans le fichier) ; de plus, elle ne permettrait pas de relire ultérieurement le fichier (à moins de connaître par ailleurs les longueurs de chacune des informations y figurant !).

Comme précédemment, nous créons un objet de type *DataOutputStream*. Mais, cette fois, nous pouvons appliquer la méthode *writeUTF* aux chaînes correspondant au nom et au prénom.

```
import java.io.* ;
public class CrFich2
{ public static void main (String args[]) throws IOException
 { String chNom, chPrenom ;
  int annee ;

  String nomFichier ;
```

```
  System.out.println ("Nom du fichier a creer : ") ;
  nomFichier = Clavier.lireString() ;
  DataOutputStream sortie = new DataOutputStream
              (new FileOutputStream (nomFichier)) ;
  int i ;
  int num = 0 ;    // pour compter les differents enregistrements

  while (true)     // on s'arretera sur nom vide
  { /* lecture infos */
   num++ ;
   System.out.print ("nom " + num + " : ") ;
   chNom = Clavier.lireString() ;
   if (chNom.length() == 0) break ;
   System.out.print ("Prenom : ") ;
   chPrenom = Clavier.lireString() ;
   System.out.print ("annee naissance : ") ;
   annee = Clavier.lireInt() ;
    /* ecriture fichier */
   sortie.writeUTF (chNom) ;
   sortie.writeUTF (chPrenom) ;
   sortie.writeInt(annee) ;
  }
  sortie.close() ;
  System.out.println ("**** fin creation fichier ****") ;
 }
}
```

Cette seconde démarche peut paraître plus souple que la première puisqu'elle n'impose aucune limite à la taille des chaînes fournies. Néanmoins, elle présente l'inconvénient de ne plus être adaptée à l'exploitation ultérieure du fichier en accès direct.

---

148 Liste séquentielle d'un fichier binaire

Écrire un programme permettant de lister en fenêtre console le contenu d'un fichier binaire tel que celui créé par l'exercice. On proposera deux solutions correspondant aux deux situations :

1. Les informations relatives au nom et au prénom ont été enregistrées dans le fichier sous la forme d'une suite de 20 caractères (comportant d'éventuels espaces à la fin).
2. Ces mêmes informations ont été enregistrées sous la forme d'une chaîne codée dans le format UTF ; aucune contrainte ne portera sur leur longueur.

Nous exploitons les méthodes de la classe flux *DataInputStream*. Pour ce faire, nous associons un objet de ce type (nommé *entree*) à un fichier dont le nom est fourni par l'utilisateur dans la chaîne *nomFichier* :

```
DataInputStream entree = new DataInputStream
             (new FileInputStream (nomFichier)) ;
```

Les informations relatives au nom et au prénom sont lues dans des tableaux de 20 caractères *nom* et *prenom* à l'aide de la méthode *readChar* de la classe *DataInputStream*.

La gestion de la fin de fichier est réalisée en interceptant l'exception *EOFException* : la boucle de lecture des informations est contrôlée par un indicateur booléen *eof* initialisé à *false* et mis à *true* par le gestionnaire d'exception.

```
import java.io.* ;

public class LecFich
{
 public static void main (String args[]) throws IOException
 { final int longMaxNom = 20 ;
  final int longMaxPrenom = 20 ;
  String chNom, chPrenom ;
```

```
  char[] nom = new char [longMaxNom] ;
  char[] prenom = new char [longMaxPrenom] ;
  int annee ;
  int i ;
  String nomFichier ;
  System.out.println ("Nom du fichier a lister : ") ;
  nomFichier = Clavier.lireString() ;
  DataInputStream entree = new DataInputStream
              (new FileInputStream (nomFichier)) ;
  System.out.println ("**** Liste du fichier ****") ;
  boolean eof = false ; // sera mis a true par gestionnaire
  exception EOFile
  while (!eof)
  { try
   { /* lecture infos */
    for (i=0 ; i<longMaxNom ; i++) nom[i] = entree.readChar () ;
    for (i=0 ; i<longMaxPrenom ; i++) prenom[i] = entree.readChar
    () ;
    annee = entree.readInt () ;
     /* affichage infos */
    for (i=0 ; i<longMaxNom ; i++) System.out.print (nom[i]) ;
    System.out.print (" ") ;
    for (i=0 ; i<longMaxPrenom ; i++) System.out.print (prenom[i]) ;
    System.out.print (" ") ;
    System.out.println (annee) ;
   }
   catch (EOFException e)
   { eof = true ;
   }
  }
  entree.close() ;
  System.out.println ("**** fin liste fichier ****") ;
 }
}
```

À titre indicatif, voici l'allure des résultats fournis par ce programme :

```
Nom du fichier a lister :
e:\repert
**** Liste du fichier ****
Carre     Thibault       1997
```

```
Dubois    Louis          1975
Dutronc   Jean Philippe  1958
Duchene   Alfred         1994
Mitenne   Thomas         2001
*** fin liste fichier ****
```

Comme précédemment, on fait appel à un objet de type *DataInputStream*. Mais les informations relatives au nom et au prénom sont lues directement à l'aide de la méthode *readUTF*. La gestion de la fin de fichier se déroule toujours de la même manière.

```
import java.io.* ;

public class LecFich2
{
 public static void main (String args[]) throws IOException
 { final int longMaxNom = 20 ;
  final int longMaxPrenom = 20 ;
  String chNom, chPrenom ;
  int annee ;
  int i ;
  String nomFichier ;
  System.out.println ("Nom du fichier a lister : ") ;
  nomFichier = Clavier.lireString() ;
  DataInputStream entree = new DataInputStream
              (new FileInputStream (nomFichier)) ;
  System.out.println ("**** Liste du fichier ****") ;
  boolean eof = false ; // sera mis a true par gestionnaire
  exception EOFile
  while (!eof)
  { try
   { /* lecture infos */
    chNom = entree.readUTF () ;
    chPrenom = entree.readUTF () ;
    annee = entree.readInt () ;
     /* affichage infos */
    System.out.print (chNom + " ") ;
    System.out.print (chPrenom + " ") ;
    System.out.println (annee) ;
```

```
    }
    catch (EOFException e)
    { eof = true ;
    }
   }
   entree.close() ;
   System.out.println ("**** fin liste fichier ****") ;
  }
}
```

Les résultats se présentent alors sous cette forme :

```
Nom du fichier a lister :
e:\reputf
**** Liste du fichier ****
Carre Thibault 1997
Dubois Louis 1975
Dutronc Jean Philippe 1958
Duchene Alfred 1994
Mitenne Thomas 2001
**** fin liste fichier ****
```

149 Synthèse : consultation d'un répertoire en accès direct

Réaliser un programme permettant de consulter un fichier du type de celui créé par la première solution à l'exercice. Le dialogue s'opérera à travers des contrôles disposés dans une fenêtre comme illustrée ci-après[a] :

L'utilisateur pourra agir indifféremment sur les champs de texte indiquant le nom de fichier ou le nom d'enregistrement. On signalera par des boîtes de message les erreurs suivantes :

- fichier inexistant,
- information de numéro d'enregistrement non numérique, négative ou supérieure à la taille du fichier.

Lorsqu'un fichier sera correctement ouvert, son nom s'affichera dans le titre de la fenêtre.

Note : pour que les contrôles soient disposés comme dans notre exemple, on pourra utiliser un gestionnaire de mise en forme de type *GridLayout* créé par *new GridLayout(5, 2)*.

a. On pourra utiliser un gestionnaire de mise en forme de type *GridBag*.

Les dimensions des tableaux de caractères sont définies par des constantes symboliques *LG_NOM* et *LG_PRENOM*. Il en va de même pour la taille d'un enregistrement (*TAILLE_ENREG*) dont on notera que le calcul doit tenir compte du fait que les caractères sont enregistrés en binaire et qu'ils occupent donc 2 octets.

La disposition des différents contrôles ne pose pas de problème particulier. On notera que, avec un gestionnaire de type *GridLayout*, le conteneur est rempli ligne par ligne, suivant l'ordre dans lequel ils sont ajoutés. Nous utilisons des champs de texte pour toutes les informations mais seuls les deux premiers sont "éditables".

Nous écoutons les événements *Focus* et *Action* des deux champs de saisie (nom de fichier et numéro d'enregistrement). Deux méthodes de service nommées *nouveauFichier* et *nouvelEnreg* nous évitent de dupliquer certaines instructions.

La demande d'ouverture d'un nouveau fichier entraîne tout d'abord la fermeture de tout autre fichier éventuellement ouvert. Puis, nous vérifions l'existence du fichier de nom indiqué en traitant convenablement l'exception générée par sa demande d'ouverture en cas d'inexistence. Lorsque les choses se sont convenablement déroulées, nous déterminons la taille du fichier en octets (méthode *length*) et nous déterminons le nombre d'enregistrements correspondants.

Dans la demande d'un nouvel enregistrement, nous vérifions que :

- l'information fournie peut être convenablement convertie en un entier,
- qu'elle possède une valeur compatible avec la taille du fichier.

Si le numéro d'enregistrement est convenable, nous positionnons le pointeur à l'endroit correspondant du fichier (méthode *seek*). Nous lisons les différentes informations voulues et nous les affichons dans les champs appropriés. Notez que les tableaux de caractères constituant le nom et le prénom doivent être convertis en chaînes ; pour ce faire, nous utilisons un constructeur de la forme *String(char[])*.

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

class MaFenetre extends JFrame implements ActionListener,
FocusListener
{ private static final int LG_NOM = 20, LG_PRENOM = 20 ;
 private static final int TAILLE_ENREG = 2*LG_NOM + 2*LG_PRENOM +
 4 ;
 private static final String titreFenetre = "Consultation
 repertoire" ;
```

```
public MaFenetre ()
{ nom = new char[LG_NOM] ;
 prenom = new char[LG_PRENOM] ;

 setTitle (titreFenetre) ;
 setSize (400, 200) ;
 Container contenu = getContentPane() ;
 contenu.setLayout (new GridLayout(5, 2) ) ;

 labNomFichier = new JLabel (etiqNomFichier) ;
 contenu.add(labNomFichier) ;
 txtNomFichier = new JTextField (20) ;
 contenu.add(txtNomFichier) ;
 txtNomFichier.addActionListener (this) ;
 txtNomFichier.addFocusListener (this) ;
 labNumEnreg = new JLabel (etiqNumEnreg) ;
 contenu.add (labNumEnreg) ;
 txtNumEnreg = new JTextField (20) ;
 contenu.add (txtNumEnreg) ;
 txtNumEnreg.addActionListener (this) ;
 txtNumEnreg.addFocusListener (this) ;
 labNom = new JLabel (etiqNom) ;
 contenu.add (labNom) ;
 txtNom = new JTextField (20) ; txtNom.setEditable (false) ;
 contenu.add (txtNom) ;
 labPrenom = new JLabel (etiqPrenom) ;
 contenu.add (labPrenom) ;
 txtPrenom = new JTextField (20) ; txtPrenom.setEditable (false) ;
 contenu.add (txtPrenom) ;
 labAnnee = new JLabel (etiqAnnee) ;
 contenu.add (labAnnee) ;
 txtAnnee = new JTextField (20) ; txtAnnee.setEditable (false) ;
 contenu.add (txtAnnee) ;
}
public void actionPerformed (ActionEvent e)
{ Object source = e.getSource() ;
 if (source == txtNomFichier) nouveauFichier() ;
 if (source == txtNumEnreg) nouvelEnreg() ;
}
public void focusGained (FocusEvent e)
```

```
{}
public void focusLost (FocusEvent e)
{ Object source = e.getSource() ;
 if (source == txtNomFichier) nouveauFichier() ;
 if (source == txtNumEnreg) nouvelEnreg() ;
}

private void nouveauFichier()
{ try
 { if (fichierOuvert)
  { fichier.close() ;
   fichierOuvert = false ;
   setTitle (titreFenetre) ;
  }
  nomFichier = txtNomFichier.getText () ;
  fichier = new RandomAccessFile (nomFichier, "r") ;
 }
 catch (IOException e) // erreur ouverture
 { JOptionPane.showMessageDialog (null, "FICHIER INEXISTANT") ;
  txtNomFichier.setText ("") ;
  return ;
 }
 fichierOuvert = true ;
 setTitle (titreFenetre + " " + nomFichier) ;
 try
 { tailleFichierOctets = fichier.length() ;
  tailleFichierEnreg = tailleFichierOctets/TAILLE_ENREG ;
 }
 catch (IOException e) {}
 txtNumEnreg.setText("") ; txtNom.setText("") ;
 txtPrenom.setText("") ; txtAnnee.setText("") ;
}
private void nouvelEnreg()
{ if (!fichierOuvert)
 { JOptionPane.showMessageDialog (null, "Pas de fichier ouvert") ;
  txtNumEnreg.setText ("") ;
  return ;
 }

  /* lecture numero enregistrement et controles validite */
```

```
  String chNumEnreg = txtNumEnreg.getText () ;
  boolean converti = false ;
  try
  { num = Integer.parseInt (chNumEnreg) ;
   converti = true ;
  }
  catch (NumberFormatException e) {}
  if (!converti || (num<=0) || (num>tailleFichierEnreg))
  { JOptionPane.showMessageDialog (null, "Numero enreg incorrect") ;
   txtNumEnreg.setText ("") ; txtNom.setText("") ;
   txtPrenom.setText("") ; txtAnnee.setText("") ;
   return ;
  }
   /* numero correct - lecture de l'enregistrement correspondant */
  try
  { numEnreg = num ;
   fichier.seek (TAILLE_ENREG*(numEnreg-1)) ;
   for (int i=0 ; i<LG_NOM ; i++) nom[i] = fichier.readChar() ;
   for   (int   i=0   ;   i<LG_PRENOM   ;   i++)   prenom[i]   =
   fichier.readChar() ;
   annee = fichier.readInt () ;
    /* conversion des informations en chaine et affichage */
   String chNom = new String (nom) ;
   String chPrenom = new String (prenom) ;
   String chAnnee = String.valueOf (annee) ;
   txtNom.setText (chNom) ;
   txtPrenom.setText (chPrenom) ;
   txtAnnee.setText (chAnnee) ;
 }
 catch (IOException e) {}
}
private boolean fichierOuvert = false ;
private String nomFichier ;
private RandomAccessFile fichier ;
private long tailleFichierEnreg, tailleFichierOctets ;
private int numEnreg, num ;
private char[] nom, prenom ;
private int annee ;
private  JLabel  labNomFichier,  labNumEnreg,  labNom,  labPrenom,
labAnnee ;
```

```
private JTextField txtNomFichier, txtNumEnreg, txtNom, txtPrenom,
txtAnnee ;
 static     private     String     etiqNomFichier     =     "Nom
 fichier :            ",
            etiqNumEnreg = "Numero enregistrement :   ",
            etiqNom = "Nom :                          ",
            etiqPrenom = "Prenom :                    ",
            etiqAnnee = "Annee naissance :            ",
}
public class ListAD
{ public static void main (String args[])
 { MaFenetre fen = new MaFenetre() ;
  fen.setVisible(true) ;
 }
}
```

---

1. En vertu des règles relatives à la redéfinition d'une méthode, il n'est pas possible de mentionner de clause *throws IOException* dans les méthodes *actionPerformed* ou *focusLost*. Dans ces conditions, il est nécessaire d'y traiter (ici artificiellement) l'exception *IOException*.

2. On constate qu'en cas d'anomalie (fichier inexistant, numéro d'enregistrement incorrect), on obtient deux fois l'affichage du message correspondant. Ceci provient de la mise à blanc des champs correspondants. Par souci de simplicité, nous n'avons pas cherché à régler le problème (par exemple, en recourant à des indicateurs booléens).

---

150 Synthèse : liste d'un fichier texte avec numérotation des lignes

Écrire un programme qui liste en fenêtre console le contenu d'un fichier texte en en numérotant les lignes. On prévoira 4 caractères pour l'affichage du numéro de ligne. Les lignes de plus de 60 caractères seront affichées sur plusieurs lignes d'écran comme dans cet exemple

```
Donnez le nom du fichier texte a lister : e:\book\essai.txt
   1 Ceci est la premiere ligne d'un exemple de fichier texte
   2 Il contient des lignes de chiffres de longueurs variables
    dont une de 59 caracteres, une de 60 caracteres et une de 61
    caracteres
   3 12345678901234567890
                                                              4
 1234567890123456789012345678901234567890123456789012345678901234567890
   5 123456789012345678901234567890123456789012345678901234567890123456789
                                                             6
 1234567890123456789012345678901234567890123456789012345678901234567890
    1
   7 12345678901234567890123456789012345678901234567890
   8 la ligne suivante est vide
   9
  10 les deux lignes suivantes sont egalement vides
  11
  12
  13 Ceci est la derniere ligne du fichier
  *** fin liste fichier ***
```

Rappelons que, pour la lecture d'un fichier texte, il n'existe pas de classe parfaitement symétrique de la classe *PrintWriter*. Il faut se contenter de la classe *FileReader* (symétrique de *FileWriter*, classe plus rudimentaire que *PrintWriter*) qu'on couple avec la classe *BufferedReader*, laquelle dispose d'une méthode *readLine* de lecture d'une ligne. Nous créons donc un objet de ce type nommé *entree* en procédant ainsi

(*nomfich* étant la chaîne correspondant au nom du fichier) :

```
BufferedReader entree = new BufferedReader (new FileReader (nomfich)) ;
```

La méthode *readLine* de la classe *BufferedReader* fournit une référence à une chaîne correspondant à une ligne du fichier. Si la fin de fichier a été atteinte avant que la lecteur n'ait commencé, autrement dit si aucun caractère n'est disponible (pas même une fin de ligne !), *readLine* fournit la valeur *null*. Il est donc possible de parcourir les différentes lignes du fichier, sans avoir besoin de recourir à la gestion des exceptions.

En ce qui concerne l'affichage du numéro de ligne (*numLigne*), il est nécessaire de convertir l'entier le représentant en une suite de 4 caractères. Pour ce faire, nous employons un tableau de 4 caractères nommé *charNumLigne* que nous initialisons avec des caractères "espace", avant d'y introduire, à partir de la fin, les caractères de la chaîne obtenue par conversion de la valeur de *numLigne*.

La gestion des lignes de plus de 60 caractères se fait simplement en affichant un changement de ligne et une suite de 4+1 espaces.

