



Interface Graphique en Java 1.6

Dessin

Sébastien Paumier



Le contexte graphique

- pour dessiner en Swing, il faut utiliser un contexte graphique
- un objet **Graphics** permet de:
 - dessiner dans le composant concerné (**JComponent** ou **Image**)
 - définir la couleur courante: **set/getColor()**
 - définir la fonte courante: **set/getFont()**
 - gérer une translation:
translate(int x,int y)



Les primitives de dessin

- chacune des méthodes suivantes utilise l'état courant:
 - `drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`
 - `drawImage(Image img, int x, int y, ImageObserver observer)`
 - `drawLine(int x1, int y1, int x2, int y2)`
 - `drawOval(int x, int y, int width, int height)`
 - `drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`
 - `drawRect(int x, int y, int width, int height)`
 - `drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`



Les primitives de dessin

- et puis aussi:
 - `drawString(String str, int x, int y)`
 - `fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`
 - `fillOval(int x, int y, int width, int height)`
 - `fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`
 - `fillRect(int x, int y, int width, int height)`
 - `fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
- et bien d'autres encore!



Récupérer un contexte

- 1 façon implicite:
 - récupérer le **Graphics** passé à la méthode **paintComponent** d'un **JComponent**
- 2 façons explicites:
 - récupérer le **Graphics** d'un composant ou d'une image avec **getGraphics()**
 - obtenir une copie d'un **Graphics** existant avec **createGraphics()**



Récupérer un contexte

- les ressources système pour le dessin pouvant être limitées, il faut faire très attention à bien libérer les contextes obtenus explicitement avec `dispose()`
- il faut le faire **rapidement**
- la création explicite de contextes n'est pas un usage habituel !!!!
 - ne pas le faire sans y avoir bien réfléchi



Comment dessiner

- avec **paintComponent**: on dessine itérativement, en changeant les paramètres du **Graphics** si nécessaire

```
public class Pacman extends JComponent {  
  
    @Override protected void paintComponent(Graphics g) {  
        /* super.paintComponent(g); here is useless since  
         * we don't extend an existing component */  
        super.paintComponent(g);  
        if (isOpaque()) {  
            g.setColor(getBackground());  
            g.fillRect(0,0,getWidth(),getHeight());  
        }  
        g.setColor(Color.YELLOW);  
        g.fillArc(0,0,getWidth(),getHeight(),35,280);  
        g.setColor(Color.BLACK);  
        g.fillOval((int)(getWidth()*0.65),(int)(getHeight()*0.15),  
                  (int)(getWidth()*0.08),(int)(getHeight()*0.08));  
    }  
}
```





Contrat d'opacité

- si on n'hérite pas d'un composant existant et si le composant ne remplit pas toute la zone de dessin, il faut obéir au contrat d'opacité:
 - si le composant est opaque, on remplit la zone avec la couleur de fond
 - sinon, on ne fait rien

```
if (isOpaque()) {  
    g.setColor(getBackground());  
    g.fillRect(0,0,getWidth(),getHeight());  
}
```



Comment dessiner

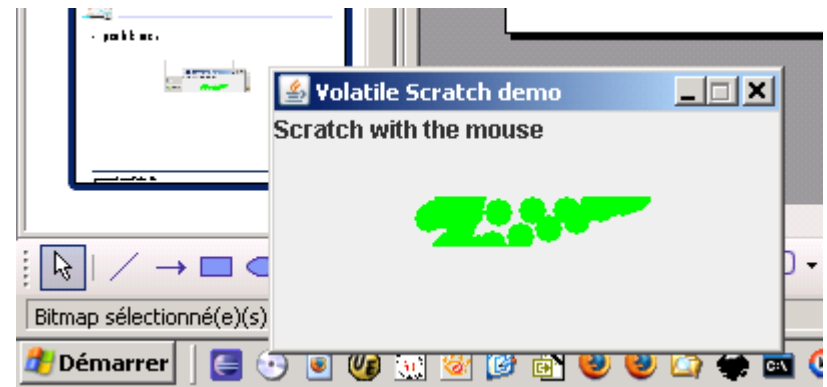
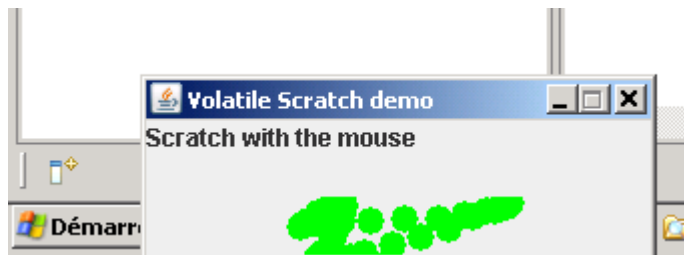
- dessin en récupérant le **Graphics** d'un composant:

```
public class VolatileScratch extends JComponent {  
  
    public VolatileScratch(int width,int height) {  
        setPreferredSize(new Dimension(width,height));  
        addMouseListener(new MouseMotionAdapter() {  
            @Override  
            public void mouseDragged(MouseEvent e) {  
                Graphics g=getGraphics();  
                try {  
                    g.setColor(Color.GREEN);  
                    g.fillOval(e.getX()-7,e.getY()-7,14,14);  
                } finally {  
                    g.dispose();  
                }  
            }  
        });  
    }  
  
    ...  
}
```



Comment dessiner

- problème: quand on doit rafraîchir la fenêtre, on perd ce qui n'était plus visible





Comment dessiner

- solution: dessiner dans une image

```
public class PersistentScratch extends JComponent {  
  
    private final BufferedImage image;  
  
    public PersistentScratch(int width,int height) {  
        setPreferredSize(new Dimension(width,height));  
        /* We just want to keep a reference on image, not on the whole component */  
        final BufferedImage image=new BufferedImage(  
            width,height,BufferedImage.TYPE_INT_ARGB);  
        this.image=image;  
        addMouseListener(new MouseMotionAdapter() {  
            @Override public void mouseDragged(MouseEvent e) {  
                Graphics g=image.getGraphics();  
                try {  
                    g.setColor(Color.GREEN);  
                    g.fillOval(e.getX()-7,e.getY()-7,14,14);  
                    paintImmediately(0,0,getWidth(),getHeight());  
                } finally {  
                    g.dispose();  
                }  
            }  
        });  
    }  
    ...  
}
```

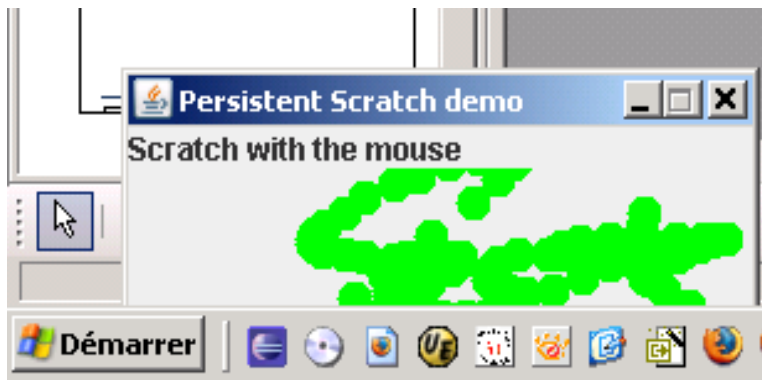
image transparente



Comment dessiner

- dans ce cas, le rafraîchissement consiste juste à afficher l'image:

```
@Override protected void paintComponent(Graphics g) {  
    g.drawImage(image, 0, 0, null);  
}
```



Yes!





ImageObserver

- le dernier paramètre de `drawImage` est un `ImageObserver`, qui sert à être prévenu du chargement de l'image
- dans notre cas, `null` signifie que l'on s'en fiche:

```
@Override protected void paintComponent(Graphics g) {  
    g.drawImage(image, 0, 0, null);  
}
```



Le clipping

- zone de clipping=pochoir appliqué au dessin qui va suivre
- on peut définir soit une zone rectangulaire:
 - `setClip(int x,int y,int width,int height)`
- soit une zone quelconque:
 - `setClip(Shape clip)`



Le clipping

- exemple: utilisation d'un disque pour n'afficher qu'une portion d'image

```
@Override
protected void paintComponent(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0,0,getWidth(),getHeight());
    if (x==-1 && y==-1) {
        return;
    }
    Shape clip=new Ellipse2D.Float(
        x-60,y-60,120,120);
    g.setClip(clip);
    g.drawImage(image,0,0,null);
}
```



ce qui est dessiné avant **setClip** n'est pas concerné (ici, remplissage avec du noir)



Le clipping

- on met à jour les coordonnées en fonction de la position de la souris

```
public class LookThroughTheHole extends JComponent {

    final Image image;
    int x,y;

    public LookThroughTheHole(ImageIcon icon) {
        setPreferredSize(new Dimension(icon.getIconWidth(),icon.getIconHeight()));
        final Image image=icon.getImage();
        this.image=image;
        addMouseListener(new MouseMotionAdapter() {
            @Override public void mouseMoved(MouseEvent e) {
                x=e.getX(); y=e.getY();
                paintImmediately(0,0,getWidth(),getHeight());
            }
        });
        addMouseListener(new MouseAdapter() {
            @Override public void mouseExited(MouseEvent e) {
                x=-1; y=-1;
                paintImmediately(0,0,getWidth(),getHeight());
            }
        });
    }

    ...
}
```



