Plan du cours

- Chapitre 1 : un survol de Java.
- Chapitre 2 : compléments sur le langage.

  (classes abstraites, interfaces, String et StringBuffer, Threads, le package d'entrées/sorties java.io)
- **Chapitres 3, 4, 5 et 6 : Java Swing.**
  (composants, afficheurs, événements, dessins, images , dialogues et animation).

Chapitre 6 : Interface Graphique

- Dessins
- Modèle MVC
- Images
- Dialogues
- Animation

Dessin dans les composants

Rappels

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

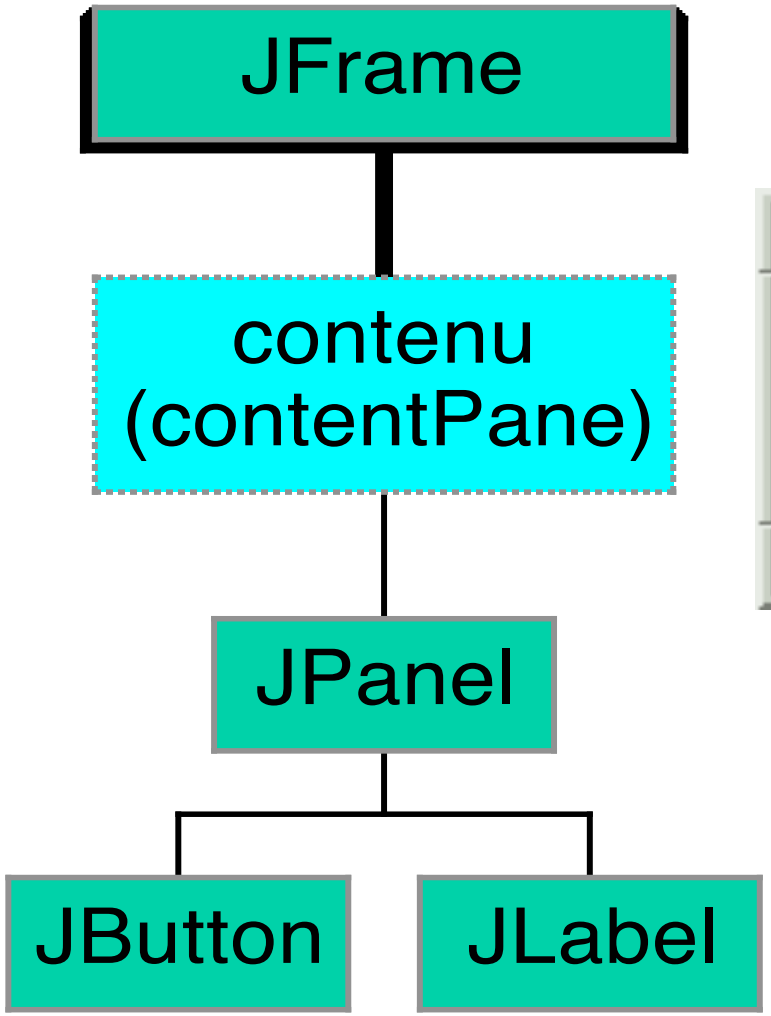
Redessine tout le composant (dans le thread de gestion des événements), en appelant sa méthode `paint()` après que tous les événements en attente aient été traités.

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. affichages (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.

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.