```
import java.io.* ;

public class ListText
{ public static void main (String args[]) throws IOException
 { final int longNumLigne = 4 ;  // nombre de caracteres utilises pour
                // afficher le numero de ligne
  final int nbCarParLigne = 60 ;
  String nomfich ;
  String ligne ;       // ligne courante du fichier texte
  char charNumLigne[] = new char[longNumLigne] ;   // pour les caracteres
             // du numero de ligne
  System.out.print ("Donnez le nom du fichier texte a lister : ") ;
  nomfich = Clavier.lireString() ;
  BufferedReader entree = new BufferedReader (new FileReader (nomfich)) ;
  int numLigne = 0 ;
   do
   { /* lecture d'une ligne du fichier */
    ligne = entree.readLine() ;
    if (ligne == null) break ;
    numLigne++ ;
     /* determination des caracteres correspondant au numero de
      ligne */
```

```
    String ch = String.valueOf (numLigne) ;
    int i, j ;  // pour parcourir le numero de ligne
    for (i=0 ; i<longNumLigne-ch.length() ; i++) charNumLigne[i] = '
    ' ;
    for  (j=0  ;  i<longNumLigne  ;  i++,  j++)  charNumLigne[i]  =
    ch.charAt(j) ;
     /* affichage numero de ligne suivi d'un espace*/
    for   (i=0   ;   i<longNumLigne   ;   i++)   System.out.print
    (charNumLigne[i]) ;
    System.out.print (' ') ;
     /* affichage ligne courante */
    int n=0 ;  // pour parcourir la ligne courante
    while (n < ligne.length())
    { if ((n != 0) && (n%nbCarParLigne == 0)) /* on change de ligne
    */
              { System.out.println () ;
              for (int k=0 ; k<longNumLigne+1 ; k++)
              System.out.print (' ') ;
              }
     System.out.print (ligne.charAt(n)) ;
     n++ ;
    }
    System.out.println () ;
   }
  while (ligne != null) ;
  entree.close () ;
  System.out.println ("*** fin liste fichier ***");
 }
}
```

151 Liste d'un répertoire

Écrire un programme qui affiche le contenu d'un répertoire (dont le nom est fourni au clavier), en précisant pour chaque nom s'il s'agit d'un sous-répertoire ou d'un fichier ; dans ce dernier cas, il en fournira également la taille en octets.

```
nom du repertoire : e:\truc
Nom incorrect (inexistant ou non repertoire)
nom du repertoire : e:\book\exosjav
evbn.fm FICHIER 84992 octets
control.fm FICHIER 96256 octets
divers REPERTOIRE
menuac.fm FICHIER 112640 octets
  .....
classes REPERTOIRE
essai.txt FICHIER 5120 octets
fichiers.fm FICHIER 82944 octets
ap.fm FICHIER 35840 octets
```

Il nous suffit de recourir aux possibilités offertes par la classe *File*. Plus précisément, à partir du nom fourni par l'utilisateur dans la chaîne *nomRepert*, nous créons un objet *objRep* de type *File* :

```
objRep = new File (nomRepert) ;
```

La méthode *isDirectory* nous permet de savoir si ce nom correspond bien à un répertoire. Notez qu'il n'est pas nécessaire ici de recourir à la méthode *exists*, dans la mesure où nous n'avons pas cherché à distinguer le cas d'un nom ne désignant pas un répertoire du cas d'un nom inexistant.

Lorsque le nom correspond bien à un répertoire, nous faisons appel à la méthode *listFiles* qui nous fournit un tableau d'objets de type *File*, chaque élément correspondant à un des membres du répertoire. Il nous suffit alors d'appliquer à chacun d'entre eux les méthodes *isDirectory*, *getName* et *length* pour obtenir les informations voulues.

```
import java.io.* ;     // pour la classe File
```

```
public class ListRep
{ public static void main (String args[])
 { String nomRepert ;
  File objRep ;
  boolean ok ;
   /* lecture nom de repertoire */
  ok = false ;
  do
  { System.out.print ("nom du repertoire : ") ;
   nomRepert = Clavier.lireString () ;
   objRep = new File (nomRepert) ;
   if (objRep.isDirectory())
    ok = true ;
   else
    System.out.println   ("Nom   incorrect   (inexistant   ou   non
    repertoire)") ;
  }
  while (!ok) ;

   /* affichage des informations correspondantes */
  File [] membres = objRep.listFiles() ;
  for (int i=0 ; i<membres.length ; i++)
  { String type ;
   System.out.print (membres[i].getName()+ " ") ;
   if (membres[i].isFile())
    System.out.println  ("FICHIER  "  +  membres[i].length()  +  "
    octets") ;
   else
    System.out.println ("REPERTOIRE ") ;
  }
 }
}
```

---

1. L'utilisateur peut fournir indifféremment un nom relatif (au répertoire courant) ou un nom absolu.

2. Au lieu de la méthode *listFiles*, nous aurions pu aussi utiliser *list* qui fournit un tableau de chaînes dans lequel chaque élément représente un nom de membre. Il aurait alors fallu créer les objets de type *File* correspondants pour obtenir les

informations voulues.

Chapitre 16

La programmation générique

Connaissances requises

- Notion de classe générique et de paramètre de type
- Définition et utilisation d'une classe générique
- Notion d'effacement du paramètre de type et les limitations qui en découlent (instanciation d'un objet d'un type générique, tableaux d'objets d'un type paramétré, champs statiques d'un type paramétré)
- Notion de méthode générique
- Limitation des paramètres de type d'une classe générique ou d'une méthode générique
- Différentes possibilités de dérivation d'une classe générique
- Relation de « faux héritage » : si *T'* dérive de *T*, *C<T'>* ne dérive pas de *C<T>*
- Notion de joker simple
- Joker avec contraintes

Note : La programmation générique n'est disponible qu'à partir du JDK 5.0.

152 Classe générique à un paramètre de type

Écrire une classe générique *Triplet* permettant de manipuler des triplets d'objets d'un même type. On la dotera :

- d'un constructeur à trois arguments (les objets constituant le triplet),
- de trois méthodes d'accès *getPremier*, *getSecond* et *getTroisieme*, permettant d'obtenir la référence de l'un des éléments du triplet,
- d'une méthode *affiche* affichant la valeur des éléments du triplet.

Écrire un petit programme utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes.

La définition d'une classe générique se fait à l'aide d'un symbole (ici, *T*) représentant un type classe quelconque que l'on précise dans le nom de la classe comme dans :

```
class Triplet<T>
```

On utilise ce symbole *T* dans la suite de la définition de la classe, comme s'il s'agissait d'un type donné.

Voici comment nous pouvons définir la classe générique *Triplet* :

```
class Triplet<T>
{ private T x, y, z ;      // les trois éléments du triplet
 public Triplet (T premier, T second, T troisieme)
 { x = premier ; y = second ; z = troisieme ;
 }
 public T getPremier ()
 { return x ;
 }
 public T getSecond ()
 { return y ;
 }
 public T getTroisieme ()
```

```
 { return z ;
 }
 public void affiche ()
 { System.out.println ("premiere valeur : " + x + " - deuxieme
 valeur : " + y
             + " - troisieme valeur : " + z) ;
 }
}
```

Notez que dans la méthode *affiche* nous nous fondons implicitement sur la méthode *toString* des objets concernés.

Voici un petit programme utilisant cette classe *Triplet* :

```
public class TstTriplet
{ public static void main (String args[])
 { Integer oi1 = 3 ;    // équivalent à : Integer oi1 = new Integer
 (3) ;
  Integer oi2 = 5 ;    // équivalent à : Integer oi2 = new Integer
  (5) ;
  Integer oi3 = 12 ;   // équivalent à : Integer oi3 = new Integer
  (12) ;
  Triplet <Integer> ti = new Triplet<Integer> (oi1, oi2, oi3) ;
        // on aurait aussi pu écrire directement :
        // Triplet <Integer> ti = new Triplet<Integer> (3, 5, 12) ;
  ti.affiche () ;
  Triplet <Double> td = new Triplet <Double> (2.0, 12.0, 2.5) ;
       // on peut fournir des arguments de type double qui seront
       // convertis automatiquement en Double
  td.affiche() ;
  Integer n = ti.getTroisieme() ;
  System.out.println("troisieme element du triplet ti = " + n ) ;
  Double p = td.getPremier () ;
  System.out.println ("premier element du triplet td = " + p ) ;
  }
}
```

```
premiere valeur : 3 - deuxieme valeur : 5 - troisieme valeur : 12
premiere valeur : 2.0 - deuxieme valeur : 12.0 - troisieme valeur :
2.5
troisieme element du triplet ti = 12
premier element du triplet td = 2.0
```

153 Classe générique à plusieurs paramètres de type

Écrire une classe générique *TripletH* semblable à celle de l'exercice précédent, mais permettant cette fois de manipuler des triplets d'objets pouvant être chacun d'un type différent. Écrire un petit programme utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes.

Dans la définition de la classe, il suffit de prévoir cette fois trois paramètres de type. Si nous les nommons *T*, *U* et *V*, ils seront annoncés ainsi dans le nom de classe :

```
class TripletH <T, U, V>
```

Voici ce que pourrait être la définition de *TripletH* :

```
class TripletH <T, U, V>
{ private T x ; private U y ; private V z ;   // les trois éléments du triplet
 public TripletH (T premier, U second, V troisieme)
 { x = premier ; y = second ; z = troisieme ;
 }
 public T getPremier ()
 { return x ;
 }
 public U getSecond ()
 { return y ;
 }
 public V getTroisieme ()
 { return z ;
 }
 public void affiche ()
 { System.out.println ("premiere valeur : " + x + " - deuxieme valeur : " + y
             + " - troisieme valeur : " + z) ;
 }
```

```
}
```

Et en voici un petit programme d'utilisation :

```
public class TstTripletH
{ public static void main (String args[])
 { Integer oi = 3 ;
  Double od = 5.25 ;
  String os ="hello" ;
  TripletH <Integer, Double, String> tids
    = new TripletH <Integer, Double, String> (oi, od, os) ;
  tids.affiche () ;

  Integer n = tids.getPremier() ;
  System.out.println("premier element du triplet ti = " + n ) ;
  Double d = tids.getSecond () ;
  System.out.println ("second element du triplet td = " + d ) ;
 }
}
```

```
premiere valeur : 3 - deuxieme valeur : 5.25 - troisieme valeur :
hello
premier element du triplet ti = 3
second element du triplet td = 5.25
```

154 Conséquences de l’effacement (1)

Repérer les erreurs commises dans les instructions suivantes :

```
class C <T>
{ T x ;
 T[] t1 ;
 T[] t2 ;
 public static T inf ;
 public static int compte ;
 void f ()
 { x = new T () ;
  t2 = t1 ;
  t2 = new T [5] ;
 }
}
```

Rappelons que, lors de la compilation, la technique dite « de l’effacement », consiste à remplacer un type générique par un « type brut ». En l’absence d’indications contraires (limitations des paramètres de type), ce type brut est tout simplement *Object*. Dans ces conditions, un certain nombre d’opérations sont impossibles, notamment :

- définition d’un champ statique d’un type générique,
- instanciation d’un type générique ou, a fortiori, d’un tableau d’un type générique.

```
class C <T>
{ T x ;                          // OK
 T[] t1 ;                       // OK
 T[] t2 ;                       // OK
 public static T inf ;          // champ statique d'un type générique interdit
 public static int compte ;
 void f ()
```

```
 { x = new T () ;           // instanciation d'un type générique
 impossible
  t2 = t1 ;                 // OK
  t2 = new T [5] ;         // instanciation d'un tableau d'un type
  générique
            // impossible
 }
}
```

155 Conséquences de l'effacement (2)

Quels seront les résultats fournis par ce programme ?

