5

Les flots

1. Généralités
2. Flots d'octets, flots de caractères
3. Les filtres
4. Comment lire un entier
5. Manipulation de fichiers
6. Flots d'objets ou sérialisation

MCours.com

Généralités

Un *flot* (*stream*) est un canal de communication dans lequel on peut lire ou écrire. On accède aux données séquentiellement.

Les flots prennent des données, les transforment éventuellement, et sortent les données transformées.

Pipeline ou filtrage

Les données d'un flot d'entrées sont prises dans une *source*, comme l'entrée standard ou un fichier, ou une chaîne ou un tableau de caractères, ou dans la sortie d'un autre flot d'entrée.

De même, les données d'un flot de sortie sont mises dans un *puit*, comme la sortie standard ou un fichier, ou sont transmises comme entrées dans un autre flot de sortie.

En Java, les flots manipulent soit des octets, soit des caractères. Certains manipulent des données typées.

Les classes sont toutes dans le paquetage `java.io`.
Les classes de base sont

```
File
RandomAccessFile

InputStream
OutputStream

Reader
Writer

StreamTokenizer
```

Les `Stream`, `Reader` et `Writer` sont abstraites.

Les `Stream` manipulent des octets, les `Reader` et `Writer` manipulent des caractères.

Hiérarchie des classes

Fichiers

```
File
FileDescriptor
RandomAccessFile
```

Streams

```
InputStream
   ByteArrayInputStream
   FileInputStream
   FilterInputStream
      BufferedInputStream
      DataInputStream
      LineNumberInputStream
      PushbackInputStream
   ObjectInputStream
   PipedInputStream
   SequenceInputStream

OutputStream
   ByteArrayOutputStream
   FileOutputStream
   FilterOutputStream
      BufferedOutputStream
      DataOutputStream
      PrintStream
   ObjectOutputStream
   PipedOutputStream
```

Reader

```
Reader
   BufferedReader
      LineNumberReader
   CharArrayReader
   FilterReader
      PushbackReader
   InputStreamReader
```



```
      FileReader
   PipedReader
   StringReader
```

Writer

```
Writer
   BufferedWriter
   CharArrayWriter
   FilterWriter
   OutputStreamWriter
      FileWriter
   PipedWriter
   PrintWriter
   StringWriter
```

Flots d'octets et de caractères

Les flots d'octets en lecture

Objet d'une classe dérivant de `InputStream`.
`System.in` est un flot d'octets en lecture.
Méthodes pour lire *à partir* du flot :

- `int read()` : lit un octet dans le flot, le renvoie comme octet de poids faible d'un `int` ou renvoie -1 si la fin du flot est atteinte;
- `int read(byte[] b)` : lit au plus `b.length` octets dans le flot et les met dans `b`;
- `int read(byte[] b, int off, int len)` : lit au plus `len` octets dans le flot et les met dans `b` à partir de `off`;
- `int available()` : retourne le nombre d'octets dans le flot;
- `void close()` : ferme le flot.

Les flots d'octets en écriture

Objet d'une classe dérivant de `OutputStream`.
`System.out` est de la classe `PrintStream`, qui dérive de `FilterOutputStream` qui dérive de `OutputStream`.
Méthodes pour écrire *dans* le flot:

- `void write(int b)` : écrit dans le flot l'octet de poids faible de `b`;
- `void write(byte[] b)` : écrit dans le flot tout le tableau;

- `int read(byte[] b, int off, int len)` : écrit dans le flot `len` octets à partir de l`off`;
- `void close()` : ferme le flot.

Lire un octet

```
import java.io.*;

public class Lire {
  public static void main(String[] args){
    try {
      int i = System.in.read();
      System.out.println(i);
    } catch (IOException e) {};
  }
}
```

On obtient :

```
$$ java Lire
a
97
```

Lire des octets

```
public class Lire {
  static int EOF = (int) '\n';
  public static void main(String[] args) throws IOException {
    int i;
    while ((i = System.in.read()) != EOF)
      System.out.print(i + " ");
    System.out.println("\nFin");
  }
}
```

On obtient :

```
$$ java Lire
a é
97 32 233 10
Fin
```

Les flots de caractères en lecture

Objet d'une classe dérivant de `Reader`.
Méthodes pour lire *à partir* du flot :

- `int read()` : lit un caractère dans le flot, le renvoie comme octet de poids faible d'un `int` ou renvoie -1 si la fin du flot est atteinte;
- `int read(char[] b)` : lit au plus `b.length` caractères dans le flot et les met dans `b`;
- `int read(char[] b, int off, int len)` : lit au plus `len` caractères dans le flot et les met dans `b` à partir de `off`;
- `int available()` : retourne le nombre d'octets dans le flot;
- `void close()` : ferme le flot.

