Chapitre 6 : Graphiques

- Dessins
- Modèle MVC (pour un logiciel de dessin)
- Boîtes de dialogues
- Compléments sur les dessins
- Images
- Threads et Animation

Dessin dans les composants

Rappels

> On dessine dans un composant awt en implémentant sa méthode `paint(g).`

> on dessine dans un `JComponent` en redéfinissant sa méthode `paintComponent(g)` et on invoque `super.paintComponent(g)` au début de la méthode.

> l'argument `g` des méthodes de dessin est un objet de type `Graphics.`

> Ces méthodes sont appelées (sauf bugs) automatiquement quand le composant est affiché ou quand il doit être réaffiché (changement de taille).

> On pourrait les appeler, en récupérant d'abord le contexte avec `getGraphics()`. Mais si on veut redessiner (= ou dessiner dans) un composant à un autre moment, on utilisera en fait une méthode appelée `repaint`.

repaint

> `repaint()`

Redessine tout le composant (dans le thread de gestion des événements), en appelant sa méthode `paint()` après que tous les événements en attente aient été traités.

L'affichage d'un JComponent

L'affichage d'un JComponent s'effectue dans le thread de gestion des événements en 4 étapes:

1. affichage du fond (s'il est opaque)
2. affichage « personnalisé » (du programmeur)
3. affichage des bords
4. affichage (récursif) des enfants (dans le cas où il s'agit d'un conteneur)

Pour SwingApplication

repaint

> `repaint(Rectangle rect)`

> Le composant sera repeint après que tous les événements en attente aient été traités.

> La région définie par le rectangle est ajoutée aux zones devant être repeintes (clip): permet de limiter la zone effective du dessin.

repaint

> `repaint( long msec )`
> demande que le composant soit redessiné dans les prochaines `msec` millisecondes.

repaint

> `repaint( int msec, int x, int y, int width, int height)`
> Ajoute la région définie par les coordonnées à la liste des zones du composant devant être repeintes (clip). Il y aura redessin par appel à `paint()` quand tous les événements auront été traités ou quand `msec` millisecondes se seront écoulées.

Que peut-on faire avec Java 2D ?

> Dessin de lignes et de courbes,
> Remplissage avec des gradients et des textures,
> Transformation simples de textes et de graphiques,
> Composition, overlapping de textes et de graphiques,
> Manipulations d'images et de bitmaps.

Que peut-on faire avec Java 2D ?

Java 2D API

> Grande variété de primitives géométriques.
> Mécanismes de hits sur les textes, les graphiques et les images.
> Modèle de composition et de niveaux (layers).
> Gestion de la couleur.
> Impression des documents.

L'objet `Graphics`

Il fournit des primitives de dessin.

> l'objet `Graphics` sert aussi à définir le contexte d'interprétation des primitives de dessin sur un composant.

> selon les primitives, des attributs spécifiques de ce « contexte graphique » sont utilisés (ex. épaisseur de ligne pour le dessin des segments ou des rectangles, fonte pour le dessin de texte, etc.)

Graphics2D

> Pour utiliser les classes de Java2D, on utilise un objet `Graphics2D`. Cette classe fournit les utilitaires permettant de dessiner en 2D dans un `Component`. Elle permet de dessiner des droites, des courbes, des gradients, etc.

> `Graphics2D` étend `Graphics` : les anciennes méthodes de awt pour dessiner subsistent.

```
public void paintComponent (Graphics g) {
                Graphics2D g2 = (Graphics2D) g;
                ...
            }
```


Exemple d'utilisation

```
import javax.swing.*;
import java.awt.*;      // pour la classe Graphics
class SketcherView extends JPanel {
        public void paintComponent (Graphics g) {
                super.paintComponent(g);
                Graphics2D g2D = (Graphics2D) g;
                g2D.setPaint(Color.red);
                g2D.draw3DRect(50, 50, 150, 100, true);
                g2D.drawString("Un joli rectangle", 60, 100);
        }
}
```

Exemple

Graphics2D

> Le contexte graphique `Graphics2D` se charge des changements de coordonnées selon que l'on dessine sur l'écran ou sur un autre dispositif.

> Il maintient l'information sur la couleur du dessin, l'épaisseur des traits, le gradient, les textures, etc., et permet aussi d'effectuer des transformations (rotation, etc.) sur le dessin initial.