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

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
}
```

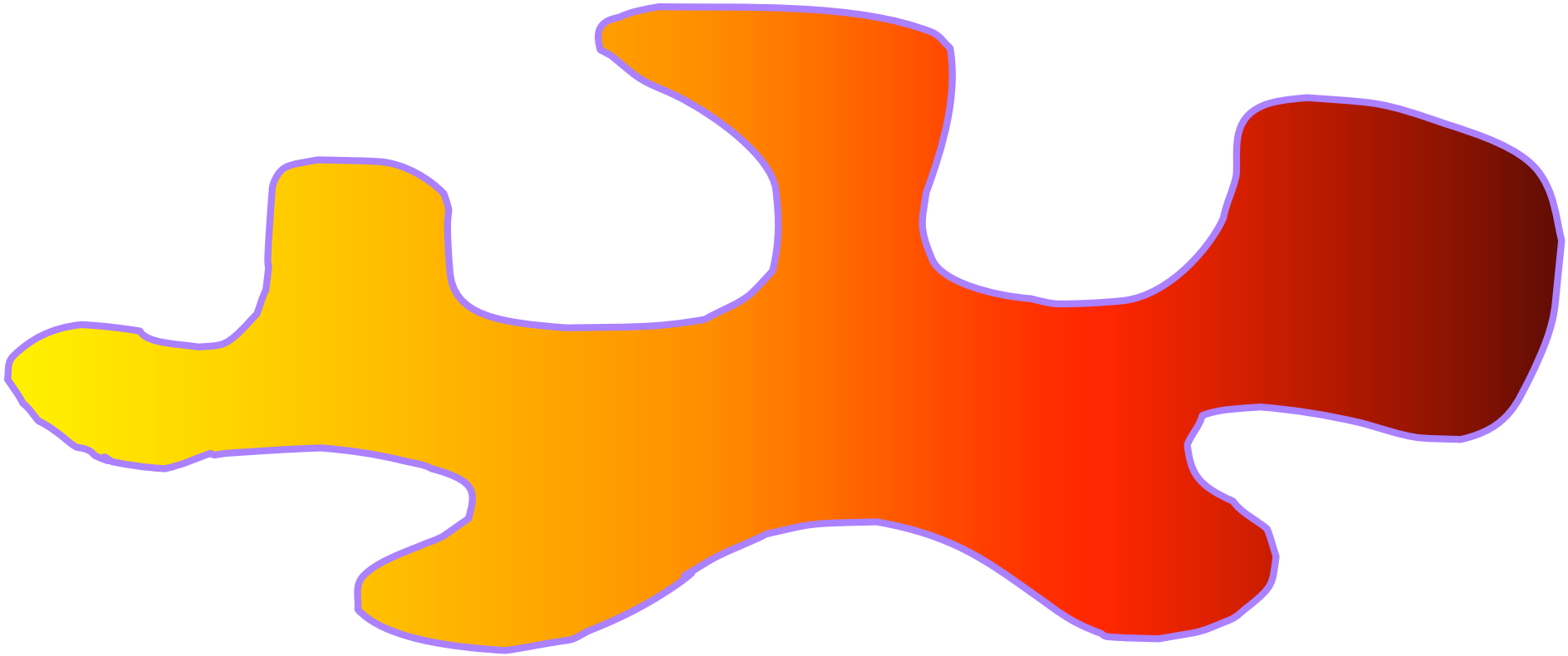
Graphics2D: attribut paint

> ***paint*** : couleur des lignes, et couleurs ou motif des remplissages.

→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)



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>.`

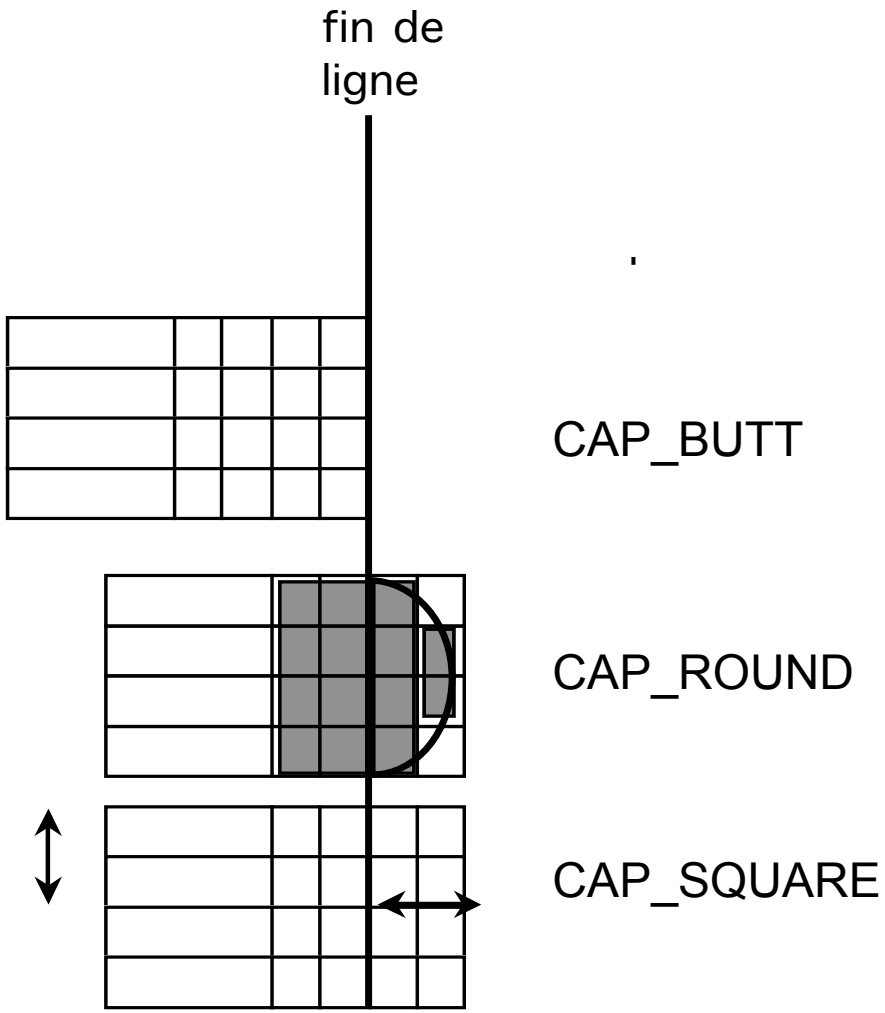
  -> `setStroke(Stroke s).`

> `BasicStroke(), // dans awt`