Les flots d'octets en écriture

Objet d'une classe dérivant de `Writer`.
Les méthodes sont analogues à celles des flots d'octets.

Les filtres

Un *filtre* est un flot qui *enveloppe* un autre flot.

Les données sont en fait lues (ou écrites) dans le flot enveloppé après un traitement (codage, bufferisation, etc). Le flot enveloppé est passé en argument du constructeur du flot enveloppant.

Les filtres héritent des classes abstraites :

- `FilterInputStream` (ou `FilterReader`);
- `FilterOutputStream` (ou `FilterWriter`).

Filtres prédéfinis :

- `DataInputStream`, `DataOutputStream` : les méthodes sont `write`*Type*`()`, `read`*Type*`()`, où *Type* est `Int, Char, Double, ...`;
- `BufferedInputStream` : permet de bufériser un flot;
- `PushBackInputStream`: permet de replacer des données lues dans le flot avec la méthode `unread()`;
- `PrintStream` : `System.out` est de la classe `PrintStream`.
- `InputStreamReader` : transforme un `Stream` en `Reader`;
- `BufferedReader` : buférise un flot de caractères;
- `LineNumberReader` : pour une lecture de caractères ligne par ligne;

Lire des entiers

Un entier avec `LineNumberReader`

```
class Lire {
  public static int lireInt() throws IOException{
    InputStreamReader in = new InputStreamReader(System.in);
    LineNumberReader data = new LineNumberReader(in);
    String s = data.readLine();
    return Integer.parseInt(s);
  }
}

class LireUnEntierLineB{
  public static void main(String[] args) throws IOException{
    int i = Lire.lireInt();
    System.out.println(i);
  }
}
```

La classe `LineNumberReader` dérive de la classe `BufferedReader`.

La méthode `String readLine()` de la classe `BufferedReader` retourne la ligne suivante.

Lire une suite d'entiers avec `StreamTokenizer`

Un **StreamTokenizer** prend en argument un flot (reader) et le fractionne en “token” (lexèmes). Les attributs sont

- `nval` contient la valeur si le lexème courant est un nombre (double)
- `sval` contient la valeur si le lexème courant est un mot.
- `TT_EOF, TT_EOL, TT_NUMBER, TT_WORD` valeurs de l'attribut `ttype`. Si un token n'est ni un mot, ni un nombre, contient l'entier représentant le caractère.

```
class LireMulti {
  public static void lire() throws IOException{
    StreamTokenizer in;
    InputStreamReader w = new InputStreamReader(System.in);
    in = new StreamTokenizer(new BufferedReader(w));
    in.quoteChar('/');
    in.wordChars('@','@');
    do {
      in.nextToken();
      if (in.ttype == (int) '/')
        System.out.println(in.sval);
      if (in.ttype == StreamTokenizer.TT_NUMBER)
         System.out.println((int) in.nval); // normalement double
      if (in.ttype == StreamTokenizer.TT_WORD)
        System.out.println(in.sval);
    } while (in.ttype != StreamTokenizer.TT_EOF);
  }
}

class TestLireMulti{
  public static void main(String[] args) throws IOException{
    LireMulti.lire();
  }
}
```



```
$$ cat Paul
0 @I1@ INDI
1 NAME Paul /Le Guen/
0 TRLR

$$ java TestLireMulti < Paul
0
@I1@
INDI
1
NAME
Paul
Le Guen
0
TRLR
```

Manipulation de fichiers

Les *sources* et *puits* des stream et reader sont

- les entrées et sorties standard (`printf`)
- les String (`sprintf`)
- les fichiers (`fprintf`)

Pour les `String`, il y a les `StringReader` et `StringWriter`. Pour les fichiers, il y a les stream et reader correspondants.

- La classe `java.io.File` permet de manipuler le système de fichiers;
- Les classes `FileInputStream` (et `FileOutputStream`) définissent des flots de lecture et d'écriture de fichiers d'octets, et les classes `FileReader` (et `FileWriter`) les flots de lecture et d'écriture de fichiers de caractères.