Processus d'interprétation

La primitive de dessin est interprétée dans le contexte graphique selon un processus de « rendu » (*rendering process*) en 4 étapes :

1. déterminer le dessin initial (source)
2. déterminer la zone atteinte (clip)
3. déterminer les couleurs initiales (pixels source)
4. les combiner avec celles de la destination pour produire réellement les pixels modifiés

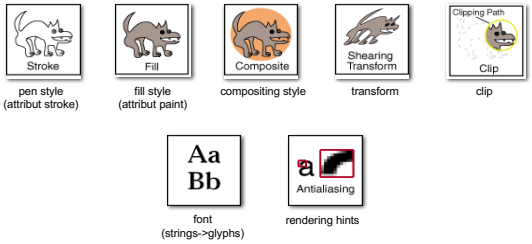
Graphics2D *rendering context*

> L'objet `Graphics2D` enregistre un certain nombre d'attributs qui forment le ***`Graphics2D`*** *rendering context* ou contexte de rendu.

> Pour dessiner, on définit le contexte, puis on appelle les routines de rendu `draw` ou `fill` par exemple.

Graphics2D *rendering context*

Exemples d'attributs du contexte :



Comment définir le contexte ?

> Utiliser les méthodes du type setAttribute de la classe Graphics2D:

- `setStroke`
- `setPaint`
- `setComposite`
- `setTransform`
- `setClip`
- `setFont`
- `setRenderingHints`

Dessiner des formes avec Graphics2D

Exemple

```
gp = new GradientPaint(0f,0f,blue,0f,30f,green);
g2.setPaint(gp);
```

Primitives de dessin

Pour dessiner, la classe Graphics2D propose entre autres méthodes:

```
>draw(Shape shape)
>fill(Shape shape)
>drawString(String text,
        int x, int y)
>drawImage(...)
```


draw

> Dessine la frontière d'une forme selon les attributs `stroke` et `paint`.

fill

> Dessine l'intérieur d'une forme en fonction de l'attribut `paint`.

drawString

> Convertit une chaîne de caractères en *glyphs*, qui sont ensuite dessinés selon l'attribut `paint`.

Graphics2D

> Puisque la classe `Graphics2D` hérite de la classe `Graphics`, les méthodes de la classe `Graphics` sont disponibles. Exemples: `drawOval`, `fillRect`.

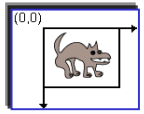
Systèmes de coordonnées

> Java2D gère deux systèmes de coordonnées:
  - *user space*,
  - *device space*.

> Les primitives graphiques et les formes sont dans user space. L'autre espace dépend d'un périphérique.

User space

> Par défaut:



Graphics2D

Attention: le repère lié au composant est situé en haut à gauche dans son parent mais les bords sont inclus dans le composant.

```
public void paintComponent(Graphics g) {
        ...
        Insets insets = getInsets();
        int currentWidth = getWidth() - insets.left - insets.right;
        int currentHeight = getHeight() - insets.top - insets.bottom;
        ...
        // Le premier dessin apparaît en haut à gauche en (x,y)
        // avec  insets.left ≤ x   et  insets.top ≤ y
    }
```

Les formes de `java.awt.geom`

> Les classes du package java.awt.geom fournissent des formes typiques:
  - `Arc2D, Ellipse2D, QuadCurve2D,`
  - `Area, GeneralPath, Rectangle2D,`
  - `CubicCurve2D, Line2D, RectangularShape, Dimension2D,`
  - `Point2D, RoundRectangle2D`.

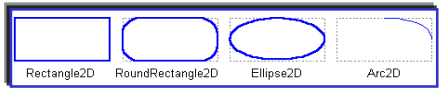
`java.awt.geom`

> Certaines de ces classes implémentent l'interface `Shape` qui fournit une méthodologie standard d'inspection des formes 2D.

> Avec ces classes, vous pouvez définir n'importe quelle forme puis la dessiner par l'intermédiaire d'un objet `Graphics2D` en invoquant les méthodes `draw` ou `fill`.

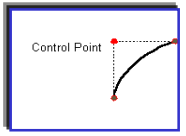
Formes rectangulaires

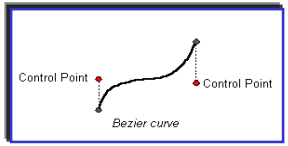
> Classes `Rectangle2D`, `RoundRectangle2D`, `Arc2D`, et `Ellipse2D`, sous-classes de `RectangularShape`.



Courbes splines quadratiques et cubiques

> `QuadCurve2D`, `CubicCurve2D`.





Dessiner des formes: l'interface Shape

> C'est une interface très puissante.