```
public class TstStatic
{ public static void main (String args[])
 { C<Integer> ci = new C<Integer> () ;
  ci.affiche() ;
  C<Double> cd = new C<Double> () ;
  ci.affiche() ; cd.affiche() ;
  Class cci = ci.getClass() ;
  Class ccd = cd.getClass() ;
  if (cci == ccd) System.out.println
    ("ci et cd sont de la meme classe") ;
  else System.out.println ("ci et cd ne sont pas de la meme
  classe") ;
  System.out.println (cci.getName() + " " + ccd.getName()) ;
 }
}
class C<T>
{ public C () {compte++ ;}
 public void affiche ()
 { System.out.println ("compte = " + compte) ;
 }
 public void aff ()
 { System.out.println ("compte = " + compte) ;
 }
 private static int compte=0 ;
}
```

Compte tenu de l'effacement, lors de l'exécution, il n'existe qu'une seule classe correspondant au type brut de *C<Integer>* ou *C<Double>*, à savoir simplement *C*. Le

champ statique *compte* n'est finalement qu'un champ statique de cette classe *C*. Il n'existe donc qu'un seul « compteur » nommé *compte* pour tous les objets de type *C*<*T*>, quelle que soit la valeur de *T*. De même, la méthode *getClass* appliquée à ces différents objets fournit la même valeur, à savoir la référence à un objet de type *Class* dont le nom est *C*. Voici finalement les résultats fournis par ce programme :

```
compte = 1
compte = 2
compte = 2
ci et cd sont de la meme classe
C C
```

156 Méthode générique à un argument

Écrire une méthode générique fournissant en retour un objet tiré au hasard dans un tableau fourni en argument. Écrire un petit programme utilisant cette méthode.

Il suffit de réaliser une méthode générique possédant un seul paramètre de type, ayant un entête de la forme suivante :

```
static <T> T hasard (T [] valeurs)
```

Le choix d'un élément se fait en tirant sa position au hasard, en recourant à la méthode *Math.random* qui fournit une valeur au hasard dans l'intervalle [0, 1[. Voici la définition de notre méthode accompagnée d'un petit programme de test :

```
public class Hasard
{ static <T> T hasard (T [] valeurs)
  { if (valeurs == null) return null ;
   int n = valeurs.length ;
   if (n == 0) return null ;
   int i = (int) (n * Math.random() ) ;
   return valeurs[i] ;
  }
  public static void main(String args[])
  { Integer[] tabi = { 1, 7, 8, 4, 9} ;  // ici boxing automatique
   System.out.println ("hasard sur tabi = " + hasard (tabi) ) ;
   String[] tabs = {"Java", "C", "C++", "C#", "Visual Basic"} ;
   System.out.println ("hasard sur tabs = " + hasard (tabs) ) ;
  }
}
```

```
hasard sur tabi = 4
hasard sur tabs = Visual Basic
```

157 Méthode générique et effacement

Écrire une méthode qui renvoie au hasard un objet choisi parmi deux objets de même type fournis en argument. Écrire un petit programme utilisant cette méthode.

Là encore, il suffit de réaliser une méthode générique à un seul paramètre de type, et à deux arguments de ce type :

```
public static <T> T hasard (T x, T y)
 { double v = Math.random () ;
  if (v < 0.5) return x ;
        else return y ;
 }
```

En revanche, cette fois, compte tenu de l'effacement, cette méthode sera compilée comme si on l'avait écrite de la façon suivante :

```
public static Object hasard (Object x, Object y)
 { double v = Math.random () ;
  if (v < 0.5) return x ;
        else return y ;
 }
```

Ainsi, des appels de *hasard* avec des arguments de types différents seront acceptés par le compilateur. Il reste cependant possible de forcer le compilateur à s'assurer que les arguments effectifs sont d'un même type, ou d'un type compatible avec un type donné. On le précise lors de l'appel à l'aide d'une syntaxe de la forme suivante, dans laquelle *nomClasse* correspond au nom de la classe où la méthode générique est définie :

nomClasse<Type>.nomMéthode (arguments)

Nous en fournissons quelques exemples en commentaires du petit programme de test de la méthode *hasard* :

```
public class MethGen2arg
{ public static void main (String args[])
 { Integer i1 = 3 ; Integer i2 = 5 ;
```

```
  System.out.println ("hasard (i1, i2) = " + hasard (i1, i2)) ;
  String s1 = "Salut" ; String s2 = "bonjour" ;
  System.out.println ("hasard (s1, s2) = " + hasard (s1, s2)) ;
  System.out.println ("hasard (i1, s1) = " + hasard (i1, s1)) ;
  // Les appels suivants seront rejetés en compilation :
  //   MethGen2arg.<Integer> hasard (i1, s1) ;
  //   MethGen2arg.<String> hasard (i1, s1) ;
  // En revanche, ceux-ci seront acceptés :
  //   MethGen2arg.<Integer> hasard (i1, i2) ;
  //   MethGen2arg.<Number> hasard (i1, i2) ;
 }
 public static <T> T hasard (T x, T y)
 { double v = Math.random () ;
  if (v < 0.5) return x ;
        else return y ;
 }
}
```

158 Dérivation de classes génériques

On dispose de la classe générique suivante :

```
class Couple<T>
{ private T x, y ;      // les deux éléments du couple
 public Couple (T premier, T second)
 { x = premier ; y = second ;
 }
  public void affiche ()
 { System.out.println ("premiere valeur : " + x
           + " - deuxieme valeur : " + y ) ;
 }
}
```

1. Créer, par dérivation, un classe *CoupleNomme* permettant de manipuler des couples analogues à ceux de la classe *Couple<T>*, mais possédant, en outre, un nom de type *String*. On redéfinira convenablement les méthodes de cette nouvelle classe en réutilisant les méthodes de la classe de base.

2. Toujours par dérivation à partir de *Couple<T>*, créer cette fois une « classe ordinaire » (c'est-à-dire une classe non générique), nommée *PointNomme*, dans laquelle les éléments du couple sont de type *Integer* et le nom, toujours de type *String*.

3. Écrire un petit programme de test utilisant ces deux classes *CoupleNomme* et *PointNomme*.

1. Il suffit d'exploiter les possibilités de dérivation de classes génériques, en créant une nouvelle classe possédant le même paramètre de type que la classe de base. Voici ce que pourrait être la définition de notre classe *CoupleNomme* :

```
class CoupleNomme <T> extends Couple <T>
{ private String nom ;
 public CoupleNomme (T premier, T second, String nom)
 { super (premier, second) ;
  this.nom = nom ;
```

```
  }
 public void affiche ()
 { System.out.print ("nom = " + nom + " - " ) ;
  super.affiche() ;
 }
}
```

2. Cette fois, on crée une classe non générique, dérivant d'un classe générique, dans laquelle on fixe le paramètre de type (ici *T = Integer*). Voici ce que pourrait être la défintion de notre classe *PointNomme* :

```
class PointNomme extends Couple <Integer>
{ private String nom ;
 public PointNomme (Integer premier, Integer second, String nom)
 { super (premier, second) ;
  this.nom = nom ;
 }
 public void affiche ()
 { System.out.print ("nom = " + nom + " - " ) ;
  super.affiche() ;
 }
```

3. Voici un programme utilisant ces deux nouvelles classes, accompagné d'un exemple d'exécution :

```
public class TstDerivCouple
{ public static void main (String args[])
 { Couple <Double> cd1 = new Couple <Double> (5.0, 2.5) ;
  cd1.affiche () ;
  Couple <Double> cd2 = new Couple <Double> (5.0, 2.5) ;
  cd2.affiche () ;
  CoupleNomme <String> cns
        = new CoupleNomme <String> ("hello", "bonjour", "saluts") ;
  cns.affiche () ;
  CoupleNomme <Couple<Double>> cnd
        = new CoupleNomme <Couple<Double>> (cd1, cd2, "cf1") ;
  cnd.affiche () ;
  PointNomme p1 = new PointNomme (2, 5, "Point1") ;
  p1.affiche() ;
  }
 }
```

```
premiere valeur : 5.0 - deuxieme valeur : 2.5
```

```
premiere valeur : 5.0 - deuxieme valeur : 2.5
nom = saluts - premiere valeur : hello - deuxieme valeur : bonjour
nom = cf1 - premiere valeur : Couple@923e30 - deuxieme valeur : Couple@130c19b
nom = Point1 - premiere valeur : 2 - deuxieme valeur : 5
```

Notez qu'ici, nous avons exploité les possibilités de « composition » dans l'instanciation de la classe générique *cnd*, en créant un objet de type *CoupleNomme*, dans lequel les éléments sont d'un type *Couple<Double>*. On constate que la méthode *affiche* fournit alors simplement les adresses des deux éléments (de type *Couple<Double>*) du couple. En effet, ici, cette méthode se contente d'utiliser implicitement la méthode *toString* du type concerné (*Couple<Double>*).

159 Les différentes sortes de relation d'héritage

On suppose qu'on a défini une classe générique nommée *C* :

```
class C <T> { ..... }
```

ainsi qu'une classe ordinaire nommée *X*.

Pour chacune des définitions suivantes, donner les relations d'héritage existant entre les classes mentionnées en commentaires :

```
class D<T> extends C<T> { ..... } /*                définition 1
*/
 // C<Object>, C<Double>, D<Object>, D<Double>
class D<T, U> extends C<T> { ..... } /*             définition 2
*/
 // C<Double>, D(Double, Integer), D(Double, Double),
 // D(Integer, Double)
class D<T extends Number> extends C<T> { ..... } /*   définition
3 */
 // D<Double>, C<Double>, D<String>, C<String>
class D<T> extends X { ..... } /*                   définition 4
*/
 // D<Double>, X, D<String>
class D<T> extends C<String> /*                     définition 5
*/
 // D<String>, D<Integer>, C<String>, C<Integer>
```

1. *D<Double>* dérive de *C<Double>*

D<Object> dérive de *C<Object>*

En revanche, il n'existe aucune relation d'héritage entre *D<Double> et D<Object>*, pas plus qu'entre *C<Double>* et *C<Object>*.

2. *D<Double, Integer>* dérive de *C<Double>*

D<Double, Double> dérive de *C<Double>*

En revanche, *D<Integer, Double>* et *C<Double>* ne sont pas liés par une relation

d’héritage.

3. *D*<*Double*> dérive de *C*<*Double*> car *Double* implémente bien l’interface *Number*. En revanche, *D*<*String*> ne dérive pas de *C*<*String*> puisque *String* n’implémente pas *Number*.

4. *D*<*Double*> dérive de *X*

D<*String*> dérive de *X*

5. *D*<*String*> dérive de *C*<*String*>

D<*Integer*> dérive e *C*<*String*>

En revanche, *D*<*Integer*> ne possède aucun lien d’héritage avec *C*<*Integer*>.

160 Limitations des paramètres de type d'une méthode

Ecrire une méthode générique déterminant le plus grand élément d'un tableau, la comparaison des éléments utilisant l'ordre induit par la méthode *compareTo* de la classe des éléments du tableau.

On pourrait envisager pour notre méthode, nommée *max*, un en-tête de cette forme :

```
static <T> T max (T[] valeurs)
```

Mais, dans ce cas, le compilateur refuserait l'application de la méthode *compareTo* à des éléments de type *T*. Pour que ce soit possible, il est nécessaire de préciser que le type *T* implémente l'interface *Comparable*<*T*>, en employant un en-tête de cette forme

```
static <T extends Comparable<T> > T max (T[] valeurs)
```

Voici la définition de la méthode et un exemple d'utilisation :

```
public class MaxTab
{ public static void main (String args[])
 { Integer [] td = {2, 8, 1, 7, 4, 9 } ;
  System.out.println( "maxi de td = " + max (td) ) ;
  String [] ts = {"bonjour", "hello", "salut"} ;
  System.out.println ("maxi de ts = " + max (ts) ) ;
 }
 static <T extends Comparable<T> > T max (T[] valeurs)
 { if (valeurs == null) return null ;
   if (valeurs.length == 0) return null ;
   T maxi = valeurs[0] ;
   for (T v : valeurs) if (v.compareTo(maxi) > 0) maxi = v ;
   return maxi ;
 }
}
```

```
maxi de td = 9
maxi de ts = Visual Basic
```

En toute rigueur, dans certains cas, la spécification *Comparable<T>* de l'en-tête de *max* pourra poser des problèmes et il faudra recourir à des jokers de type super, en la remplaçant par *< T extends Comparable <? super T> >*, à l'instar de ce qui se fait dans certaines méthodes relatives aux collections. Ce point, dont la justification sort du cadre de ce manuel, concerne essentiellement les développeurs de bibliothèques génériques.

161 Redéfinition de la méthode compareTo

Compléter la classe *Point* suivante, de manière à ce que l'on puisse appliquer la méthode générique *max* précédente à un tableau d'objets de type *Point*. On conviendra que les points sont ordonnés par leur distance à l'origine.

```
class Point
{ private int x, y ;
 Point (int x, int y)
 { this.x = x ; this.y = y ;
 }
 public void affiche()
 { System.out.println ("coordonnees : " + x + " " + y ) ;
 }
}
```

Il faut faire implémenter à la classe *Point*, l'interface *Comparable* <*Point*> dont l'unique méthode a pour en-tête :

public int compareTo (Point p)

D'où la nouvelle définition de notre classe *Point* (ne pas oublier de mentionner que, dorénavant, la classe *Point* implémente *Comparable* <*Point*> :

```
class Point implements Comparable <Point>
{ private int x, y ;
 Point (int x, int y)
 { this.x = x ; this.y = y ;
 }
 public void affiche()
 { System.out.println ("coordonnees : " + x + " " + y ) ;
 }
 public int compareTo (Point p)
 { int norme1 = x * x + y * y ;
```

```
   int norme2 = p.x * p.x + p.y * p.y ;
   if (norme1 == norme2) return 0 ;
   if (norme1 > norme2) return 1 ;
            else return -1 ;
  }
```

Voici un petit programme appliquant la méthode *max* à des objets du nouveau type *Point* (par souci de lisibilité, nous avons reproduit la liste de la méthode *max*) :

```
public class MaxTabPoints
{ public static void main (String args[])
 { Point p1 = new Point (0, 5) ;
  Point p2 = new Point (3, 1) ;
  Point p3 = new Point (0, 12) ;
  Point p4 = new Point (3, 5) ;
  Point [] tp = {p1, p2, p3, p4} ;
  Point maxp = max (tp) ;
  System.out.println ("Point maxi : ") ;
  maxp.affiche() ;
 }
 static <T extends Comparable <T> > T max (T[] valeurs)
 { if (valeurs == null) return null ;
   if (valeurs.length == 0) return null ;
   T maxi = valeurs[0] ;
   for (T v : valeurs) if (v.compareTo(maxi) > 0) maxi = v ;
   return maxi ;
 }
}
```

```
point maxi :
coordonnees : 0 12
```

Chapitre 17

Les collections et les tables associatives

Connaissances requises

- Principales méthodes de l'interface *Collection*, indépendantes d'un itérateur : *add*, *size*, *contains*, *addAll*, *retainAll* et *removeAll*
- Interface *Iterator* et méthodes *next*, *hasNext* et *remove*
- Interface *ListIterator* et méthodes *previous*, *hasPrevious*, *set* et *add*
- Ordre des éléments d'une collection ; interface *Comparable* et méthode *compareTo* ; objets comparateurs
- Relativité de la notion d'égalité de deux éléments d'une même collection ; rôle de la méthode *equals*
- Utilisation de la boucle *for… each* sur des collections
- Classe *LinkedList* et méthodes spécifiques *removeFirst* et *removeLast*
- Classe *ArrayList* et méthodes spécifiques travaillant avec une position donnée : *get*, *set*, *add* et *remove*
- Classe *HashSet* et méthodes *hashCode* et *equals*
- Classe *TreeSet* et méthode *compareTo*

Note : nous supposons que nous travaillons avec une version Java 5 ou postérieure, ce qui nous permet d'utiliser des collections et des tables génériques, l'emballage et le déballage automatique, ainsi que la boucle dite *for…each*. En revanche, nous ne ferons pas appel aux spécificités du JDK 8, lesquelles ne seront exploitées que dans le chapitre suivant.

• Algorithmes applicables aux collections : tri, recherche de maximum ou de minimum, recherche binaire, copie ; rôle de la méthode *compareTo* ou d'un objet comparateur

• Tables associatives, interface *Map*, classes *HashMap* et *TreeMap*, méthodes *put*, *get* et *remove*

• Notion de vue associée à une table : classe *Map.Entry* et méthodes *entrySet*, *keySet*, *getKey* et *getValue*

162 Dépendance ou indépendance d'un itérateur

Quels résultats fournira ce programme :

```
import java.util.*;
public class Essai
{ public static void main (String args[])
 { LinkedList<Integer> liste = new LinkedList <Integer> () ;
  liste.add (3) ; liste.add (5) ; nliste.add (3) ;
  liste.add (12) ;nliste.add (3) ;
  System.out.println (liste) ;
  liste.remove (3) ; System.out.println (liste) ;
  liste.remove (new Integer(3)) ; System.out.println (liste) ;
  Iterator <Integer> it = liste.iterator () ;
  while (it.hasNext())if (it.next()==3) it.remove() ;
  System.out.println (liste) ;
 }
}
```

Rappelons tout d'abord que, depuis Java 5, les possibilités dites d'emballage et de déballage automatiques (*autoboxing*) permettent le recours automatique à des classes enveloppes. Ainsi, l'appel :

```
liste.add (3) ;
```

remplace avantageusement :

```
liste.add (new Integer(3)) ;
```

Cependant, cette démarche ne s'applique plus dans l'appel :

```
liste.remove (3) ;
```

En effet, il existe une méthode *remove (int)* qui, compte tenu des règles relatives à la surdéfinition, se trouvera appelée ici. Elle supprime l'élément de rang 3 de *liste*.

En revanche :

```
liste.remove (new Integer(3)) ;
```

supprime bien le premier élément de la liste dont la valeur est égale à 3. Rappelons que l'égalité se fonde sur la méthode *equals* laquelle, dans le cas des classes enveloppes, considère bien la valeur des objets.

Enfin, la boucle :

