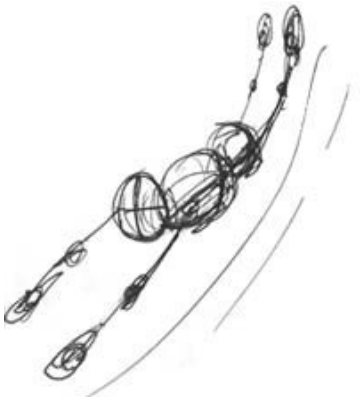
Dessiner en JAVA



→ http://java.sun.com/docs/books/tutorial/2d

Contexte Graphique: ***Graphics***

- Pour pouvoir dessinner, il faut un ***contexte graphique***, instance de la classe **`Graphics`**. Elle encapsule l'information nécessaire, sous forme d'***état graphique***.

  Celui-ci comporte :

  - la zone de dessin (le ***composant***), pour les méthodes **`draw*()`** et **`fill*()`**
  - une éventuelle translation d'origine
  - le rectangle de découpe (***clipping***)
  - la couleur courante
  - la fonte courante
  - l'opération de dessin (simple ou xor)
  - la couleur du xor, s'il y a lieu.

- Chaque composant graphique peut accéder implicitement et explicitement à un contexte graphique.

Récupérer un contexte graphique

- On obtient un *contexte graphique*
  - ***implicitement***, dans une méthode `paint()` ou `update()` : AWT construit un contexte graphique passé en paramètre,
  - ***explicitement***, dans un composant ou dans une image, par `getGraphics()`

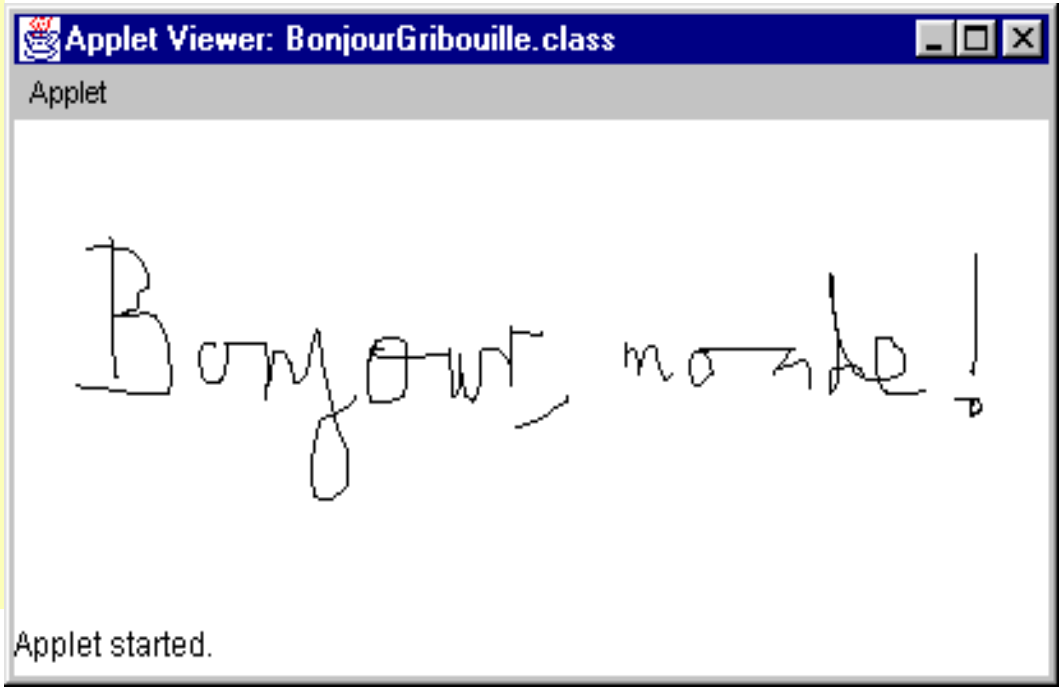
- Un contexte graphique utilise des ressources systèmes. L'acquisition explicite doit être accompagnée, *in fine*, par une *libération explicite* au moyen de `dispose()`.

- L'acquisition explicite d'un contexte graphique est souvent le signe d'une programmation *maladroite*. Normalement les opérations de dessin se font dans la méthode paint des objets graphiques.

Exemple : application de griboullage

```
public class BonjourGribouille
  extends Applet {
  int xd, yd;

  public void init() {
    addMouseListener(new Appuyeur());
    addMouseMotionListener(new Dragueur());
  }
  class Appuyeur extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      xd = e.getX(); yd = e.getY();
    }
  }
  class Dragueur extends MouseMotionAdapter
   {
    public void mouseDragged(MouseEvent e) {
      int x = e.getX(), y = e.getY();
      Graphics g = getGraphics();
      g.drawLine(xd, yd, x, y);
      xd = x; yd = y;
      g.dispose();
    }
  }
}
```

- Accès *explicite* à un contexte graphique.
- A chaque `getGraphics()`, un *nouveau* contexte est fourni, ne connaissant rien du précédent.



Affichage d'un composant

- À la demande du système
  - quand le composant est rendu visible pour la première fois
  - quand le composant a été recouvert puis découvert

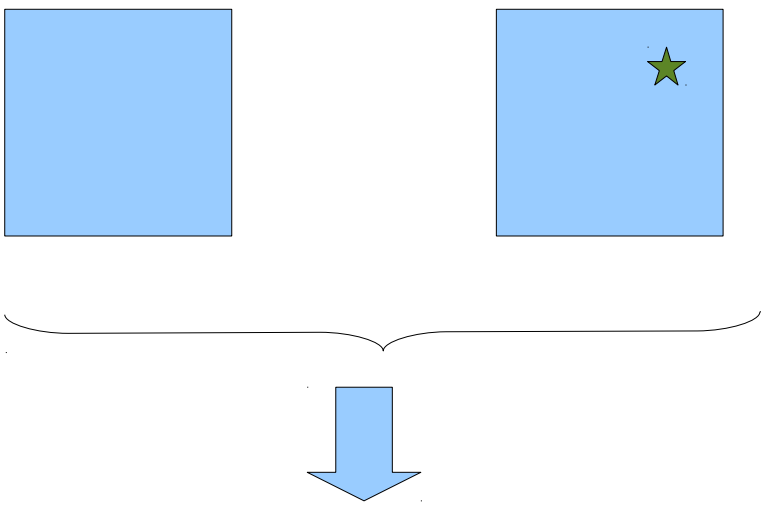
→ appel de la méthode ***paint***() du composant

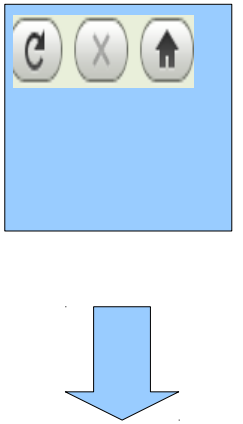
- À la demande de l'application
  - quand le programme ou un composant Swing détermine que le composant doit être ré-affiché
    - en interne dans les Swing (changement d'un texte, d'une couleur, ...)
    - dans votre propre programme en faisant une demande explicite de ré-affichage

→ appel de la méthode ***repaint***() du composant

Que fait la méthode ***paint*** ?

1) dessiner le fond 2) dessin spécifique 3) dessiner bordure 4) dessiner les fils





`paintComponent``(Graphics g)`

Dessine le composant lui-même
• déjà implémentée pour les composants standards (fenêtres, boutons, …)
→ **doit être redéfinie (overriden) pour créer vos propres composants**

`paintBorder``(Graphics g)`

Dessine bordures ajoutées au composant (en utilisant setBorder)

`paintChildren``(Graphics g)`

Ne pas appeler directement cette méthode ni la redéfinir.

Méthode repaint

- Reste la ***méthode par excellence*** pour rafraîchir un affichage !