> ex: des méthodes `contains()` permettent de savoir si un point ou un rectangle sont à l'intérieur d'une forme Shape.

interface Shape

> La plupart des formes qu'on souhaite dessiner implémentent l'interface `Shape.`

> Il y a par exemple des points, des lignes, divers rectangles qui sont des objets implémentant Shape:

- `Point2D,`
- `Line2D,`
- `Rectangle2D, RoundRectangle2D.`

interface Shape

Une forme 2D est décrite avec un objet **`PathIterator`** qui en exprime le contour, et une **règle** qui détermine ce qui est à l'intérieur (ou à l'extérieur) de la forme ayant ce contour.

On peut aussi faire des **opérations booléennes** sur les rectangles, et plus généralement savoir si un rectangle intersecte avec, ou est contenu dans une forme donnée.

interface Shape

> `Arc2D, Ellipse2D:` arcs et ellipses.

> `QuadCurve2D, CubicCurve2D` : courbes quadratiques et cubiques. (notion de point de contrôle).

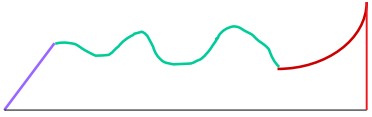
> `GeneralPath:` permet d'avoir des formes composites.

GeneralPath

> La classe `GeneralPath` offre une forme quelconque en définissant une suite de positions sur la frontière. Ces positions sont reliées indépendamment à une ligne géométrique.

GeneralPath

> Pour faire une forme composite dont le contour est formée de droites, de courbes (cubiques ou quadratiques), ou d'arcs, etc.

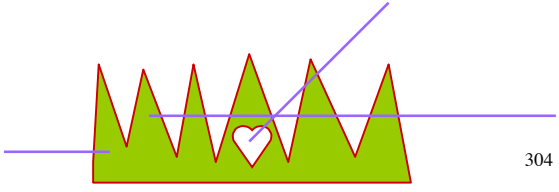


GeneralPath

> Quand vous définissez un `GeneralPath`, vous précisez au constructeur la manière dont l'intérieur est défini avec une règle de remplissage (*rule*).
> Ensuite, vous ajoutez les différents composants du contour.
> (On peut aussi définir un GeneralPath à partir d'une forme Shape).

Règle WIND_EVEN_ODD

> Un point est intérieur si une demi-droite d'origine le point croise le contour de la forme un nombre impair de fois. (Fonctionne avec les formes ayant des trous).



Règle WIND_NON_ZERO

> On oriente le tracé du `GeneralPath`.
> Le point est intérieur si la différence entre le nombre de fois que la demi-droite d'origine le point est croisée par une limite de gauche à droite et le nombre de fois où elle est croisée de droite à gauche n'est pas égale à zéro.
> Ne fonctionne pas pour des formes à trous.

Styles de ligne et de remplissage

> Les classes `Stroke` et `Paint` permettent de définir tous les styles usuels.
> Le style des lignes se définit dans le `Stroke`.
> Méthode générale: créer un objet `BasicStroke` et le passer à la méthode `Graphics2D.setStroke`.

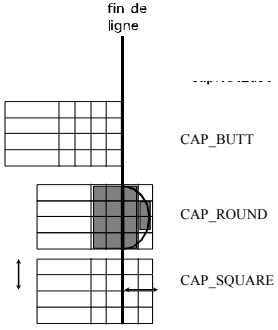
Graphics2D: attribut stroke

> ***stroke*** définit les caractéristiques du trait du dessin (épaisseur, extrémités des lignes, jointures des lignes, pointillés, etc.). Il est utilisé par les primitives en `draw<FORME>.`

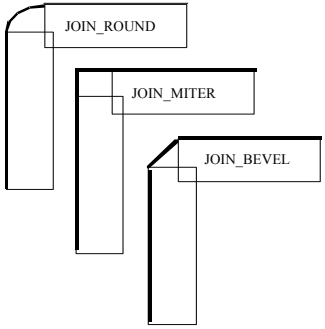
-> `setStroke(Stroke s).`

```
> BasicStroke(), // dans awt
> BasicStroke(float width),
> BasicStroke(float width,
    int cap, int join)
```


stroke: attribut cap



stroke: attribut join



stroke: attribut dash (tiret)

> `BasicStroke(float width, int cap, int join, float miterlimit, float[] dash, float dash_phase)`

> permet de définir l'alternance des longueurs de tirets d'un motif de pointillés en fournissant le tableau `dash` des longueurs de tirets successifs. L'argument `dash_phase` donne le décalage d'origine du motif.