```
while (it.hasNext())if (it.next()==3) it.remove() ;
```

permet de supprimer tous les éléments (restants) dont la valeur est égale à 3. Rappelons que la méthode *remove* supprime l'élément courant (c'est-à-dire celui désigné par l'itérateur).

Enfin, dans une instruction telle que :

```
System.out.println (liste) ;
```

il y a appel de la méthode *toString* de l'objet *liste*. Celle-ci, comme pour toute collection, appelle la méthode *toString* de chacun de ses éléments.

En définitive, ce programme fournit les résultats suivants :

```
[3, 5, 3, 12, 3]
[3, 5, 3, 3]
[5, 3, 3]
[5]
```

163 Manipulation d'un tableau de type ArrayList

On dispose d'un objet *tab* déclaré ainsi :

```
ArrayList <Integer> tab ;
```

Écrire les instructions réalisant les actions suivantes sur les valeurs de *tab* :

- affichage dans l'ordre naturel (on proposera au moins 4 solutions) ;
- affichage dans l'ordre inverse (au moins 2 solutions) ;
- affichage des éléments de rang pair (0, 2, 4…) (au moins 2 solutions) ;
- mise à zéro des éléments de valeur négative (au moins 2 solutions).

L'affichage dans l'ordre naturel peut se faire :

- en utilisant la boucle dite *for… each* :

```
for (int elem : tab) System.out.print (elem + " ") ;
```

- en utilisant le recours automatique à la méthode *toString* de *ArrayList* :

```
System.out.println (tab) ;
```

- en recourant à la méthode *get* pour parcourir les différents éléments du tableau :

```
for (int i=0 ; i<tab.size() ; i++) System.out.print (tab.get(i)+ "
") ;
```

- en utilisant un itérateur :

```
ListIterator <Integer> it = tab.listIterator() ;
while (it.hasNext()) System.out.print (it.next() + " ") ;
```

L'affichage dans l'ordre inverse ne peut plus utiliser les deux premières démarches. Les deux dernières restent applicables :

```
for (int i=tab.size()-1 ; i>=0 ; i--) System.out.print (tab.get(i)+
" ") ;
ListIterator <Integer> itr = tab.listIterator(tab.size()) ; // fin
de liste
while (itr.hasPrevious()) System.out.print(itr.previous() + " ") ;
```

Il en va de même pour l'affichage des éléments de rang pair :

```
for (int i=0 ; i<tab.size() ; i+=2) System.out.print (tab.get(i)+ "
") ;

System.out.println ("\nelements de rang pair - methode 2") ;
while (itp.hasNext())
{ System.out.print(itp.next() + " ") ;
  itp.next();
}
```

La mise à zéro des éléments négatifs ne peut, là encore, se faire qu'en utilisant soit les méthodes *get* et *set*, soit un itérateur :

```
for (int i=0 ; i<tab.size() ; i++) if (tab.get(i) < 0) tab.set (i,
0) ;

ListIterator <Integer> itz = tab.listIterator() ;
while (itz.hasNext())if (itz.next() < 0) itz.set(0) ;
```

Notez bien que la boucle *for… each* ne permet que des consultations des éléments d'une collection. Elle n'est donc pas utilisable ici. Bien que correcte sur un plan syntaxique, l'instruction suivante se contenterait d'agir à plusieurs reprises sur la valeur de *elem* (la mettant à zéro lorsqu'elle est négative), mais laisserait inchangée la valeur correspondante du tableau :

```
for (int elem : tab) if (elem < 0) elem = 0 ;
```

Voici un exemple de programme complet reprenant ces diverses démarches, accompagné d'un exemple d'exécution. Notez que, pour tester les démarches de mise à zéro des éléments négatifs, nous avons dû travailler sur une copie du tableau initial.

```
import java.util.* ;
public class Tableau
{ public static void main (String args[])
 { int t[] = { 3, -5, 9, 2, 0, -8, 12, 7, 3, 12 } ;
  ArrayList <Integer> tab = new ArrayList<Integer> () ;
  for (int elem : t) tab.add (elem) ;

   // affichage ordre naturel
  System.out.println ("ordre naturel - methode 1") ;
  for (int elem : tab) System.out.print (elem + " ") ;
  System.out.println ("\nordre naturel - methode 2") ;
  System.out.println (tab) ;
  System.out.println ("ordre naturel - methode 3") ;
  for (int i=0 ; i<tab.size() ; i++) System.out.print (tab.get(i)+ "
  ") ;
```

```
  System.out.println ("\nordre naturel - methode 4") ;
  ListIterator <Integer> it = tab.listIterator() ;
  while (it.hasNext()) System.out.print(it.next() + " ") ;

   // affichage ordre inverse
  System.out.println ("\nordre inverse - methode 1") ;
  for   (int   i=tab.size()-1   ;   i>=0   ;   i--)   System.out.print
  (tab.get(i)+ " ") ;
  System.out.println ("\nordre inverse - methode 2") ;
  ListIterator <Integer> itr = tab.listIterator(tab.size()) ; // fin
  de liste
  while (itr.hasPrevious()) System.out.print(itr.previous() + " ") ;

   // affichage éléments de rang pair
  System.out.println ("\nelements de rang pair - methode 1") ;
  for (int i=0 ; i<tab.size() ; i+=2) System.out.print (tab.get(i)+
  " ") ;
  System.out.println ("\nelements de rang pair - methode 2") ;
  ListIterator <Integer> itp = tab.listIterator() ;
  while (itp.hasNext())
  { System.out.print(itp.next() + " ") ;
   itp.next();
  }
   // mise a zero d'une copie de tab
  ArrayList <Integer> tab1 = new ArrayList<Integer> (tab) ;
  System.out.println ("\nmise a zero - methode 1") ;
  for (int i=0 ; i<tab1.size() ; i++) if (tab1.get(i) < 0) tab1.set
  (i, 0) ;
  System.out.println (tab1) ;

  tab1 = new ArrayList<Integer> (tab) ;
  System.out.println ("mise a zero - methode 2") ;
  ListIterator <Integer> itz = tab1.listIterator() ;
  while (itz.hasNext())if (itz.next() < 0) itz.set(0) ;
  System.out.println (tab1) ;
 }
}
```

```
ordre naturel - methode 1
3 -5 9 2 0 -8 12 7 3 12
ordre naturel - methode 2
[3, -5, 9, 2, 0, -8, 12, 7, 3, 12]
```

```
ordre naturel - methode 3
3 -5 9 2 0 -8 12 7 3 12
ordre naturel - methode 4
3 -5 9 2 0 -8 12 7 3 12
ordre inverse - methode 1
12 3 7 12 -8 0 2 9 -5 3
ordre inverse - methode 2
12 3 7 12 -8 0 2 9 -5 3
elements de rang pair - methode 1
3 9 0 12 3
elements de rang pair - methode 2
3 9 0 12 3
mise a zero - methode 1
[3, 0, 9, 2, 0, 0, 12, 7, 3, 12]
mise a zero - methode 2
[3, 0, 9, 2, 0, 0, 12, 7, 3, 12]
```

164 Tri d'une collection (1)

On dispose de la classe *Cercle* suivante :

```
class Cercle
{ public Cercle (int x, int y, double rayon)
 { this.x = x ; this.y = y ; this.rayon = rayon ; }
 public void affiche ()
 { System.out.println ("Coordonnees : " + x + ", " + y
           + " ; rayon : " + rayon) ;
 }
 public double getRayon () { return rayon ; }
 public int getX () { return x ; }
 private int x, y ;
 double rayon ;
}
```

Écrire les instructions permettant de trier, sans modifier la classe *Cercle*, un tableau (de type *ArrayList*) d'objets de type *Cercle* :

- suivant les valeurs croissantes de leur rayon ;
- suivant les valeurs croissantes de leur abscisse.

La classe *Collections* fournit différents algorithmes de tri d'une collection quelconque implémentant l'interface *List*, ce qui est le cas de *ArrayList*. L'ordre de tri y est défini soit par la méthode *compareTo* de la classe concernée qui doit alors implémenter l'interface *Comparable*, soit par ce que l'on nomme un objet comparateur, fourni en argument de l'algorithme de tri.

Manifestement ici, la classe *Cercle* n'implémentant pas l'interface *Comparator*, il faut se tourner vers la seconde démarche. Ici, notre objet comparateur devra implémenter l'interface *Comparator<Cercle>*, c'est-à-dire disposer d'une méthode *compare (Cercle, Cercle)* renvoyant un entier (dont la valeur exacte est sans importance) :

- négatif si le premier argument est considéré comme inférieur au second ;
- nul si le premier argument est considéré comme égal au second ;

• positif si le premier argument est considéré comme supérieur au second.

Voici ce que pourrait être le code demandé. Ici, nous avons choisi (arbitrairement) d'utiliser une classe comparateur pour le premier tri et une classe anonyme pour le second.

```
import java.util.* ;
public class EssaiComparateur
{ public static void main (String args[])
 { ArrayList <Cercle> liste = new ArrayList <Cercle> () ;
  Cercle c1 = new Cercle (5, 3, 5.0) ;
  Cercle c2 = new Cercle (1, 9, 3.5) ;
  Cercle c3 = new Cercle (2, 9, 2.5) ;
  liste.add (c1) ; liste.add (c2) ; liste.add (c3) ;
   // tri suivant le rayon du cercle
  Collections.sort (liste, new Comparateur1 () ) ;
  System.out.println ("-- Cercles tries par rayon croissant") ;
  for (Cercle c : liste) c.affiche () ;
   // tri suivant l'abcisse du cercle
  Collections.sort (liste, new Comparator <Cercle> ()
     { public int compare (Cercle c1, Cercle c2)
      { double x1 = c1.getX () ; double x2 = c2.getX () ;
         if (x1 < x2) return -1 ;
        else if (x1 == x2) return 0 ;
         else return 1 ;
      }
     } ) ;
  System.out.println ("-- Cercles tries par abscisse croissante") ;
  for (Cercle c : liste) c.affiche () ;
 }
}
class Comparateur1 implements Comparator <Cercle>
{ public int compare (Cercle c1, Cercle c2)
 { double r1 = c1.getRayon () ;
  double r2 = c2.getRayon () ;
      if (r1 < r2 ) return -1 ;
    else if ( r1 == r2) return 0 ;
        else return 1 ;
 }
}
```

```
class Cercle
{ public Cercle (int x, int y, double rayon)
 { this.x = x ; this.y = y ; this.rayon = rayon ; }
public void affiche ()
{ System.out.println ("Coordonnees : " + x + ", " + y
            + " ; rayon : " + rayon) ;
}
public double getRayon () { return rayon ; }
public int getX () { return x ; }
private int x, y ;
double rayon ;
}
```

```
-- Cercles tries par rayon croissant
Coordonnees : 2, 9 ; rayon : 2.5
Coordonnees : 1, 9 ; rayon : 3.5
Coordonnees : 5, 3 ; rayon : 5.0
-- Cercles tries par abscisse croissante
Coordonnees : 1, 9 ; rayon : 3.5
Coordonnees : 2, 9 ; rayon : 2.5
Coordonnees : 5, 3 ; rayon : 5.0
```

Notez que l'écriture de la classe anonyme dans l'appel de *Collections.sort* pourrait également utiliser la méthode *compareTo* de la classe *Integer*, moyennant l'emploi de conversions de *int* en *Integer* :

```
Collections.sort (liste, new Comparator <Cercle> ()
   { public int compare (Cercle c1, Cercle c2)
    {       return       ((Integer)(c1.getX())).compareTo((Integer)
    (c2.getX())) ;
    }
   } ) ;
```

Il en va de même pour la méthode *compare* de la classe *Comparateur1* :

```
public int compare (Cercle c1, Cercle c2)
{         return         ((Double)(c1.getRayon())).compareTo((Double)
(c2.getRayon())) ;
}
```

---

165 Tri d'une collection (2)

Modifier la classe *Cercle* de l'exercice précédent, de manière à ce que l'appel (*liste* étant un objet de type *ArrayList<Cercle>*) :

```
Collections.sort (liste) ;
```

trie les éléments de *liste*, suivant les valeurs croissantes de leur rayon.

Cette fois, l'énoncé nous interdit d'employer un objet comparateur, comme nous l'avons fait dans l'exercice précédent. Il faut donc obligatoirement que la clase *Cercle* implémente l'interface *Comparable* et qu'elle définisse la méthode *compareTo*, de façon appropriée.

Voici la classe *Cercle* modifiée dans ce sens, accompagnée d'un petit programme de test et d'un exemple d'exécution.

```
import java.util.* ;
public class EssaiTri
{ public static void main (String args[])
 { ArrayList <Cercle> liste = new ArrayList <Cercle> () ;
  Cercle c1 = new Cercle (5, 3, 5.0) ;
  Cercle c2 = new Cercle (1, 9, 3.5) ;
  Cercle c3 = new Cercle (2, 9, 2.5) ;
  liste.add (c1) ; liste.add (c2) ; liste.add (c3) ;
   // tri suivant le rayon du cercle
  Collections.sort (liste) ;
  System.out.println ("-- Cercles tries par rayon croissant") ;
  for (Cercle c : liste) c.affiche () ;
  }
}
class Cercle implements Comparable<Cercle>
{ public Cercle (int x, int y, double rayon)
 { this.x = x ; this.y = y ; this.rayon = rayon ; }
 public void affiche ()
 { System.out.println ("Coordonnees : " + x + ", " + y
            + " ; rayon : " + rayon) ;
```

```
 }
 public int compareTo (Cercle c)
 { if (rayon < c.rayon ) return -1 ;
  else if ( rayon == c.rayon) return 0 ;
  else return 1 ;
 }
  // on peut aussi utiliser compareTo sur des Double :
  // return ((Double)(rayon)).compareTo ((Double)(c.rayon))
 public double getRayon () { return rayon ; }   // inutilisee ici
 public int getX () { return x ; }              // inutilisee ici
 private int x, y ;
 double rayon ;
}
```

```
-- Cercles tries par rayon croissant
Coordonnees : 2, 9 ; rayon : 2.5
Coordonnees : 1, 9 ; rayon : 3.5
Coordonnees : 5, 3 ; rayon : 5.0
```

Notez que, cette fois, les méthodes *getX* et *getRayon* de la classe *Cercle* ne sont plus utilisées, puisque notre méthode *compareTo* a bien accès aux champs privés.

Par ailleurs, l'ordre de tri est défini, une fois pour toutes, dans la classe *Cercle* elle-même. Si l'on souhaite effectuer d'autres sortes de tris, il faudra quand même recourir à la démarche de l'exercice précédent en fournissant un objet comparateur, indépendant de la classe *Cercle*.

166 Réalisation d’une liste triée en permanence

Réaliser une classe nommée *ListeTriee* permettant de manipuler une liste de chaînes de caractères, en s’arrangeant pour qu’elle soit triée en permanence. Outre le constructeur, on la dotera des méthodes :

- *ajoute* qui ajoute un nouvel élément à la bonne place ;
- *affiche* qui affiche les éléments de la liste.

L’énoncé n’impose pas le type de collection à utiliser pour conserver les chaînes. L’interface *List* nous convient tout à fait puisqu’elle permet de parcourir les éléments de la collection et d’insérer un nouvel élément entre deux autres. Nous pouvons indifféremment employer un objet de type *ArrayList* ou de type *LinkedList*.

La méthode *ajoute (String ch)* devra rechercher dans la liste le premier élément supérieur à *ch*. Si un tel élément existe, on ajoutera *ch* avant (il faudra utiliser *previous* pour "reculer" l’itérateur). Si un tel élément n’existe pas, il suffira d’ajouter *ch* à la position courante de l’itérateur, et ceci que la liste soit vide ou non.

Voici ce que pourrait être la classe demandée, implémentée ici avec un objet de type *LinkedList*, accompagnée d’un petit programme de test et d’un exemple d’exécution.