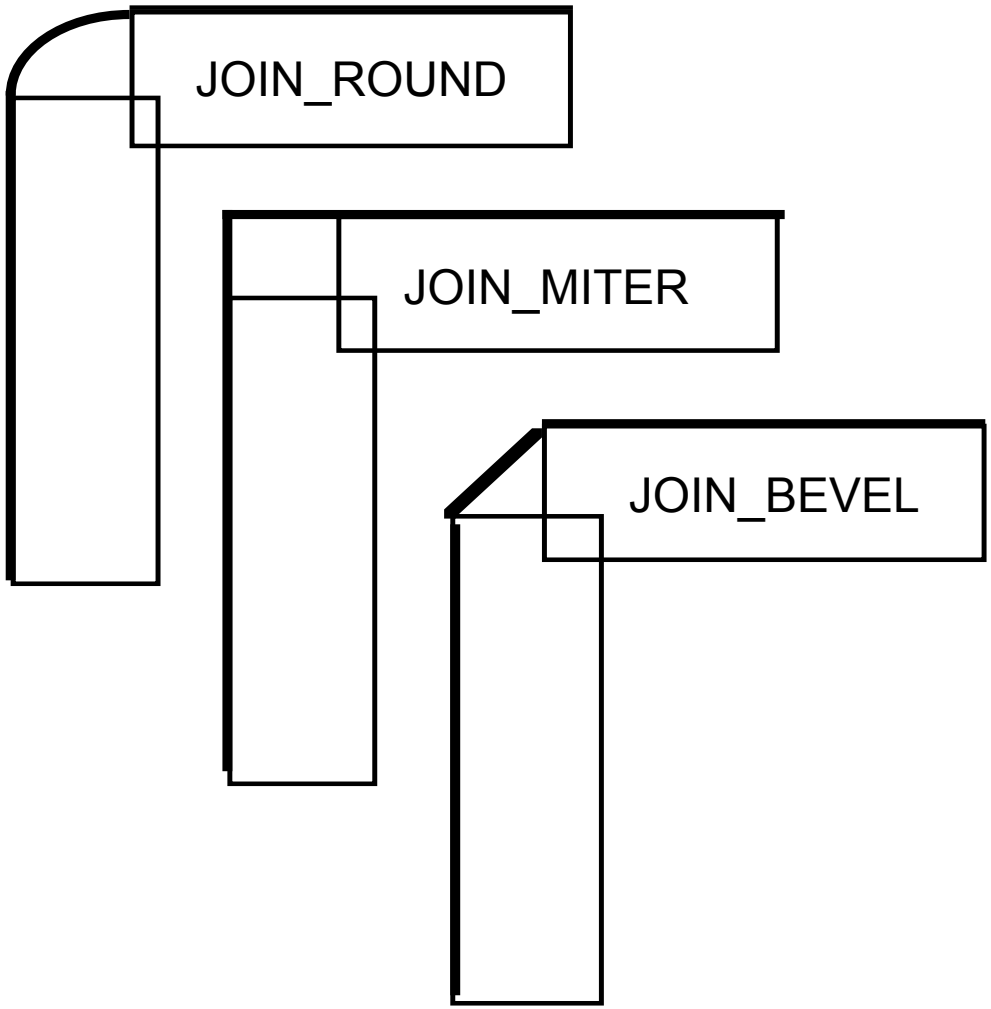
> `BasicStroke(float width),`

> `BasicStroke(float width,`
`int cap, int join).`

stroke: attribut cap



stroke: attribut join



stroke: 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.

Graphics2D (autres attributs)

> ***font*** -> `setFont(Font font)`

> ***transform***

-> `setTransform(AffineTransform Tx)`

> ***clip*** -> `setClip()`

> ***composite*** détermine comment les pixels du dessin source et les pixels de la destination (a priori