Motif de remplissage (fill patterns)

> Définir un objet implémentant l'interface `Paint` et passer cet objet à la méthode `setPaint` de `Graphics2D`.

> 3 classes implémentent l'interface `Paint`:

- `Color`
- `GradientPaint`
- `TexturePaint`

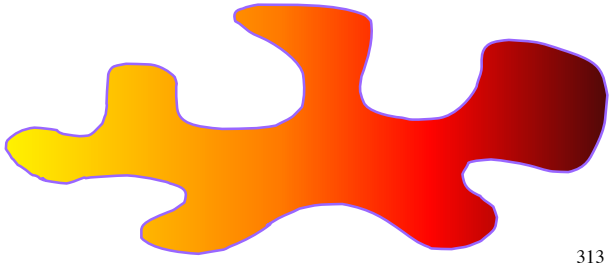
Graphics2D: attribut paint

> ***paint*** : couleur des lignes, et couleur ou motif du remplissage des formes.

→La couleur du `foreground` d'un composant fournit l'attribut `paint` par défaut de son contexte graphique. Pour dessiner avec d'autres couleurs : `setPaint(Paint paint)`

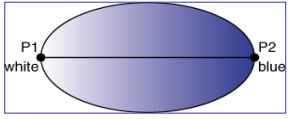
→Pour basculer (sur la destination) entre une couleur `c1` fournie par `paint` et une autre couleur : `setXORMode(Color c2)`

setPaint: GradientPaint (dégradés) ou TexturePaint (images)



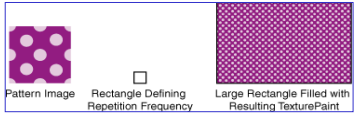
Classe `GradientPaint`

> Donner une couleur de début et une couleur de fin. Il y a interpolation entre les deux.



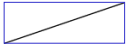
Classe `TexturePaint`

> On donne une image `BufferedImage`, et un rectangle dans lequel l'image est répliquée.



Exemples: stroke et fill de base

```
// draw Line2D.Double
g2.draw(new Line2D.Double(x, y+rectHeight-1,x + rectWidth, y));
```



Exemples: stroke et fill de base

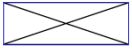
```
// draw Rectangle2D.Double
g2.setStroke(stroke);
g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
```

Exemples: stroke et fill de base

```
// draw RoundRectangle2D.Double
g2.setStroke(dashed);
g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
```

Exemples: stroke et fill de base

```
//draw GeneralPath (polygon)
int x1Points[] = {x, x+rectWidth, x, x+rectWidth};
int y1Points[] = {y, y+rectHeight, y+rectHeight, y};
GeneralPath polygon = new GeneralPath(GeneralPath.WIND_EVEN_ODD, x1Points.length);
polygon.moveTo(x1Points[0], y1Points[0]);
for (int index = 1; index < x1Points.length; index++) {
    polygon.lineTo(x1Points[index], y1Points[index]);
    };
polygon.closePath();
g2.draw(polygon);
```



Exemples: stroke et fill de base

```
// fill RoundRectangle2D.Double
g2.setPaint(redtowhite);
g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
```

Exemple à faire en TP : un logiciel de dessin

Exemple extrait
du livre de Ivor Norton,
"Maitrisez Java 2"

Rappel: interface Observer

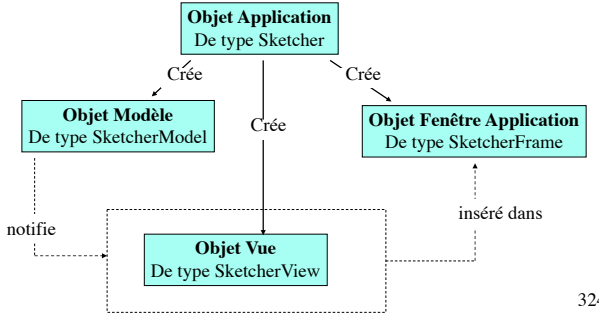
```
public class View implements Observer {

    public void update(Observable object, Object arg) {
            // cette méthode est déclenchée
            //  quand l'objet observable est modifié
    }
    // reste de la définition de la classe
}
public class Model extends Observable {
            // définition + appels à NotifyObservers
}
```


Rappel: classe Observable

```
> addObserver(Observer o)
> deleteObserver(Observer o)
> deleteObservers()
> notifyObservers(Object arg)
> notifyObservers()
> int countObservers()
> protected setChanged()
> boolean hasChanged()
> protected clearChanged()
```