```
import java.util.* ;
public class TestListeTriee
{ public static void main (String args [])
 { ListeTriee liste = new ListeTriee () ;
  liste.ajoute ("c") ;
  liste.affiche() ;
  liste.ajoute ("b") ;
  liste.affiche() ;
  liste.ajoute ("f") ;
  liste.affiche() ;
  liste.ajoute ("e") ;
```

```
  liste.affiche() ;
 }
}
class ListeTriee
{ public ListeTriee ()
 { liste = new LinkedList <String> () ;  // ou ArrayList
 }
 public void ajoute (String ch)
{ ListIterator <String> it = liste.listIterator () ;
  boolean trouve = false ;
  while ((it.hasNext()) && ! trouve)
  { if (it.next().compareTo(ch) > 0) trouve = true ;
  }
  if (trouve) it.previous() ;     // ici, il y obligatoirement un
  precedent
  it.add (ch) ;
 }
 public void affiche ()
 {
  for (String ch : liste) System.out.print (ch + " ") ;
  System.out.println () ;
 }
 private LinkedList <String> liste ;    // ou ArrayList
}
```

```
c
b c
b c f
b c e f
```

167 Création d'un index

Réaliser une classe nommée *Index* permettant de gérer un index d'ouvrage. Un tel index associe une *entrée* (mot ou suite de mots) à un ou plusieurs numéros de page (contrairement à ce qui se passe dans les index de la plupart des ouvrages, on ne prévoira pas d'entrées à plusieurs niveaux).

La classe *Index* disposera, en plus d'un constructeur, des méthodes :

- *ajouter* pour introduire une nouvelle entrée, associée à un numéro de page ;
- *liste* pour afficher la liste de l'index, par ordre alphabétique des entrées, la liste des numéros de page d'une même entrée étant triée par valeur croissante ; l'affichage d'une entrée d'index se présentera sur une même ligne sous la forme :

```
Java : 12 25
```

On prendra bien soin de n'associer qu'une seule fois un même numéro de page à une entrée donnée.

Pour représenter notre index, nous utiliserons une table associative. Rappelons qu'il s'agit d'un ensemble de paires, formant chacune une association entre une clé et une valeur. Ici, la clé sera une entrée d'index (de type *String*). Quant à la valeur, elle correspondra à la liste des numéros associés à une entrée. Ceux-ci pourraient être conservés dans un objet de type *List*, mais ce dernier devrait alors être trié au moment de l'affichage de l'index et de plus, il faudrait y éviter les doublons. Il est plus simple d'utiliser un ensemble (ce qui élimine les doublons) de type *TreeSet* (il sera alors trié en permanence).

Quant à la table associative elle-même, nous choisirons également le type *TreeMap*, de sorte qu'elle sera toujours triée sur les entrées d'index. Rappelons que, dans une table associative (quel que soit son type exact), les clés sont toujours uniques.

En définitive, notre index sera conservé dans un objet du type :

```
TreeMap <String, TreeSet <Integer> >
```

Pour ajouter une nouvelle entrée à l'index, la méthode

```
ajouter (String entree, int numero)
```

recherchera tout d'abord l'ensemble des numéros déjà associés à la clé *entree*, en

recourant à la méthode *get (entree)*. Si cette entrée n'existe pas (la méthode *get* fournira alors la valeur *null*), on introduira une nouvelle paire dans l'index, à l'aide de la méthode *put*, à laquelle on fournira comme clé *entree* et comme valeur un nouvel ensemble formé du seul *numero*. Dans le cas contraire (*get* fournira alors une valeur non nulle correspondant à la référence sur la valeur associée à la clé fournie), on ajoutera *numero* à l'ensemble des numéros existants et l'on utilisera également la méthode *put* pour introduire la paire voulue dans l'index ; comme les clés sont uniques, la paire ainsi ajoutée prendra bien la place de l'ancienne.

Une table associative ne dispose pas d'itérateur. Pour effectuer la liste de notre index, nous devrons utiliser la méthode *entrySet* qui permet de "voir" la table comme un ensemble de paires (clé, valeur). Chaque paire est un élément de type *Map.Entry* dont les méthodes *getKey* et *getValue* permettent d'obtenir respectivement la clé et la valeur correspondante.

En définitive, voici notre classe *Index*, accompagnée d'un petit programme de test :

```
import java.util.* ;
public class TestIndex
{ public static void main (String args[])
 { Index monIndex = new Index () ;
  monIndex.ajouter ("Java", 25) ;
  monIndex.ajouter ("C++", 45) ;
  monIndex.ajouter ("Java", 12) ;
  monIndex.ajouter ("objet", 15) ;
  monIndex.ajouter ("polymorphisme", 45) ;
  monIndex.liste() ;
 }
}
class Index
{ public Index ()
 { index = new TreeMap <String, TreeSet <Integer> > () ; }
 public void ajouter (String entree, int numero)
 {   // si entree n'existe pas dans l'index, on l'ajoute, associe au
 numero
   // sinon, on ajoute le numero de page a l'ensemble des numeros
   // deja associes a entree
  TreeSet <Integer> numerosExistants = index.get(entree) ;
  if (numerosExistants == null)
  { TreeSet <Integer> numeroNouveauNom = new TreeSet <Integer> () ;
   numeroNouveauNom.add (numero) ;
   index.put (entree, numeroNouveauNom) ;
```

```
  }
  else
  { numerosExistants.add (numero) ;
   index.put(entree,  numerosExistants)  ;    //  remplace  l'entree
   precedente
  }
 }
 public void liste ()
 { Set <Map.Entry <String, TreeSet <Integer> > >
         elementsIndex = index.entrySet () ;
  for  (Map.Entry  <String,  TreeSet  <Integer>  >  elementCourant  :
  elementsIndex)
  { String entreeCourante = elementCourant.getKey () ;
   TreeSet <Integer> numeros = elementCourant.getValue () ;
   System.out.print (entreeCourante + " : ") ;
   for (int num : numeros) System.out.print (num + " ") ;
   System.out.println () ;
  }
 }
 private TreeMap <String, TreeSet <Integer> > index ;
}
```

```
C++ : 45
Java : 12 25
objet : 15
polymorphisme : 45
```

168 Inversion d'un index

Modifier la classe *Index* précédente, de sorte qu'elle dispose de deux méthodes supplémentaires :

• *creationIndexPage* créant un "index inversé", associant un numéro de page donné à la liste des entrées correspondantes ; on se limitera aux numéros de page associés à au moins une entrée ;

• *listeIndexPage* affichant la liste de cet index inversé, par ordre croissant des numéros de page, les entrées d'une même page étant triées par ordre alphabétique, sous la forme :

```
25 : Java langage
```

Nous dotons notre classe d'un nouveau champ *indexPage* destiné à représenter l'index inversé. Nous utiliserons également une table associative ; cette fois, la clé correspondra à un numéro de page (type *Integer*) tandis que la valeur correspondra à la liste des entrées associées à la page correspondante. Là encore, nous pourrions utiliser un objet de type *List*, mais un *TreeSet <String>* permettra d'éliminer les doublons et de conserver la liste des entrées d'une page triée par ordre alphabétique. Une nouvelle fois, pour la table associative, nous choisirons le type *TreeMap*, ce qui lui permettra d'être triée automatiquement sur le numéro de page. En définitive, notre champ *indexPage* sera déclaré ainsi :

```
private TreeMap <Integer, TreeSet <String> > indexPage ;
```

La méthode *creationIndexPage* devra parcourir chacune des paires de l'index initial. Pour ce faire, on utilisera la méthode *entrySet* qui servira à "voir" notre index comme un ensemble de paires (clé, valeur), chaque paire étant un élément de type *Map.Entry* dont les méthodes *getKey* et *getValue* permettant d'obtenir respectivement la clé (chaîne) et la valeur associée (ensemble de numéros de page).

Parallèlement, on créera l'index inversé, en procédant de façon similaire à ce que nous avions fait dans la méthode *ajouter* de l'exercice précédent. Cette fois, pour chaque numéro de page *n* associé à une entrée *e*, on recherchera dans l'index inversé un élément de clé *n*. S'il existe, on ajoutera à sa liste d'entrées l'entrée *e* ; dans le cas contraire, on créera un nouvel élément de clé *n*, dont la valeur sera un ensemble formé

de la seule entrée *e*.

En définitive, voici notre nouvelle classe *Index* (nous n'avons pas reproduit les méthodes *ajouter* et *liste*), accompagnée d'un petit programme de test.

```
import java.util.* ;
public class TestIndexParPage
{ public static void main (String args[])
 { Index monIndex = new Index () ; monIndex.ajouter ("Java", 25) ;
  monIndex.ajouter ("C++", 45) ; monIndex.ajouter ("Java", 12) ;
  monIndex.ajouter ("objet", 15) ; monIndex.ajouter
  ("polymorphisme", 45) ;
  monIndex.ajouter ("objet", 45) ; monIndex.ajouter ("langage",
  25) ;
  monIndex.creationIndexPage () ;
  monIndex.listeIndexPage () ;
 }
}
class Index
{ public Index () { // comme precedemment }
 public void ajouter (String entree, int numero) { // comme
 precedemment }
 public void liste () { // comme precedemment }
 public void creationIndexPage ()
 { indexPage = new TreeMap <Integer, TreeSet <String> > () ;
  Set <Map.Entry <String, TreeSet <Integer> > >
        elementsIndex = index.entrySet () ;
  for (Map.Entry <String, TreeSet <Integer> > elementCourant :
  elementsIndex)
  { String entreeCourante = elementCourant.getKey () ;
   TreeSet <Integer> pagesCourantes = elementCourant.getValue () ;
for (Integer numero : pagesCourantes)
   { TreeSet <String> entreesExistantes = indexPage.get(numero) ;
    if (entreesExistantes == null)
    { TreeSet <String> entreeNouveauNumero = new TreeSet <String>
    () ;
     entreeNouveauNumero.add(entreeCourante) ;
     indexPage.put(numero, entreeNouveauNumero) ;
    }
    else
    { entreesExistantes.add(entreeCourante) ;
     indexPage.put (numero, entreesExistantes) ;
```

```
    }
   }
  }
 }
 public void listeIndexPage ()
 { if (indexPage == null) return ;
  Set <Map.Entry <Integer, TreeSet <String> > >
         elementsIndexPage = indexPage.entrySet () ;
  for   (Map.Entry   <Integer,   TreeSet   <String>   >   numero   :
  elementsIndexPage)
  { Integer numeroCourant = numero.getKey () ;
   TreeSet <String> entrees = numero.getValue () ;
   System.out.print (numeroCourant + " : ") ;
   for (String entree : entrees) System.out.print (entree + " ") ;
   System.out.println () ;
  }
 }
 private TreeMap <String, TreeSet <Integer> > index ;
 private TreeMap <Integer, TreeSet <String> > indexPage ;
}
```

```
12 : Java
15 : objet
25 : Java langage
45 : C++ objet polymorphisme
```

Chapitre 18

Les expressions lambda et les streams

Connaissances requises

- Syntaxe des expressions lambda
- Interface fonctionnelle ; les principales interfaces fonctionnelles standards
- Références de méthodes
- L'interface *Comparator* ; méthodes *comparing*, *reversed*, *reversedOrder*
- Stream séquentiel et stream parallèle
- Les différentes sources pour un stream : collection, liste ou tableau de valeurs, génération
- Les méthodes intermédiaires : *filter*, *map*, *sorted*, *limit*, *peek*
- Les méthodes terminales : *forEach*, *forEachOrdered*, *count*, *sum*, *min*, *max*, *average*
- La méthode *reduce* (forme usuelle) ; la méthode *collect* et les objets *Collectors : toList*, *toMap*, *groupingBy* et *joining*

169 Lambda et interfaces prédéfinies

Écrire la méthode *affichage_selectif* afin que le programme suivant affiche les éléments positifs du tableau *tab* :

```
public class Affichage
{ public static void main (String [] args)
 { int [] tab = {1, 4, -2, 9, -3, 5, -3 } ;
  System.out.print ("-- Les positifs de tab : ") ;
  affichage_selectif (tab, ee -> ee > 0) ;
 }
}
```

On proposera deux solutions, l'une utilisant une interface personnalisée, l'autre une interface prédéfinie.

On voit que la méthode *affichage_selectif* doit recevoir en second argument une implémentation d'une interface fonctionnelle dont la méthode fonctionnelle reçoit un argument de type *int* et fournit un résultat de type booléen. Avec une interface personnalisée, il pourrait s'agir de (les noms *Filtrage* et *filtre* étant arbitraires) :

```
interface Filtrage
{ public Boolean filtre (int n) ;
}
```

Voici ce que pourrait être alors la méthode *affiche_selectif* :

```
public static void affichage_selectif (int [] t, Filtrage f)
 { for (int val : t) if (f.filtre (val)) System.out.print (val + "
 * ") ;
  System.out.println () ;
 }
```

On peut se passer de définir l'interface *Filtrage* en recourant à l'interface prédéfinie *IntPredicate* et à sa méthode fonctionnelle *test* (qui reçoit un *int* et renvoie un booléen). Voici ce que deviendrait notre programme complet dans ce cas :

```
import java.util.function.* ;
```

```
public class Affichage
{ public static void main (String [] args)
 { int [] tab = {1, 4, -2, 9, -3, 5, -3 } ;
  System.out.print ("-- Les positifs de tab : ") ;
  affichage_selectif (tab, ee -> ee > 0) ;
 }
 public static void affichage_selectif (int [] t, IntPredicate f)
 { for (int val : t) if (f.test (val)) System.out.print (val + " *
 ") ;
  System.out.println () ;
 }
}
```

```
-- Les positifs de tab : 1 * 4 * 9 * 5 *
```

170 Lambda et références

Compléter la dernière solution de l'exercice précédent, de manière que le programme affiche, en plus des nombres positifs du tableau *tab*, les nombres négatifs, puis les nombres pairs :

```
-- Les positifs :
1 * 4 * 9 * 5 * 12 * 7 * 6 *
-- Les negatifs :
-2 * -3 * -3 * -11 *
-- Les pairs :
4 * -2 * 12 * 0 * 6 *
```

On proposera tout d'abord une solution utilisant des expressions lambda, puis une solution utilisant des références à des méthodes qu'on écrira.

La première solution consiste simplement à utiliser comme deuxième argument de la méthode *affichage_selectif* l'une des expressions lambda suivantes :

```
ee -> ee < 0
ee -> 2 * (ee/2) == ee )
```

D'où les instructions supplémentaires :

```
System.out.print ("-- Les negatifs : ") ;
affichage_selectif (tab, ee -> ee < 0) ;
System.out.print ("-- Les pairs : ") ;
affichage_selectif (tab, ee -> 2 * (ee/2) == ee ) ;
```

La deuxième solution requiert que l'on puisse utiliser à la place des expressions lambda, la référence d'une méthode. Ici, nous utiliserons tout naturellement des méthodes statiques, recevant un argument de type *int* et renvoyant un résultat de type booléen. Nous les nommerons *filtragePositifs*, *filtrageNegatifs* et *filtragePairs*.

Voici ce que pourrait être le programme complet dans lequel nous avons fait figurer les deux solutions (expression lambda et référence) :

```
import java.util.function.* ;
public class Affichage
{ public static void main (String [] args)
```

```
 { int [] tab = {1, 4, -2, 9, -3, 5, -3, 12, 7,-11, 0, 6 } ;
  System.out.println ("-- Les positifs : ") ;
  affichage_selectif (tab, ee -> ee > 0 ) ;                // lambda
  affichage_selectif  (tab,  Affichage::filtragePositifs)  ;     //
  reference
  System.out.println ("-- Les negatifs : ") ;
  affichage_selectif (tab, ee -> ee < 0) ;                  // lambda
  affichage_selectif  (tab,  Affichage::filtrageNegatifs)  ;     //
  reference
  System.out.println ("-- Les pairs : ") ;
  affichage_selectif (tab, ee -> 2 * (ee/2) == ee ) ;       // lambda
  affichage_selectif  (tab,  Affichage::filtragePairs)  ;         //
  reference
 }
 public static void affichage_selectif (int [] t, IntPredicate f)
 { for (int val : t) if (f.test (val)) System.out.print (val + " *
 ") ;
  System.out.println () ;
 }
 public static Boolean filtrageNegatifs (int n) { return n < 0 ; }
 public static Boolean filtragePositifs (int n) { return n > 0 ; }
 public static Boolean filtragePairs (int n) { return 2*(n/2)==n ; }
}
```