Le clipping

- problème: on voudrait pouvoir cacher le curseur
- on doit créer notre propre curseur invisible

```
final Cursor noCursor=Toolkit.getDefaultToolkit().createCustomCursor(  
    new BufferedImage(1,1,BufferedImage.TYPE_INT_ARGB),new Point(0,0), "");
```

taille minimum
d'une image

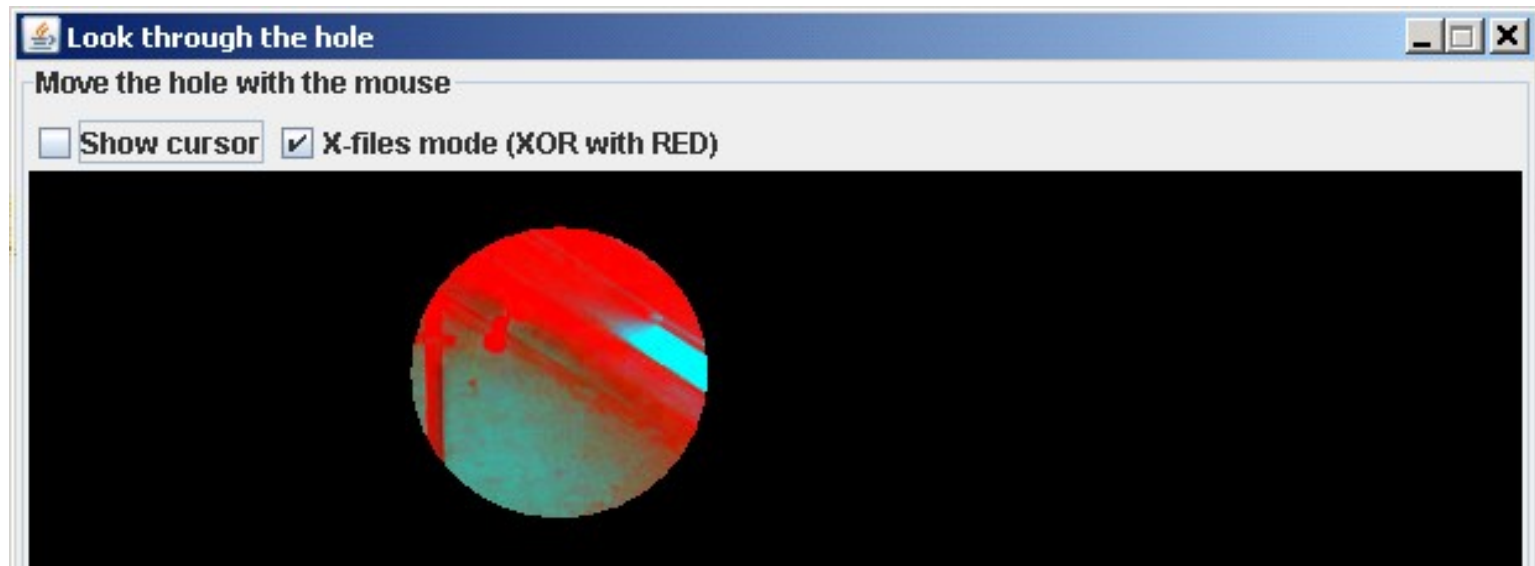
image
transparente

hotspot du
curseur



Le XOR mode

- on peut définir le mode de dessin:
 - peinture en écrasement (mode par défaut): `setPaintMode()`
 - combinaison avec la couleur des pixels en dessous: `setXORMode(Color c)`





Graphics2D

- **Graphics2D** étend **Graphics** et permet de gérer de nouvelles choses:
 - le pinceau à utiliser
 - la transparence
 - le dessin de formes complexes
 - les options de rendus
 - les transformations affines
- en Swing, c'est toujours un **Graphics2D**, qui est *réellement* reçu; on peut donc toujours caster un **Graphics** en **Graphics2D**



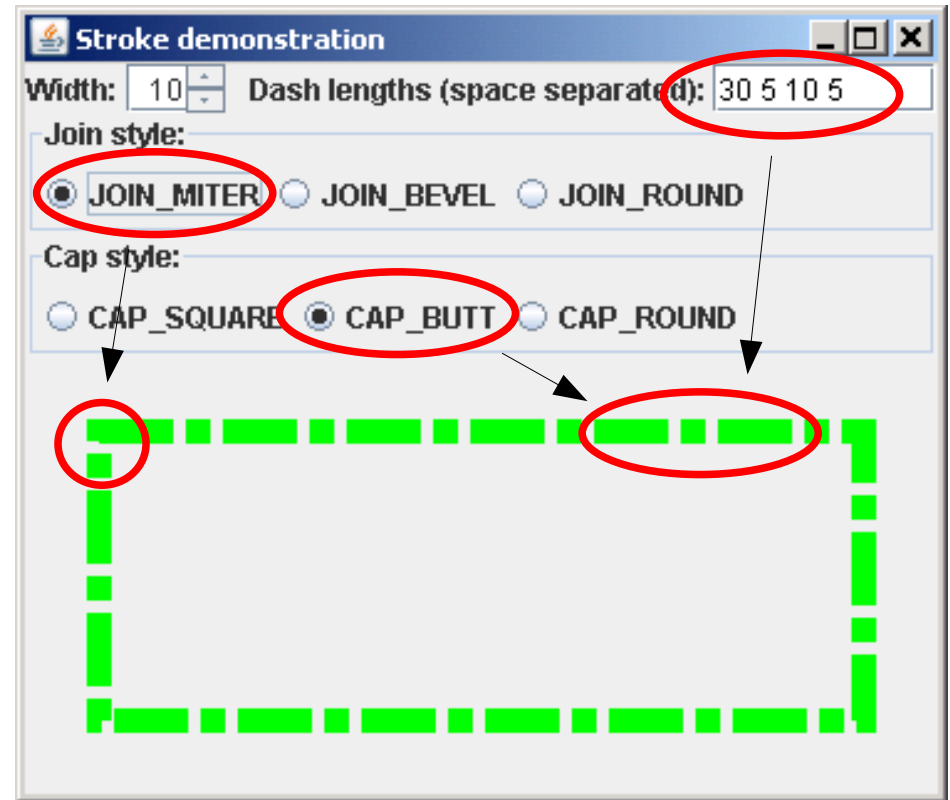
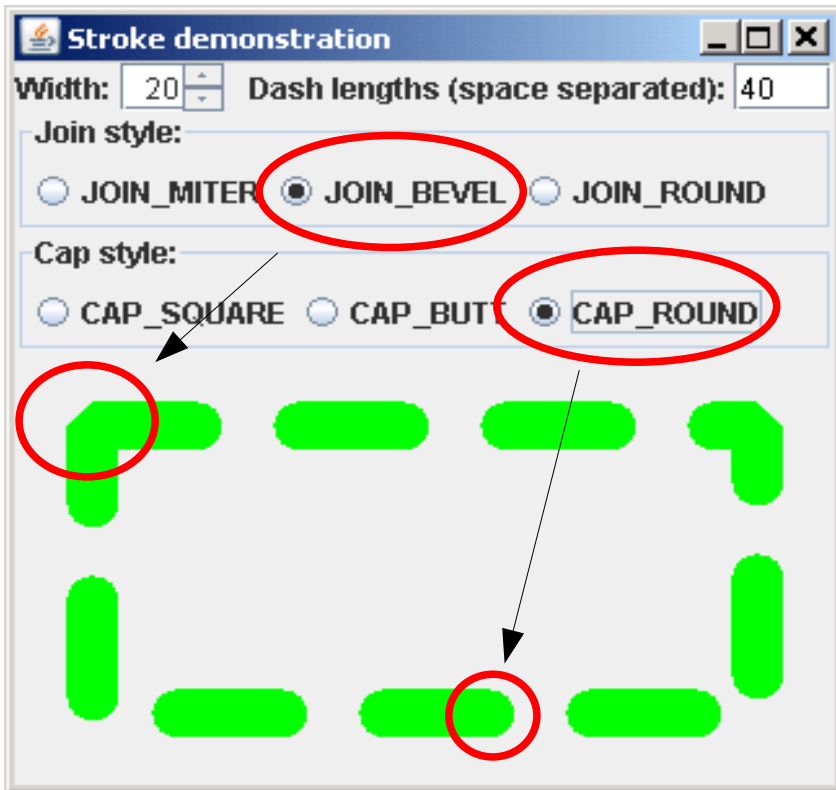
Le pinceau

- défini par l'interface **Stroke**
implémentation=**BasicStroke**
- on peut gérer:
 - l'épaisseur du trait
 - la forme des extrémités de segment
 - la forme des angles d'un polygone
 - le dessin en pointillé



Le pinceau

- exemples:





La transparence

- on peut définir la transparence du dessin de 2 façons:
 - en utilisant une couleur translucide, obtenue par exemple avec le constructeur:
`new Color(int r,int g,int b,int a)`
 - en définissant un `Composite`, obtenu avec `AlphaComposite.getInstance(int rule,float alpha)`:
 - `rule`=mode de combinaison entre le dessin et l'arrière-plan (cf. doc de `AlphaComposite`)
 - `alpha`: `0.0`=transparent `1.0`=opaque



Le ghost button

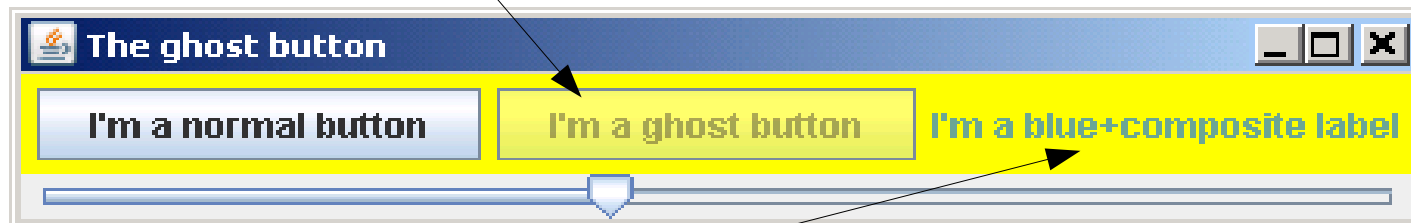
- on peut utiliser un **Composite** pour modifier la transparence globale d'un composant:

```
public class GhostButton extends JButton {  
  
    private Composite composite=AlphaComposite.getInstance(  
                                                AlphaComposite.SRC_OVER,0.5f);  
  
    public GhostButton(String text) {  
        super(text);  
        /* We don't want the background of the button to be painted */  
        setOpaque(false);  
    }  
  
    @Override protected void paintComponent(Graphics g) {  
        Graphics2D g2=(Graphics2D)g;  
        Composite old=g2.getComposite();  
        g2.setComposite(composite);  
        super.paintComponent(g);  
        g2.setComposite(old);  
    }  
  
    ...  
}
```



La transparence

- exemples:
 - un bouton dont la transparence est réglée par un **JSlider**



- un **JLabel** dont la couleur est transparente:

```
JLabel label=new JLabel("I'm a blue+composite label");  
label.setForeground(new Color(0,100,255,150));
```



Shape

- on peut définir des formes à l'aide de l'interface **Shape**
- formes prédéfinies:
 - **Ellipse2D**
 - **Rectangle2D**
 - **RoundRectangle2D**
 - **Line2D**
 - **CubicCurve2D, QuadCurve2D**
 - ...