l'écran) se combinent pour donner les pixels (couleurs) finalement modifiés sur la destination.

Dessiner des formes avec Graphics2D

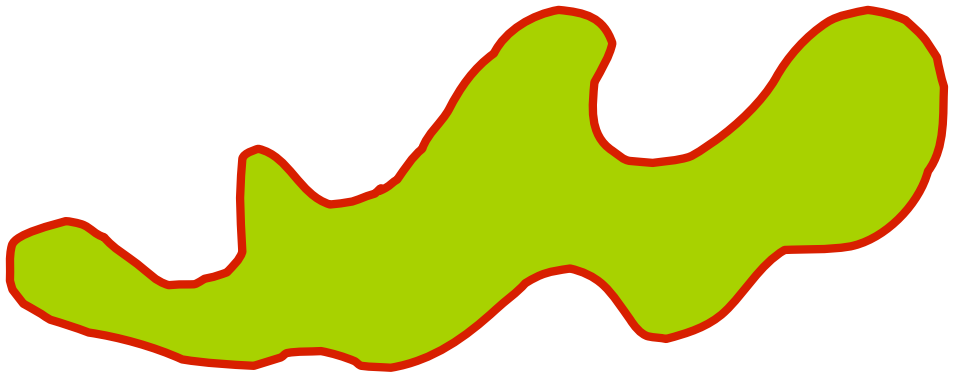
Primitives de dessin

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

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.

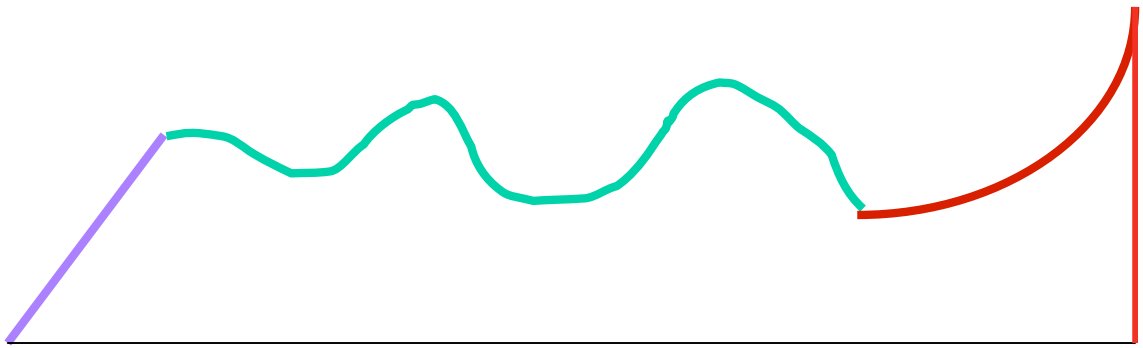
On peut aussi faire des **opérations booléennes** sur les rectangles, et plus généralement savoir si un rectangle intersecte avec, ou est contenu dans une forme donnée.