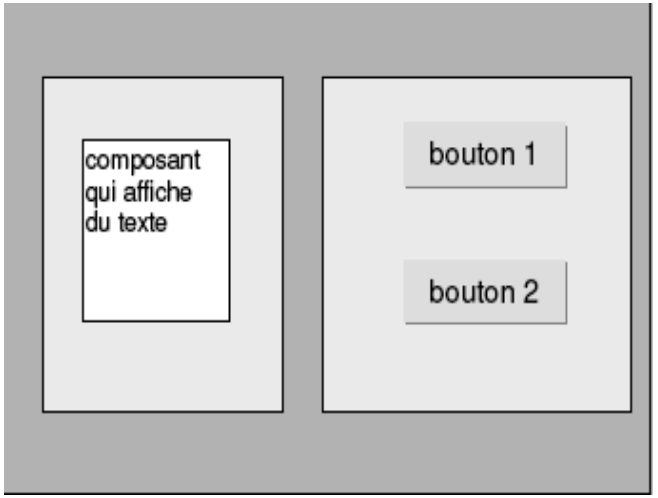
Code simplifié de ComponentUI.java

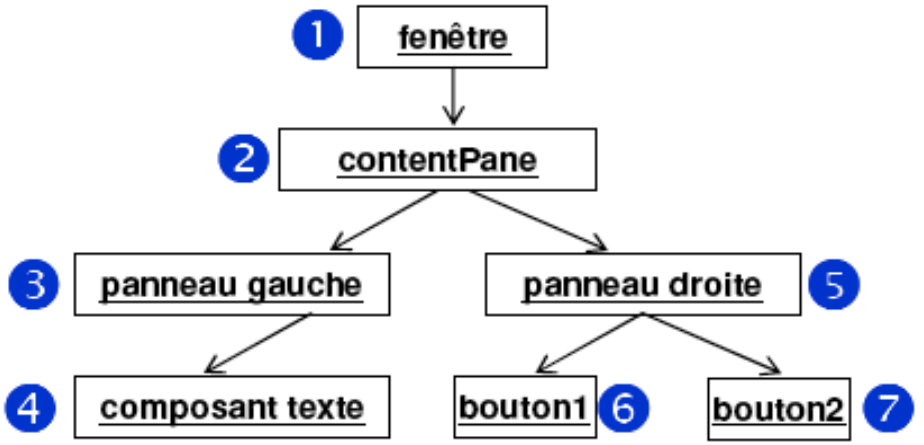
```
public void update(Graphics g, JComponent c) {
  if (c.isOpaque()) {
    g.setColor(c.getBackground());
    g.fillRect(0,0, c.getWidth(),
   c.getHeight());
  }
  paint(g, c);
}
```

- **`repaint()`** poste un appel à **`update()`**. Plusieurs appels peuvent être groupés.
- **`update()`** appelle **`paint()`**.
- **`paint()`** appelle successivement
  - `paintComponent()` pour le dessin (le `paint()` en AWT)
  - `paintBorder()`
  - `paintChildren()`.
- **`paintComponent()`** par défaut appelle **`ComponentUI.update()`** qui efface et redessine le fond ***si le composant est opaque*** (**`JPanel`** l'est par défaut).
- Pour dessiner, on redéfinit **`paintComponent()`** et il est utile d'appeler **`super.paintComponent()`**.

Affichage récursif des composants

- Un composant se dessine avant chacun des composants qu'il contient
  - affichage d'une interface Swing s'effectue récursivement en descendant la hiérarchie des containers





En résumé

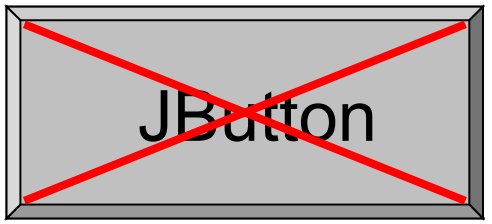
- Chaque composant graphique possède une méthode qui définit comment il doit se dessiner
  - public void **paint**(Graphics g) pour les composants awt
  - public void **paintComponent**(Graphics g) pour les composants ***Swing***
- Pour les composants standards (fenêtres, boutons, …) il est inutile de définir comment ils doivent s'afficher
  - une fenêtre affichera son cadre et son fond puis affichera tout les composants qu'elle contient
  - un conteneur affichera son fond puis affichera récursivement tous les composants qu'il contient

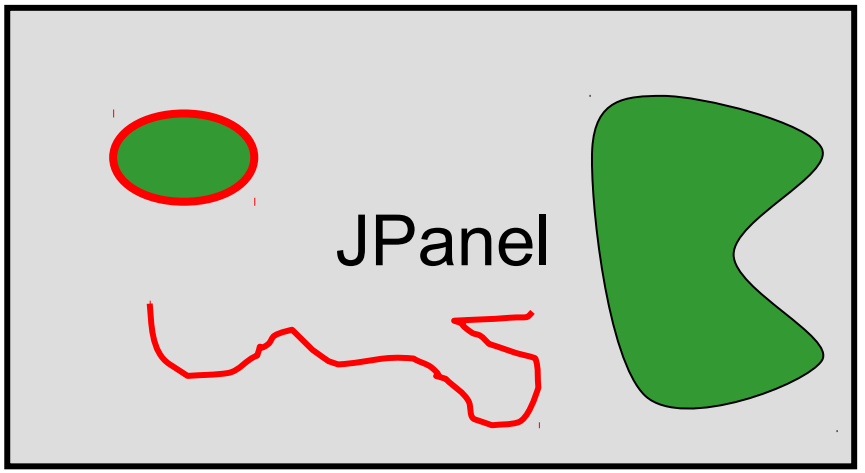
Mais dès que l'application gère ses propres graphiques via un contexte graphique (objet *Graphics*) elle devra se soucier de leur rafraichissement et redéfinir *paintComponent !!!*

Dessiner sur un Composant

- Ne pas dessiner sur un composant standard...

  → ***JPanel*** est le composant de choix pour faire du dessin





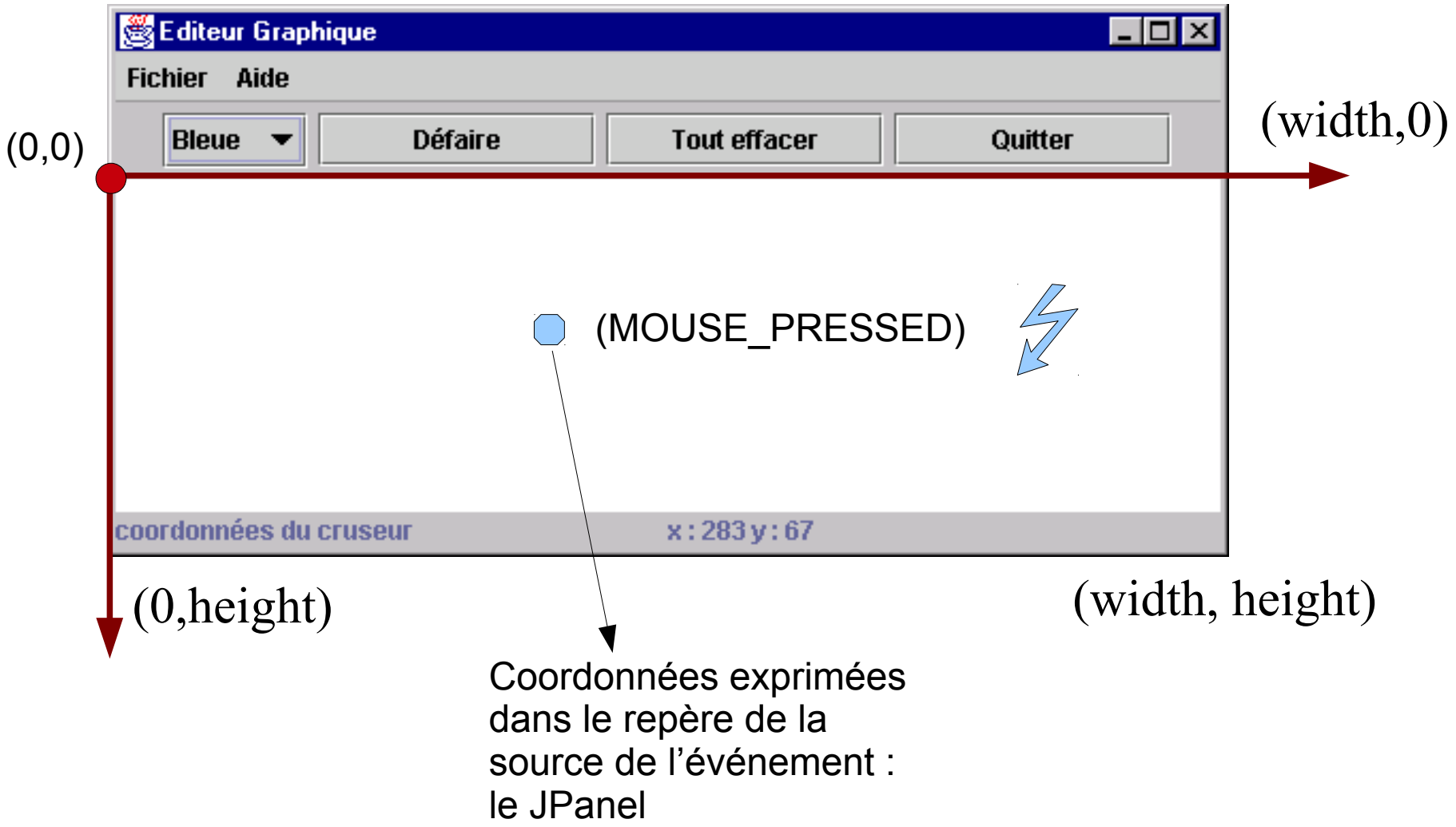
Exemple

```
class Dessin extends JPanel {
  int theta = 45, del =45;
  public void paintComponent(Graphics g) {
    int largeur = getSize().width;
    int hauteur = getSize().height;
    int dl = largeur/2, dh = hauteur/2;
    int [] polx = { 0, dl, largeur, dl};
    int [] poly = {dh, 0, dh, hauteur};
    Polygon pol = new Polygon(polx,poly,4);

    g.setColor(Color.black);
    g.fillRect(0,0,largeur,hauteur);
    g.setColor( Color.yellow);
    g.fillPolygon(pol);
    g.setColor( Color.red);
    g.fillRect(dl/2, dh/2, dl,dh);
    g.setColor( Color.green);
    g.fillOval(dl/2, dh/2, dl,dh);
    g.setColor( Color.blue);
    g.fillArc(dl/2, dh/2, dl, dh,theta,
   del);
  }
...
```

```
public class Losange extends JApplet
   {
  public void init(){
            setContentPane(new
   Dessin());
   }
```