La classe `File` décrit une représentation d'un fichier.

```
import java.io.*;

public class InfoFichier{
  public static void main(String[] args) throws Exception{
    info(args[0]);
  }

  public static void info(String nom)
      throws FileNotFoundException{
    File f = new File(nom);
    if (!f.exists())
      throw new FileNotFoundException();
    System.out.println(f.getName());
    System.out.println(f.isDirectory());
    System.out.println(f.canRead());
    System.out.println(f.canWrite());
    System.out.println(f.length());
  }
}
```

On obtient :

```
monge : > ls -l toto
-rw-r--r-- 1 beal institut 488 Sep 21 12:13 toto
monge : > java InfoFichier toto
toto
false
true
true
488
```


Lecture d'un fichier

Un lecteur est le plus souvent défini par

`FileReader f = new FileReader(nom);`

où `nom` est le nom du fichier. La lecture se fait par les méthodes de la classe `InputStreamReader`.
Lecture par bloc.

```
FileInputStream in = new FileInputStream(nomIn);
FileOutputStream out = new FileInputStream(nomOut);
int readLength;
byte[] block = new byte[8192];
while ((readLength = in.read(block)) != -1)
  out.write(block, 0, readLength);
```

Lecture d'un fichier de texte, ligne par ligne.

```
public String readFile(String f) throws IOException {
  FileReader fileIn = new FileReader(nom);
  BufferedReader in = new BufferedReader(fileIn);

  StringBuffer s = new StringBuffer();
  String line;
  while ((line = in.readLine()) != null)
    s.append(line + "\n");
  fileIn.close();
  return s.toString();
}
```


Les flots d'objets ou sérialisation

- Un *flot d'objets* permet d'écrire ou de lire des objets Java dans un flot.
- On utilise pour cela les filtres `ObjectInputStream` et `ObjectOutputStream`. Ce service est appelé *sérialisation*.
- Les applications qui échangent des objets via le réseau utilisent la sérialisation.
- Pour sérialiser un objet, on utilise la méthode d'un flot implémentant l'interface `ObjectOutput` : `void writeObject(Object o)`.
- Pour désérialiser un objet, on utilise la méthode d'un flot implémentant l'interface `ObjectInput` : `Object readObject()`.
- Pour qu'un objet puisse être inséré dans un flot, sa classe doit implémenter l'interface `Serializable`. Cette interface ne contient pas de méthode.

La première fois qu'un objet est sauvé, tous les objets qui peuvent être atteints à partir de cet objet sont aussi sauvés. En plus de l'objet, le flot sauvegarde un objet appelé *handle* qui représente une référence locale de l'objet dans le flot. Une nouvelle sauvegarde entraîne la sauvegarde du handle à la place de l'objet.

Exemple de sauvegarde

```
import java.io.*;

public class Point implements Serializable {
  private int x, y;
  public Point(int xx,int yy){ x = xx; y = yy; }
  public String toString(){ return "(" + x + "," + y + ")"; }

  public void sauvePoint(String nom) throws Exception {
    File f = new File(nom);
    ObjectOutputStream out;
    out = new ObjectOutputStream(new FileOutputStream(f));
    out.writeObject(this);
    out.close();
    // fin de la partie sauvegarde

    ObjectInputStream in;
    in = new ObjectInputStream(new FileInputStream(f));
    Point oBis = (Point) in.readObject();
    in.close();
    System.out.println(this);
    System.out.println(oBis);
    System.out.println(this.equals(oBis));
  }

  public static void main(String[] args) throws Exception{
    Point o = new Point(1,2);
    o.sauvePoint(args[0]);
  }
}
```

On obtient :

```
monge :> java Point toto
(1,2)
(1,2)
false
```


Redéfinir l'objet de sauvegarde

Au moment de la sauvegarde, il est possible de remplacer un objet par un autre.

- On définit pour cela la méthode `Object writeReplace()` dans la classe de *l'objet à remplacer*.
- Au moment de la désérialisation, on utilise la méthode `Object readResolve()` de la classe de *l'objet remplacé* pour retourner un objet compatible avec l'original.

