Le Dessin

- Composant et Graphique
- Graphics & Graphics2D
- Chaîne de traîtement
- Shape, Curve, Area
- Transformation affine

Le Dessin

- Il est possible en Java de dessiner pour créer un rendu visuel à partir de forme de base (ligne, rectangle, ellipse, etc.).
- Le dessin est possible en AWT (uniquement sur des Canvas) et en Swing (sur tous JComponent).
- Graphics (JDK 1.0)
  Graphics2D (JDK 1.2)

	AWT	Swing
Graphics	Texte, Forme géométrique simple	Double buffer, Dirty région, Clipping
Graphics2D	Shape, Path, Transformation	Shape, affichage sophistiqué

Contexte Graphique

- L'outil de dessin est le ***contexte graphique***, objet de la classe `Graphics`. Il encapsule l'information nécessaire, sous forme d'***états graphiques***.

- 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 pour chaque pixel (simple ou XOR)
  - la couleur du XOR, s'il y a lieu.

Contexte Graphique - Etats

- Pour changer l'état du contexte graphique.
- Translater la zone d'affichage
  - translate(x,y)
- Changer la zone de clipping (forcément + petite)
  - get/setClip(x,y,width,height)
- Changer la couleur courante
  - get/setColor(color)
- Changer la fonte courante
  - get/setFont(font)
- Changer le mode de dessin
  - setPaintMode()
  - setXORMode(color)

Contexte Graphique – Primitives de Dessin

- Les primitives de dessin utilisent l'état courant :
- Afficher une ligne, rectangle, ellipse ou polygone
  - drawLine(x1,y1,x2,y2)
  - drawRect(x,y,width,height)
  - drawOval(x,y,width,height)
  - drawPolygon(polygon)
- Afficher remplit un rectangle, ellipse ou polygone
  - fillRect(x,y,width,height)
  - fillOval(x,y,width,height)
  - fillPolygon(polygon)

Contexte Graphique – Primitives de Dessin (2)

- Autres primitives de dessin utilisant l'état courant :
- Afficher un chaîne de caractère
  - drawString(x,y,text)
- Afficher une image
  - drawImage(Image,x,y,ImageObserver)
- L'**`ImageObserver`** est un objet qui va demander un chargement de l'image si celle-ci n'est pas chargée et affiché l'image pendant le chargement.
  - L'ImageObserver peut être null.
  - Les Composants (**`Component`**) implante cette interface.

Création du Contexte Graphique

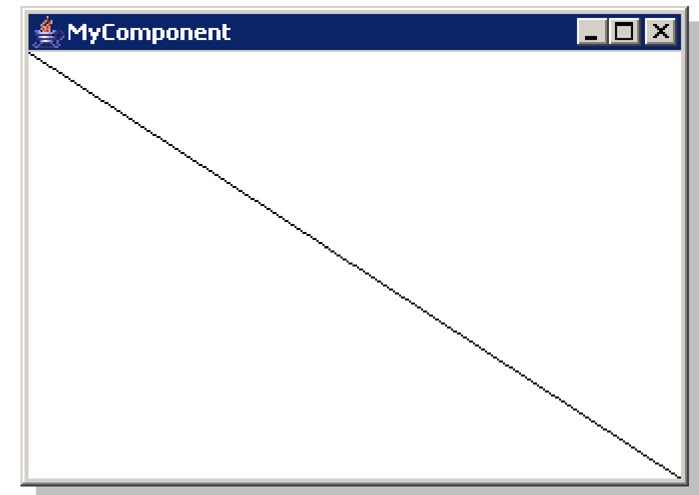
- La création du contexte graphique respecte des règles.
- Trois façons d'obtenir le **`Graphics`** :
  - **Implicitement**, dans une méthode **`paint()`**, **`paintComponent()`** servant au dessin du composant.
  - **Explicitement**, en demandant au composant ou à une image un contexte graphique (**`getGraphics()`**).
  - **Explicitement**, en appelant la méthode **`createGraphics()`** à partir d'un **`Graphics`** existant.
- Les **`Graphics`** obtenu **explicitement** doivent être libérés avec la méthode **`dispose()`**.

Création du Contexte Graphique (2)

- Un contexte graphique utilise des ressources systèmes.
- Pour certain environnement, en particulier Windows, le nombre de ressources utilisables pour le dessin est faible.
- Il faut donc utiliser la méthode `dispose()` pour rendre les ressources au système **rapidement**.
- La création explicite de contexte graphique est donc réservée à des utilisations particulières et n'est en aucun cas une utilisation habituelle.