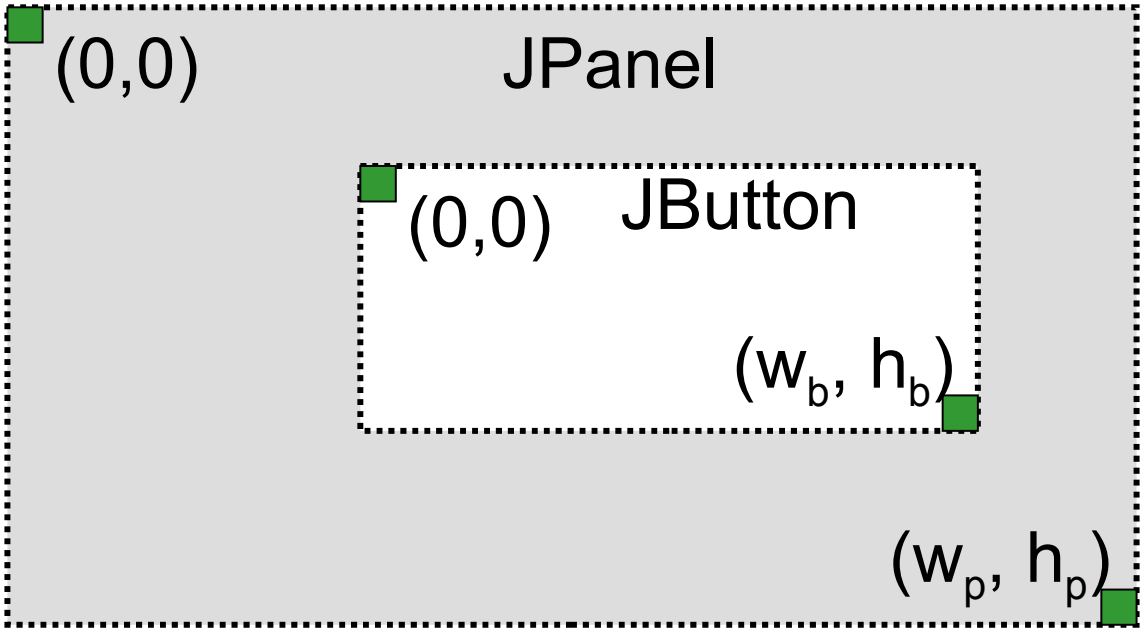
```
...
   public Dessin() {
    addMouseListener( new MouseAdapter() {
      public void mousePressed(MouseEvent e)
   {
        theta = (theta + 10)%360;
        repaint();
      }
    });
   }
}
```

Système de coordonnées



Coordonnées dans une hiérarchie de fenêtres

- Chaque composant d'une interface à sa propre fenetre
  - Qui est une zone rectangulaire à l'intéreur du composant parent
  - Qui possède son propre système de coordonnées



Couleurs

- Constants (ORANGE, PINK, CYAN, MAGENTA, YELLOW, BLACK, WHITE, GRAY, RED, GREEN, BLUE)
- Example
  - `g.setColor(Color.MAGENTA);`
- RGB (Red Green Blue Values)
  - 0 to 255 (0 no color, 255 full color)
  - 0.0 to 1.0
- Example
  - `g.setColor(new Color(255, 0 ,0));`
  - `g.setColor (new Color (0.0f, 1.0f, 0.0f)`

Gestion des polices

- Java Font Styles (PLAIN, BOLD, ITALIC)
- Java Fonts
  - Serif (Times)
  - Monospaced (Courier)
  - SanSerif (Helvetica)
- Examples:
  - `g.setFont(new Font (“Serif”, Font.BOLD, 12));`
  - `g.setFont(new Font(“Monospaced”, Font.ITALIC + Font.BOLD, 24)`

Drawing Lines, Rectangles, Ovals

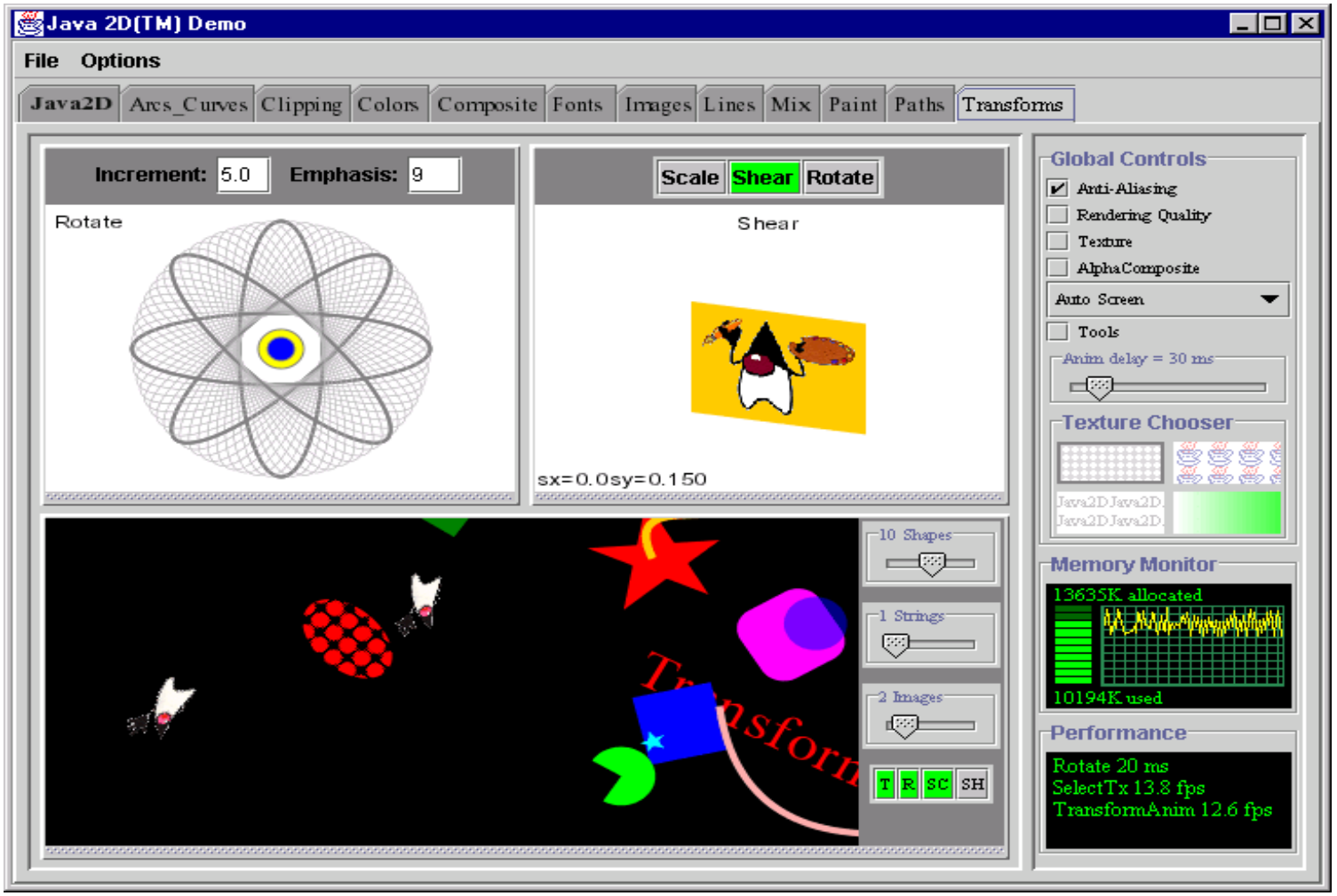
- Basic Drawing Methods
  - drawLine (x1, y1, x2, y2)
  - drawRect (x, y, width, height)
  - drawOval (x, y, width, height)
    - (x,y) is top-left corner of bounding box of oval
- Fill Shapes
  - Call setColor method before filling shapes
  - fillRect (x, y, width,height)
  - fillOval(x, y, width, height)

De AWT à Java2D

- Sérieuses limitations des possibilités graphiques de AWT
  - primitives graphiques limitées (lignes, rectangles, ovales...)
  - dessin des lignes avec épaisseur d'un seul pixel
  - peu de polices de caractères disponibles
  - pour appliquer une rotation ou une translation à quelque chose il faut le faire soi-même
  - support rudimentaire pour les images
  - contrôle de la transparence très difficile …
- Introduction de nouvelles API pour le graphique avec version 2 de Java : « Java 2D »