```
-- Les positifs :
1 * 4 * 9 * 5 * 12 * 7 * 6 *
1 * 4 * 9 * 5 * 12 * 7 * 6 *
-- Les negatifs :
-2 * -3 * -3 * -11 *
-2 * -3 * -3 * -11 *
-- Les pairs :
4 * -2 * 12 * 0 * 6 *
4 * -2 * 12 * 0 * 6 *
```

171 L'interface Comparator

On dispose de la classe *Point* définie ainsi :

```
class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
 public int getX() { return x ; }
 public int getY() { return y ; }
 public void affiche ()
  { System.out.print(" [ " + x + ", " + y + "] ") ; }
 private int x, y ;
}
```

Écrire une méthode statique nommée *traiteListe* recevant en premier argument une liste de points sur laquelle elle réalise successivement trois opérations paramétrées par les trois arguments suivants :

- une sélection des éléments réalisant une condition ;
- un tri (suivant un critère variable) des éléments sélectionnés ;
- un affichage des éléments ainsi triés.

La méthode *traiteListe* s'utilisera ainsi (le type des arguments étant à préciser) :

```
traiteListe (liste, selection, tri, affichage) ;
```

On utilisera cette méthode :

- pour sélectionner les points d'abscisse positive, les trier sur les valeurs de leurs abscisses et les afficher suivant cette forme :

```
[ 2, 5] [ 2, 3] [ 6, -3]
```

- pour sélectionner tous les éléments, les trier suivant la somme de leurs coordonnées et les afficher ainsi :

```
(abs = -3, ord = 4) (abs = 6, ord = -3) (abs = 2, ord = 3)
```

Ici, on évitera d'utiliser des streams.

Le premier argument de la méthode *traiteListe* est naturellement de type *ArrayList<Point>*. Les suivants sont des implémentations des interfaces fonctionnelles

prédéfinies suivantes :

- *Predicate<Point>*, dont la méthode fonctionnelle est *test* ;
- *Comparator<Point>* ;
- *Consumer<Point>* dont la méthode fonctionnelle se nomme *accept*.

Comme l'énoncé nous impose de ne pas utiliser de *stream*, il est nécessaire de créer localement une liste nommée ici *liste2* contenant les éléments sélectionnés pour la soumettre au tri. Voici ce que pourrait être notre méthode *traiteListe* :

```
public static void traiteListe (ArrayList<Point> liste,
                Predicate<Point> selec,
                Comparator<Point> comp,
                Consumer<Point> aff)
 { ArrayList<Point> liste2 = new ArrayList <Point>() ;
  liste.forEach (ee -> { if (selec.test(ee)) liste2.add(ee) ; }) ;
  liste2.sort(comp);
  liste2.forEach(ee -> aff.accept(ee)) ;
 }
```

Pour le premier traitement, la sélection peut se faire à l'aide de l'expression lambda :

```
ee -> ee.getX()>0
```

Pour le comparateur à fournir en troisième argument, nous avons plusieurs possibilités :

- utiliser une expression lambda pour implémenter la méthode fonctionnelle *compare* de l'interface *Comparator<Point>*, ce qui conduit à une expression assez compliquée, notamment à cause des conversions en *Integer* dues à ce que la méthode *getX* fournit un *int* et non un *Integer* :

```
(pp1, pp2) -> ((Integer)(pp1.getX()))
        .compareTo (((Integer)(pp2.getX())))
```

- Utiliser la méthode *Comparator.comparing* pour créer un comparateur à partir d'une expression lambda :

```
Comparator.comparing (pp-> pp.getX())
```

- Utiliser à la fois la méthode *Comparator.comparing* et une référence de méthode :

```
Comparator.comparing(Point::getX)
```

Nous choisirons la dernière pour sa simplicité (mais, à titre indicatif, nous programmerons également la première dans notre exemple complet).

Quant à l'affichage, il peut se faire simplement ici à l'aide de la méthode *Point::affiche*.

Pour le deuxième traitement, il nous faut sélectionner tous les points ; nous utilisons

l’expression lambda :

```
xx -> true
```

Pour le comparateur, nous ne pouvons plus utiliser de référence à une méthode puisque la comparaion porte dorénavant sur la somme des coordonnées (à moins de créer une méthode supplémentaire à cet effet). Nous utiliserons donc la méthode *Comparator.comparing* de cette manière :

```
Comparator.comparing (xx -> xx.getX() + xx.getY())
```

Enfin, l’affichage ne peut plus recourir à la méthode *affiche* de la classe *Point*. Nous aurions pu créer une méthode statique et transmettre sa référence en argument. Ici, nous avons choisi une expression lambda.

En définitive, notre programme pourrait se présenter ainsi :

```
import java.util.function.* ;
import java.util.* ;
public class TraiteListe
{ public static void main (String [] args)
 { Point [] tab = { new Point(2, 5), new Point(-3, 4),
           new Point(2, 3), new Point(6, -3) } ;
  ArrayList<Point> l = new ArrayList<Point>() ;
  for (Point p : tab) l.add(p) ;
    // selection des points d'abscisse positive,
    // tri sur l'abscisse utilisant Comparator.comparing
  traiteListe (l, ee -> ee.getX()>0,
         Comparator.comparing(Point::getX),
         Point::affiche) ;
  System.out.println () ;
    // meme chose avec un comparateur sous forme d’expression lambda
  traiteListe (l, ee -> ee.getX()>0,
         (pp1, pp2) -> ((Integer)(pp1.getX()))
         .compareTo (((Integer)(pp2.getX()))),
         Point::affiche) ;
  System.out.println () ;
    // tri de tous les points suivant la somme des coordonnees
    // avec Comparator.comparing
  traiteListe (l, xx -> true,
         Comparator.comparing (xx -> xx.getX() + xx.getY()),
         (xx -> System.out.print ("(abs = " + xx.getX()
              + ", ord = " + xx.getY() + ") ") )) ;
}
```

```
 public static void traiteListe (ArrayList<Point> liste,
              Predicate<Point> selec,
              Comparator<Point> comp,
              Consumer<Point> aff)
 { ArrayList<Point> liste2 = new ArrayList <Point>() ;
  liste.forEach (ee -> { if (selec.test(ee)) liste2.add(ee) ; }) ;
  liste2.sort(comp);
  liste2.forEach(ee -> aff.accept(ee)) ;
 }
}
class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
 public int getX() { return x ; }
 public int getY() { return y ; }
 public void affiche () { System.out.print(" [ " + x + ", " + y + "]
 ") ; }
 private int x, y ;
}
```

```
[ 2, 5] [ 2, 3] [ 6, -3]
[ 2, 5] [ 2, 3] [ 6, -3]
(abs = -3, ord = 4) (abs = 6, ord = -3) (abs = 2, ord = 3) (abs = 2,
ord = 5)
```

172 Les méthodes usuelles des streams

Que produit le programme suivant :

```
import java.util.stream.* ;
public class ExoStream
{ public static void main (String args[])
 { int [] tab = { 3, 5, -3, 8, 12, 4, 7, 4, 8, 3 } ;
  long     nb     =     IntStream.of(tab).filter(xx     ->     xx
  >0).count() ;       // 1
  System.out.println ("nb = " + nb) ;
  IntStream.of(tab).filter(xx          ->          xx          >
  3).sorted()                // 2
          .forEach(xx -> System.out.print (xx + " ")) ;
  System.out.println();
  IntStream.of(tab).filter(xx               ->               xx
  >3).sorted().distinct()       // 3
          .forEach(xx -> System.out.print (xx + " ")) ;
  int           s           =IntStream.of(tab).map(xx           ->
  Math.abs(xx))               // 4
              .map(xx -> xx * xx).sum() ;
  System.out.println ("\nresultat = " + s) ;
 }
}
```

L'instruction 1 crée un stream à partir du tableau d'entiers *tab*. La méthode *filter* filtre les éléments positifs et ceux-ci sont comptabilisés par la méthode terminale *count*.

L'instruction 2 crée le même stream, en sélectionnant cette fois les éléments de valeur supérieure à 3, en les triant suivant l'ordre naturel et en les affichant.

L'instruction 3 fait la même chose que l'instruction 2 avec cette différence que, après le tri, on évite de conserver des valeurs en double, grace à la méthode *distinct*.

Enfin, l'instruction 4 effectue une première transformation associant à chaque nombre sa valeur absolue, puis une seconde transformation associant à chaque élément son carré (notez qu'ici on obtiendrait le même résultat, en supprimant la méthode *map*). Enfin, la méthode terminale *sum* effectue la somme de ces derniers.

En définitive, ce programme affiche :

```
nb = 9
```

```
4 4 5 7 8 8 12
4 5 7 8 12
resultat = 405
```

173 Traitement de liste avec un stream

L'exercice 171 proposait de réaliser une méthode statique de traitement de liste, laquelle devait nécessairement créer une nouvelle liste.

Écrire un programme effectuant les mêmes opérations, en utilisant un stream.

On proposera deux solutions :

- l'une utilisant toujours la méthode *traiteListe* ;
- l'autre n'utilisant plus cette méthode et se contentant d'effectuer le traitement directement au sein de la méthode *main*.

La première démarche consiste simplement à créer un *stream* dans la méthode *traiteListe* :

```
public static void traiteListe (ArrayList<Point> liste,
                Predicate<Point> selec,
                Comparator<Point> comp,
                Consumer<Point> aff)
 { liste.stream().filter(selec).sorted(comp).forEach(aff) ;
 }
}
```

On notera que, cette fois, il n'est plus nécessaire de créer une nouvelle liste puisque les différentes opérations réalisées par le stream n'en modifient pas la source (liste d'origine). Le reste du code est inchangé.

La deuxième démarche possède, là encore, plusieurs variantes suivant la manière dont on fournit les arguments aux méthodes *filter*, *sorted* et *forEach*. Voici une formulation possible :

```
import java.util.* ;
public class ExoTraiteLlistStream
{ public static void main (String [] args)
 { Point p1 = new Point (2, 5), p2 = new Point (-2, 3),
     p3 = new Point (6, -3), p4 = new Point (-3, -2) ;
  ArrayList<Point> l = new ArrayList<Point>() ;
  l.add(p1) ; l.add(p2) ; l.add(p3) ; l.add(p4) ;
    // sélection des points d'abscisse positive, tri sur l'abscisse
  l.stream().filter(ee -> ee.getX() > 0)
```

```
        .sorted(Comparator.comparing (xx -> xx.getX()))
        .forEach(Point::affiche) ;
  System.out.println () ;
    // tri de tous les points suivant la somme des coordonnees
  l.stream().sorted  (Comparator.comparing  (xx  ->  xx.getX()  +
  xx.getY()))
        .forEach (xx -> System.out.print ("(abs=" + xx.getX()
              + ", ord=" + xx.getY()) ) ;
 }
}
class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
 public int getX() { return x ; }
 public int getY() { return y ; }
 public void affiche ()
  { System.out.print(" [ " + x + ", " + y + "] ") ; }
 private int x, y ;
}
```

```
[ 2, 5] [ 6, -3]
(abs=-3, ord=-2) (abs=-2, ord=3) (abs=6, ord=-3) (abs=2, ord=5)
```

174 Répertoire

On dispose de la classe *Personne* suivante :

```
class Personne
{ public Personne (String prenom, String nom, int annee)
 { this.nom = nom ; this.prenom = prenom ; annee_naissance =
 annee ; }
  public String getNom() { return nom ; }
  public String getPrenom() { return prenom ; }
  public int getAnnee() { return annee_naissance ; }
  private String nom, prenom ;
  private int annee_naissance ;
 }
}
```

Réaliser les opérations suivantes sur un tableau d'objets de type *Personne*, en utilisant un stream :

- afficher le nom des personnes nées après 1985 ;
- afficher le nom des personnes nées avant 2000, triés par ordre alphabétique sur leur nom, et afficher leur nombre ;
- afficher tous les noms et prénoms, triés par ordre alphabétique sur leur nom et leur prénom.

Si *tab* désigne le tableau d'objets en question, on pourra créer un stream associé à ce tableau, à l'aide de la méthode statique *of* de la classe *Stream*, en procédant ainsi :

```
Stream.of(tab)
```

Pour la première opération, on lui appliquera un filtre pour sélectionner les personnes correspondant à la condition voulue :

```
.filter(pp -> pp.getAnnee() > 1985)
```

Enfin, l'affichage sera provoqué par l'opération terminale *forEach* :

```
.forEach(pp -> System.out.print (pp.getPrenom() + ", ")
```

Pour la seconde opération, on procédera de façon similaire, en ajoutant une opération intermédiaire de tri, à l'aide de la méthode *sorted*, à laquelle on fournira le comparateur voulu. Plusieurs démarches sont possibles, la plus simple étant de recourir à la méthode *Comparator.comparing* en lui fournissant la référence d'une méthode

existant déja dans la classe *Personne*, à savoir ici *Personne::getNom*.

En revanche, cette fois, on ne peut plus utiliser *forEach* pour afficher les noms demandés puisque cette opération terminale déclencherait l'exécution du *stream*, alors que nous voulons pouvoir l'utiliser pour lui appliquer la méthode *count* (à moins d'utiliser deux streams consécutifs, l'un pour l'affichage, l'autre pour le comptage). On peut recourir à la méthode *peek* qui, tout en attendant un *Consumer* comme *forEach*, présente la particularité d'être une méthode intermédiaire laissant le stream inchangé (on retrouve en sortie le stream fourni en entrée).

Enfin, pour la troisième opération, on utilisera également la méthode *sorted*, en lui fournissant toujours un comparateur créé par la méthode *Comparator.comparing* mais, cette fois, il n'est plus possible de lui fournir une référence de méthode (à moins d'en écrire une spécialement pour cela). On procèdera ainsi :

```
sorted(Comparator.comparing (pp -> pp.getNom()+pp.getPrenom()))
```

Voici un exemple de programme complet créant un "mini-répertoire" de cinq personnes (nous n'avons pas rappelé la classe *Personne*) :

```
import java.util.stream.* ;
import java.util.* ;
public class ExoRepert
{ public static void main (String [] args)
 { Personne[] tab = { new Personne ("thibault", "Rougier", 2001),
            new Personne ("thomas", "Niesseron", 1987),
            new Personne ("thifaine", "Mitenne", 1959),
            new Personne ("maxime", "Forest", 1995),
            new Personne ("jules", "Forest", 1995) } ;
 System.out.println ("--- Nes apres 1985 : ") ;
 Stream.of(tab).filter(pp -> pp.getAnnee() > 1985)
        .forEach(pp -> System.out.print (pp.getPrenom() + ", ")) ;
 System.out.println ("\n--- Nes avant 2000 :") ;
 long nombre = Stream.of(tab).filter(pp -> pp.getAnnee() < 2000)
             .sorted(Comparator.comparing(Personne::getNom))
             .peek(pp -> System.out.print (pp.getNom() + " "))
             .count() ;
  System.out.println ("\n Ils sont "+nombre) ;
  System.out.println ("--- Tous tries sur nom + prenom : ") ;
  Stream.of(tab).sorted(Comparator.comparing
               (pp -> pp.getNom() + pp.getPrenom()))
        .forEach(pp -> System.out.print ("(" + pp.getNom() + ", "
             + pp.getPrenom() +") ")) ;
 }
```

}

--- Nes apres 1985 :

thibault, thomas, maxime, jules,

--- Nes avant 2000 :

Forest Forest Mitenne Niesseron

    Ils sont 4

--- Tous tries sur nom + prenom :

(Forest, jules) (Forest, maxime) (Mitenne, thifaine) (Niesseron, thomas)

(Rougier, thibault)

175 Répertoire (bis)

On suppose qu'on dispose de la classe *Personne* de l'exercice précédent.

Écrire un programme qui, à partir d'une liste de personnes (*List<Personne>*) utilise un stream pour afficher l'année de naissance de la plus jeune. On proposera deux formulations :

- l'une ne recourant pas à un comparateur et affichant seulement l'année concernée ;
- l'autre affichant le nom, prénom et année de naissance de la personne concernée (on peut, cette fois, utiliser un comparateur).

Dans les deux démarches imposées, il faut manifestement recourir à la méthode *min* d'un stream. Lorsqu'on l'applique à un *Stream<T>*, il est nécessaire de lui fournir un comparateur. En revanche, lorsqu'on l'applique à un *IntStream*, la méthode *min* ne dispose d'aucun argument et utilise l'ordre naturel des entiers.

Dans les deux cas, nous travaillerons sur un *Stream<Personne>*. Dans le premier, nous utiliserons la méthode *mapToInt* pour extraire seulement les années de naissance, sous forme d'un *IntStream* auquel nous appliquerons la méthode *max*.

Dans le deuxième cas, nous appliquerons directement la méthode *min* sur ce *Stream<Personne>*, en fournissant un comparateur approprié, ce qui nous permettra de récupérer l'ensemble des informations sous forme d'un objet de type *Personne*.

Enfin, il faut ajouter que, dans les deux cas, la méthode *min* tient compte du fait que le stream peut être vide, en fournissant un objet de type *OptionalInt* (pour le premier cas) ou *Optional<Personne>* (pour le deuxième cas). Il faut alors utiliser la méthodes *isPresent* pour savoir s'il y a bien présence d'une valeur que l'on récupère, le cas échéant, par *getAsInt* (dans le premier cas) ou *get* (dans le deuxième).

En définitive, voici un exemple complet de programme créant, là encore, un mini-répertoire de cinq personnes. Notez que, l'énoncé imposant une liste, nous avons d'abord créé un tableau de personnes, que nous transformons en liste à l'aide de la méthode *asList* de la classe utilitaire *Arrays*.