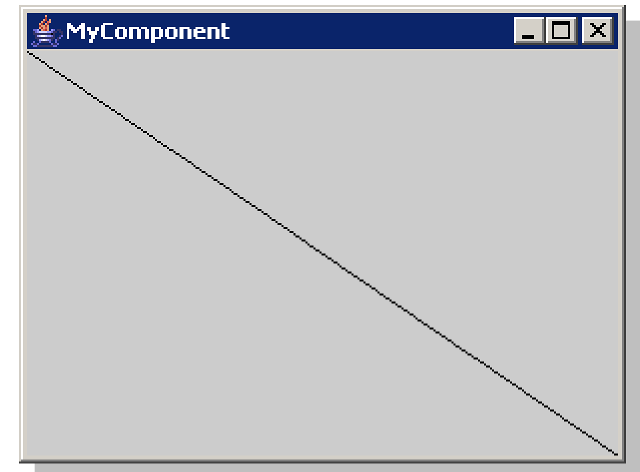
- L'évènement de dessin sur une composant n'est pas interceptable avec un listener.
- Doit hériter de **`Canvas`**.
- Redéfinir la méthode **`paint()`** du **`Canvas`**.

```
import java.awt.*;

public class MyComponent extends Canvas {
  public void paint(Graphics g) {
    g.drawLine(0,0,getWidth(),getHeight());
  }

  public static void main(String[] args) {
    Frame frame=new Frame("MyComponent");
    frame.add(new MyComponent());
    frame.setSize(400,300);
    frame.show();
  }
}
```



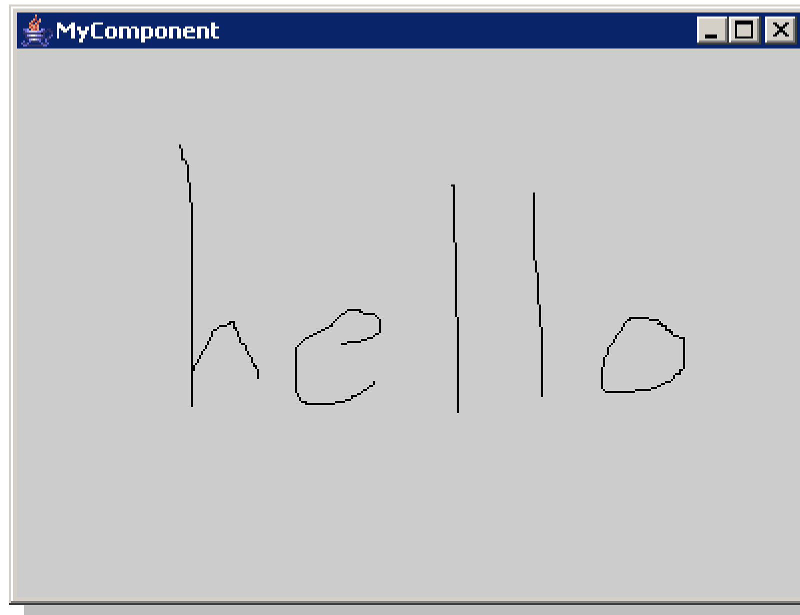
- Peut hériter de n'importe quel composant Swing, toutes les sous-classes de **`JComponent`**.
- Doit redéfinir la méthode **`paintComponent()`**.
- Doit faire un appel à **`super.paintComponent()`**.

```
import java.awt.*;

public class MyJComponent extends JComponent {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawLine(0,0,getWidth(),getHeight());
  }
  public static void main(String[] args) {
    JFrame frame=new JFrame("MyComponent");
    MyJComponent component=new MyJComponent();

    frame.getContentPane().add(component);
    frame.setSize(400,300);
    frame.show();
  }
}
```



Context graphique Explicite

- Exemple de dessin à la main.



- Sauvegarde la où l'utilisateur clique sans relacher.
- Trace une ligne tant que l'utilisateur bouge et ne relache pas le bouton de la souris.

Context graphique Explicite (2)

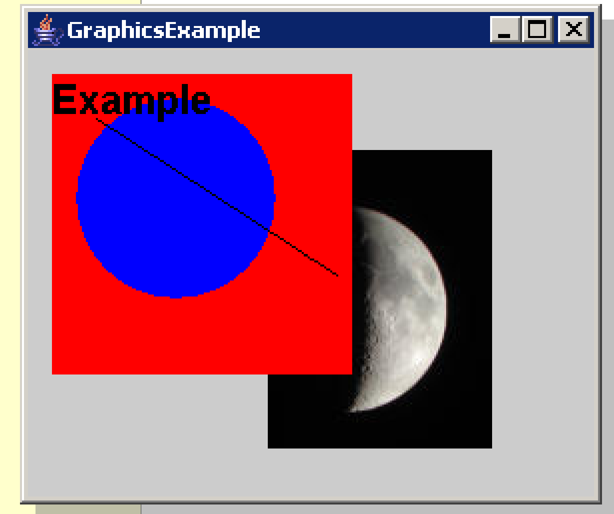
- Récupère un contexte graphique par le composant.

```
public class DrawJComponent extends JComponent {
  public static void main(String[] args) {
    JFrame frame=new JFrame("MyComponent");
    final DrawJComponent draw=new DrawJComponent();
    class MouseManager extends MouseAdapter implements MouseMotionListener {
      public void mouseDragged(MouseEvent event) {
        Graphics g=draw.getGraphics();
        try {
          int newX=event.getX();
          int newY=event.getY();
          g.drawLine(x,y,newX,newY);
          x=newX;
          y=newY;
        }
        finally {
          g.dispose();
        }
      }
      public void mouseMoved(MouseEvent e) {
      }
      public void mousePressed(MouseEvent event) {
        x=event.getX();
        y=event.getY();
      }
      private int x,y;
    };
    MouseManager manager=new MouseManager();
    draw.addMouseListener(manager);
    draw.addMouseMotionListener(manager);
    frame.getContentPane().add(draw);
    frame.setSize(400,300);
    frame.setVisible(true);
  }
}
```