Architecture Modèle/Vue dans un logiciel de dessin



Architecture Modèle/Vue

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

public class Sketcher {          // les objets application
        private SketcherModel model;
        private SketcherView view;
        private static SketcherFrame frame;
        private static Sketcher  theApp; // pour mémoriser une instance

        public static void main(String[] args) {
                theApp = new Sketcher();
                theApp.init();
        }
        public SketcherFrame getFrame() {
                return frame;
        }
```


Architecture Modèle/Vue

```
        // définir ici de même getModel() et getView()

        public void init() {
                // initialise le cadre de la fenêtre principale
                window = new SketcherFrame("Dessin", this);
                window.setBounds(20, 20, 800, 500);
                // ou le code pour centrer la fenêtre

                model = new SketcherModel();
                view = new SketcherView(this);
                model.addObserver((Observer)view);
                window.getContentPane().add(view,
                                BorderLayout.CENTER);
                window.setVisible(true);
        }
}
```


Architecture Modèle/Vue

```
// dans SketcherFrame (précédent programme) modifier le
// constructeur et rajouter la variable theApp instance de l'appli.

        …
        private Sketcher theApp; // pour mémoriser l'instance

        public SketcherFrame(String titre, Sketcher theApp) {
                setTitle(titre);
                this.theApp = theApp; // on mémorise l'instance
                setJMenuBar(menuBar);
                setDefaultCloseOperation(EXIT_ON_CLOSE);

        // reste du constructeur comme auparavant, menus, etc.
        // on va le revoir plus loin pour la gestion des événements

        }
```


Architecture Modèle/Vue

```
import javax.swing.*;
import java.util.*;
class SketcherView extends JPanel implements Observer {
        private Sketcher theApp;        // Objet Application
        public SketcherView(Sketcher theApp) {
                this.theApp = theApp; // on mémorise l'instance
        }
        public void update(Observable o, Object rectangle) {
        // réaffichage du modèle: on appellera repaint sur le rectangle
        }
        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                Iterator elements = theApp.getModel().getIterator();
                // affichage du model élément par élément
}       }
```

Architecture Modèle/Vue

```
class SketcherModel extends Observable {
        protected LinkedList ListeElements = new LinkedList();
        public void add(Element element) {
                ListeElements.add(element);
                setChanged();
                notifyObservers(element.getBounds());
        }
        … // même style de code pour remove avec une précaution
        // si l'objet supprimé n'existe pas dans le modèle …
        public Iterator getIterator() {
                return ListeElements.listIterator();
        }
}
```


Gestion des événements

```
public class SketcherFrame extends JFrame implements Constants {
        protected JTextArea output ;
        protected JScrollPane scrollPane;
        private JMenuBar menuBar;

        private Color elementColor = DEFAULT_ELEMENT_COLOR ;
        private int elementType = DEFAULT_ELEMENT_TYPE;
        protected String newline = "\n";

        private Sketcher theApp;

        public SketcherFrame(String titre, Sketcher theApp) {
                setTitle(titre);
                this.theApp = theApp; // on mémorise l'instance d'appli
```


Gestion des événements

```
                JMenu menu;
                JMenuItem menuItem;
                JRadioButtonMenuItem rbMenuItem;
                // Construire le menu Elements …
                menu = new JMenu("Elements");
                menu.setMnemonic(KeyEvent.VK_E);
                menuBar.add(menu);

                ButtonGroup group = new ButtonGroup();
rbMenuItem = new JRadioButtonMenuItem("Ligne », elementType == LIGNE);
                rbMenuItem.setMnemonic(KeyEvent.VK_L);
                group.add(rbMenuItem);
                rbMenuItem.addActionListener(new TypeListener(LIGNE));
                menu.add(rbMenuItem);
```


Gestion des événements

```
                // etc. pour les autres types RECTANGLE, etc.
                …
        } // fin du constructeur
    class TypeListener implements ActionListener {
        private int type;

        TypeListener(int type) {
                this.type = type;       // mémorise le type
        }

        public void actionPerformed(ActionEvent e) {
                elementType = type; // affectation de la variable
                // correspondant au type de l'élément courant
```

Gestion des événements