`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.

Classe GeneralPath

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

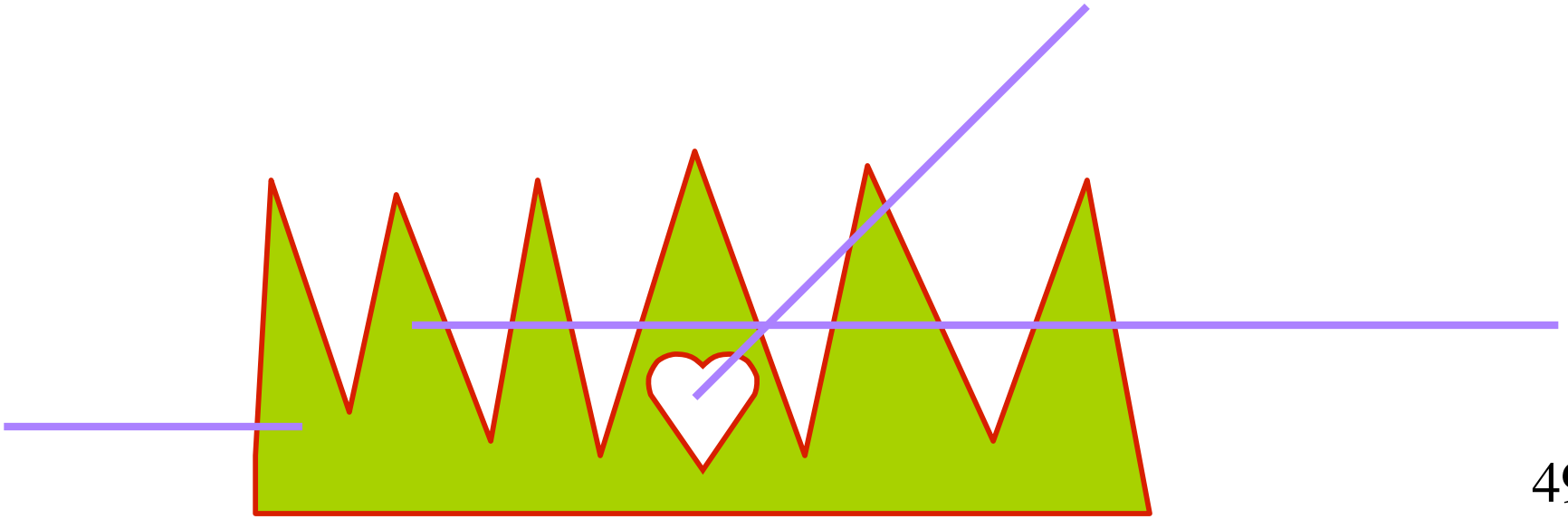


Classe GeneralPath

- 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.

Exemple à faire en TP : un logiciel de dessin

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

Rappel: interface Observer

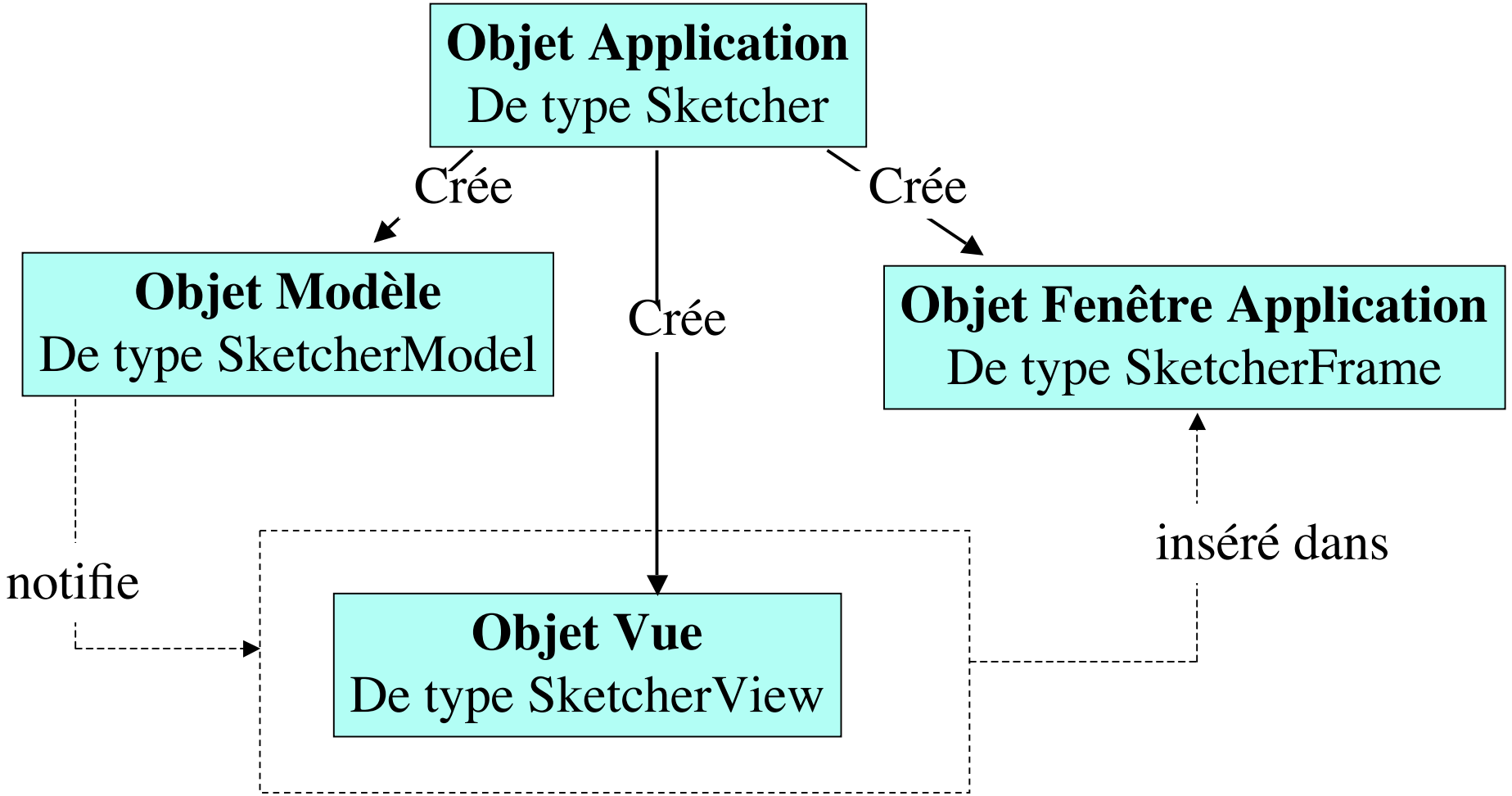
```
public class View implements Observer {