Context graphique Implicite

- Récupère un contexte graphique par le composant.

```
public class GraphicsExample extends JComponent {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(image, 120, 50, this);
    g.setColor(Color.RED);
    g.fillRect(12,12,150,150);
    g.setColor(Color.BLUE);
    g.fillOval(24, 24, 100, 100);
    g.setColor(Color.BLACK);
    g.drawLine(34,34,154,112);
    g.setFont(font);
    g.drawString("Example",12,32);
  }
  private final Font font=new Font("SansSerif",Font.BOLD,20);
  private final Image image=new
ImageIcon("moon.jpg").getImage();
  public static void main(String[] args) {
    JFrame frame=new JFrame("GraphicsExample");
    GraphicsExample component=new GraphicsExample();
    frame.getContentPane().add(component);
    frame.setSize(400,300);
    frame.show();
  }
}
```



Context graphique étendu

- La classe **`Graphics2D`** définie de nouvelles fonctionnalités et refactorise celles de **`Graphics`**.
- Le processus de traitement est en plusieurs étapes
  - déterminer ce qui doit être affiché (formes, textes, images)
  - appliquer les transformations géométriques et le clipping
  - déterminer la couleur
  - combiner avec ce qui se trouve sur la surface

- Pour chacune de ces étapes, des multiples possibilités existent.

Chaîne de traitement - 1

- On obtient un objet `Graphics2D` par conversion

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

- On choisit le propriété d'affichage, par ex: antialiasing

```
RenderingHints hints = ...
g2.setRenderingHints(hints);
```

- On choisit l'outil de dessin des traits (contours)

```
Stroke stroke = ...
g2.setStroke(stroke);
```

Chaîne de traitement - 2

- On choisit l'outil de remplissage (couleur, dégradé, texture)

```
Paint paint = ...
g2.setPaint(paint);
```

- On définit la zone d'affichage (clipping)

```
Shape clip = ...
g2.setClip(clip);
```

- On définit une transformation géométrique entre l'espace utilisateur et espace d'écran

```
AffineTransfrom transform = ...
g2.transform(transform);
```

Chaîne de traitement - 3

- On compose le dessin résultant avec le dessin existant

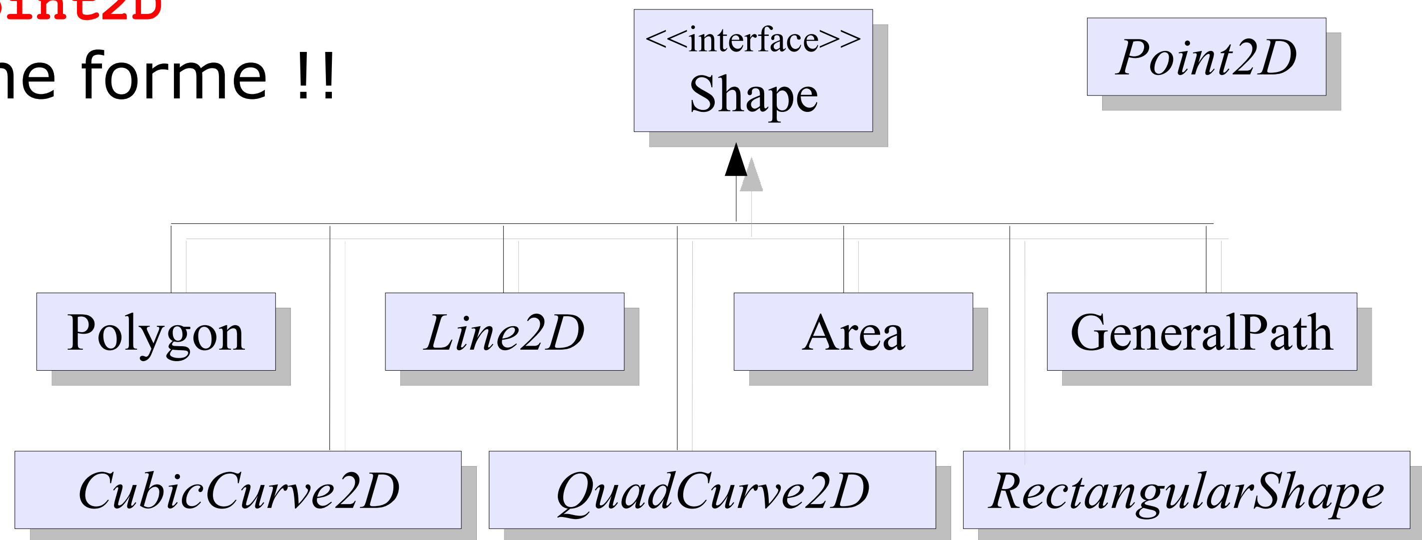
```
Composite composite = ...
g2.setComposite(composite);
```

- On définit la forme à dessiner et on l'affiche

```
Shape shape = ...
g2.fill(shape);
// et/ou
g2.draw(shape);
```