```
import java.util.* ;
public class OptionalPersonne
```

```
{ public static void main (String [] args)
 { Personne[] tab = { new Personne ("thibault", "Rougier", 2001),
            new Personne ("thomas", "Niesseron", 1987),
            new Personne ("thifaine", "Mitenne", 1959),
            new Personne ("maxime", "Forest", 1995),
            new Personne ("jules", "Forest", 1995) } ;
  List<Personne> liste = Arrays.asList(tab) ;

   //  utilisation  d'un  Stream<Personne>  transforme  par  map  en
   IntStream
  OptionalInt anneeJeune = liste.stream()
        .mapToInt(pp -> pp.getAnnee()).max() ;
  if (anneeJeune.isPresent())
     System.out.println ("--- Methode 1 - Le plus jeune est ne en :
    "
              + anneeJeune.getAsInt()) ;
  else System.out.println ("--- Liste vide") ;

     // recherche de min sur un Stream<Personne>
  Optional<Personne> personneJeune =liste.stream()
        .max(Comparator.comparing(Personne::getAnnee)) ;
  if (personneJeune.isPresent())
  { Personne pj = personneJeune.get();
   System.out.println ("--- Methode 2 - Le plus jeune est : "
        +  pj.getNom()  +  "  "  +  pj.getPrenom()  +  "  "  +
        pj.getAnnee()) ;
  }
  else System.out.println ("--- Liste vide") ;
 }
}
class Personne
{ public Personne (String prenom, String nom, int annee)
 { this.nom = nom ; this.prenom = prenom ; annee_naissance = annee ;
 }
 public String getNom() { return nom ; }
 public String getPrenom() { return prenom ; }
 public int getAnnee() { return annee_naissance ; }
 private String nom, prenom ;
 private int annee_naissance ;
}
```

```
--- Methode 1 - Le plus jeune est ne en : 2001
--- Methode 2 - Le plus jeune est : Rougier thibault 2001
```

176 Reduce

Écrire un programme qui utilise un stream pour générer un nombre donné de nombres réels compris dans l'intervalle [0.5, 1.5[et en calculer le produit. On affichera les nombres compris dans l'intervalle [1 -e, 1+e], e étant une petite valeur (par exemple 0.01).

Faire la même chose en générant des nombres compris dans l'intervalle [0, 2[et en filtrant ceux compris entre 0.5 et 1.5.

Nous commençons par créer un stream d'éléments de type double (*DoubleStream*) à l'aide de la méthode génératrice *generate*, à laquelle on fournit l'expression lambda (la méthode *random* fournit un *double* compris dans l'intervalle [0, 1[) :

```
() -> (Math.random()+0.5)
```

Nous limitons les valeurs ainsi produites à l'aide de la méthode intermédiaire *limit*. Ensuite, il nous faut imprimer certaines valeurs, sans modifier le stream ; nous utilisons pour cela la méthode *peek*, à laquelle nous fournissons l'instruction effectuant l'impression des valeurs voulues :

```
.peek(xx -> { if (xx > 1-EPS && xx < 1+EPS)
        System.out.print (xx + " ") ;
        })
```

Enfin, comme nous ne disposons pas de méthode "toute faite" pour effectuer l'opération de réduction demandée (calcul du produit des éléments), nous recourons à la méthode *reduce*, en utilisant 1 comme valeur initiale (et donc, comme élément neutre de la réduction) et une expression lambda pour "l'accumulateur" :

```
.reduce(1, (xx, yy) -> xx * yy )
```

La deuxième question se résout de façon comparable. Il faut simplement effectuer un filtrage préalable des valeurs tirées au hasard, avant d'en limiter les valeurs par *limit*.

```
import java.util.stream.* ;
public class ExoReduce1
{ public static void main (String args[])
 { final int NVALEURS = 200 ;
  final double EPS = 1e-2 ;
```

```
  double     produit1     =     DoubleStream.generate     (()     ->
  (Math.random()+0.5))
           .limit(NVALEURS)
           .peek(xx -> { if (xx > 1-EPS && xx < 1+EPS)
              System.out.print (xx + " ") ;
              })
           .reduce(1, (xx, yy) -> xx * yy ) ;
  System.out.println ("\nProduit 1 : " + produit1) ;
  double produit2 = DoubleStream.generate ( () -> 2*Math.random())
             .filter (xx -> xx>0.5 && xx <1.5)
             .limit(NVALEURS)
             .peek(xx -> { if (xx > 1-EPS && xx < 1+EPS)
              System.out.print (xx + " ") ;
              })
             .reduce(1, (xx, yy) -> xx * yy ) ;
  System.out.println ("\nProduit 2 : " + produit2) ;
 }
}
```

```
0.9975467498452386        0.998138364621028        0.99188663131875
1.0083743070821198
0.9928318680893621
Produit 1 : 2.697255477885424E-5
0.9902255816404553       0.9940031238252993       0.9997538300614459
1.003265020181365
Produit 2 : 5.127710428111979E-4
```

177 Collect et Collectors

On suppose qu'on dispose de la clase *Point* suivante :

```
class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
 public int getX() { return x ; }
 public int getY() { return y ; }
 private int x, y ;
}
```

Écrire un programme qui réalise les opérations suivantes :

- À partir d'un tableau de points, créer un ensemble formé des points dont les abscisses sont positives ;
- À partir d'un tableau d'entiers, créer tout d'abord un *Stream<Point>* dont chaque élément a pour abscisse un élément du tableau et pour ordonnée son double ; utiliser ce stream pour créer un map dans lequel à chaque clé correspond la liste des points ayant cette clé comme abscisse.
- À partir d'un tableau de chaînes, créer une chaîne unique formée de la concaténation des chaînes de longueur supérieure à 4, séparées par le caractère "|".
- À partir de ce même tableau de chaînes, créer un map où à chaque clé représentant une lettre de l'aphabet correspond la liste des mots commençant par cette lettre.

Si notre tableau de points se nomme *tabPoints*, on voit qu'il faut créer un stream à l'aide de la méthode *of* :

```
Stream.of(tabPoints)
```

et le filtrer avec la condition :

```
.filter(xx -> getX()>0)
```

puis, pour créer l'ensemble voulu, on utilise la méthode *collect* en lui fournissant comme paramètre *Collectors.toSet()*, ce qui permet de collecter les valeurs du stream ainsi obtenu dans un *Set<Point>*.

Pour la seconde question, nous créons le *Stream<Point>* demandé en appliquant la méthode *map* de cette manière :

```
.map(xx -> new Point(xx, 2*xx) )
```

Puis, comme précédemment, nous collectons les points obtenus dans une liste à l'aide de la méthode *collector* à qui l'on fournit le paramètre *Collectors.toList()*.

Pour la question suivante, nous créons un *Stream<String>* à partir du tableau de chaînes, puis nous filtrons les éléments voulus par :

```
.filter(xx -> xx.length()>4)
```

Nous concaténons alors les chaînes sélectionnées à l'aide de la méthode *Collectors.joining()* fournie en argument de la méthode *collect*.

Pour la dernière question, nous choisirons des clés de type *String* plutôt que de type *Character*, ce qui nous évitera des opératons de conversion dans l'écriture du comparateur. Nous créerons donc un *Map<String, List<String>*, là encore à l'aide de la méthode *collect*, mais cette fois nous employons *Collectors.groupingBy* pour effectuer le regroupement voulu, la clé étant la première lettre des éléments.

Voici ce que pourrait être le programme.

```
import java.util.* ;
import java.util.stream.* ;
public class ExoCollect
{ public static void main (String [] args)
 { Point[] tabPoints = { new Point(2, 4), new Point(3, 8), new
 Point(1, 3),
             new Point(-2, 4), new Point(3, 8), new Point(1,3)} ;
 Set<Point> ens = Stream.of(tabPoints).filter(xx -> xx.getX()>0)
               .collect(Collectors.toSet()) ;
 System.out.print("Ensemble : ") ;
 ens.forEach (pp -> System.out.print ("["+pp.getX()+", "
               + pp.getY() + "] ")) ;
 Integer [] tab = { 2, 15, -3, 2, -5, 23, -8, 12 } ;
 List<Point> liste = Stream.of(tab).map(xx -> new Point (xx, 2*xx))
              .collect(Collectors.toList()) ;
 System.out.print("\nListe : ") ;
 liste.forEach    (pp    ->    System.out.print    ("["+pp.getX()+",
 "+pp.getY()+"] ")) ;
 String [] mots = {"bonjour", "hello", "buongiorno", "hi", "chao",
          "bom dia", "guten tag" } ;
 String                          mots_longs                          =
 Stream.of(mots).collect(Collectors.joining("|")) ;
 System.out.println ("\nchaine des mots longs : "+mots_longs) ;
```

```
 Map <String, List<String>> map = Stream.of(mots)
         .filter(xx -> xx.length()>4)
         .collect(Collectors.groupingBy(xx ->(xx.substring(0,1)))) ;
 System.out.println ("MAP: "+map) ;
 }
}
class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
 public int getX() { return x ; }
 public int getY() { return y ; }
 private int x, y ;
}
```