→ ***Graphics2D*** fourni des possibilités de dessin beaucoup plus élaborées

Démonstration des possibilités de ***Graphics2D***

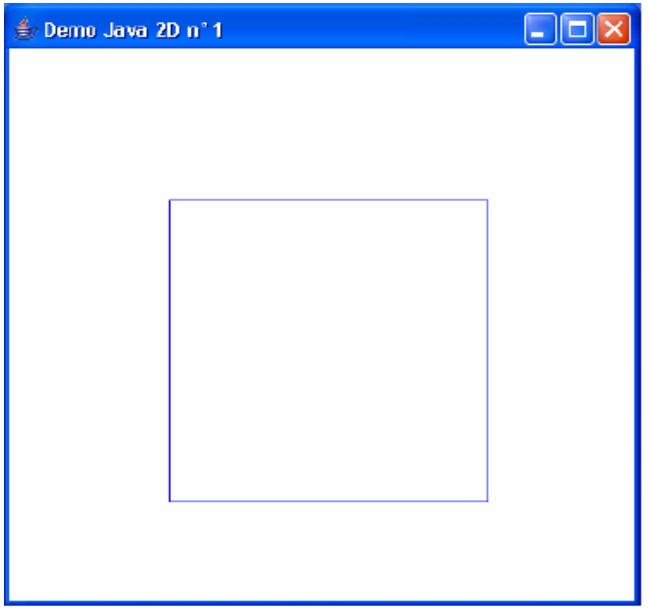


programme de démonstration dans $JAVA_HOME/demo/jfc/JAVA2D/Java2Demo.jar

Exemple

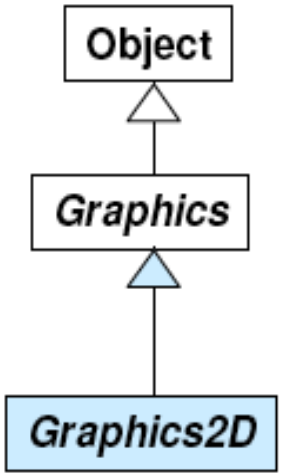
```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame; import javax.swing.JPanel;

 public class DemoJava2D1 extends JPanel(
   public DemoJava2D_1() {
     setBackground(Color.WHITE);
   }

   public void paintComponent(Graphics g) {
       super.paintComponent(g);
       g.setColor(Color.BLUE);
       g.drawRect(100,100,100,100);
   }

   public static void main(String[] args) {
       JFrame f = new JFrame("Demo Java 2D n°1");
       f.addWindowListener(new WindowAdapter() {
              public void windowClosing(WindowEvent we){
                   System.exit(0);
              }
           });
   f.add(new DemoJava2D_1());
   f.setSize(new Dimension(400,400));
   f.setVisible(true);
  }
}
```

Pour illustrer les possibilités de l'API Java2D, on va débuter par un petit programme de dessin qui au départ n'utilise pas Java 2D et qui sera ensuite modifié



Utiliser un ***Graphics2D***

- public void ***paintComponent***(***Graphics*** g)



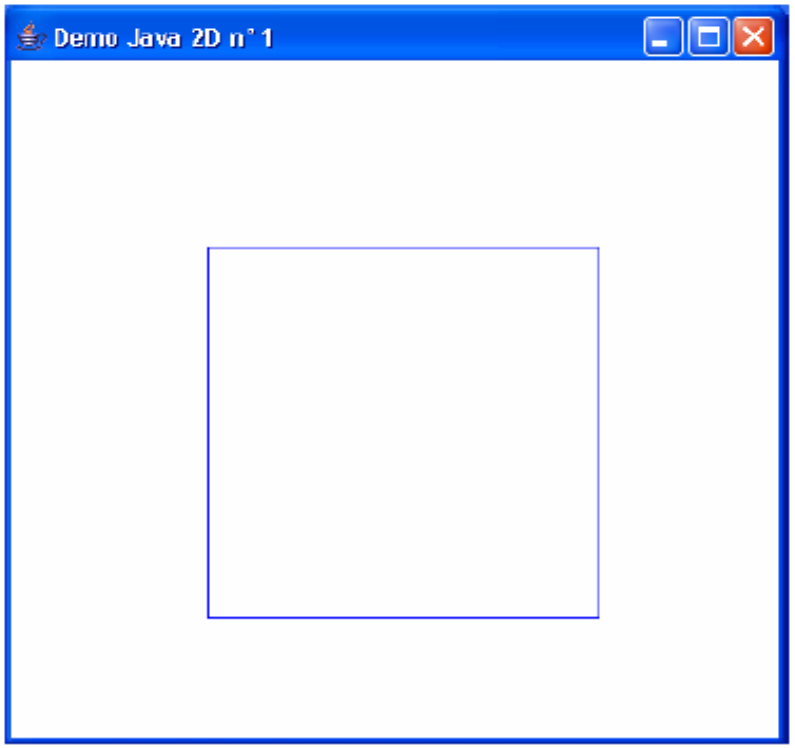
En fait g est un objet Graphics2D
(Graphics2D est une sous classe de Graphics)
On utilise le type Graphics pour des raisons de compatibilité avec les versions antérieures de l'API java

Si on veut utiliser les fonctionnalités de Graphics2D il suffit de « downcaster » g

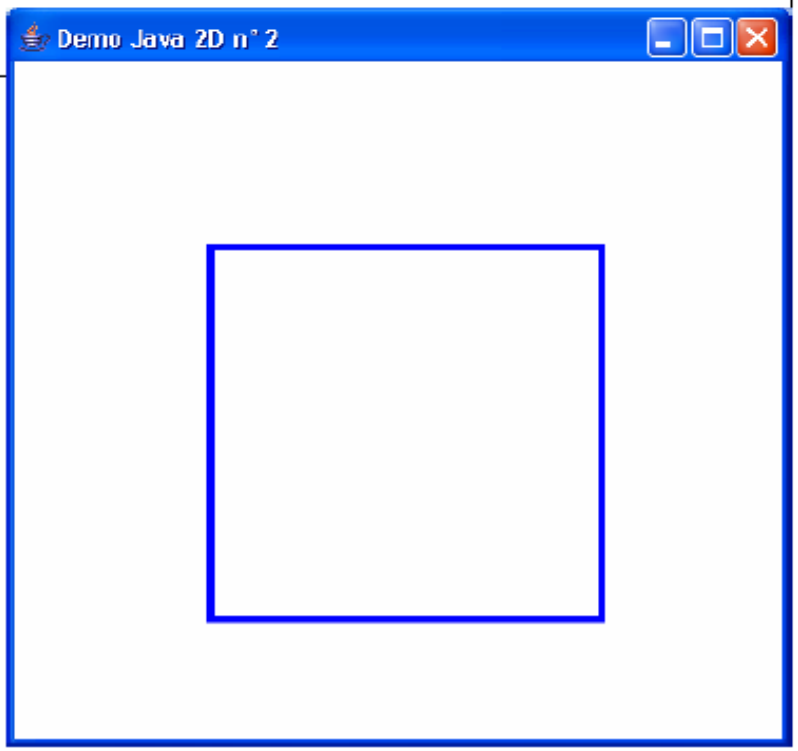
```
Graphics2D g2D = (Graphics2D) g;
g.draw(new Polygone(tabX, tabY, nbPts));
```

Exemple → Java2D

```
...

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  g.setColor(Color.BLUE);
  g.drawRect(100,100,100,100);
 }

...
```



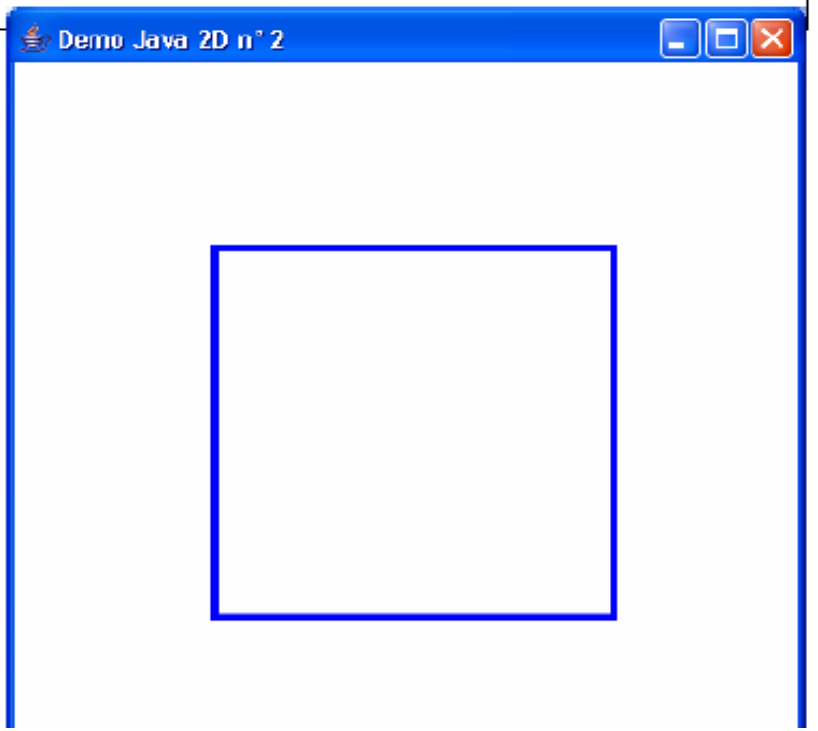

```
...

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;
  g2d.setColor(Color.BLUE);
  g2d.setStroke(new BasicStroke(4.0f));
  g2d.drawRect(100,100,100,100);
 }

...
```