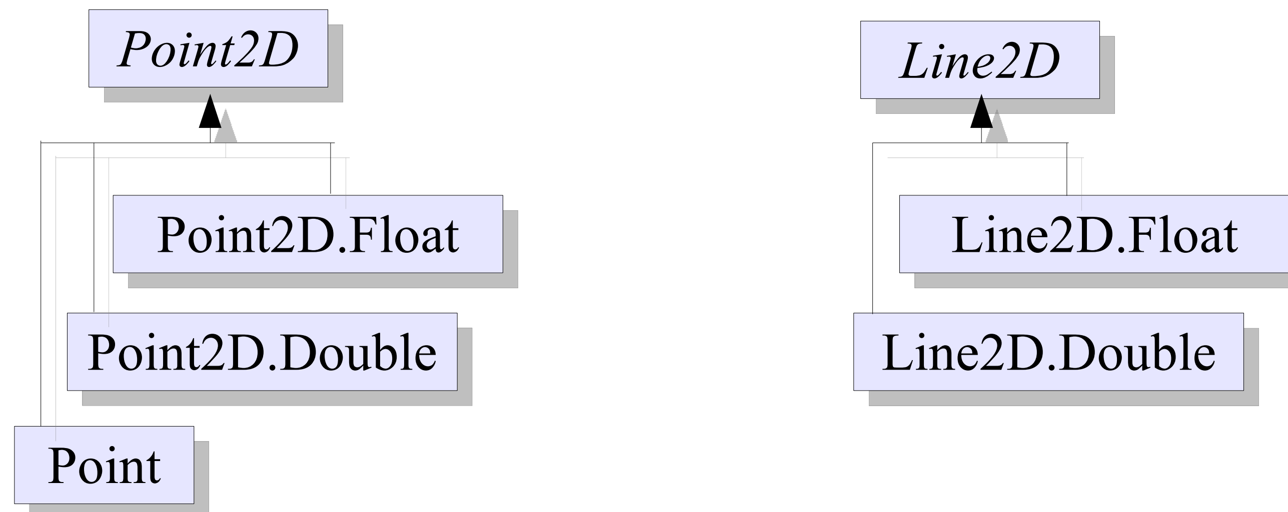
Shape

- **Shape** est une interface regroupant l'ensemble des formes qui peuvent être affichées.
- Dans le paquetage **java.awt.geom**
- **Polygon** est devenu un **Shape**.
- La classe **Point2D** n'est pas une forme !!



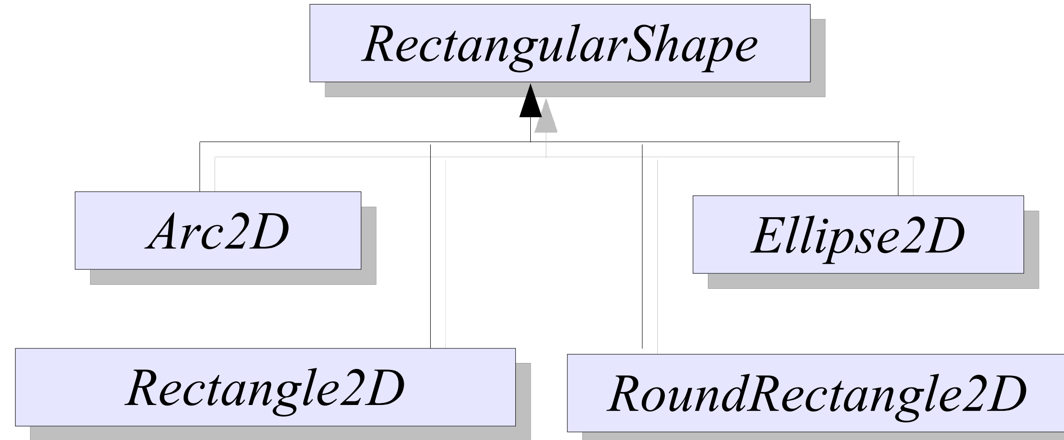
Cordonnées réelles

- Les coordonnées des formes sont des valeurs réelles ce qui permet d'indiquer des fractions de pixel
- Pour les transformations, l'anti-aliasing.
- Les classes abstraites **`Point2D`**, **`Line2D`**, **`CubicCurve2D`**, **`QuadCurves2D`** possèdent chacune deux classes internes Float et Double qui stocke les coordonnées sous forme de float ou de double.



RectangularShape

- **`RectangularShape`** est la classe abstraite de base des formes qui sont décrites par un rectangle.
- Les classes abstraites **`Arc2D`**, **`Ellipse2D`**, **`Rectangle2D`**, **`RoundRectangle2D`** possèdent elles aussi des coordonnées réelles.



Dessiner une forme

- La classe `Graphics2D` est une sous-classe de `Graphics`.
- La méthode `JComponent.paintComponent(Graphics g)` reçoit en fait un `Graphics2D`. De même pour `paint()`.
- On utilise un cast pour obtenir le `Graphics2D`

```
Graphics2D g2 = (Graphics2D)g;
```