Dans l'exemple suivant, une liste d'entiers est remplacée, au moment de son écriture dans un fichier, par un objet de la classe `ListeString` qui contient la liste des entiers sous forme de chaîne de caractères.


```
class Serial {
  public static void main(String[] args) throws Exception{
    Liste l  = new Liste(1, new Liste(2,null));
    l.ecrireListe(args[0]);
    l.lireListe(args[0]);
  }
}

class Liste implements Serializable{
  int val;
  Liste next;
  public Liste(int val, Liste next){
    this.val = val ;
    this.next = next;
  }
  Object writeReplace() throws ObjectStreamException{
    Liste tmp;
    StringBuffer buffer = new StringBuffer();
    for (tmp = this; tmp != null; tmp = tmp.next)
      buffer.append(" " + tmp.val);
    return new ListeString(buffer.toString());
  }
  public String toString(){
    return "(" + val + "," + next + ")";
  }
  public void ecrireListe(String nom) throws Exception {
    ObjectOutputStream out;
    out = new ObjectOutputStream(new FileOutputStream(nom));
    out.writeObject(this);
    out.close();
  }
  public void lireListe(String nom) throws Exception {
    ObjectInputStream in;
    in = new ObjectInputStream(new FileInputStream(nom));
    Liste l = (Liste) in.readObject();
    System.out.println(l);
  }
}
```



```
class ListeString implements Serializable {
  String s;
  ListeString(String s) { this.s = s; }

  Object readResolve() throws ObjectStreamException {
    StringTokenizer st = new StringTokenizer(s);
    return resolve (st);
  }
  Liste resolve(StringTokenizer st) {
    if (st.hasMoreTokens()) {
      int val = Integer.parseInt(st.nextToken());
      return new Liste(val, resolve(st));
    }
    return null;
  }
}
```

On obtient :

```
monge :> java Serial toto
(1,(2,null))
monge :> file toto
toto: Java serialization data, version 5
```

6

Les composants AWT de Java

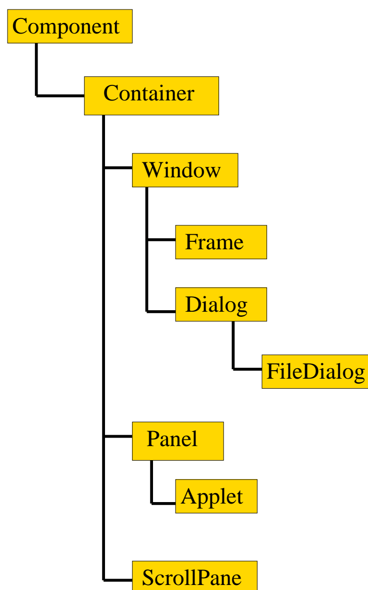
(Abstract Windowing Toolkit)

1. Composants et conteneurs
2. Événements
3. Dessins

Composants et conteneurs

- Un *composant* est un objet de base de l'interface utilisateur de Java. C'est un objet d'un classe dérivée de la classe `java.awt.Component` (exemple : fenêtres, boutons, zones de dessin, menus, barres de défilement ...);
- Un composant est en principe inséré dans un *conteneur*. C'est un objet de la classe `java.awt.Container` qui regroupe des composants.
- Un *gestionnaire de placement* (Layout Manayer) gère la géométrie des composants. Il indique comment sont disposés les composants dans un conteneur.

Différents types de conteneurs



Gestionnaires de placement

Les gestionnaires de placement sont des objets de classes implémentant l'interface `LayoutManager`. Les principales sont :

- `FlowLayout`;
- `GridLayout`;
- `BorderLayout`;
- `BoxLayout`.

Chaque conteneur a un gestionnaire de placement par défaut. On peut le modifier par la méthode `setLayout()` de la classe `Container` (Ex : `setLayout(new FlowLayout());`).
Lorsque des objets sont ajoutés dans un conteneur, la mise à jour peut se demander par la méthode `validate()` de la classe `Component`.

Événements

Les programmes à interfaces graphiques sont pilotés par des évènements. Un thread spéciale (`EventDispatchThread`) est créée par la machine virtuelle, en charge de la boucle de lecture et de distribution des évènements.

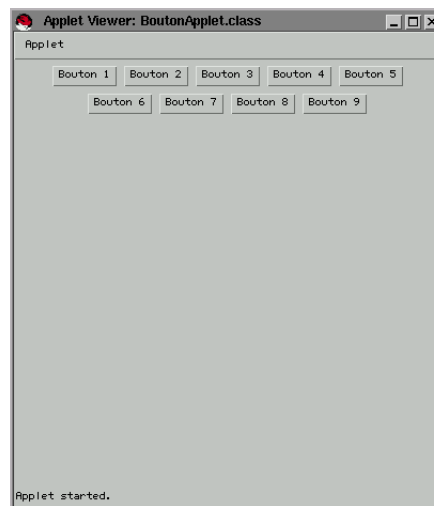
Un *événement* correspond à une action simple (enfoncement de la souris, pression d'une touche, ...) : on dit qu'il s'agit d'un *événement élémentaire*.