Utilisation de la classe ***Shape***

1. Initialise le contexte graphique
2. Défini et dessine un objet (Rectangle)

```
...
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;
  g2d.setColor(Color.BLUE);
  g2d.setStroke(new BasicStroke(4.0f));
  g2d.drawRect(100,100,100,100);
}
...
```



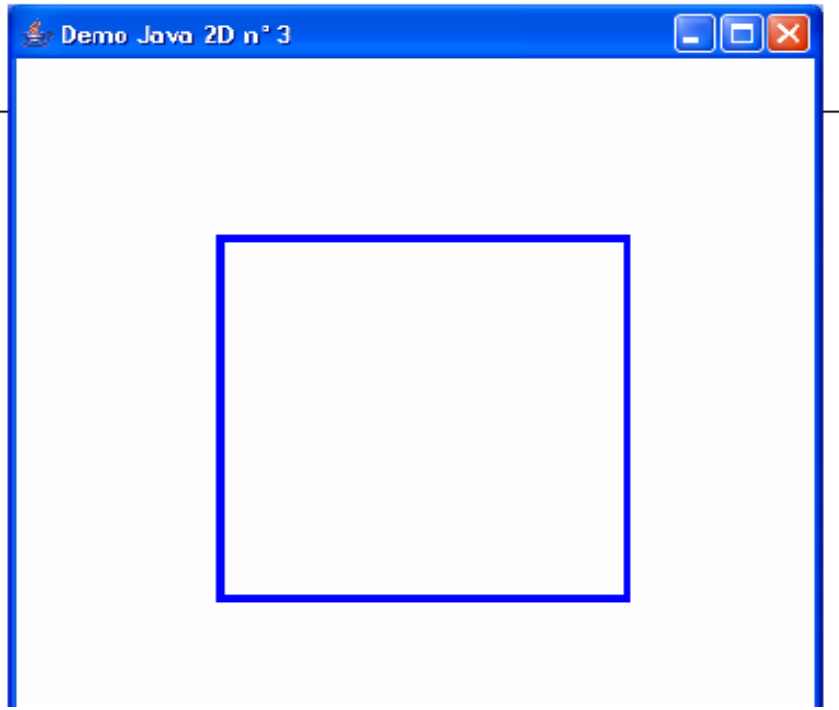
Dans l'API Java 2D l'affichage peut être séparé de la définition de l'objet.

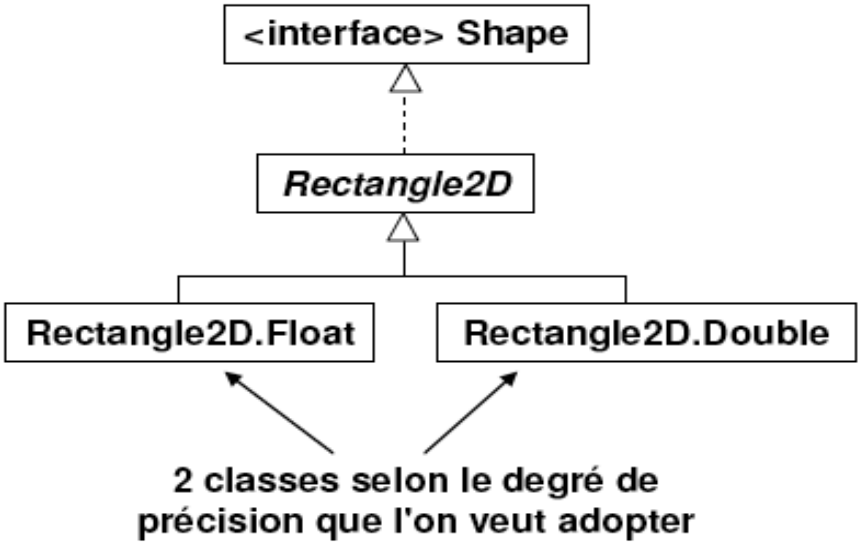
- *en général 3 étapes pour dessiner des objets*
  1. *Initialiser le contexte graphique*
  2. *Définir l'objet à dessiner*
  3. *Appeler l'une des méthodes de rendu de* ***`Graphics2D`*** *pour afficher l'objet*
- *Les objets implémentent l'interface* ***`java.awt.Shape`***
- *Le package* ***`java.awt.geom`*** *propose différentes classes d'implémentation de* ***`Shape`***

Utilisation de la classe ***Shape***

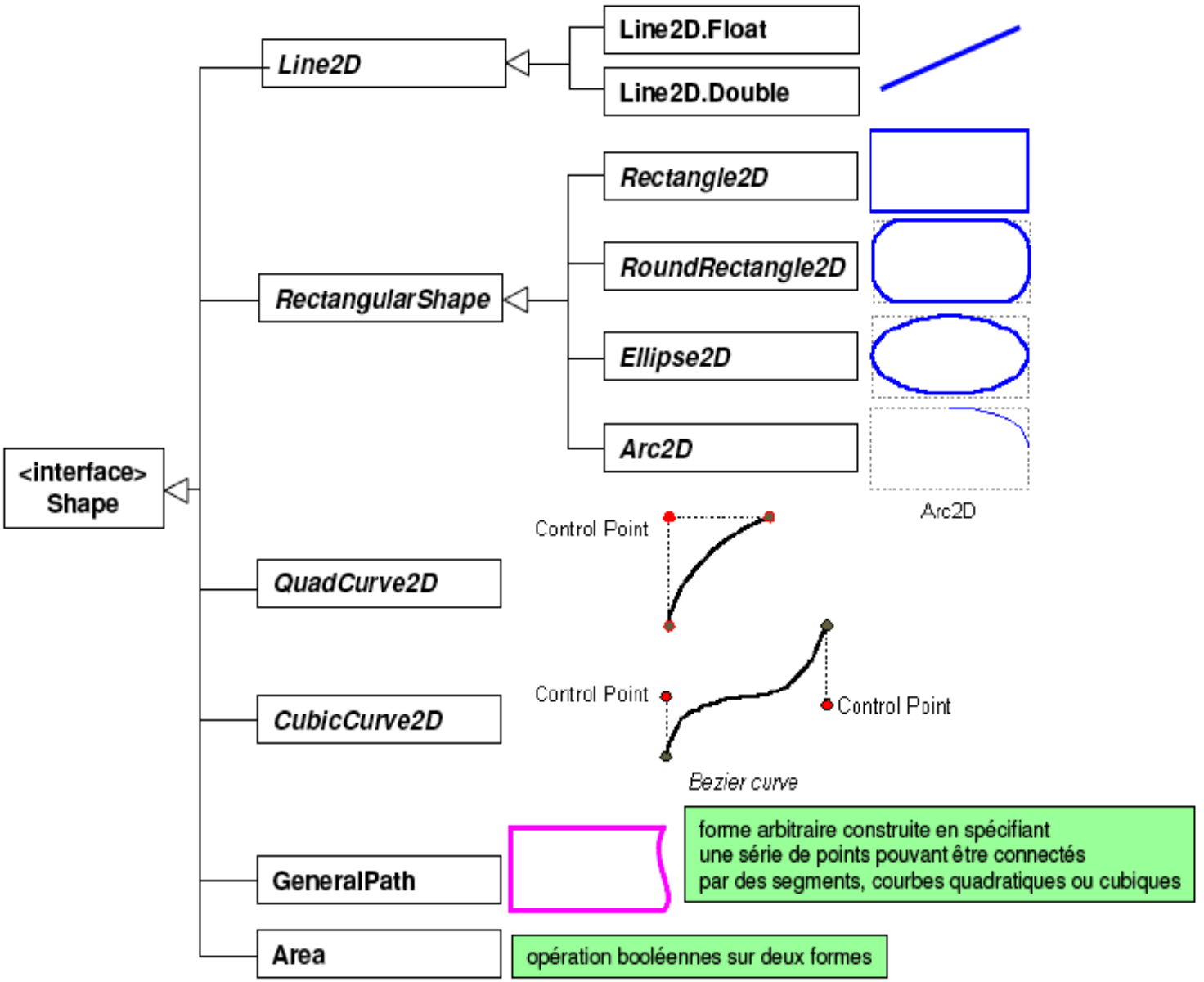
1. Initialise le contexte graphique
2. Défini un objet (Rectangle2D)
3. Appel de l'une des méthodes de rendu de Graphics2D

```
...
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D) g;
  g2d.setColor(Color.BLUE);
  g2d.setStroke(new BasicStroke(4.0f));
  Rectangle2D r = new Rectangle2D.Float(100.0f,
            100.0f, 200.0f, 200.0f);
  g2d.draw(r);
}
...
```