- On utilise des méthodes

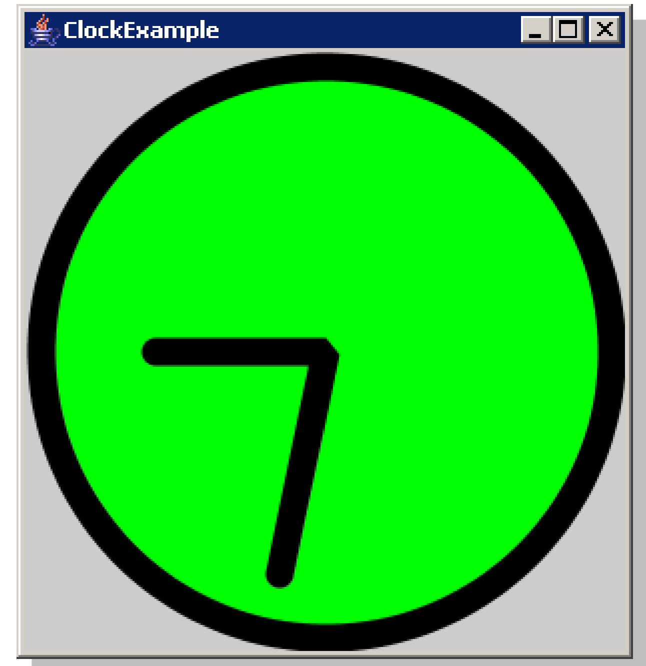
```
fill(Shape s)
draw(Shape s)
```

- Exemple

```
fill(new Rectangle2D.Float(x, y, 100, 100));
```

Trait

- Les traits se dessinent avec une plume de 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));
```

Exemple de l'horloge

- Le Bord de l'horloge et les aiguilles utilisent un *stroke*.

```
public class ClockExample extends JComponent {
  protected void paintComponent(Graphics graphics) {
    super.paintComponent(graphics);
    Graphics2D g= (Graphics2D)graphics;
    g.clearRect(0, 0, getWidth(), getHeight());
    g.setRenderingHint(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

    int width= Math.min(getWidth()-15,
      getHeight()-15);
    g.translate(getWidth()/2,getHeight()/2);
    g.setColor(Color.GREEN);
    g.fillOval(-width/2, -width/2,width,width);
    g.setColor(Color.BLACK);
    g.setStroke(new BasicStroke(
      14,
      BasicStroke.CAP_ROUND,
      BasicStroke.JOIN_BEVEL));
    g.drawOval(-width/2, -width/2, width,width);
```

```
    calendar.setTimeInMillis(
      System.currentTimeMillis());
    int hour= calendar.get(Calendar.HOUR);
    int minute= calendar.get(Calendar.MINUTE);
    double angleHour=2*Math.PI*(hour-3)/12;
    double angleMinute=2*Math.PI*(minute-
15)/60;

    GeneralPath path= new GeneralPath();
    path.moveTo(
      (int)(0.3*width*Math.cos(angleHour)),
      (int)(0.3*width*Math.sin(angleHour)));
    path.lineTo(0, 0);
    path.lineTo(
      (int)(0.4*width*Math.cos(angleMinute)),
      (int)(0.4*width*Math.sin(angleMinute)));
    g.draw(path);
  }
}
```

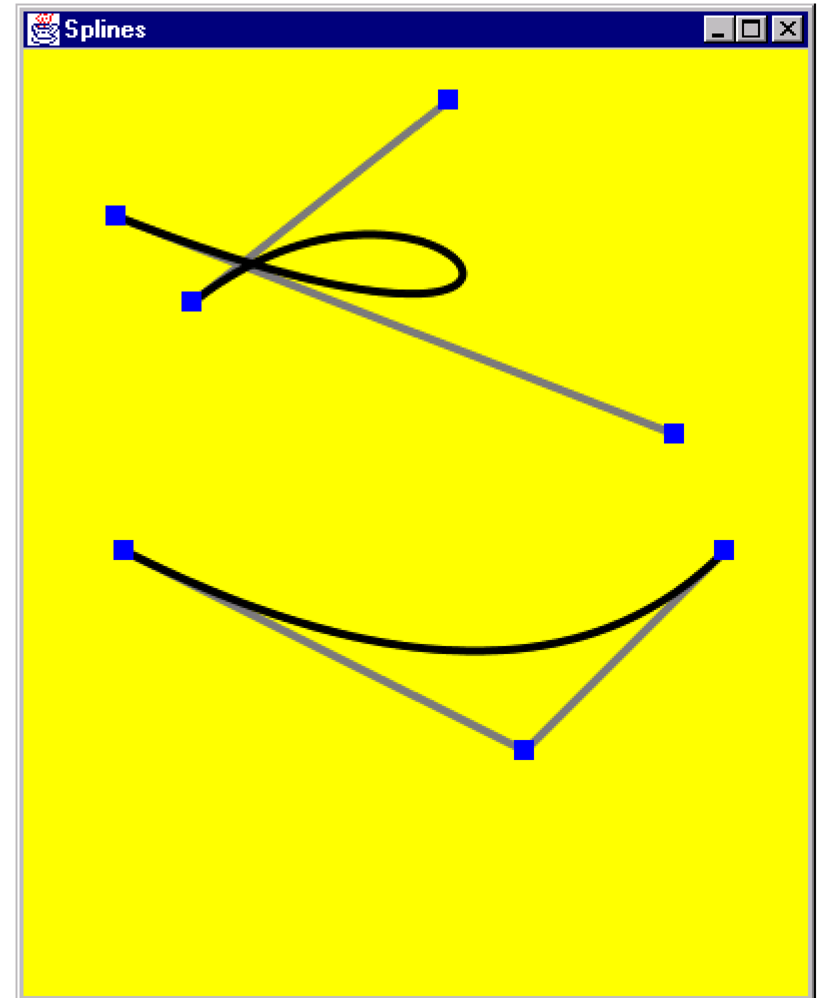
Exemple de l'horloge - 2

- On utilise **`paintImmediately()`** à la place de **`repaint()`**.