```
        // le reste consiste à imprimer des infos sur
        // l'evenement qui s'est produit dans la JTextArea
                JMenuItem source
                        = (JMenuItem)(e.getSource());
                String s = "ActionEvent detecté dans
TypeListener."
                        + newline
                        + "    Event source: "
                        + source.getText()
                        + " (une instance de "
                        + getClassName(source) + ")";
                output.append(s + newline);
        }
`
```


Boîtes de dialogues

Boîtes de dialogue

> Une boîte de dialogue s'affiche dans le contexte d'une autre fenêtre : son parent. Elle gère la saisie de données et/ou affiche des messages.

> `JDialog` est une fenêtre `Window` spécialisée permettant de définir soi-même une boîte de dialogue.

> `JOptionPane` offre une manière simple de créer des boîtes de dialogues standard.

Modal et non modal

> Les boîtes de dialogues ont deux types de fonctionnement distincts :

> modal : c'est un dialogue qui inhibe toutes les autres fenêtres de l'application, jusqu'à ce qu'on ferme la fenêtre de dialogue (ex. boîte de saisie).

> non modal : la fenêtre de dialogue peut rester à l'écran sans bloquer les autres fenêtres de l'application.

constructeurs JDialog

constructeur	barre de titre	parent	mode
JDialog()	vide	shared hidden frame	non modal
JDialog(Frame parent)	vide	parent	non modal
JDialog(Frame parent, String title)	title	parent	non modal
JDialog(Frame parent, boolean modal)	vide	parent	modal (si true)
JDialog(Frame parent, String title, boolean modal)	title	parent	modal (si true)

JDialog

Quelque soit le constructeur utilisé, on peut modifier le `JDialog`. On dispose des méthodes :

```
setModal(boolean)
boolean isModal()
setVisible(boolean)
setTitle(String)
String getTitle()
setResizable(boolean)
```


JOptionPane

> La classe `JOPtionPane` définit un certain nombre de **méthodes statiques** pour créer et afficher des boîtes de dialogue modales standards.

JOptionPane

> Pour afficher un message, on utilisera `showMessageDialog`.
> Pour confirmer une réponse, comme yes/no/cancel, `showConfirmDialog.`
> Pour saisir une entrée, `showInputDialog.`
> Pour une union des trois précédentes : `showOptionDialog`

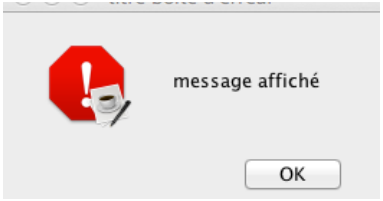
showMessageDialog(Component parent, Object message)

- Affiche une boîte de dialogue modale avec le titre «Message».
- `parent` est utilisé pour positionner la boîte au centre du `Frame` contenant le parent. S'il est `null` le dialogue sera positionné au centre de l'écran.
- L'argument `message` de type `String`, `Icon`, ou `Component` est affiché en sus du bouton OK. En cas d'autre type, `toString()` est invoquée.
- On peut passer un tableau d'objets. Chaque élément est alors disposé en pile verticale.

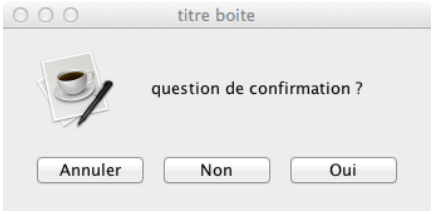
showMessageDialog(…, String title, int messagetype)

- Même chose, sauf que l'on a un titre en plus.
- `messagetype` peut valoir `ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE.`
- Ce type détermine le style du message et une icône associée par défaut.
- Une autre version rajoute un argument `Icon`.

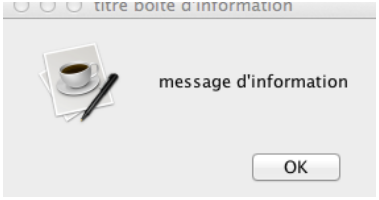
Message d'erreur



Boîte de confirmation



Message d'information



ShowInputDialog(…)

> `showInputDialog(Object message)` : le message sert de légende au champ texte de saisie. Cette méthode retourne une `String`, qui est saisie quand on clique sur OK. Si on clique sur Cancel, elle renvoie `null`.

> Il y a des variantes où l'on peut préciser le parent ou le type de message, ou une liste d'options.

Boîte de saisie

Boîtes de dialogue: résumé

> Pour utiliser les boîtes de dialogues modales standards, on fera appel à une méthode statique de la classe **JOptionPane**

> Pour créer sa propre boîte de dialogue et la configurer comme on le souhaite, on utilisera la classe **JDialog** comme cadre de base