    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 window;
        private static Sketcher  theApp; // pour mémoriser une instance

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

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 bien 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;

        public SketcherFrame(String titre, Sketcher theApp) {
                setTitle(titre);
                this.theApp = theApp;
                setJMenuBar(menuBar);
                setDefaultCloseOperation(EXIT_ON_CLOSE);

        // reste du constructeur comme auparavant, menus, etc.

        }
        …
```

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;
        }
        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();
        }
}
```

Dessiner des images

java 2 (swing): classe ImageIcon

- 9 constructeurs `ImageIcon()`
- contient aussi une référence à un objet `Image` (de java.awt) récupérable par `getImage()`.
- formats lus : GIF, JPEG, PNG.
  - GIF: Graphics Interchange Format. Limité à 256 couleurs.
  - JPEG: Joint Photographic Experts Group.Plus perfectionné. Comprime les données avec perte.
  - PNG: Portable Network Graphics. Pour des images crées par ordinateur.

classe ImageIcon

`ImageIcon` utilise un **`MediaTracker`** pour suivre le chargement de l'image. C'est un peu plus simple qu'un `ImageObserver` et en général suffisant.

On crée une instance de `MediaTracker`, on lui demande de suivre une (ou plusieurs images) et on peut l'interroger avec :

```
> int getImageLoadStatus()
   // ABORTED, COMPLETE, ERRORED, LOADING
```

interface ImageObserver

Pour un suivi plus fin, on peut aussi utiliser l'interface `ImageObserver` :

il faut implanter une méthode `imageUpdate` et s'assurer de l'enregistrement de l'objet comme observateur (habituellement quand on spécifie un `ImageObserver` à une méthode `drawImage`).