```
  public Dimension getPreferredSize() {
    return new Dimension(300,300);
  }
  private final Calendar calendar=
Calendar.getInstance();
  public static void main(String[] args) {
    Toolkit.getDefaultToolkit().setDynamicLayout(true);
    JFrame frame= new JFrame("ClockExample");
    final ClockExample component= new ClockExample();
    new Timer(999,new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        component.paintImmediately(0,0,
          component.getWidth(),component.getHeight());
      }
    }).start();
    frame.setContentPane(component);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Courbes de Bézier

- Elles sont quadratiques ou cubiques.
- Elles ont trois ou quatre points de contrôle, dont deux sont des extrémités.
- **QuadCurve2D**, trois points de contrôle.
- **CubicCurve2D**, quatres points de contrôle.



Courbes de bezier (2)

- Un constructeur des courbes, en double :

```
QuadCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1,
  double x2, double y2)
CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1,
  double ctrlx2, double ctrly2, double x2, double y2)
```

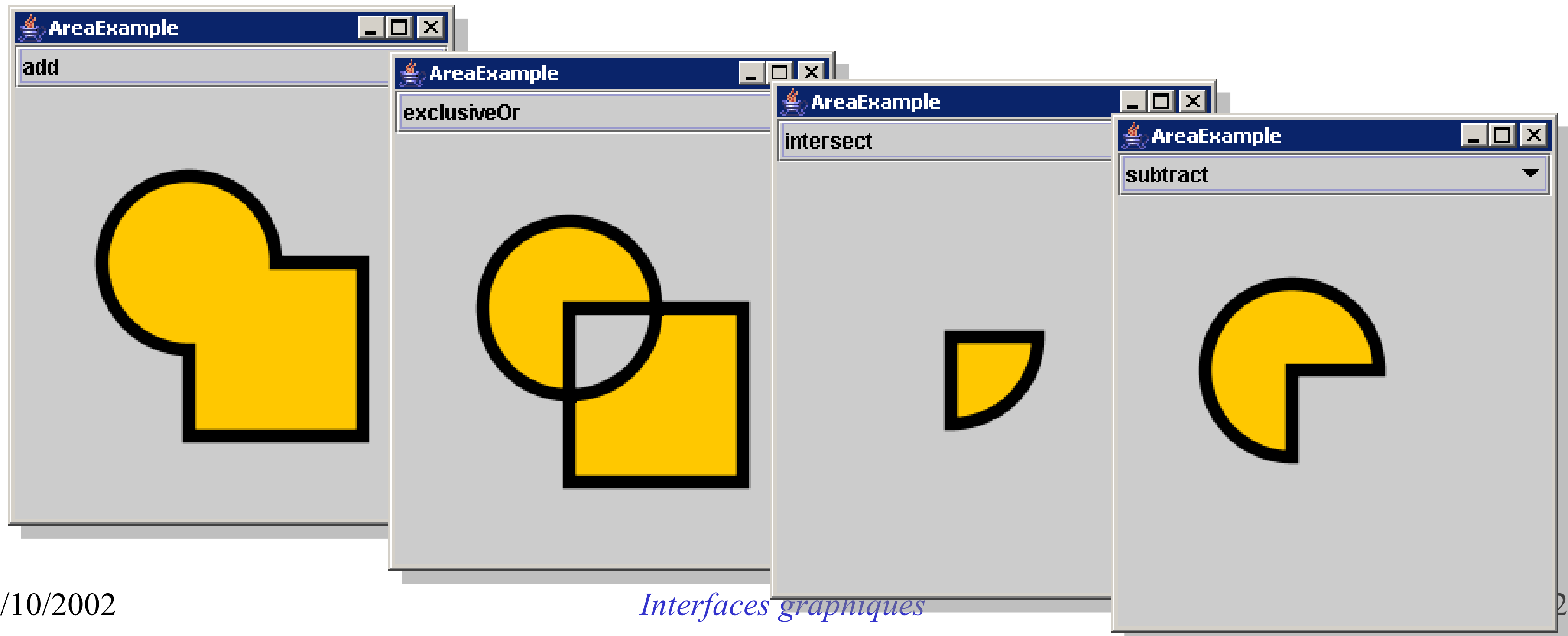
- Variantes, utiles pour changer les points,

```
setCurve(double[] coords, int offset)
setCurve(Point2D[] pts, int offset)
setCurve(Point2D p1, Point2D cp1, Point2D p2)
setCurve(Point2D p1, Point2D cp1, Point2D cp2, Point2D p2)
```

marchent pour les courbes quadratiques et cubiques.