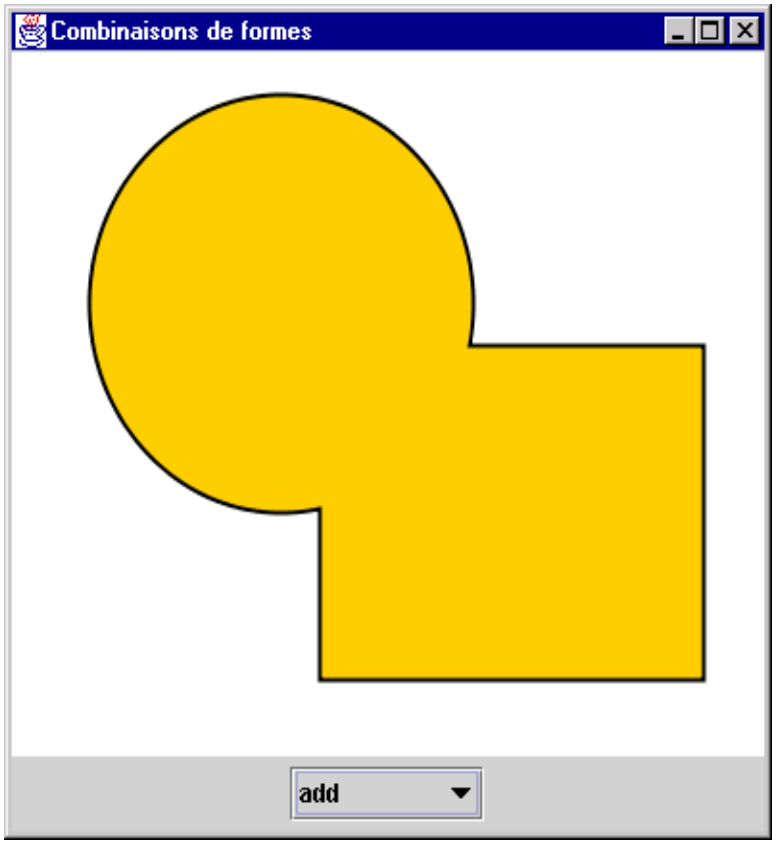


Shapes : package java.awt.geom



Composition de Formes (*Shapes*)

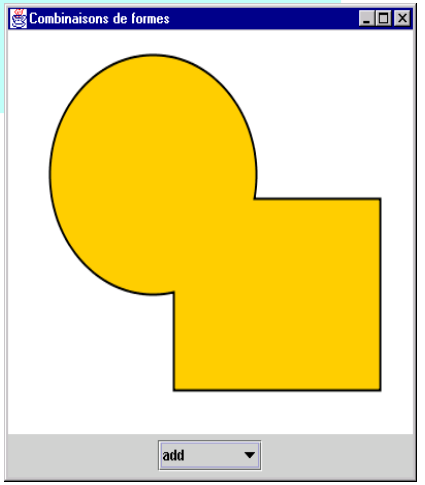
- Formes que l'on peut composer par des opérations booléennes (***constructive solid geometry*** en 2D)
- Toute forme est un composant
- Les opérations sont
  - add
  - subtract
  - intersect
  - ...



Java 2D also provides an Area class that allows you to create new shapes by combining existing objects that implement the Shape interface.

Un exemple de composition

```
public void paintComponent(Graphics g) {
  Graphics2D g2 = (Graphics2D)g;
  ...
  Area areaOne = new Area(ellipse);
  Area areaTwo = new Area(rectangle);
  if (option.equals("add")) areaOne.add(areaTwo);
  else if (option.equals("intersection")) areaOne.intersect(areaTwo);
  else if (option.equals("subtract")) areaOne.subtract(areaTwo);
  else if (option.equals("exclusive or")) areaOne.exclusiveOr(areaTwo);

  g2.setPaint(Color.orange);
  g2.fill(areaOne);
  g2.setPaint(Color.black);
  g2.draw(areaOne);
}
```



Dessiner et remplir une forme

- **Painting** : remplir l'intérieur d'une forme (shape) avec une couleur, un dégradé ou une texture , méthode *fill*() de Graphics2D
- **Stroking** : dessiner le contour (outline) d'une forme en spécifiant épaisseur du trait (line width), le style de trait, méthode *draw*() de Graphics2D

```
Graphics2D g2 = (Graphics2D)g;
double x = 15, y = 50, w = 70, h = 70;
Ellipse2D e = new Ellipse2D.Double(x, y, w, h);
```

```
GradientPaint gp = new GradientPaint(75, 75,
   Color.white,95, 95, Color.gray, true);
// Fill with a gradient.
g2.setPaint(gp); 1
g2.fill(e);      2
```

```
// Stroke with a solid color.
e.setFrame(x + 100, y, w, h);
g2.setStroke(new BasicStroke(8)); 1
g2.setPaint(Color.black);         2
g2.draw(e);                       3
```

```
// Stroke with a gradient.
e.setFrame(x + 200, y, w, h);
g2.setPaint(gp);
g2.draw(e);
```

remplir une forme

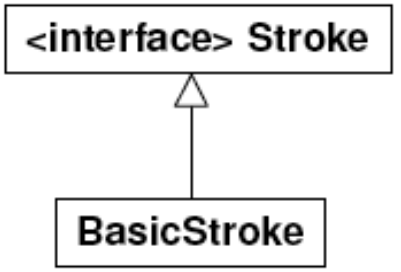
1. indique au contexte graphique comment remplir les formes. gp un objet qui implémente interface `java.awt.Paint`
2. demande au contexte graphique de remplir la forme en la passant à la méthode fill

dessiner le contour d'une forme

1. indique au contexte graphique comment dessiner le contour des formes à l'aide d'un objet qui implémente interface `java.awt.Stroke`
2. indique au contexte graphique comment remplir le contour à l'aide d'objet qui implémente interface `java.awt.Paint`
3. demande au contexte graphique de dessiner le contour de la forme en la passant à la méthode draw

Stroke (coup de pinceau)

- **setStroke(Stroke s)**



- **Epaisseur du trait (line width) c'est un *float* (1.0 environ 1 pixel 1/72 pouce lorsque transformation par défaut est appliquée)**

- **style de jointure (joint style) manière dont segments se raccordent**

- **style de terminaison (end-cap) manière dont les extrémité des segments sont affichées**

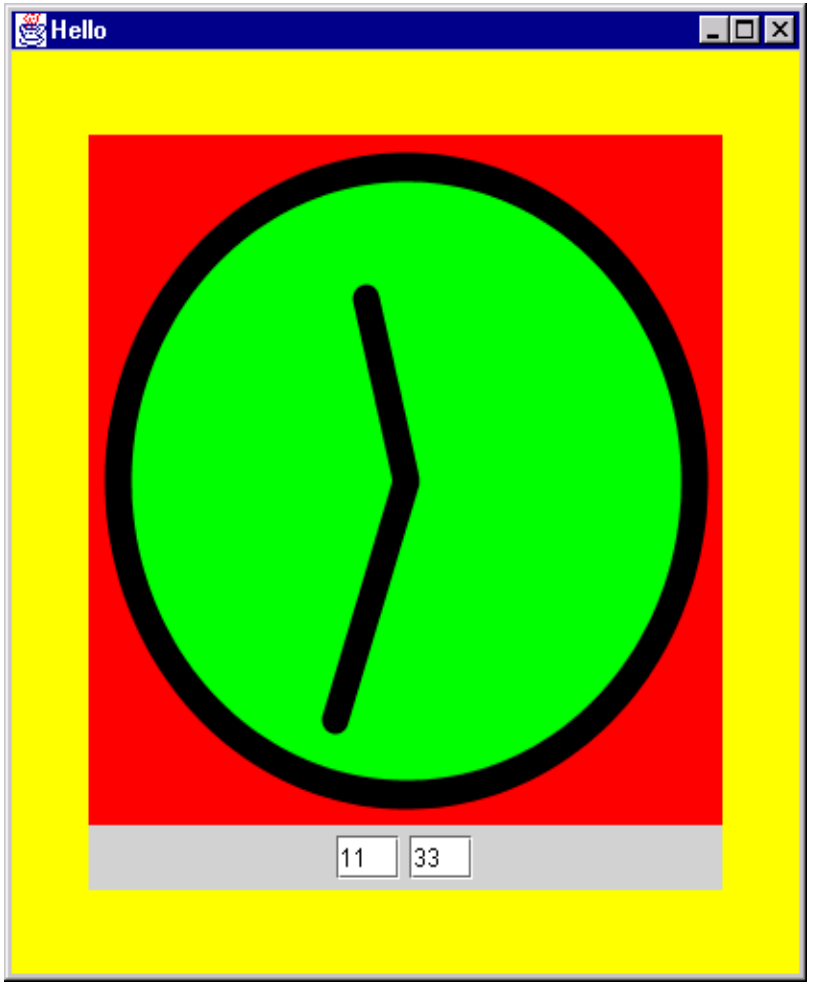
- **style de pointillé (dash style)**

Exemple Horloge

- Les traits se dessinent avec une *plume* de la l'interface Stroke, implémentée par BasicStroke.
- Les attributs sont
  - l'épaisseur (width)
  - fins de traits (end caps)