Les Binou2D

- pour les `...2D`, on a 2 implémentations qui diffèrent par leur précision:
 - `...2D.Float`
 - `...2D.Double` (pas encore très bien géré par les cartes graphiques)
- pourquoi pas des coordonnées entières ?
- parce que c'est utile quand on fait de l'antialiasing (entre autres)



Dessiner ses propres formes

- on peut créer des **Polygon**, en ajoutant les points que l'on souhaite
- le dernier est relié au premier

```
/**
 * Creates a house-shaped polygon. x and y represents the center of the house.
 */
private Polygon createHouse(int x,int y) {
    Polygon p=new Polygon();
    p.addPoint(x-40,y+45);
    p.addPoint(x-40,y);
    p.addPoint(x-50,y);
    p.addPoint(x,y-45);
    p.addPoint(x+50,y);
    p.addPoint(x+40,y);
    p.addPoint(x+40,y+45);
    return p;
}
```



Dessiner ses propres formes

- si on veut combiner des courbes en plus de segments de droites, il faut utiliser les **GeneralPath**:
 - **moveTo**: déplace le curseur
 - **lineTo**: trace un segment
 - **curveTo**: trace une courbe de Bézier cubique
 - **quadTo**: trace une courbe de Bézier quadratique



Shape

- on peut dessiner le contour d'une Shape, en utilisant le pinceau courant et la couleur courante: `g.draw(shape);`
- ou bien la remplir avec la couleur courante: `g.fill(shape);`
- dans les deux cas, ça se combine avec le XOR mode, s'il y en a un, ou le composite



Area

- grâce à la classe `Area`, on peut combiner des formes entre elles
- on crée une `Area`
 - soit avec `new Area(Shape s)`
 - soit par combinaison (`a` est modifié):
 - `a.add(b)`
 - `a.intersect(b)`
 - `a.exclusiveOr(b)`
 - `a.subtract(b)`



Area

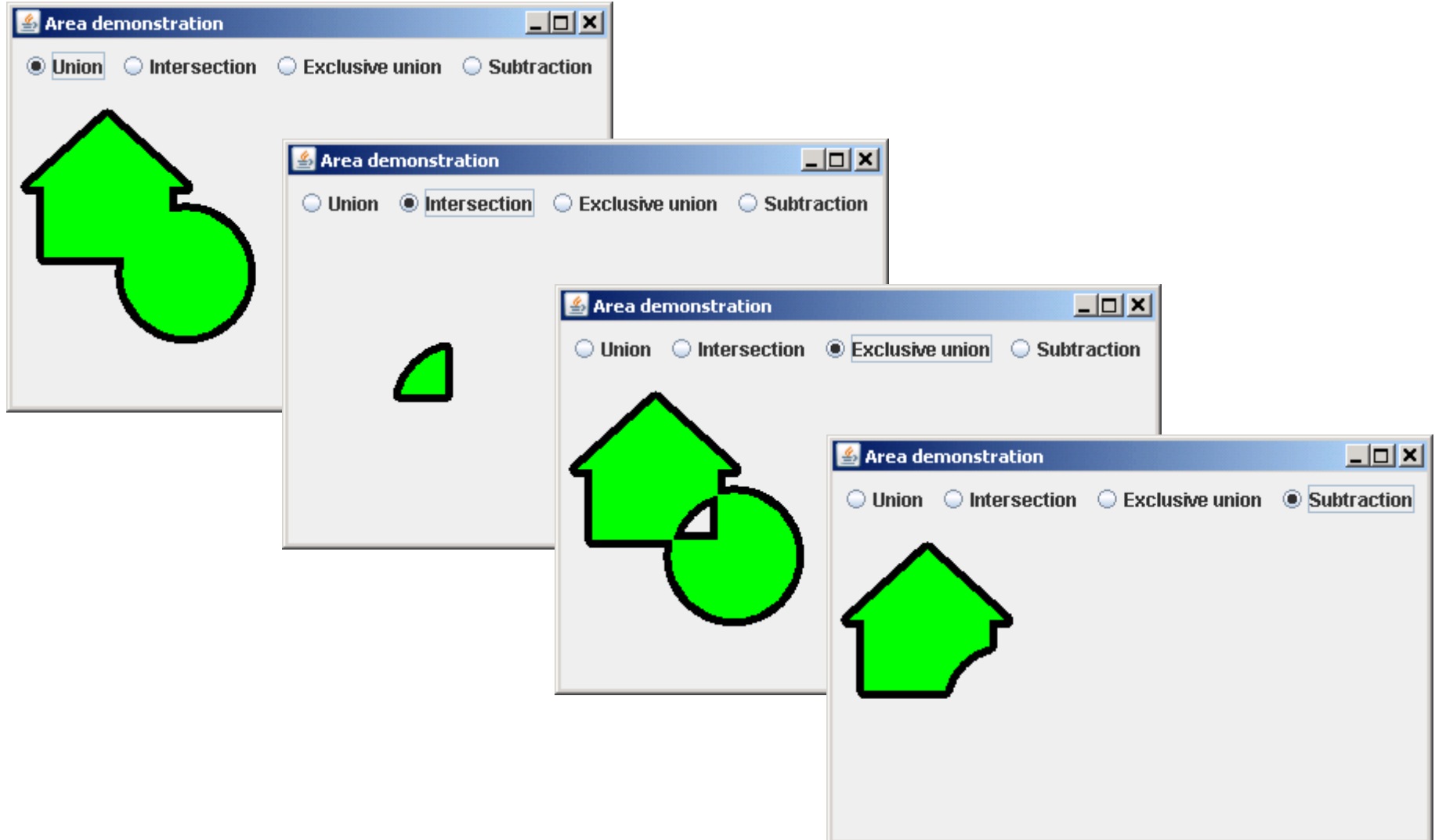
```
private final Area circle=new Area(new Ellipse2D.Float(70,75,80,80));
private final Area house=new Area(createHouse(60,60));
private final BasicStroke stroke=new BasicStroke(
    10,BasicStroke.CAP_ROUND,BasicStroke.JOIN_ROUND);

public static final int ADD_OP=0,INTER_OP=1,XOR_OP=2,SUB_OP=3;
private int operation=ADD_OP;

@Override protected void paintComponent(Graphics g) {
    if (isOpaque()) {
        g.setColor(getBackground());
        g.fillRect(0,0,getWidth(),getHeight());
    }
    /* We don't want to modify this area permanently */
    Area area=(Area) house.clone();
    switch (operation) {
        case ADD_OP: area.add(circle); break;
        case INTER_OP: area.intersect(circle); break;
        case XOR_OP: area.exclusiveOr(circle); break;
        case SUB_OP: area.subtract(circle); break;
    }
    Graphics2D g2=(Graphics2D)g;
    g2.setStroke(stroke);
    g2.setColor(Color.BLACK);
    g2.draw(area);
    g2.setColor(getForeground());
    g2.fill(area);
}
```



Area





Area

- l'interface **Shape** prévoit des méthodes pour tester l'intersection de formes:

```
intersects(double x, double y,  
           double w, double h)
```

```
intersects(Rectangle2D r)
```

- pratique quand on utilise des formes complexes
- exemple: la maison qui rebondit



Area

- on peut mettre la maison où on veut:

```
AffineTransform old=g2.getTransform();  
g2.translate(x,y);  
g2.rotate(theta);  
g2.fill(house);
```

- on peut ensuite obtenir une forme (**Shape** ou **Area**) correspondant à la maison telle qu'elle est après les transformations affines:

```
Area tmp=house.createTransformedArea(g2.getTransform());
```