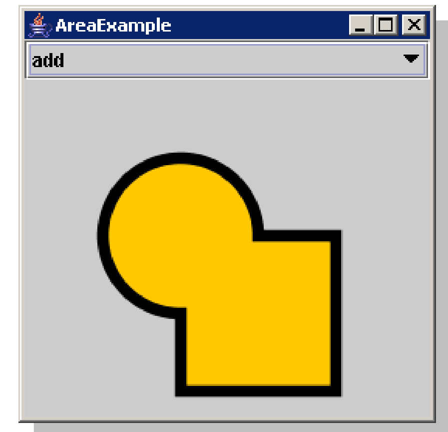
Aires

- Aires que l'on peut composer par des opérations booléennes (***constructive solid geometry*** en 2D)
- Un objet de la classe **`Area`** peut être construit à partir d'une forme (**`Shape`**) ou d'une opération sur des aires.
- Les opérations possible sont :



Aires - Exemple

- Les méthodes modifient la première aire.




```
public class AreaExample extends JComponent {
  protected void paintComponent(Graphics graphics) {
    super.paintComponent(graphics);
    Graphics2D g= (Graphics2D)graphics;
    g.clearRect(0, 0, getWidth(), getHeight());
    g.setRenderingHint(
      RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
    Area area=(Area)first.clone();
    op.apply(area, second);
    g.setColor(Color.ORANGE);
    g.fill(area);
    g.setColor(Color.BLACK);
    g.setStroke(stroke);
    g.draw(area);
  }
```

```
  public void setOp(CAGOp op) {
    this.op=op;
    paintImmediately(0,0,getWidth(), getHeight());
  }
  private CAGOp op;
  private static final BasicStroke stroke=
    new BasicStroke(7.5f);
  private static final Area first=
    new Area(new Ellipse2D.Float(50,50,100,100));
  private static final Area second=
    new Area(new Rectangle(100,100,100,100));

  interface CAGOp {
    void apply(Area area, Area anotherArea);
  }
```

Aires – Exemple - 2

```java
enum CAGOps implements CAGOP {
  add {
    public void apply(Area a1, Area a2) {
      a1.add(a2);
    }
  }, exclusiveOr {
    public void apply(Area a1, Area a2) {
      a1.exclusiveOr(a2);
    }
  }, intersect {
    public void apply(Area a1, Area a2) {
      a1.intersect(a2);
    }
  }, substract {
    public void apply(Area a1, Area a2) {
      a1.subtract(a2);
    }
  }
}
```

```java
  public static void main(String[] args) {
    JFrame frame= new JFrame("AreaExample");
    final AreaExample component= new AreaExample();
    CAGOp ops=CAGOps.values();
    component.setOp(ops[0]);
    final JComboBox comboBox=new JComboBox(ops);
    comboBox.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        component.setOp((CAGOp)comboBox.
          getSelectedItem());
      }
    });

    frame.getContentPane().add(comboBox,
      BorderLayout.NORTH);
    frame.getContentPane().add(component);
    frame.setSize(400,300);
    frame.setVisible(true);
  }
}
```

Clipping

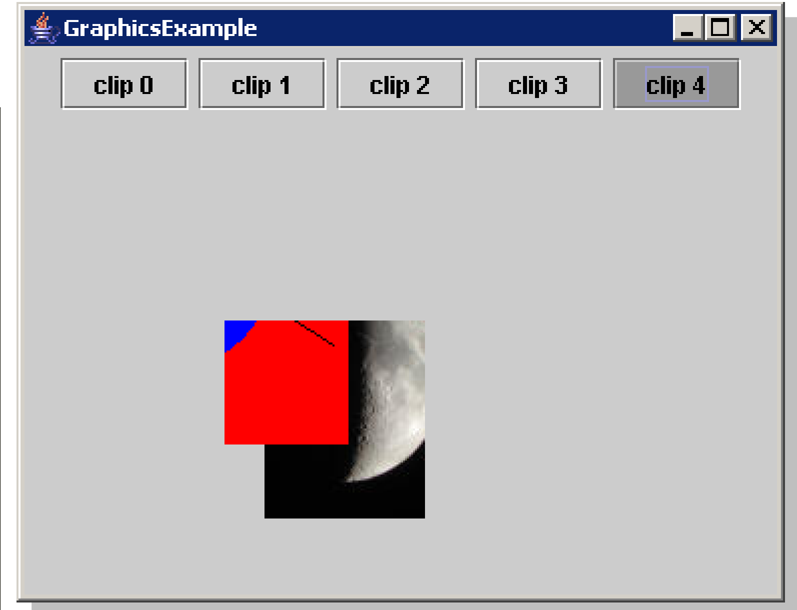
- Permet de restreindre la zone dans lequel le dessin sera effectué.

- Deux façons de procéder :
  - Sur le **`Graphics`**, **`setClip(Shape shape)`**.
  - Appeler la méthode **`repaint(x,y,width,height)`** ou **`paintImmediately(x,y,width,height)`** qui effectuent automatiquement le clipping.

- Le clipping sert à deux choses :
  - Eviter de *déborder* en dessinant (par exemple, les composants).
  - Effectuer un affichage plus rapide.

Exemple de Clipping

◆ Sélectionne la zone visible.




```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (shape!=null)
      g.setClip(shape);

    g.drawImage(image, 120, 50, this);
    g.setColor(Color.RED);
    g.fillRect(12,12,150,150);
    g.setColor(Color.BLUE);
    g.fillOval(24, 24, 100, 100);
    g.setColor(Color.BLACK);
    g.drawLine(34,34,154,112);
    g.setFont(font);
    g.drawString("Example",12,32);
  }
  public void setShape(Shape shape) {
    this.shape=shape;
    paintImmediately(0,0,getWidth(),getHeight());
  }
  private Shape shape;
```

Couleur

- La classe `Color` permet de gérer les couleurs.
- Constantes (prendre celles en MAJUSCULE)

  BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW.
- Constructeurs RGB,

```
Color(int red,int green,int blue)
```