    `CAP_BUTT, CAP_ROUND, CAP_SQUARE`
  - lien entre traits (join caps)

    `JOIN_BEVEL, JOIN_MITER, JOIN_ROUND`
  - pointillé (dash)
- Par défaut
  - trait continue d'épaisseur `1, CAP_SQUARE, JOIN_MITER`, miter limit 10



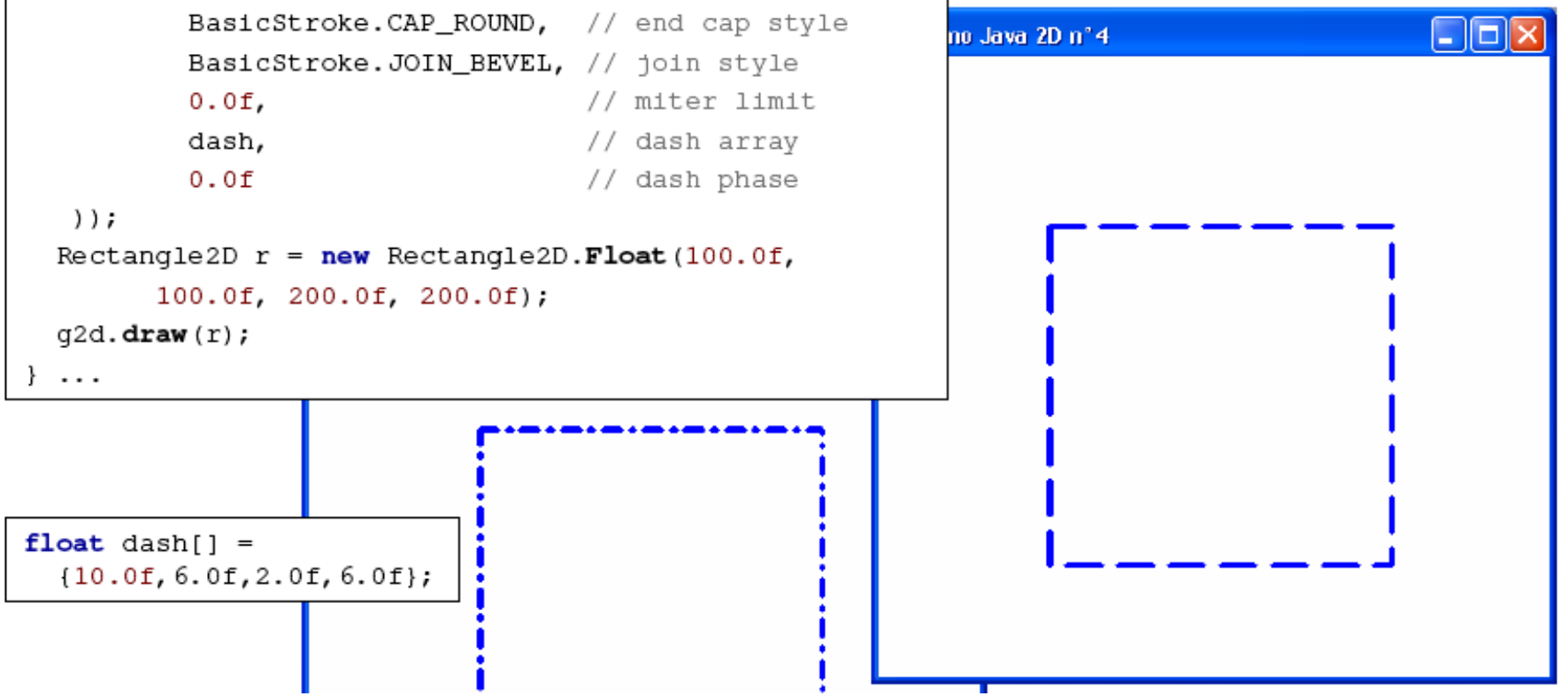

```
g2.setStroke(new BasicStroke(14,
    BasicStroke.CAP_ROUND, BasicStroke.JOIN_BEVEL));
```

Un autre exemple...

```
...
public void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2d = (Graphics2D) g;
   g2d.setPaint(Color.BLUE);
   float dash[] = {20.0f, 10.0f};
   g2d.setStroke(new BasicStroke(
          4.0f,                      // line width
          BasicStroke.CAP_ROUND,     // end cap style
          BasicStroke.JOIN_BEVEL,    // join style
          0.0f,                      // miter limit
          dash,                      // dash array
          0.0f                       // dash phase
   ));
  Rectangle2D r = new Rectangle2D.Float(100.0f,
        100.0f, 200.0f, 200.0f);
  g2d.draw(r);
} ...
```

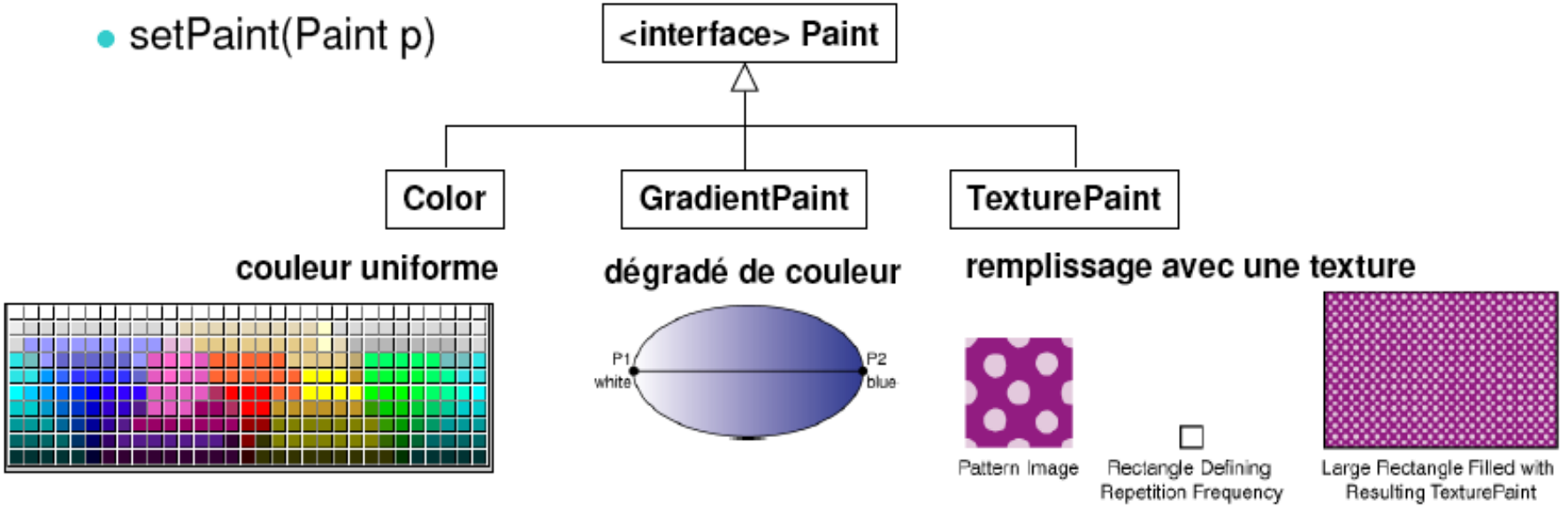
Tableau de flottant qui représente la longueur des sections tracées et non tracées du pointillé
- index pair (trait plein)
- index impair (espace)

```
float dash[] =
  {10.0f,6.0f,2.0f,6.0f};
```



La méthode ***setPaint***

- setPaint(Paint p)



couleurs spécifiées par composantes primaires Rouge, Vert et Bleu

```
public Color(int r, int g, int b)   0 .. 255
public Color(float r, float g, float b)   0.0 .. 1.0
public Color(int rgb)
```

Color.RED	255 0 0
Color.GREEN	0 255 0
Color.BLUE	0 0 255
Color.WHITE	255 255 255
Color.BLACK	0 0 0
Color.CYAN	0 255 255
Color.MAGENTA	255 0 255
Color.YELLOW	255 255 0