Area

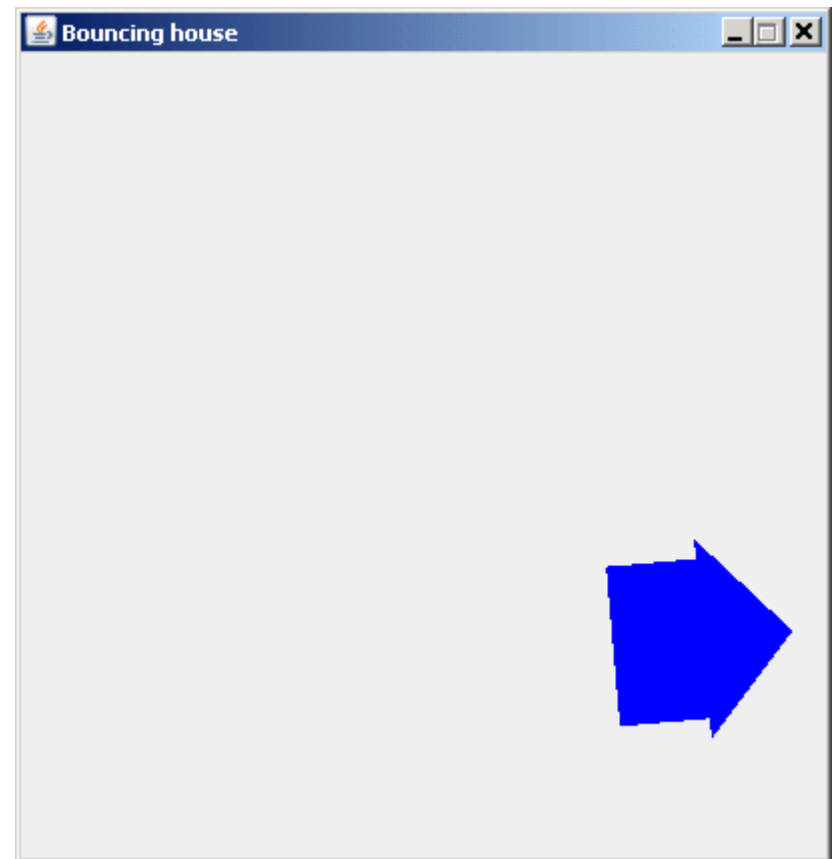
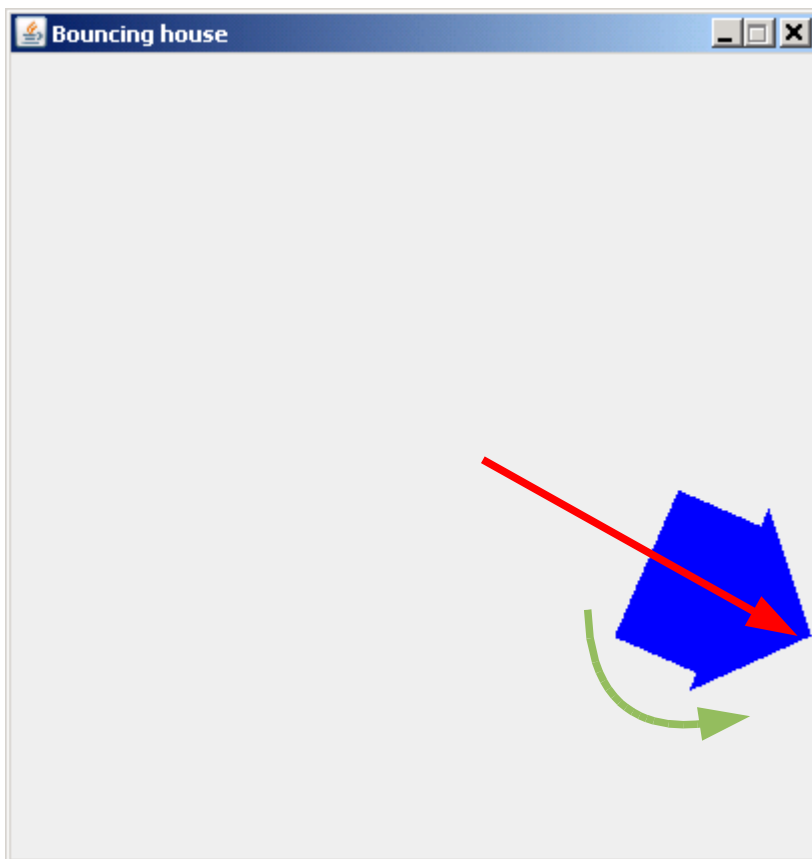
- avec une forme dans le repère d'origine, on peut facilement tester l'intersection avec les bords du composant:

```
if (tmp.intersects(w,0,w,h)) {  
    dx=-dx;  
    if (dy>0) dtheta=-0.1f;  
    else dtheta=0.1f;  
}  
if (tmp.intersects(0,0,1,h)) {  
    dx=-dx;  
    if (dy>0) dtheta=0.1f;  
    else dtheta=-0.1f;  
}  
...
```



Area

- on obtient ainsi une magnifique maison qui rebondit:





Les options de rendus

- **RenderingHints**=ensemble de paires clés,valeurs
- chaque clé représente un paramètre de rendu, pour lequel on peut définir une ou plusieurs valeurs:
 - soit clé par clé:
`setRenderingHint (key , value)`
 - soit en une seule fois:
`setRenderingHints (hints)`

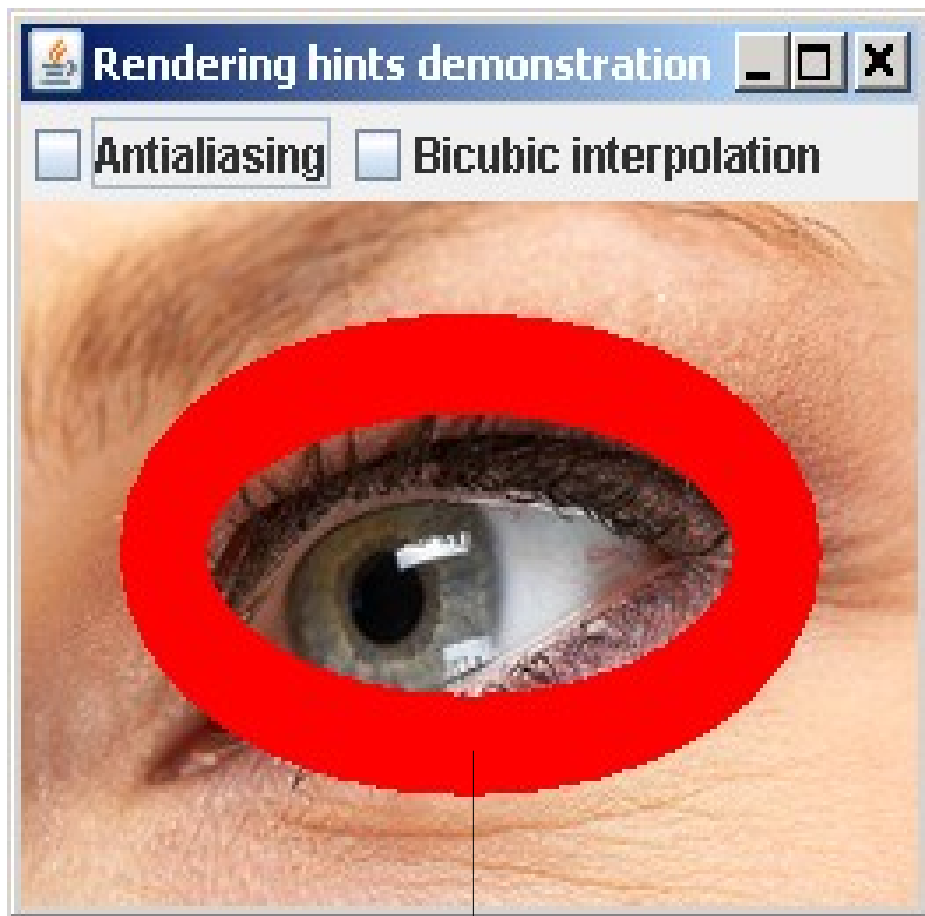


Les options de rendus

- exemple avec 2 clés:
- lissage du dessin:
`RenderingHints.KEY_ANTIALIASING`
 - `RenderingHints.VALUE_ANTIALIASING_ON`
 - `RenderingHints.VALUE_ANTIALIASING_OFF`
- interpolation quand on étire une image:
`RenderingHints.KEY_INTERPOLATION`
 - `RenderingHints.VALUE_INTERPOLATION_BICUBIC`
(c'est fait en soft, donc c'est lent)