Un *événement sémantique* consiste en une séquence d'événements élémentaires (clic de souris, ...) synthétisés en un évènement unique.

Le modèle émetteur-auditeur

- Un événement est émis (*fired*) par un composant;
- Un événement est transmis aux *auditeurs* enregistrés.
- Les auditeurs exécutent des méthodes en fonction de l'événement reçu.
- Plusieurs auditeurs peuvent être enregistrés pour un événement.
- Il existent différentes classes d'auditeurs (*listeners*) qui ont leurs propres méthodes. Ces classes sont des interfaces à implémenter.

Créer des boutons en cliquant sur le premier




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

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    setLabel("Bouton "+ numero);
    numero++;
  }
}

public class BoutonFrame extends Frame {

  public BoutonFrame() {
    setTitle("Encore des boutons");
    Button b = new BoutonNumerote();
    add(b);
    b.addActionListener(new BoutonListener());
    setSize(200,200);
    setVisible(true);
  }

  public static void main(String[] args) {
    new BoutonFrame();
  }

  class BoutonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      add(new BoutonNumerote());
      validate();
    }
  }
}
```


Version applette

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    setLabel("Bouton "+ numero);
    numero++;
  }
}

public class BoutonApplet extends Applet {

  public void init() {
    BoutonNumerote b = new BoutonNumerote();
    add(b);
    b.addActionListener(new BoutonListener());
  }

  class BoutonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      add(new BoutonNumerote());
      validate();
    }
  }
}
```

Remarque : La classe `BoutonListener` est une *classe interne* (*inner class*) à `BoutonApplet`.

Version plus compacte

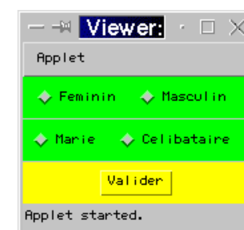
```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

class BoutonNumerote extends Button {
  static int numero = 1;

  BoutonNumerote() {
    super("Bouton "+ numero);
    numero++;
  }
}

public class BoutonApplet extends Applet
  implements ActionListener{

  public void init() {
    BoutonNumerote b = new BoutonNumerote();
    add(b);
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    add(new BoutonNumerote());
    validate();
  }
}
```

MCours.com

Compter les clics sur des boutons

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

class BoutonCompteur extends Button {
  int compteur = 0;
  String nom;

  BoutonCompteur(String s) {
    super(s);
    nom = s;
  }
}


public class Bouton2 extends Applet implements ActionListener{
  BoutonCompteur b1,b2;

  public void init() {
    setLayout(new GridLayout(1,2,1,1));
    b1 = new BoutonCompteur("Nombre de clics ici : ");
    b1.addActionListener(this);
    b2 = new BoutonCompteur("Nombre de clics la : ");
    b2.addActionListener(this);
    add(b1);
    add(b2);
  }

  public void actionPerformed(ActionEvent e) {
    BoutonCompteur b = (BoutonCompteur) e.getSource();
    b.compteur++;
    b.setLabel(b.nom+b.compteur);
  }
}
```


Cases à cocher



Sortie :

```
monge : >
Feminin
Celibataire
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Cases extends Applet implements ActionListener{
  Panel sexePanel,mariPanel,validationPanel;
  CheckboxGroup sexeGroup,mariGroup;
  public void init() {
    setLayout(new GridLayout(3,1,1,1));
    sexePanel = new Panel();
    sexePanel.setBackground(Color.green);
    sexeGroup = new CheckboxGroup();
    sexePanel.add( new Checkbox("Feminin",sexeGroup,false));
    sexePanel.add( new Checkbox("Masculin",sexeGroup,false));

    mariPanel = new Panel();
    mariPanel.setBackground(Color.green);
    mariGroup = new CheckboxGroup();
    mariPanel.add( new Checkbox("Marie",mariGroup,false));
    mariPanel.add( new Checkbox("Celibataire",mariGroup,false));