Par défaut couleurs définies comme opaques
Couleur partiellement transparente définie à l'aide canal alpha

```
public Color(int a, int r, int g, int b)   0 .. 255
public Color(float a, float r, float g, float b)   0.0 .. 1.0
public Color(int argb, boolean hasAlpha)
```

opaque

transparent

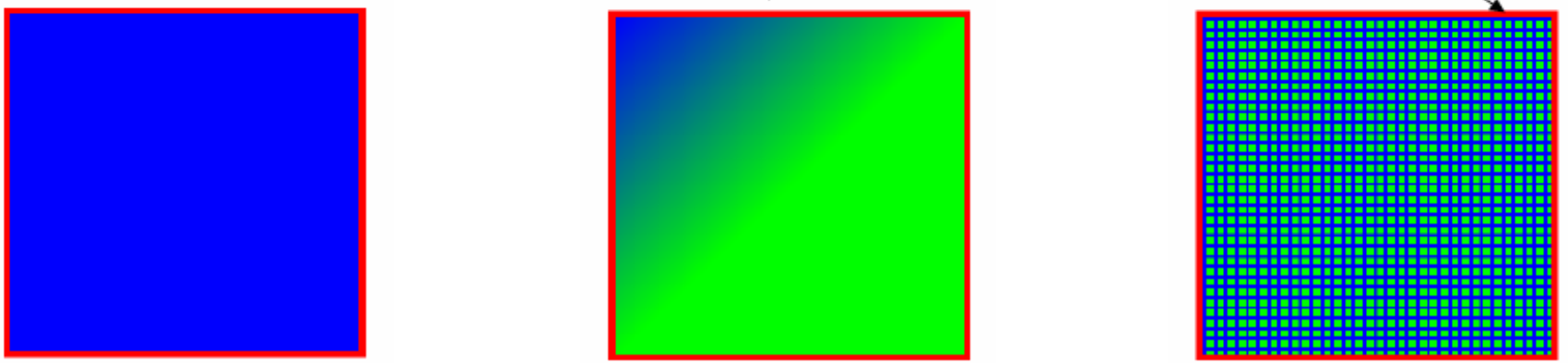
Exemple d'utilisation de ***setPaint***

```
g2d.setStroke(new BasicStroke(4.0f));
Rectangle2D r = new Rectangle2D.Float(100.0f, 100.0f, 200.0f, 200.0f);
g2d.setPaint(…);
g2d.fill(r);
g2d.setPaint(Color.RED);
g2d.draw(r);
```

Color

GradientPaint

TexturePaint

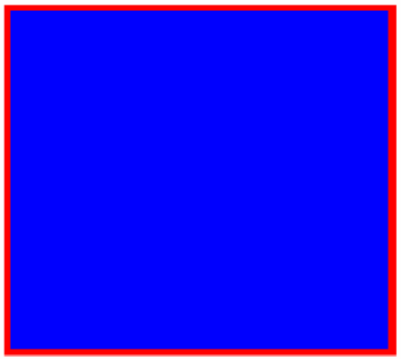


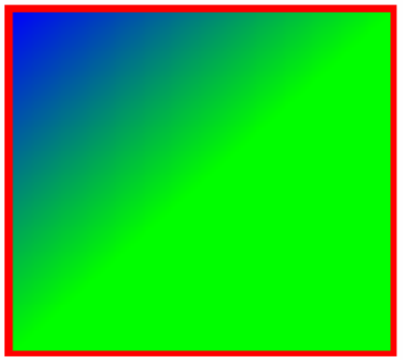
Exemple couleur unie

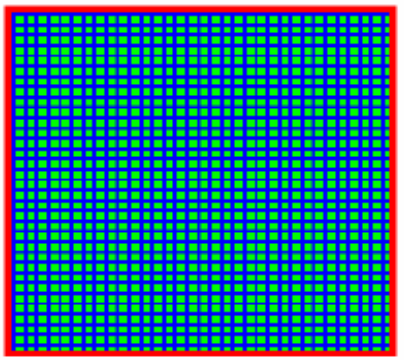
```
g2d.setStroke(new BasicStroke(4.0f));
Rectangle2D r = new Rectangle2D.Float(100.0f, 100.0f, 200.0f, 200.0f);
g2d.setPaint(…);
g2d.fill(r);
g2d.setPaint(Color.RED);
g2d.draw(r);
```

```
g2d.setPaint(Color.BLUE);
```

```
GradientPaint gp = new GradientPaint(
        100.f, 100.f, // starting point
        Color.BLUE,   // starting color
        200.f, 200.f, // ending point
        Color.GREEN // ending color );
g2d.setPaint(gp);
```







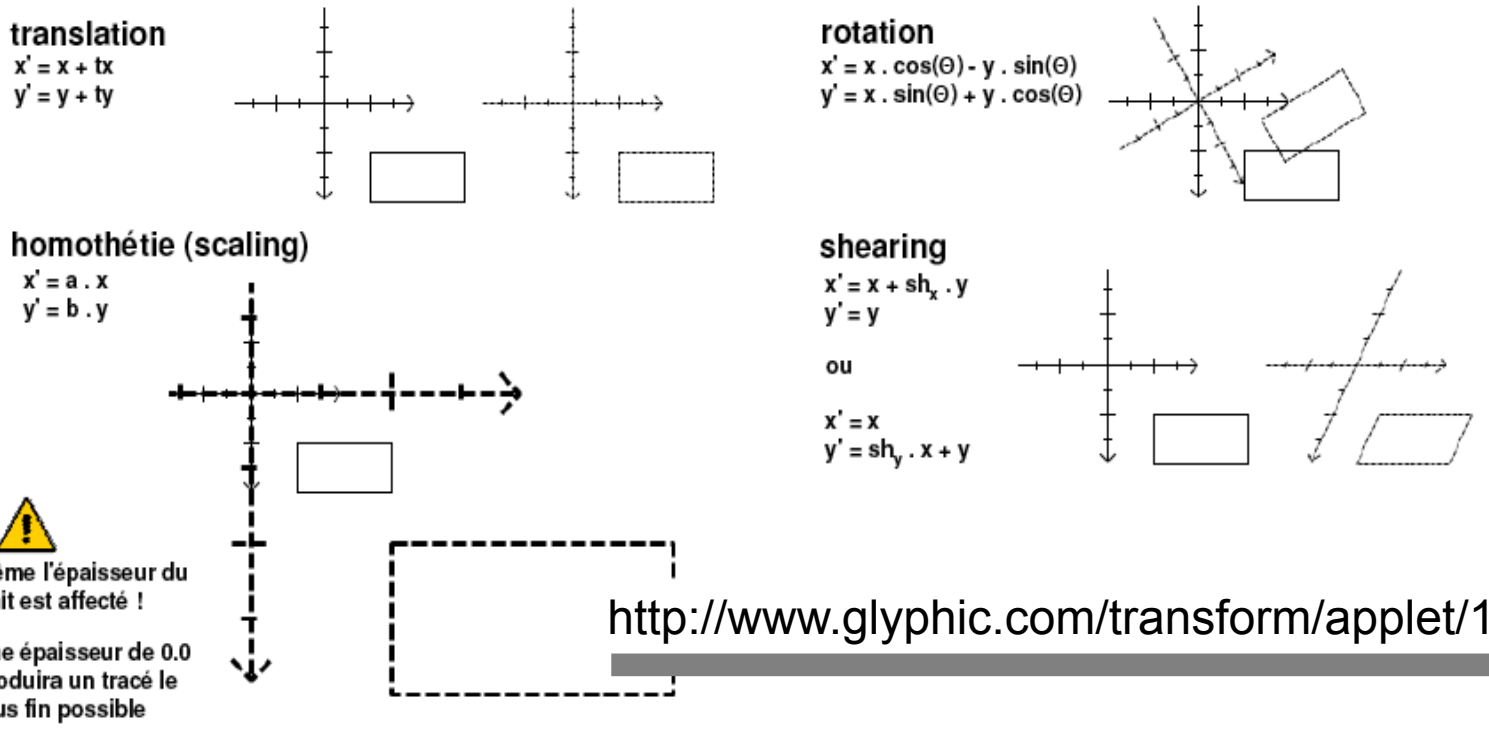
Un mot sur le clipping

- On peut restreindre la zone à (re)dessiner de deux manières
  - par la définition d'une ***forme de découpe*** (clipping)
  - par la spécification d'un rectangle de rafraîchissement dans la méthode `repaint()`.
- Un contexte graphique contient un rectangle de découpe
  - initialement toute la zone de dessin
  - modifiable par `setClip()` le rectangle ne peut que diminuer
- La méthode `repaint()` de `Component` peut prendre en argument un rectangle, et limiter ainsi l'action à ce rectangle.

```
repaint()
repaint(long tm)
repaint(int x, int y, int width, int height)
repaint(long tm, int x, int y, int width, int height)
```

Transformation géométriques

- ***Graphics2D*** maintient une transformation géométrique appliquée au primitives géométriques avant qu'elles ne soient rendues.
- Attribut défini comme une instance de ***AffineTransform***
  - transformation affine : lignes parallèles restent parallèles
  - composition de transformations élémentaires



http://www.glyphic.com/transform/applet/1intro.html

Interaction avec l'utilisateur

- Pour interagir avec les dessins affichés, nécessité de déterminer lorsque l'utilisateur clique sur l'un d'eux.
- Il y a deux manières de procéder :
  - en utilisant la méthode ***hit*** de ***Graphics2D***
    - ***public abstract boolean hit(Rectangle rect, Shape s, boolean onStroke)***

      → ***true*** si la forme s intersecte le rectangle rect exprimé en coordonnées écran (device coordinates)
  - en utilisant la méthode ***contains*** de l'objet ***Shape***
    - ***public boolean contains(double x, double y)***

      → ***true*** si la forme s contient le point x, y exprimé en coordonnées écran (device coordinates)