Les options de rendus



contour non lissé



contour lissé



Les options de rendus

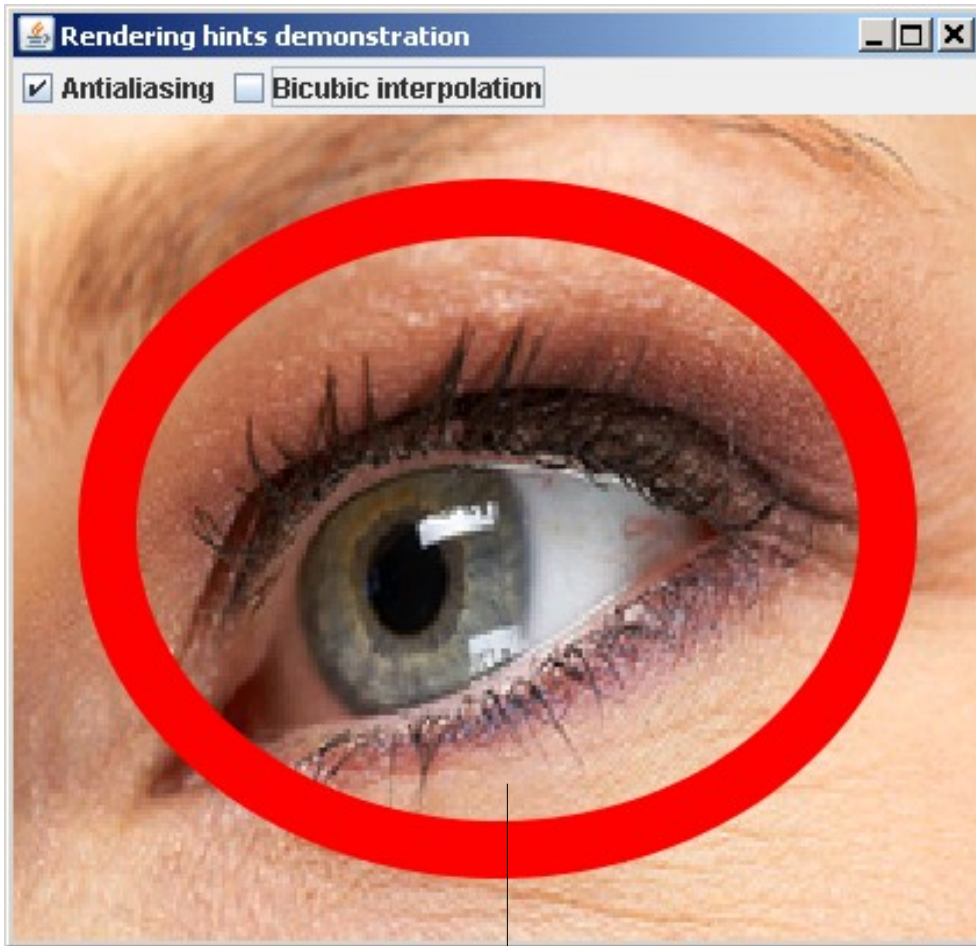


image pixellisée

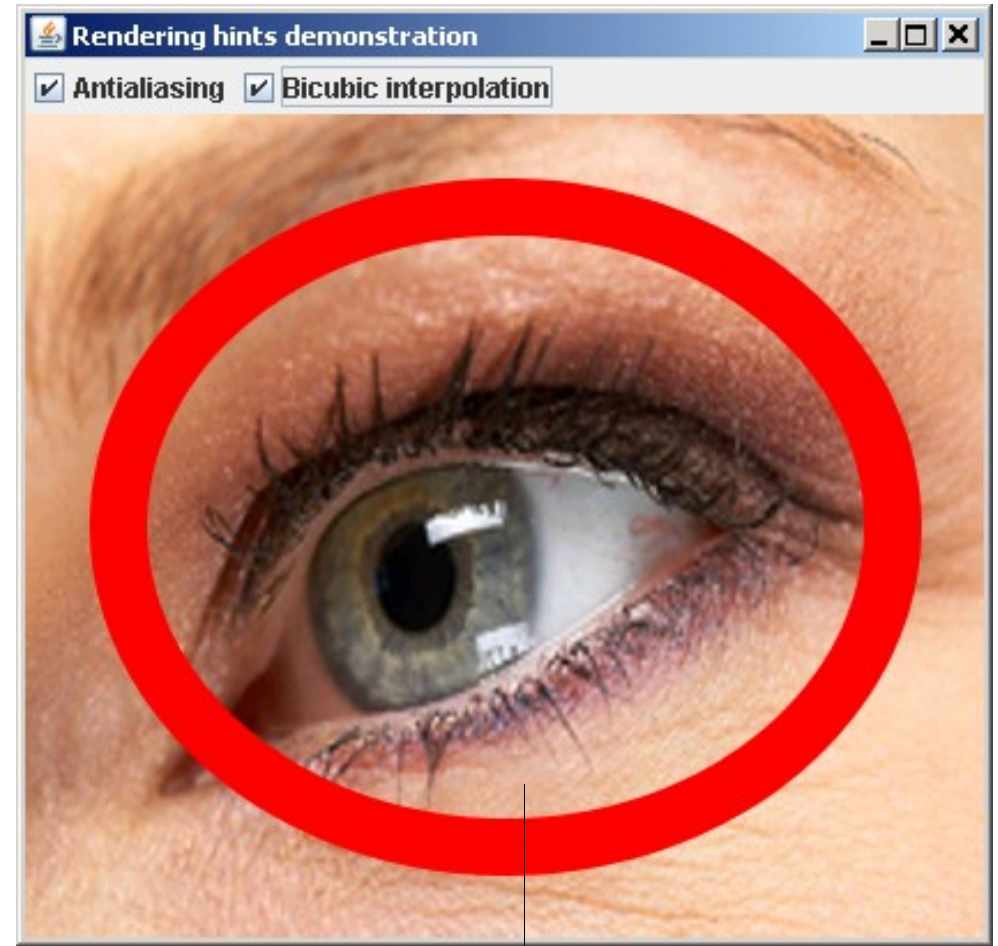


image non pixellisée



Les transformations affines

- avec un `Graphics2D`, on peut effectuer:
 - des rotations:
`rotate(double theta)`
`theta`=angle en radians dans le sens des aiguilles d'une montre
 - des homothéties:
`scale(double sx, double sy)`
`sx` et `sy`=coefficients de l'homothétie
 - des cisaillements:
`shear(double shx, double shy)`
`shx` et `shy`=coefficients du cisaillement



Les transformations affines

- pour les translations, 2 méthodes documentées un peu différemment:
 - `translate(int x,int y)`
 - le point `(x,y)` est la nouvelle origine du repère
 - `translate(double tx,double ty)`
 - `tx` et `ty` sont les longueurs des déplacements horizontaux et verticaux
- en pratique, ça revient au même



Les transformations affines

- on peut également utiliser des objets **AffineTransform**:
 - `getRotateInstance(double theta)`
 - `getScaleInstance(double sx, double sy)`
 - `getShearInstance(double shx, double shy)`
 - `getTranslateInstance(double tx, double ty)`
- pas de `getTranslateInstance(int x, int y)`



Les transformations affines

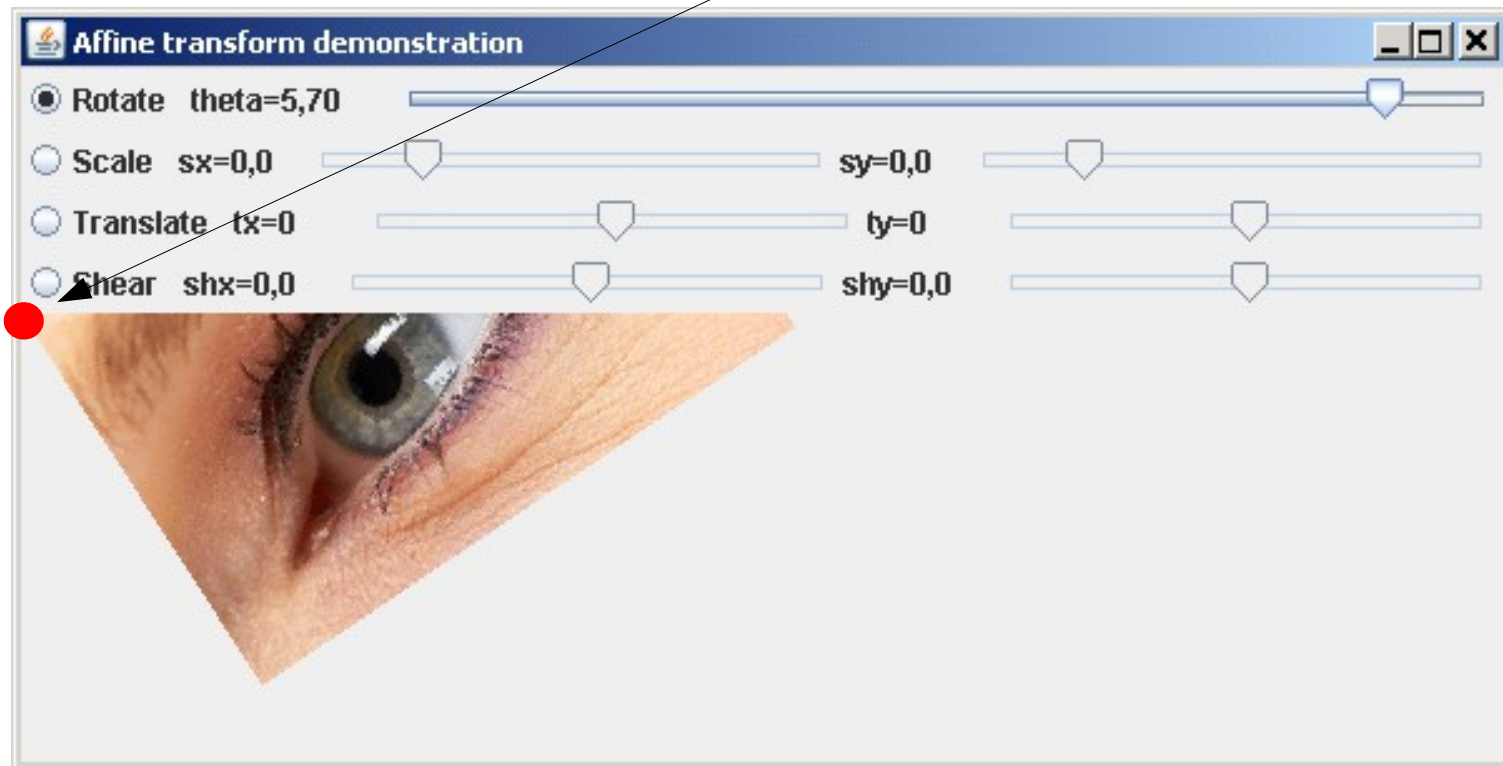
- quand on fait des transformations, il faut:
 - toujours sauvegarder l'état initial et le restaurer ensuite
 - ne **jamais** remplacer la transformation existante, mais combiner la nouvelle avec elle

```
@Override protected void paintComponent(Graphics g) {  
    Graphics2D g2=(Graphics2D)g;  
    AffineTransform old=g2.getTransform();  
    /* NEVER do g2.setTransform(transform); !!!  
     * It would ignore the possibly existing transform */  
    g2.transform(transform);  
    g2.drawImage(image,0,0,null);  
    g2.setTransform(old);  
}
```



Les transformations affines

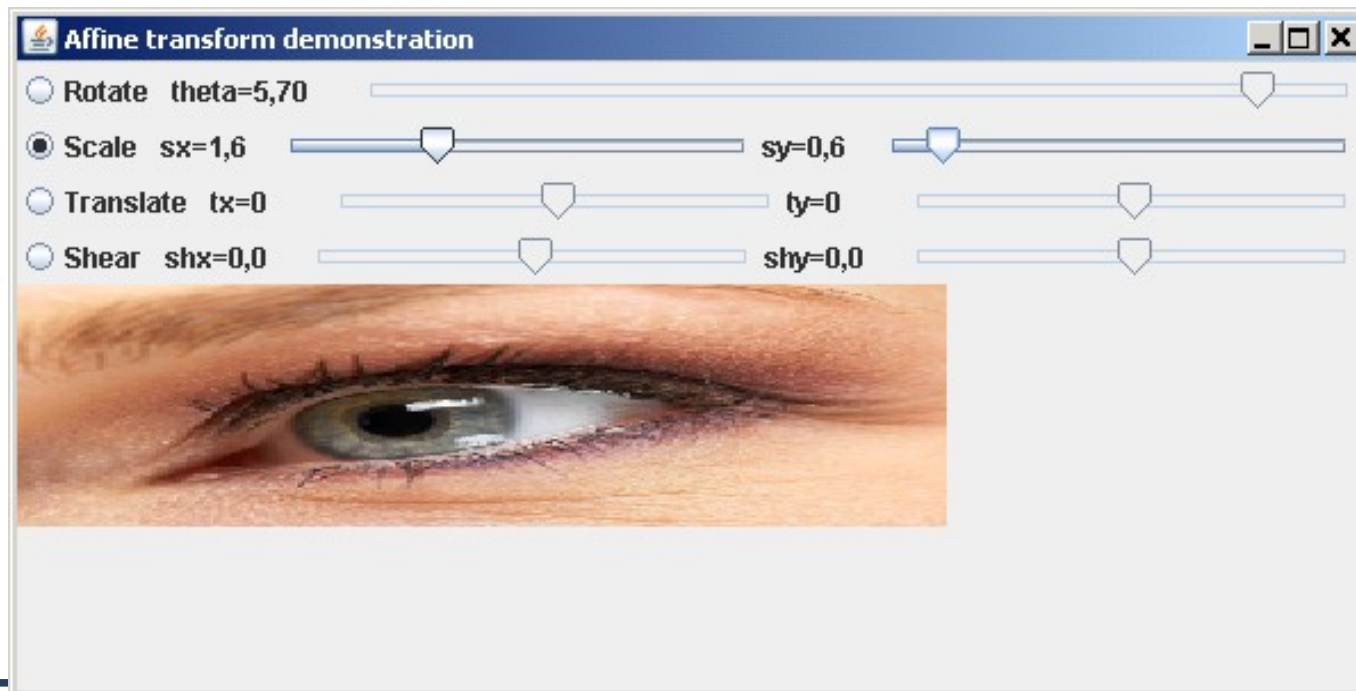
- rotation autour du point d'origine:





Les transformations affines

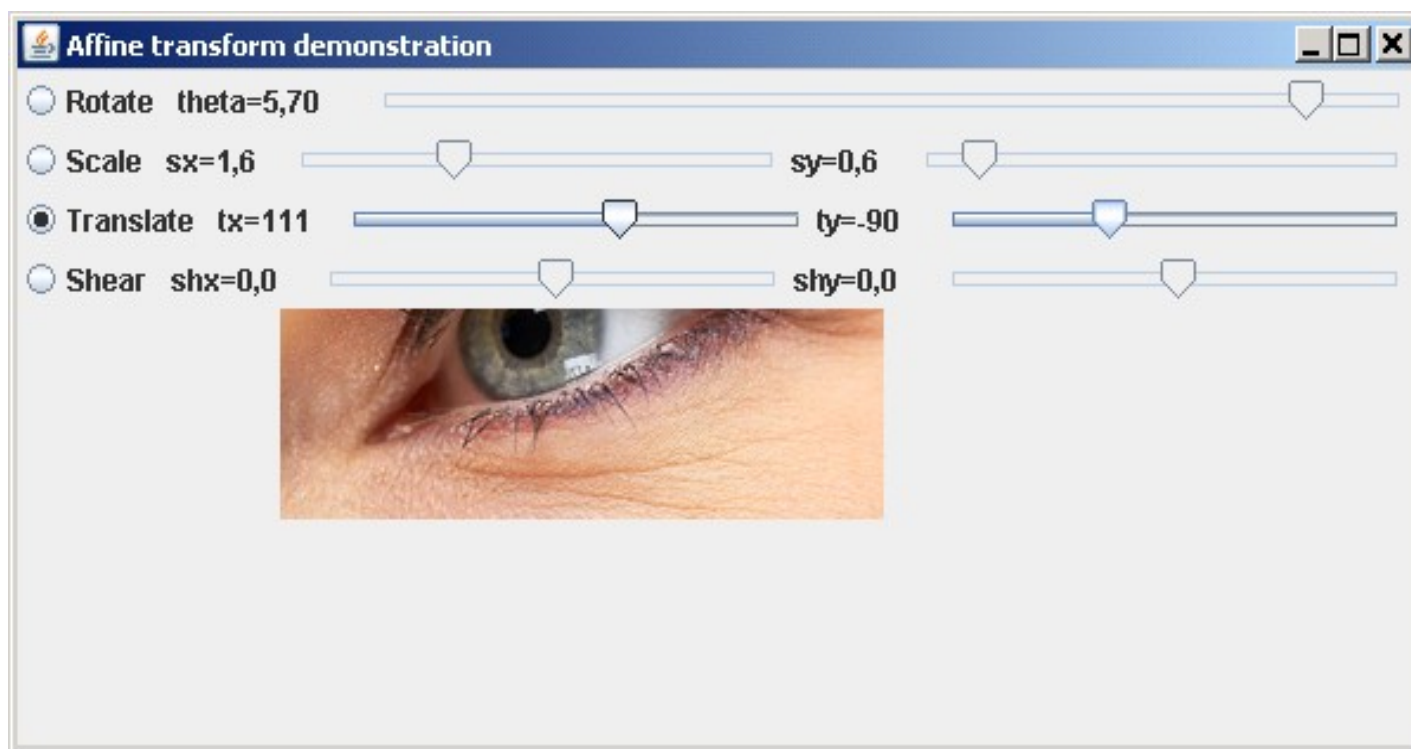
- coefficients de l'homothétie:
 1 = identité
 >1 = agrandissement
entre 0 et 1 = rétrécissement
 <0 idem, en symétrique





Les transformations affines

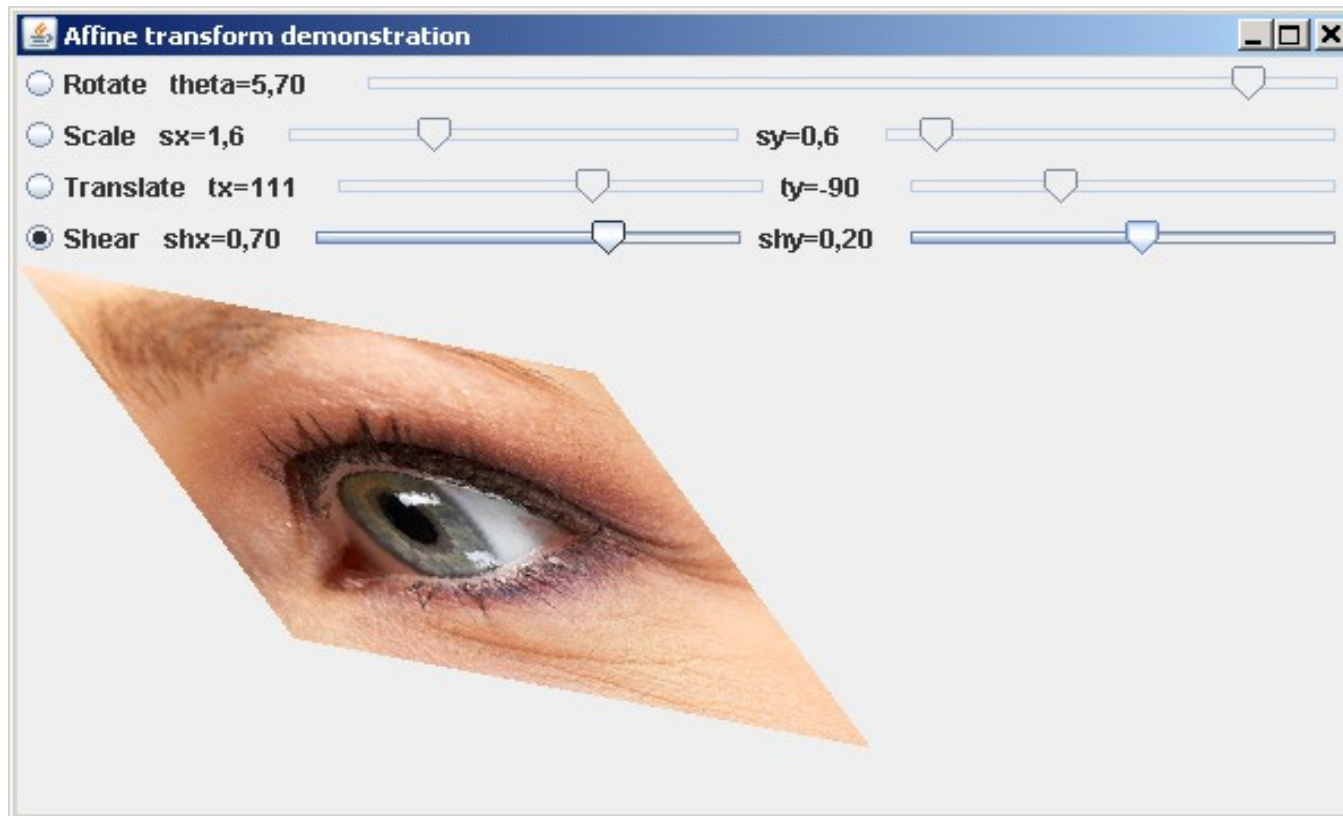
- translation:





Les transformations affines

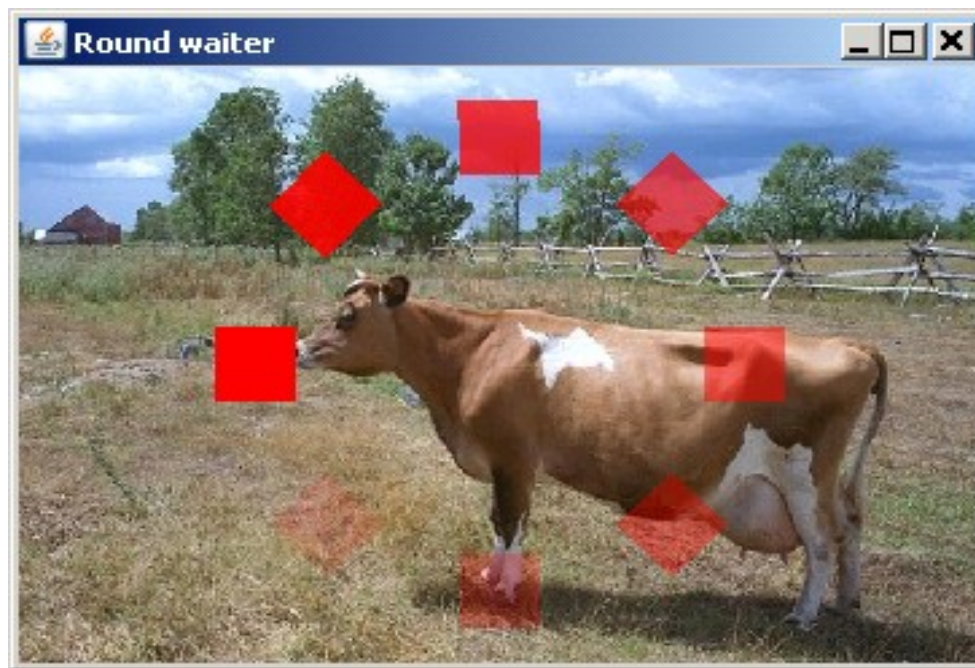
- cisaillement=déformation du repère (les axes ne sont plus orthogonaux)