Elle sera appelée chaque fois qu'une nouvelle information sur l'image devient accessible.

```
>void setImageObserver(ImageObserver)
>ImageObserver getImageObserver()
```

classe ImageIO

```
BufferedImage img = null;
try {
    img = ImageIO.read(new File("strawberry.jpg"));
} catch (IOException e) {
}
```

Classe URL

Uniform Resource Locator

La classe URL de java.net encapsule 4 données :

> Un protocole d'accès (http ou ftp)

> Un nom de domaine (java.sun.com)

> Un numéro de port

> Un chemin d'accès au fichier

Classe URL

- `URL sourceURL = new URL («http://www.wrox.fr/»);`
- `getCodeBase():` renvoie l'URL de la source où le fichier .class d'une applet est stocké.

```
URL illustre = new URL
(getCodeBase(), «dessin.gif»);
```

- `URL doc = getDocumentBase()`

Classe URL

- `URL sourceURL = new URL («http», «www.ncsa.uiuc.edu», -1,// default port «/demoweb/url-primer.html»);`
- `String getProtocol(), getHost(), getFile(); int getPort().`
- `Boolean equals(URL)`
- `InputStream openStream()// retourne // un InputStream sur le fichier défini par // l'URL`

classe ImageIO

```
// si le code tourne dans une applet
BufferedImage img = null;
try {
   URL url = new URL(getCodeBase(),
                     // url du repertoire contenant l'applet
                                   "strawberry.jpg");
    img = ImageIO.read(url);
} catch (IOException e) {
}
```

Dessiner des images

- `drawImage(Image im, AffineTransform xform, ImageObserver observer)`
- Remarque: il existait déjà des méthodes `drawImage(Image im, int x, int y,[int width, int height], [Color background], ImageObserver observer)` dans awt.Graphics.
- `observer` implémente `imageObserver` et sera notifié quand une nouvelle info sur l'image deviendra accessible.

ImageObserver

`imageUpdate(Image img, int infoflags, x, y, width, height)` est la méthode invoquée dans `observer` quand une information requise par une interface asynchrone devient accessible. ex:

`getWidth(ImageObserver)`, `drawImage(img, x, y, ImageObserver)`.

Elle retourne true si d'autres appels de mise à jour peuvent se produire sur l'image.

imageUpdate(Image img, int infoflags, x, y, width, height)

- `infoflags`: indique quelles informations sont maintenant disponibles sur l'image par un OU bit à bit:

  `WIDTH, HEIGHT, PROPERTIES, SOMEBITS, FRAMEBITS, ALLBITS, ERROR, ABORT.`

- `imageUpdate` retourne false si toutes les informations sur l'image sont maintenant disponibles.

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 une boîte de dialogue.
- `JOptionPane` offre une manière simple de créer des boîtes de dialogues.

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`.

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.

Threads et animations

Threads : Rappels

> Thread = flot d'exécution dans un processus.
Multi-processing ≠ multi-threading

> un seul thread est exécuté à la fois.

> le scheduler java gère la vie des threads avec des procédures qui permettent de passer d'un état à un autre.

Gestion des threads

> start(): lance la méthode run() du thread

> stop(): détruit le thread

> suspend()/resume(), sleep(long[,int]) …

☹ 2 threads accédant à une même donnée doivent être synchronisés.

Threads and Swing

- ☹ Les composants Swing ne peuvent être accédés que par un thread à la fois.
- ☞ Or, on a vu qu'un thread swing gère les événements sur les composants quand ils sont réalisés (=après un *appel à setVisible(), show() ou pack()*), donc tout code qui affecte ou dépend d'un composant *réalisé* doit être effectué *dans* ce thread (=via des événements).
- > Exceptions: les procédures dites *thread-safe* (elles peuvent être appelées de n'importe où).

  par exemple: repaint() ou revalidate()