```
Ensemble : [1, 3] [2, 4] [1, 3] [3, 8] [3, 8]
Liste : [2, 4] [15, 30] [-3, -6] [2, 4] [-5, -10] [23, 46] [-8, -16] [12, 24]
chaine des mots longs : bonjour|hello|buongiorno|hi|chao|bom dia|guten tag
MAP: {b=[bonjour, buongiorno, bom dia], g=[guten tag], h=[hello]}
```

Annexe A

Les constantes et fonctions mathématiques

Elles sont fournies par la classe *Math*. Les angles sont toujours exprimés en radians.

Constante (double)	Valeur
E	2.718281828459045
PI	3.141592653589793

Fonction	Rôle	En-têtes
abs	Valeur absolue	double abs (double a) float abs (float a) int abs (int a) long abs (long a)
acos	Arc cosinus (angle dans l'intervalle [- 1, 1])	double acos (double a)
asin	Arc sinus (angle dans l'intervalle [-1, 1])	double asin (double a)
atan	Arc tangente (angle dans l'intervalle [-pi/2, pi/2])	double atan (double a)
atan2	Arc tangente (a/b) (angle dans l'intervalle [-pi/2, pi/2])	double atan2 (double a, double b)
ceil	Arrondi à l'entier supérieur	double ceil (double a)
cos	Cosinus	double cos (double a)
exp	Exponentielle	double exp (double a)

floor	Arrondi à l'entier inférieur	double floor (double a)
IEEEremainder	Reste de la division de x par y	double IEEEremainder (double x, double y)
log	Logarithme naturel (népérien)	double log (double a)
max	Maximum de deux valeurs	double max (double a, double b) float max (float a, float b) int max (int a, int b) long max (long a, long b)
min	Minimum de deux valeurs	double min (double a, double b) float min (float a, float b) int min (int a, int b) long min (long a, long b)
pow	Puissance (a^b)	double pow (double a, double b)
random	Nombre aléatoire dans l'intervalle [0, 1[	double random ()
rint	Arrondi à l'entier le plus proche	double rint (double a)
round	Arrondi à l'entier le plus proche	long round (double a) int round (float a)
sin	Sinus	double sin (double a)
sqrt	Racine carrée	double sqrt (double a)
tan	Tangente	double tan (double a)
toDegrees	Conversion de radians en degrés	double toDegrees (double aRad)
toRadians	Conversion de degrés en radians	double toRadians (double aDeg)

Annexe B

Les composants graphiques et leurs méthodes

Nous présentons ici les principales classes et méthodes des paquetages *java.awt* et *javax.swing*, en particulier celles qui sont utilisées dans les exercices de cet ouvrage. On notera que :

- lorsqu'une méthode est mentionnée dans une classe, elle n'est pas rappelée dans les classes dérivées ;
- lorsqu'une classe se révèle inutilisée en pratique (exemple *Window, Frame, Dialog*), ses méthodes n'ont été mentionnées que dans ses classes dérivées ; par exemple, la méthode *setTitle* est définie dans la classe *Frame* mais elle n'est indiquée que dans la classe *JFrame*.

Nous vous fournissons d'abord l'arborescence des classes concernées, avant d'en décrire les différentes méthodes, classe par classe (pour chacune, nous rappelons la liste de ses ancêtres).

1 Les classes de composants

Les classes précédées d'un astérisque (*) sont abstraites.

*Component
  *Container
    Panel
      Applet
        JApplet
    Window
      JWindow
      Frame
        JFrame
      Dialog
        JDialog
    JComponent
      JPanel
      AbstractButton
        JButton
        JToggleButton
          JCheckBox
          JRadioButton
        JMenuItem
          JCheckBoxMenuItem
          JRadioButtonMenuItem
          JMenu
      JLabel
      JTextComponent
      JTextField
        JList
      JcomboBox

JMenuBar
JPopupMenu
JScrollPane
JToolBar

2 Les méthodes

Component

	Component ()
void	**add** (PopupMenu menuSurgissant)
void	**addFocusListener** (FocusListener écouteur)
void	**addKeyListener** (KeyListener écouteur)
void	**addMouseListener** (MouseListener écouteur)
void	**addMouseMotionListener** (MouseMotionListener écouteur)
Color	**getBackground** ()
Rectangle	**getBounds** ()
Font	**getFont** ()
FontMetrics	**getFontMetrics** (Font fonte)
Color	**getForeground** ()
Graphics	**getGraphics** ()
int	**getHeight** ()
Dimension	**getSize** ()
Toolkit	**getToolkit** ()
int	**getX** ()
int	**getY** ()
int	**getWidth** ()
boolean	**hasFocus** ()
boolean	**imageUpdate** (Image image, int flags, int x, int y, int largeur, int hauteur)
void	**invalidate** ()
boolean	**isEnabled** ()
boolean	**isFocusTraversable** ()
boolean	**isVisible** ()
void	**paint** (Graphics contexteGraphique)
void	**setBackground** (color couleurFond)
void	**setBounds** (Rectangle r)
void	**setBounds** (int x, int y, int largeur, int hauteur)
void	**setCursor** (Cursor curseurSouris)
void	**setEnabled** (boolean activé)

void	**setFont** (Font fonte)
void	**setForeground** (Color couleurAvantPlan)
void	**setSize** (Dimension dim)
void	**setSize** (int largeur, int hauteur)
void	**setVisible** (boolean visible)
void	**update** (Graphics contexteGraphique)
void	**validate** ()

Container (*Component*)

	Container ()
Component	**add** (Component composant)
void	**add** (Component composant, Object contraintes)
Component	**add** (Component composant, int rang)
Component	**add** (Component composant, Object contraintes, int rang)
void	**setLayout** (LayoutManager gestionnaireMiseEnForme
void	**remove** (int rang)
void	**remove** (Component composant)
void	**removeAll** ()

Applet (*Panel -Component - Container*)

	applet ()
void	**destroy** ()
URL	**getCodeBase** ()
Image	**getImage** (URL adresseURL)
Image	**getImage** (URL adresseURL, String nomFichier)
String	**getParameter** (String nomParamètre)
void	**init** ()
void	**resize** (Dimension dim)
void	**resize** (int largeur, int hauteur)
void	**start** ()
void	**stop** ()

Applet (*Applet -Panel - Component - Container*)

	JApplet ()
Container	**getContentPane** ()
void	**setJMenuBar** (JMenuBar barreMenus)
void	**setLayout** (LayoutManager gestionnaireMiseEnForme)

Frame (*Frame -Window - Component - Container*)

	JFrame ()
	JFrame (String titre)
Container	**getContentPane** ()
Toolkit	**getToolkit** ()
void	**setContentPane** (Container contenu)
void	**setDefaultCloseOperation** (int operationSurFermeture)
void	**setJMenuBar** (JMenuBar barreMenus)
void	**setLayout** (Layout gestionnaireMiseEnForme)
void	**setTitle** (String titre) *// héritée de Frame*
void	**update** (Graphics contexteGraphique)

Dialog (*Dialog - Window - Container*)

	JDialog (Dialog propriétaire, boolean modale)
	JDialog (Frame propriétaire, boolean modale)
	JDialog (Dialog propriétaire, String titre, boolean modale)
	JDialog (Frame propriétaire, String titre, boolean modale)
void	**dispose** ()
Container	**getContentPane** ()
void	**setDefaultCloseOperation** (int operationSurFermeture)
void	**setLayout** (LayoutManager gestionnaireMiseEnForme)
void	**setJMenuBar** (JMenuBar barreMenus)
void	**setTitle** (String titre) *// héritée de Dialog*
void	**show** ()
void	**update** (Graphics contexteGraphique)

Component (*Container - Component*)

	JComponent ()
Graphics	**getGraphics** ()
Dimension	**getMaximumSize** ()
Dimension	**getMinimumSize** ()
Dimension	**getPreferredSize** ()
void	**paintBorder** (Graphics contexteGraphique)
void	**paintChildren** (Graphics contexteGraphique)
void	**paintComponent** (Graphics contexteGraphique)
void	**revalidate** ()
void	**setBorder** (Border bordure)

void	**setMaximumSize** (Dimension dimensions)
void	**setMinimumSize** (Dimension dimensions)
void	**setPreferredSize** (Dimension dimensions)
void	**setToolTipText** (String texteBulleDAide)

Panel (*Jcomponent - Container - Component*)

JPanel ()

JPanel (LayoutManager gestionnaireMiseEnForme)

AbstractButton (*Jcomponent - Container - Component*)

	AbstractButton ()
void	**addActionListener** (ActionListener écouteur)
void	**addItemListener** (ItemListener écouteur)
String	**getActionCommand**()
String	**getText**()
boolean	**isSelected**()
void	**setActionCommand** (String chaineDeCommande)
void	**setEnabled** (boolean activé)
void	**setMnemonic** (char caractèreMnémonique)
void	**setSelected** (boolean sélectionné)
void	**setText** (String libellé)

Button (*AbstractButton - JComponent - Container - Component*)

JButton ()

JButton (String libellé)

CheckBox (*JToggleButton - AbstractButton - JComponent - Container - Component*)

JCheckBox ()

JCheckBox (String libellé)

JCheckBox (String libellé, boolean sélectionné)

RadioButton (*JToggleButton - AbstractButton - JComponent - Container - Component*)

JRadioButton (String libellé)

JRadioButton (String libellé, boolean sélectionné)

Label (*JComponent - Container - Component*)

	JLabel (String texte)
void	**setText** (String libellé)

TextField (*JTextComponent - JComponent - Container - Component*)

	JTextField ()	
	JTextField (int nombreColonnes)	
	JTextField (String texteInitial)	
	JTextField (String texteInitial, int nombreColonnes)	
Document	**getDocument** ()	*// héritée de* JTextComponent
String	**getText** ()	*// héritée de* JTextComponent
void	**setColumns** (int nombreCaractères)	
void	**setEditable** (boolean éditable)	*// héritée de* JTextComponent
void	**setText** (String texte)	*// héritée de* JTextComponent

List (*JComponent - Container - Component*)

	JList ()
	JList (Object[] données)
void	**addListSelectionListener** (ListSelectionListener écouteur)
void	**setSelectedIndex** (int rang)
int	**getSelectedIndex** ()
int[ ]	**getSelectedIndices** ()
Object	**getSelectedValue** ()
Object [ ]	**getSelectedValues** ()
boolean	**getValueIsAdjusting** ()
void	**setSelectedIndex** (int rang)
void	**setSelectedIndices** (int [ ] rangs)
void	**setSelectionMode** (int modeDeSelection)
void	**setVisibleRowCount** (int nombreValeurs)

ComboBox (*JComponent - Container - Component*)

	JComboBox ()
	JComboBox (Object[ ] données)
void	**addItem** (Object nouvelleValeur)

int **getSelectedIndex** ()
Object **getSelectedItem** ()
void **insertItemAt** (Object nouvelleValeur, int rang)
void **removeItem** (Object valeurASupprimer)
void **removeItemAt** (int rang)
void **removeAllItems** ()
void **setEditable** (boolean éditable) *// héritée de* JTextComponent
void **setSelectedIndex** (int rang)

MenuBar (*JComponent - Container - Component*)

JMenuBar ()
JMenu **add** (JMenu menu)
JMenu **getMenu** (int rang)

Menu *(JMenuItem - AbstractButton - JComponent - Container - Component*)

JMenu ()
JMenu (String nomMenu)
JMenuItem **add** (Action action)
JMenuItem **add** (JMenuItem option)
void **addMenuListener** (MenuListener écouteur)
void **addSeparator** ()
KeyStroke **getAccelerator** ()
void **insert** (Action action, int rang)
void **insert** (JMenuItem option, int rang)
void **insertSeparator** (int rang)
boolean **isSelected** ()
void **remove** (int rang)
void **remove** (JMenuItem option)
void **removeAll** ()
void **setAccelerator** (KeyStroke combinaisonTouches)
void **setEnabled** (boolean activé)
void **setSelected** (boolean sélectionné)

PopupMenu (*JComponent - Container - Component*)

JPopupMenu ()
JPopupMenu (String nom)
JMenuItem **add** (Action action)

JMenuItem	**add** (JMenuItem option)
void	**addPopupMenuListener** (PopupMenuListener écouteur)
void	**addSeparator** ()
void	**insert** (Action action, int rang)
void	**insert** (Component composant, int rang)
void	**remove** (Component composant)
void	**setVisible** (boolean visible)
void	**show** (Component composant, int x, int y)

MenuItem (*AbstractButton - JComponent - Container - Component*)

	JMenuItem ()
	JMenuItem (String nomOption)
	JMenuItem (Icon icône)
	JMenuItem (String nomOption, Icon icône)
	JMenuItem (String nomOption, int caractèreMnémonique)
void	**setAccelerator** (KeyStroke combinaisonTouches)
keyStroke	**getAccelerator** ()

CheckBoxMenuItem *(JMenuItem - AbstractButton - JComponent - Container - Component*)

JChekBoxMenuItem ()

JChekBoxMenuItem (String nomOption)

JChekBoxMenuItem (Icone icône)

JChekBoxMenuItem (String nomOption, Icon icône)

JChekBoxMenuItem (String nomOption, boolean activé)

JChekBoxMenuItem (String nomOption, Icon icône, boolean activé)

RadioButtonMenuItem *(JMenuItem - AbstractButton - JComponent - Container - Component*)

JRadioButtonMenuItem ()

JRadioButtonMenuItem (String nomOption)

JRadioButtonMenuItem (Icone icône)

JRadioButtonMenuItem (String nomOption, Icon icône)

JRadioButtonMenuItem (String nomOption, boolean activé)

JRadioButtonMenuItem (String nomOption, Icon icône, boolean activé)

ScrollPane

JScrollPane ()

JScrollPane (Component)

ToolBar

	JToolBar ()
	JToolBar (int orientation)
JButton	**add** (Action action)
void	**addSeparator** ()
void	**addSeparator** (Dimension dimensions)
boolean	**isFloatable** ()
void	**remove** (Component composant)
void	**setFloatable** (boolean flottante)

Annexe C

Les événements et les écouteurs

Nous vous fournissons tout d'abord deux tableaux de synthèse, le premier pour les événements de bas niveau, le second pour les événements sémantiques. Ils fournissent pour chacune des principales interfaces écouteurs correspondantes :

- le nom de l'interface écouteur et le nom de la classe adaptateur (si elle existe),
- les noms des méthodes de l'interface,
- le type de l'événement correspondant,
- les noms des principales méthodes de l'événement,
- les composants concernés.

Vous trouverez ensuite les en-têtes complètes des méthodes des classes événement.

3 Les événements de bas niveau

Ecouteur (*adaptateur*)	Méthode écouteur	Type événement	Méthodes événement	Composants concernés
MouseListener (*MouseAdapter*)	mouseClicked mousePressed mouseReleased mouseEntered mouseExited	MouseEvent	getClickCount getComponent getModifiers getSource getX getY getPoint isAltDown isAltGraphDown isControlDown isMetaDown isPopupTrigger isShiftDown	Component
MouseMotionListener (*MouseMotionAdapter*)	mouseDragged mouseMoved			
KeyListener (*KeyAdapter*)	keyPressed keyReleased keyTyped	keyEvent	getComponent getSource getKeyChar getKeyCode getKeyModifiersText getKeyText getModifiers isAltDown isAltGraphDown isControlDown isShiftDown isMetaDown isActionKey	Component
FocusListener (*FocusAdapter*)	focusGained focusLost	FocusEvent	getComponent getSource isTemporary	Component
WindowListener (*WindowAdapter*)	windowOpened windowClosing windowClosed windowActivated windowDeactivated windowIconified windowDeiconified	WindowEvent	getComponent getSource getWindow	Window

4 Les événements sémantiques

Dans la dernière colonne de ce tableau, les termes génériques *Boutons* et *Menus* désignent les classes suivantes

- Boutons : *JButton*, *JCheckBox*, *JRadioButton*,
- Menus : *JMenu*, *JMenuItem*, *JCheckBoxMenuItem*, *JRadioButtonMenuItem*.

Ecouteur (*adaptateur*)	**Méthode écouteur**	**Type événement**	**Méthodes événement**	**Composants concernés**
ActionListener	actionPerformed	ActionEvent	getSource getActionCommand getModifiers	*Boutons* *Menus* JTextField
ItemListener	itemStateChanged	ItemEvent	getSource getItem getStateChange	*Boutons* *Menus* JList JComboBox
ListSelection-Listener	valueChanged	ListSelectionEvent	getSource getValueIsAdjusting	JList
Document-Listener	changeUpdate insertUpdate removeUpdate	DocumentEvent	getDocument	Document
MenuListener	menuCanceled menuSelected menuDeselected	MenuEvent	getSource	JMenu
PopupMenu-Listener	popupMenuCanceled popupMenuWillBecomeVisible popupMenuWillBecomeInvisible	PopupMenuEvent	getSource	JPopupMenu

5 Les méthodes des événements

MouseEvent

int	**getClickCount** ()
Component	**getComponent** ()
int	**getModifiers** ()
Object	**getSource** ()
int	**getX** ()
int	**getY** ()
Point	**getPoint** ()
boolean	**isAltDown** ()
boolean	**isAltGraphDown** ()
boolean	**isControlDown** ()
boolean	**isMetaDown** ()
boolean	**isPopupTrigger** ()
boolean	**isShiftDown** ()

KeyEvent

Component	**getComponent** ()
Object	**getSource** ()
char	**getKeyChar** ()
int	**getKeyCode** ()
String	**getKeyText** (int codeToucheVirtuelle)
int	**getModifiers** ()
boolean	**isAltDown** ()
boolean	**isAltGraphDown** ()
boolean	**isControlDown** ()
boolean	**isMetaDown** ()
boolean	**isShiftDown** ()

FocusEvent

Component	**getComponent** ()
Object	**getSource** ()
boolean	**isTemporary** ()

WindowEvent

Component	**getComponent** ()
Object	**getSource** ()
Window	**getWindow** ()

ActionEvent

Object	**getSource** ()
String	**getActionCommand** ()
int	**getModifiers** ()

ItemEvent

Object	**getSource** ()
Object	**getItem** ()
int	**getStateChanged** ()

ListSelectionEvent

Object	**getSource** ()
boolean	**getValueIsAdjusting** ()

DocumentEvent

Document	**getDocument** ()

MenuEvent

Object	**getSource** ()

PopuMenuEvent

Object	**getSource** ()

Annexe D

La classe Clavier

Voici la liste de la classe *Clavier* présente sur le site Web d'accompagnement et que vous pouvez utiliser pour la solution à certains des exercices de cet ouvrage.

Elle fournit des méthodes permettant de lire sur une ligne une information de l'un des types *int*, *float*, *double* ou *String*. La méthode de lecture d'une chaîne est utilisée par les autres pour lire la ligne.

```
// classe fournissant des fonctions de lecture au clavier
import java.io.* ;
public class Clavier
{ public static String lireString ()   // lecture d'une chaine
 { String ligne_lue = null ;
  try
  { InputStreamReader lecteur = new InputStreamReader (System.in) ;
   BufferedReader entree = new BufferedReader (lecteur) ;
   ligne_lue = entree.readLine() ;
  }
  catch (IOException err)
  { System.exit(0) ;
  }
  return ligne_lue ;
 }
 public static float lireFloat ()   // lecture d'un float
 { float x=0 ;   // valeur a lire
  try
  { String ligne_lue = lireString() ;
   x = Float.parseFloat(ligne_lue) ;
```

```
 }
 catch (NumberFormatException err)
 { System.out.println ("*** Erreur de donnee ***") ;
  System.exit(0) ;
 }
 return x ;
}
public static double lireDouble ()   // lecture d'un double
{ double x=0 ;   // valeur a lire
 try
 { String ligne_lue = lireString() ;
  x = Double.parseDouble(ligne_lue) ;
 }
 catch (NumberFormatException err)
 { System.out.println ("*** Erreur de donnee ***") ;
  System.exit(0) ;
 }
 return x ;
}
public static int lireInt ()         // lecture d'un int
{ int n=0 ;   // valeur a lire
 try
 { String ligne_lue = lireString() ;
  n = Integer.parseInt(ligne_lue) ;
  }
 catch (NumberFormatException err)
 { System.out.println ("*** Erreur de donnee ***") ;
  System.exit(0) ;
 }
 return n ;
}

 // programme de test de la classe Clavier
public static void main (String[] args)
{ System.out.println ("donnez un flottant") ;
 float x ;
 x = Clavier.lireFloat() ;
 System.out.println ("merci pour " + x) ;
 System.out.println ("donnez un entier") ;
```

```
   int n ;
   n = Clavier.lireInt() ;
   System.out.println ("merci pour " + n) ;
  }
 }
```

Notez que, en cas d'exception de type *IOException* (rare !), on se contente d'interrompre le programme. Si nous n'avions pas traité cette exception, nous aurions dû la déclarer dans une clause *throws*, ce qui aurait obligé l'utilisateur de la classe *Clavier* à la prendre en charge.

La lecture des informations de type entier ou flottant utilise la méthode *Clavier.lireString*, ainsi que les méthodes de conversion de chaînes *Integer.parseInt*, *Float.parseFloat* et *Double.parseDouble*. Nous devons traiter l'exception *NumberFormatException* qu'elles sont susceptibles de générer. Ici, nous affichons un message et nous interrompons le programme.