Combiner les transformations

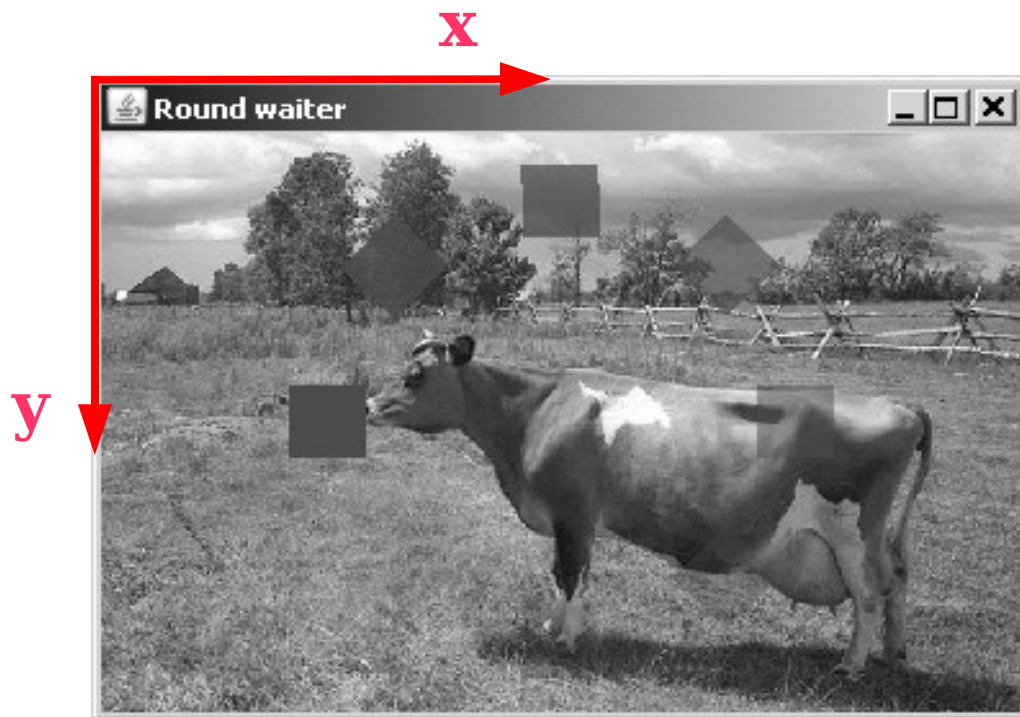
- on peut appliquer plusieurs transformations successives
- exemple: le round waiter





Combiner les transformations

- étape 0:

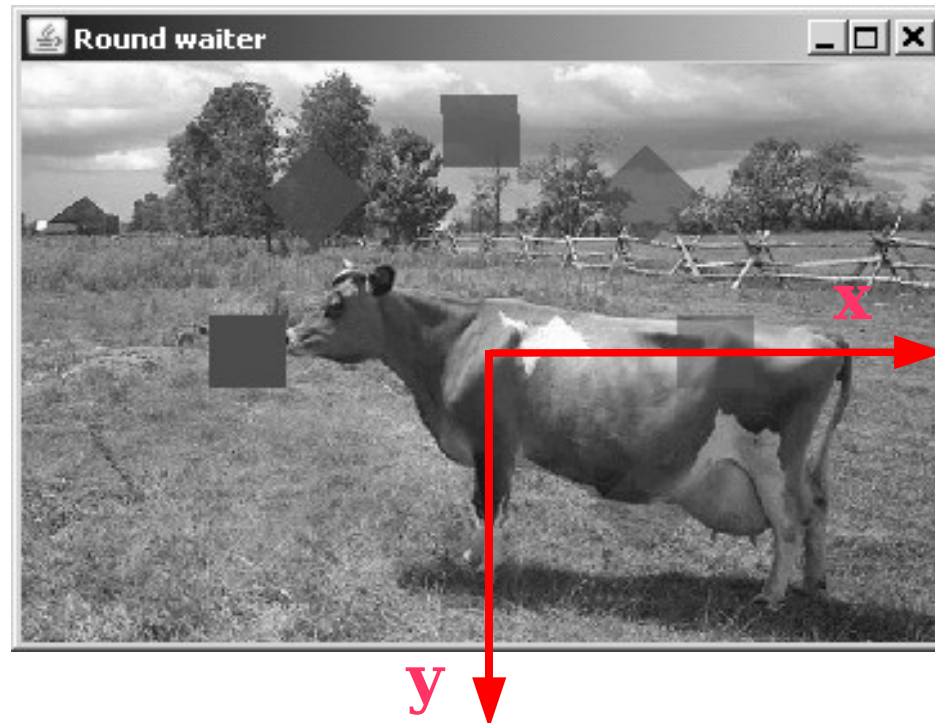




Combiner les transformations

- étape 1: se placer au centre du composant

```
g2.translate(xCenter, yCenter);
```

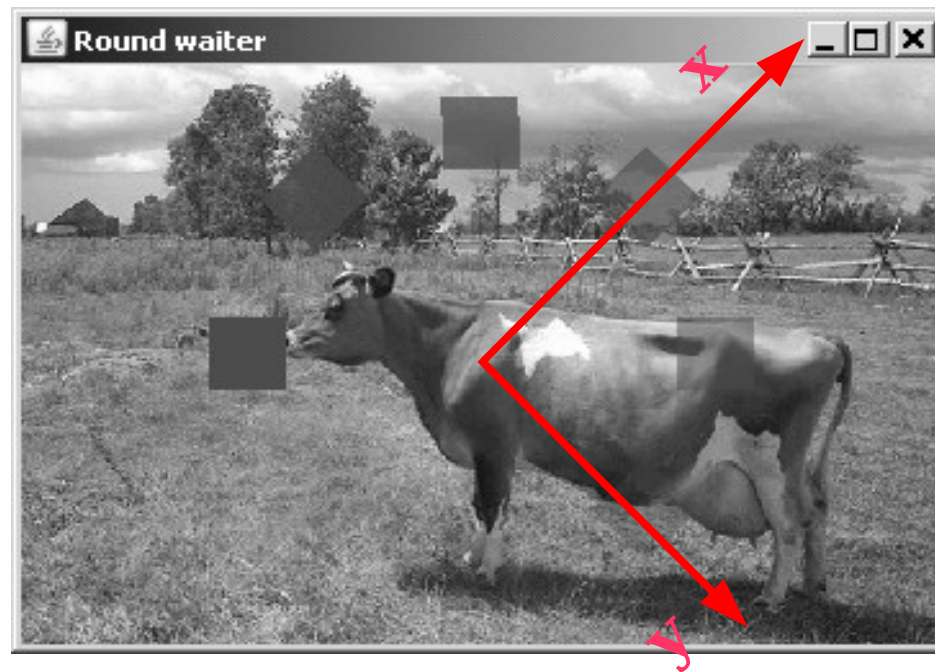




Combiner les transformations

- étape 2: pivoter d'un angle proportionnel à l'indice du carré courant

```
g2.rotate(2 * Math.PI * i / N);
```

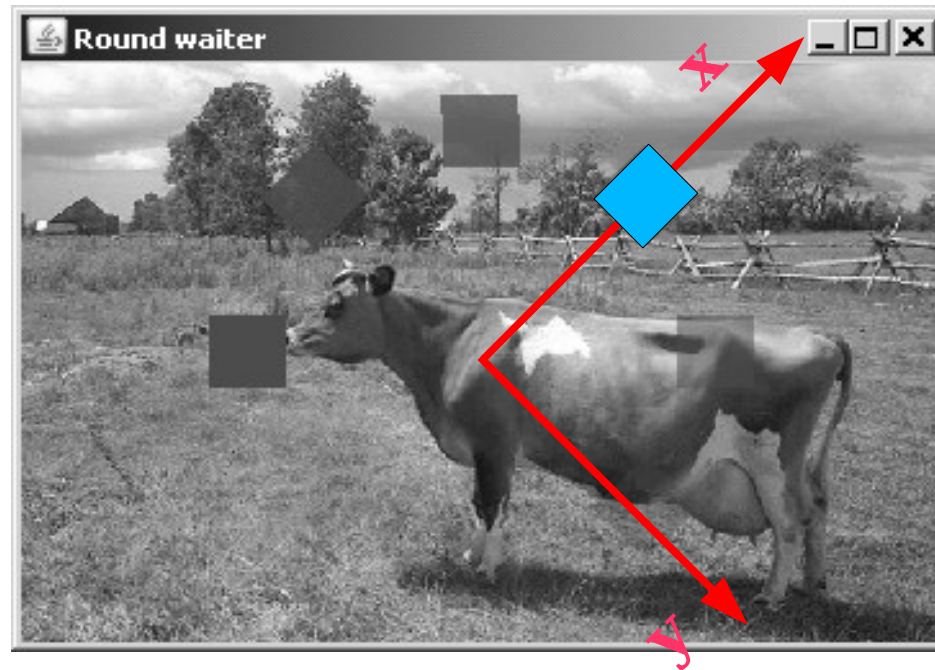




Combiner les transformations

- étape 3: dessiner le carré sur l'axe des X

```
g2.fillRect(radius-size/2,-size/2, size, size);
```