    validationPanel = new Panel();
    validationPanel.setBackground(Color.yellow);
    Button b = new Button("Valider");
    b.addActionListener(this);
    validationPanel.add(b);
    add(sexePanel);
    add(mariPanel);
    add(validationPanel);
    setSize(getPreferredSize());
  }
  public void actionPerformed(ActionEvent e) {
    Checkbox c = sexeGroup.getSelectedCheckbox();
    Checkbox d = mariGroup.getSelectedCheckbox();
    if (c != null && d !=null)
      System.out.println(c.getLabel() + "\n" + d.getLabel());
  }
}
```


Dessins

L'outil de dessin est le *contexte graphique*, un objet de la classe abstraite `Graphics`. Il encapsule des informations nécessaires concernant une zone de dessin comme :

- l'objet composant sur lequel on va dessiner;
- une translation d'origine;
- le rectangle de découpe;
- la couleur courante;
- la police de caractère courante;
- le mode opératoire logique de dessin (XOR ou Paint);
- éventuellement la couleur du XOR.

Le contexte graphique ne représente pas le dessin lui-même. On obtient un contexte graphique :

- *implicitement*, dans une méthode `paint()` ou `update()` : le contexte graphique construit est passé en paramètre;
- *explicitement*, en copiant un contexte graphique déjà existant;
- *explicitement*, en faisant un `getGraphics()` dans un composant ou une image.

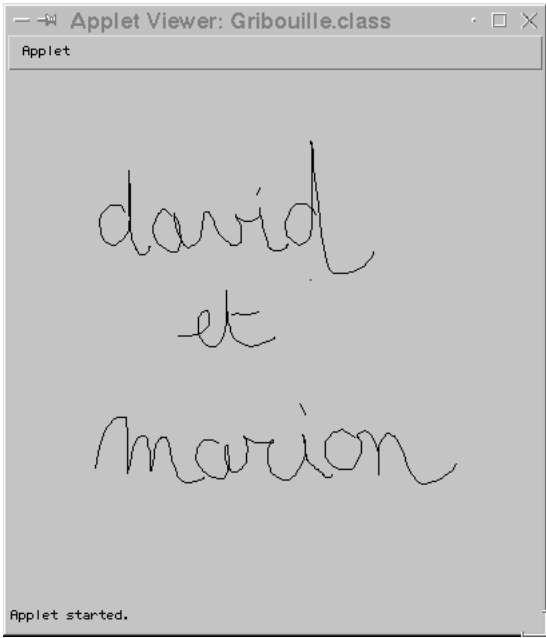
L'acquisition explicite traduit une mauvaise utilisation de `paint()` ou `update()`. Il faut ensuite libérer le contexte graphique explicitement par `dispose()`.

La triplette magique paint() repaint() update()

Ce sont des méthodes de la classe `Component`.

- `paint(Graphics gc)` : ne fait rien par défaut, en pratique contient des appels de méthodes de dessin de la classe `Graphics` sur `gc`;
- `update(Graphics gc)` : par défaut efface le dessin et appelle `paint()`;
- `repaint()` : appelle `update()` en lui fournissant un contexte graphique.

Exemple du gribouillage



```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Gribouille extends Applet{
  int x0,y0,x,y;

  public void init() {
    addMouseListener(new Appuyeur());
    addMouseMotionListener(new Dragueur());
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }

  class Appuyeur implements MouseListener {
    public void mousePressed(MouseEvent e) {
      x0 = e.getX(); y0 = e.getY();
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
  }

  class Dragueur implements MouseMotionListener{
    public void mouseDragged(MouseEvent e) {
      x = e.getX(); y = e.getY(); repaint();
    }
    public void mouseMoved(MouseEvent e) {}
  }
}
```


- L'implémentation d'une interface demande l'écriture de *toutes* ses méthodes.
- Un *adaptateur* est une classe qui implémente une interface avec un comportement par défaut.
- Le paquetage `java.awt.event` contient un adaptateur pour toute interface auditeur qui a au moins deux méthodes.


