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.

Solution 1

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 ****") ;
  }
}
```

Remarque

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*.

Remarque

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 !).

Solution 2

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 ****") ;
  }
}
```

Remarque

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.

Solution 1

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 ****
```

Solution 2

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.

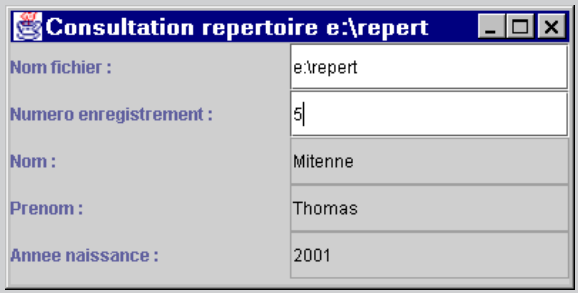
```
import java.io.* ;

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*.

Solution

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) ;
  }
}
```

Remarque

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 123456789012345678901234567890123456789012345678901234567890
   5 12345678901234567890123456789012345678901234567890123456789
   6 123456789012345678901234567890123456789012345678901234567890
     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 ***
```

Solution

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
```

Solution

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 recourrir 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 ") ;
      }
    }
  }
```

Remarque

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.

Solution

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.

Solution

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] ;
  }
}
```

Solution

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 ;
}
```

Solution

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.

Solution

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 recourrant 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.

Solution

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*.

Solution

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>
```

Solution

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.

Solution

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
```

Remarque

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 ) ;
  }
}
```

Solution

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
```

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)

Fonction	Rôle	En-têtes
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** ()

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

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

JFrame (*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)

JDialog (*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)

JComponent (*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)

JPanel (*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)

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

	JButton ()
	JButton (String libellé)

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

	JCheckBox ()
	JCheckBox (String libellé)
	JCheckBox (String libellé, boolean sélectionné)

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

	JRadioButton (String libellé)
	JRadioButton (String libellé, boolean sélectionné)

JLabel (*JComponent - Container - Component*)

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

JTextField (*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

JList (*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)

JComboBox (*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)

JMenuBar (*JComponent - Container - Component*)

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

JMenu *(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)

JPopupMenu (*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)

JMenuItem (*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** ()

JCheckBoxMenuItem *(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)

JRadioButtonMenuItem *(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)

JScrollPane

JScrollPane ()
JScrollPane (Component)

JToolBar

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) ;
  }
}
```

Remarque

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.