Threads and Swing

- ☞ On construit l'interface dans le thread principal (`main` ou `init`). Une fois le cadre réalisé `(setVisible, show ou pack)` on ne touche plus aux composants que dans les gestionnaires d'événements.
- ☺ On peut cependant appeler `repaint` ou `revalidate`, modifier la liste des écouteurs, appeler `invokeLater` ou `invokeAndWait` depuis un autre thread.

invokeAndWait

Appelée depuis un écouteur d'événement, elle peut générer une exception dans le thread de dispatching des événements :

```
void showHelloThereDialog() throws Exception {
        Runnable showModalDialog = new Runnable() {
            public void run() {
              JOptionPane.showMessageDialog(myMainFrame,
                                            "Hello There");
            }
        };
        SwingUtilities.invokeAndWait(showModalDialog);
}
```

invokeAndWait

```
// Appelée depuis un thread qui doit accéder à deux TextField
void printTextField() throws Exception {
        final String[] myStrings = new String[2];
        Runnable getTextFieldText = new Runnable() {
                public void run() {
                        myStrings[0] = textField0.getText();
                        myStrings[1] = textField1.getText();
                }
         };
        SwingUtilities.invokeAndWait(getTextFieldText);
        System.out.println(myStrings[0] + " " + myStrings[1]);
}
```

Classe SwingWorker

```
final SwingWorker worker = new SwingWorker() {
        public Object construct() {
            //      code long à exécuter
            //      il sera lancé dans un thread séparé
             return someValue; // récupérable par get() [bloquant]
         }
        public void finished() {
            //      code court à exécuter après construct
            //      placé dans le "event-dispatching thread"
         }
};
```

Classe SwingWorker

Exemple : une procédure loadImage qui charge une image peut utiliser un SwingWorker.

```
//      ici, on a des objets Photo noté pic (pour picture)
public void actionPerformed(ActionEvent e) {
        if (icon == null) {
                loadImage(imagedir + pic.filename, current);
        }       // current est un index dans un tableau de Photo
        else {
                updatePhotograph(current, pic);
        }
}
```

Classe SwingWorker

```
private void loadImage(final String imagePath, final int index) {
        final SwingWorker worker = new SwingWorker() {
            ImageIcon icon = null;

             public Object construct() {
                icon = new ImageIcon(getURL(imagePath));
                return icon;   / /   obligado mais non utilisé ici
              }
        / / (ImageIcon utilise un MediaTracker)  …
        / / la procédure finished() sera lancée après construct
```

Classe SwingWorker

```
        // la procédure finished() est lancée
        // dans le "event-dispatching thread" quand
        // construct() est terminée
            public void finished() {
                Photo pic = (Photo)pictures.elementAt(index);
                pic.setIcon(icon);
                if (index == current)
                        updatePhotograph(index, pic);
            }
        };      // end SwingWorker
}               // end loadImage
```

Classe Timer

La classe `Timer` de javax.swing permet de lancer une action après un délai, **via un événement d'action**. (La procédure déclenchée est ainsi lancée depuis le thread de gestion des événements.)

> Cela permet aussi d'effectuer des actions répétées (réarmer le timer avec l'action).

> Utilisée pour contrôler des d'animations.

> `Timer(int millisec,`
`ActionListener thisObj)`

Classe Timer

- **setDelay(int)**
- int getDelay()
- void setInitialDelay(int)
- int getInitialDelay()
- **void setRepeats(boolean)// True**
- boolean isRepeats()
- void setCoalesce (boolean)// True
- boolean isCoalesce()

Classe Timer

- `void start()`
- `void restart()` : relance le timer. En abandonnant les événements en attente de traitement.
- `void stop()` : stoppe le timer.
- `boolean isRunning()`

exemple

```
public class AnimatorClass
                        ...  implements ActionListener {
     int number = -1;
    Timer timer;
    JLabel label; / / permet d'afficher le nombre number
                ...
    timer = new Timer(delay, this);
            ...
    public synchronized void startAnimation() {
            ...
            timer.start();
            ...
     }
```

exemple

```
public synchronized void stopAnimation() {
        … // à appeler sur un clic de souris par ex.
        timer.stop();
        ...
}
public void actionPerformed(ActionEvent e) {
        number++;
        label.setText("Animation " + number);
}

// Affichage de l'interface utilisateur puis

startAnimation();
}
```