```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Gribouille2 extends Applet {
  int x0,y0,x,y;

  public void init() {
    addMouseListener(new Appuyeur());
    addMouseMotionListener(new Dragueur());
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }

  class Appuyeur extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      x0 = e.getX(); y0 = e.getY();
    }
  }
  class Dragueur extends MouseMotionAdapter {
    public void mouseDragged(MouseEvent e) {
      x = e.getX(); y = e.getY();
      repaint();
    }
  }
}
```



```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Gribouille3 extends Applet{
  int x0,y0,x,y;

  public void init() {

    addMouseListener(new MouseAdapter (){
      public void mousePressed(MouseEvent e) {
        x0 = e.getX(); y0 = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter (){
      public void mouseDragged(MouseEvent e) {
        x = e.getX(); y = e.getY();
        repaint();
      }
    });
  }

  public void update(Graphics gc) {
    paint(gc);
  }

  public void paint(Graphics gc) {
    gc.drawLine(x0,y0,x,y);
    x0 = x; y0 = y;
  }
}
```

Les classes `Appuyeur` et `Dragueur` sont devenues *anonymes*.

7

Introduction à Swing

1. Exemples
2. La structure des JFrame
3. Menus
4. Boutons

Premier exemple

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

class MyCloseableFrame extends JFrame{
  public MyCloseableFrame(){
    super("ma fenetre");
    setSize(300,200);
    addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        System.exit(0);
      }
    });
  }
}

public class MyCloseableFrameTest{
  public static void main(String[] args){
    JFrame frame = new MyCloseableFrame();
    frame.show();
  }
}
```

Remarques :

- On a utilisé une *classe interne anonyme* pour définir l'auditeur.
- La méthode `show()` est une méthode de la classe `Window` mère de `Frame`. Elle permet d'afficher un composant.
- Il n'est pas toujours facile de savoir si une méthode est définie dans la classe `Component`, `Window` ou `Frame`.
  Exemple de méthodes :

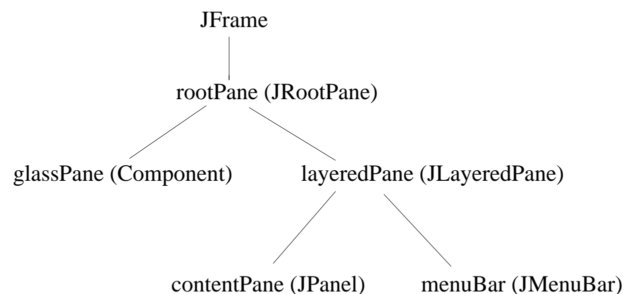
– Dans `java.awt.Component` :
  `boolean isEnabled()`
  `void setEnabled(boolean)`
  `void setVisible(boolean)`

– Dans `java.awt.Window` :
  `void toFront()`
  `void toBack()`

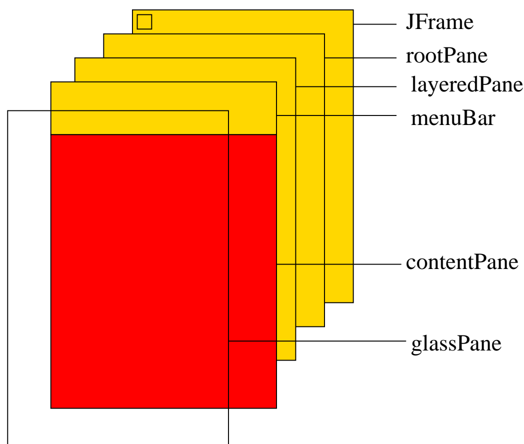
– Dans `java.awt.Frame` :
  `void setTitle(String)`

En Java, les cadres sont des conteneurs d'autres composants.

La structure (complexe) des JFrame



La structure (complexe) des JFrame



L'accès à la fenêtre `contentPane` se fait par la méthode `getContentPane()` de la classe `JFrame`.

Deuxième exemple

On ne dessine pas ou on n'écrit pas directement dans une `JFrame`, on passe par son `contentPane` qui est un `JPanel`.

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

class MyJFrame extends JFrame{
  public MyJFrame() {
    super("ma fenetre");
    setSize(300,200);
    JPanel p = new BonjourPanel();
    getContentPane().add(p);
  }
}

class BonjourPanel extends JPanel{
  public void paintComponent(Graphics gc){
    super.paintComponent(gc);
    gc.drawString("Bonjour",75,100);
  }
}