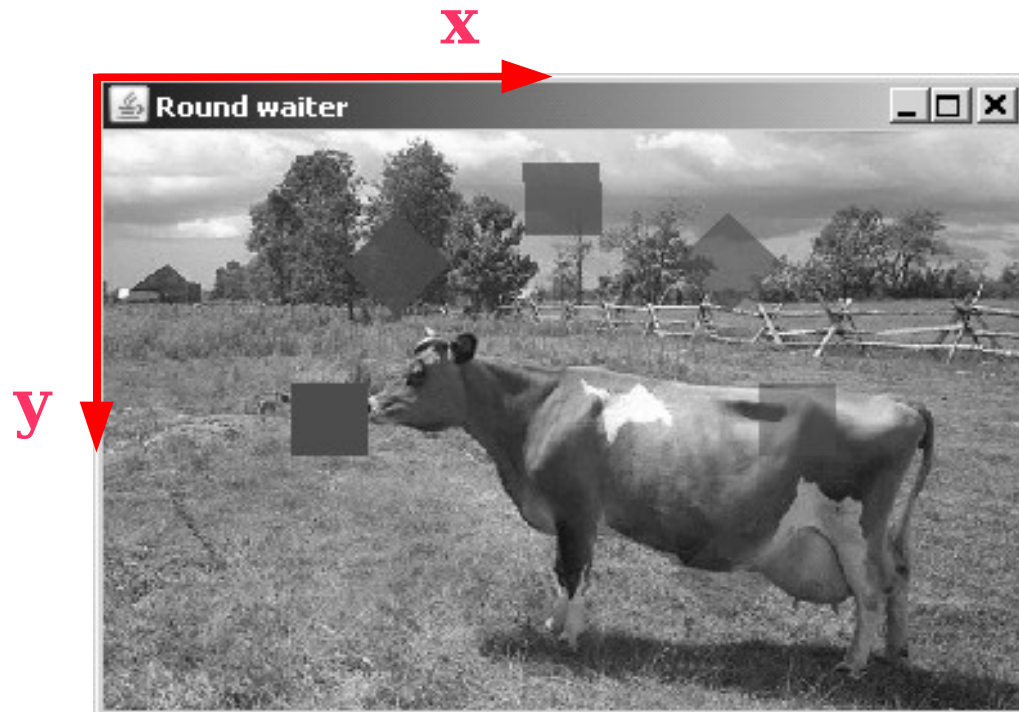




Combiner les transformations

- étape 4: rétablir la transformation d'origine pour recommencer avec le carré suivant

```
g2.setTransform(old);
```





Les peintres

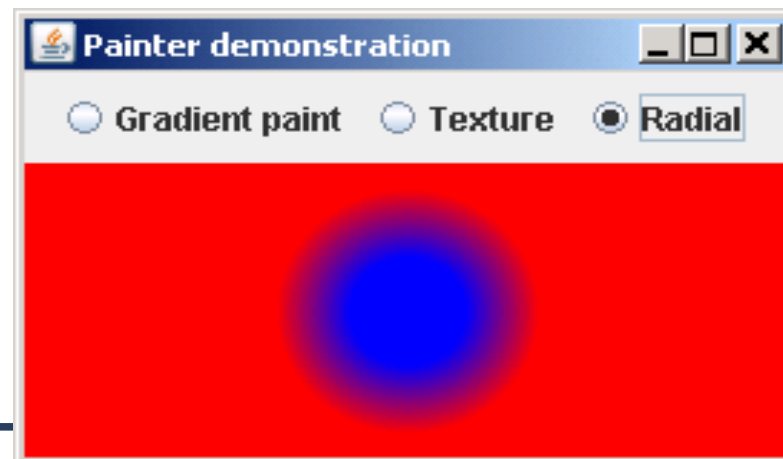
- on peut définir l'objet qui s'occupe du coloriage avec `setPaint(Paint p)`
- par défaut, c'est un objet `Color`, mais il y en a d'autres, comme:
 - `GradientPaint`: dégradé entre 2 couleurs paramétré par 2 points
 - `TexturePaint`: reproduction d'un motif défini par une `BufferedImage` et un `Rectangle2D` qui indique la taille et la position de départ du motif



Les peintres

- **LinearGradientPaint**: variante du **GradientPaint** avec plusieurs couleurs
- **RadialGradientPaint**: dégradé circulaire, paramétré par un point central et 1 rayon, des couleurs et des coefficients qui indiquent la vitesse de transition entre 2 couleurs:

```
g2.setPaint(new RadialGradientPaint(getWidth()/2,getHeight()/2,getWidth()/4,  
    new float[]{0.3f,0.7f},new Color[]{Color.BLUE,Color.RED}));
```

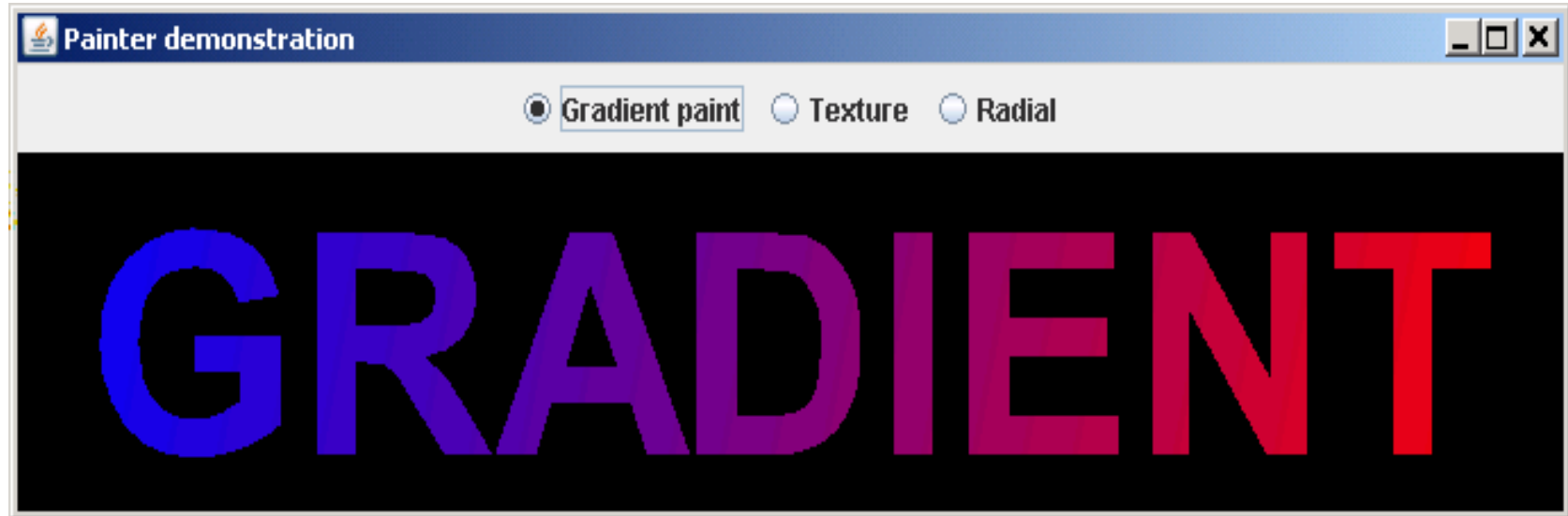




Les peintres

- exemple de `GradientPaint` du bleu vers le rouge de (0,0) vers (width,height)

```
g2.setPaint(new GradientPaint(0,0,Color.BLUE,getWidth(),getHeight(),Color.RED));
```





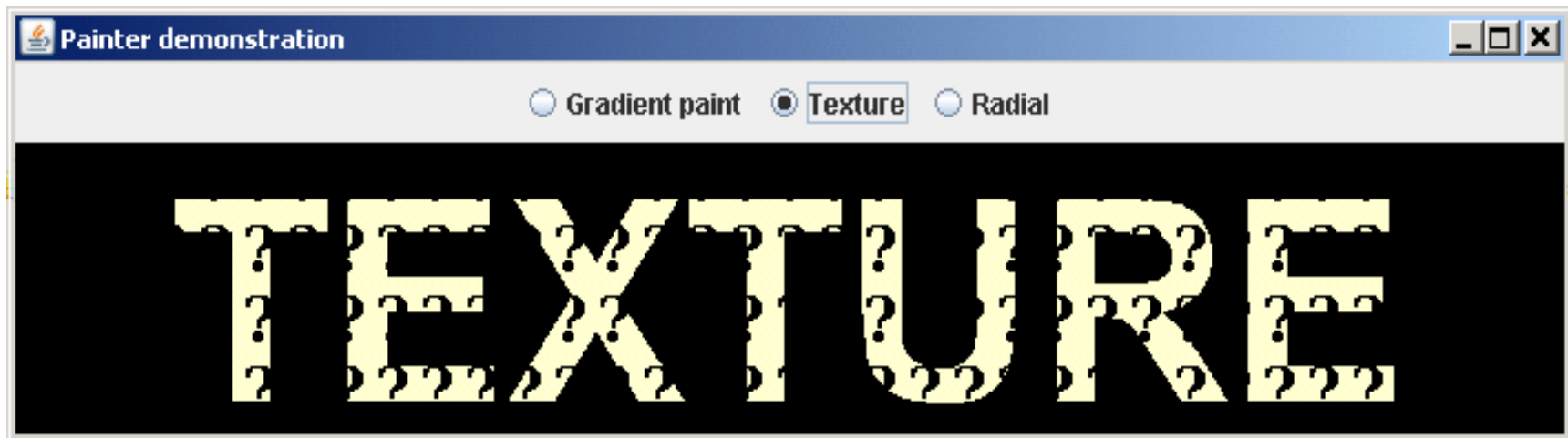
Les peintres

- exemple de **TexturePaint**:

```
private final BufferedImage texture=createTexture(  
    new ImageIcon(PainterDemo.class.getResource("texture.png")));  
  
private final TexturePaint texturePaint=new TexturePaint(texture,  
    new Rectangle2D.Float(0,0,texture.getWidth(),texture.getHeight()));  
  
private BufferedImage createBufferedImage(ImageIcon icon) {  
    BufferedImage image=new BufferedImage(icon.getIconWidth(),icon.getIconHeight(),  
        BufferedImage.TYPE_INT_RGB);  
    Graphics g=image.getGraphics();  
    try {  
        g.drawImage(icon.getImage(),0,0,null);  
    } finally {  
        g.dispose();  
    }  
    return image;  
}
```



Les peintres



- ça peut être une façon de faire du clipping avec du texte