public class MyJFrameTest{
  public static void main(String[] args){
    JFrame frame = new MyJFrame();
    frame.show();
  }
}
```

- La méthode `paintComponent()` est une méthode de la classe `javax.swing.JComponent`, qui dérive de `Container`, qui dérive elle-même de `Component`.

- Chaque fois qu'une fenêtre est redessinée, le gestionnaire d'évènements Java envoie une notification à cette fenêtre. Les méthodes `paintComponent()` de tous les composants de la fenêtre sont alors exécutés.
- Il ne faut pas appeler soi-même `paintComponent()` puisque ceci est fait automatiquement.

Pour dessiner, on peut utiliser `java.awt.Graphics`.

Actions générées par un menu

La classe abstraite **javax.swing.AbstractAction** permet de définir des actions.

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

class MyJFrameAction extends JFrame{
  public MyJFrameAction(){
    super("ma fenetre");
    setSize(300,200);
    JPanel p = new JPanel();
    getContentPane().add(p);
    Action bleuAction = new CouleurAction("Bleu",Color.blue,p);
    Action rougeAction = new CouleurAction("Rouge",Color.red,p);
    Action vertAction = new CouleurAction("Vert",Color.green,p);

    JMenu m = new JMenu("Couleur");
    m.add(bleuAction);
    m.add(rougeAction);
    m.add(vertAction);
    JMenuBar mbar = new JMenuBar();
    mbar.add(m);
    setJMenuBar(mbar);
  }
}
public class MyJFrameActionTest{
  public static void main(String[] args){
    JFrame frame = new MyJFrameAction();
    frame.show();
  }
}
```



```
class CouleurAction extends AbstractAction{
  private Component cible;
  private Color couleur;

  public CouleurAction(String nom, Color couleur, Component cible) {
    super(nom);
    this.couleur = couleur;
    this.cible = cible;
  }

  public void actionPerformed(ActionEvent e){
    cible.setBackground(couleur);
    cible.repaint();
  }
}
```

Un exemple de JToggleButton

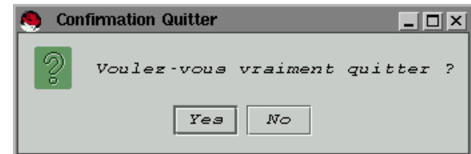
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyToggleFrame extends JFrame{

  public static void main(String args[]) {
    JFrame frame = new MyToggleFrame();
    frame.show();
  }

  public MyToggleFrame(){
    super("Choisir La Liaison PPP");
    getContentPane().add(new MyTogglePanel());

    Dimension dim = getToolkit().getScreenSize();
    setLocation(dim.width/2 - getWidth()/2,
                dim.height/2 - getHeight()/2);
    pack();
    setVisible(true);
```



```
    setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        int confirm = JOptionPane.showOptionDialog(
          MyToggleFrame.this,
          "Voulez-vous vraiment quitter ?",
          "Confirmation Quitter",
          JOptionPane.YES_NO_OPTION,
          JOptionPane.QUESTION_MESSAGE,
          null, null, null);
        if (confirm == 0)
          System.exit(0);
    }
    };
    this.addWindowListener(l);
  }
}


class MyTogglePanel extends JPanel {
  JPanel choixPanel, quitterPanel;

  public MyTogglePanel () {
    setLayout(new GridLayout(2,1,1,1));
    choixPanel = new JPanel(new GridLayout());
    quitterPanel = new JPanel(new GridLayout());
    add(choixPanel);
    add(quitterPanel);

    JToggleButton button1 = new JToggleButton("ppp0",true);
    Font bigFont = new Font("Dialog",Font.PLAIN, 24);
    button1.setFont(bigFont);
    choixPanel.add(button1);
    JToggleButton button2 = new JToggleButton("ppp1",false);
    button2.setFont(bigFont);
    choixPanel.add(button2);
    JToggleButton button3 = new JToggleButton("ppp2",false);
    button3.setFont(bigFont);
```

```
    choixPanel.add(button3);
    ButtonGroup buttonGroup = new ButtonGroup();
    buttonGroup.add(button1);
    buttonGroup.add(button2);
    buttonGroup.add(button3);

    ActionListener actionPPP  = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JToggleButton b = (JToggleButton) e.getSource();
        System.out.println("liaison "+ b.getText());
      }
    };
    button1.addActionListener(actionPPP);
    button2.addActionListener(actionPPP);
    button3.addActionListener(actionPPP);


    JButton button4 = new JButton("quitter");
    button4.setFont(bigFont);
    quitterPanel.add(button4);

    ActionListener actionQuitter  = new ActionListener(){
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    button4.addActionListener(actionQuitter);
  }
}
```

On définit de préférence un `JPanel` qui sera ensuite inséré au choix dans une `JFrame` ou une applette.



MCours.com