- Conversion **RGBtoHSB** (hue, saturation, brightness) et vice-versa.
- La classe dérivée `SystemColor` contient des noms symboliques pour les couleurs du système: contrôle, fenêtre active, menu, ombre (inspiré de Windows).
- Le coefficient alpha indique la transparence (0.0 = opaque, 1.0 = transparent)

Dégradés et Textures

- Les classes **`Color`**, **`GradientPaint`** et **`TexturePaint`** implémentent **`Paint`**.
- **`GradientPaint`** crée un dégradé entre deux couleurs données en deux points

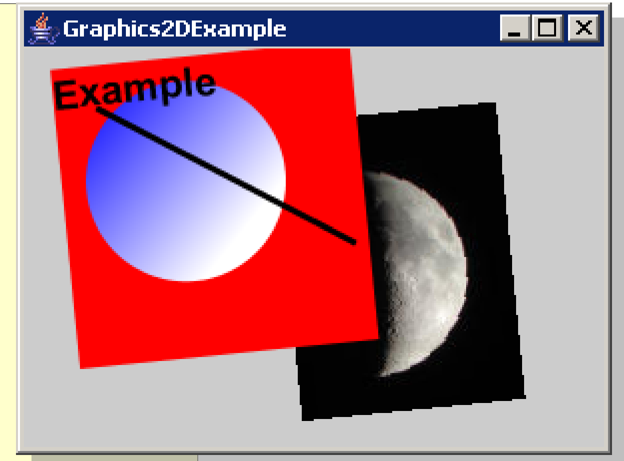
```
Paint paint = new GradientPaint(0, 0, Color.RED,
  getWidth()/2, getHeight()/2, Color.BLUE);
g2.setPaint(paint);
g2.fill(ellipse);
```

- **`TexturePaint`** répète une image plaquée dans un rectangle jusqu'à remplir la forme

```
Rectangle2D anchor = new Rectangle2D.Double(0, 0,
  4*image.getWidth(), 4*image.getHeight());
Paint paint = new TexturePaint(bufferedImage, anchor);
g2.setPaint(paint);
g2.fill(ellipse);
```

Exemple de Dégradés

- **`setPaint()`** accepte des couleurs ou un gradient.




```
protected void paintComponent(Graphics graphics) {
  super.paintComponent(graphics);
  Graphics2D g=(Graphics2D)graphics;
  g.setRenderingHint(
    RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
  g.rotate(-0.1);
  g.drawImage(image, 120, 50, this);
  g.setPaint(Color.RED);
  g.fillRect(12,12,150,150);
  g.setPaint(paint);
  g.fill(ellipse);
  g.setPaint(Color.BLACK);
  g.setStroke(stroke);
  g.drawLine(34,34,154,112);
  g.setFont(font);
  g.drawString("Example",12,32);
}

private final GradientPaint paint=
  new GradientPaint(24,24,Color.BLUE,100,100,Color.WHITE);
private final Stroke stroke=new BasicStroke(3);
private final Shape ellipse=new Ellipse2D.Float(24, 24, 100, 100);
private final Font font=new Font("SansSerif",Font.BOLD,20);
private final Image image=new ImageIcon("moon.jpg").getImage();
```

Composition

- Il y a 12 modes de composition (Porter-Duff) de l'image construite avec l'image existante, représentés par des constantes de la classe `AlphaComposite`.
- Le coefficient alpha ne change pas ces modes, mais atténue seulement l'impact de l'image construite.
- ***s*** = alpha de source, ***d*** = alpha de destination, l'apha du mélange est alors ***s(1-d)*** ou ***d(1-s)*** .
- On choisit le style de composition par :

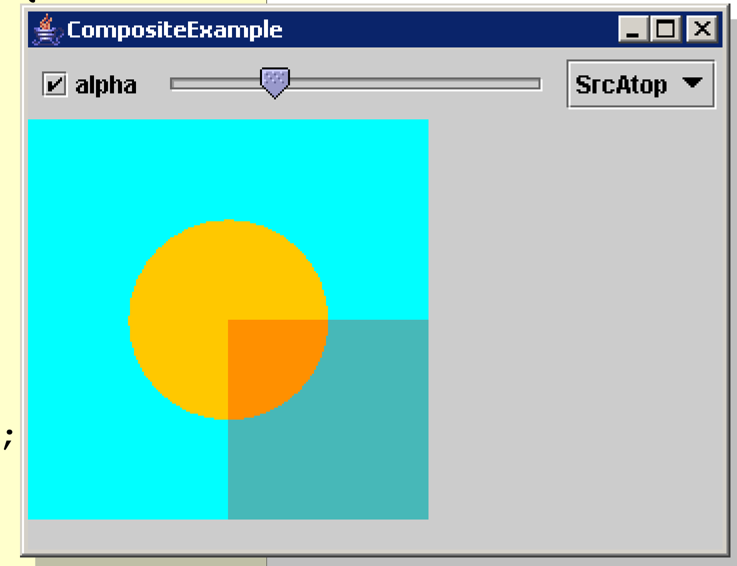
```
Composite composite = AlphaComposite.getInstance(rule, alpha);
g2.setComposite(composite);
```

- Encore faut-il que l'écran accepte une "couche alpha". En général, c'est non, et on utilise alors un buffer intermédiaire.

Exemple de Composition

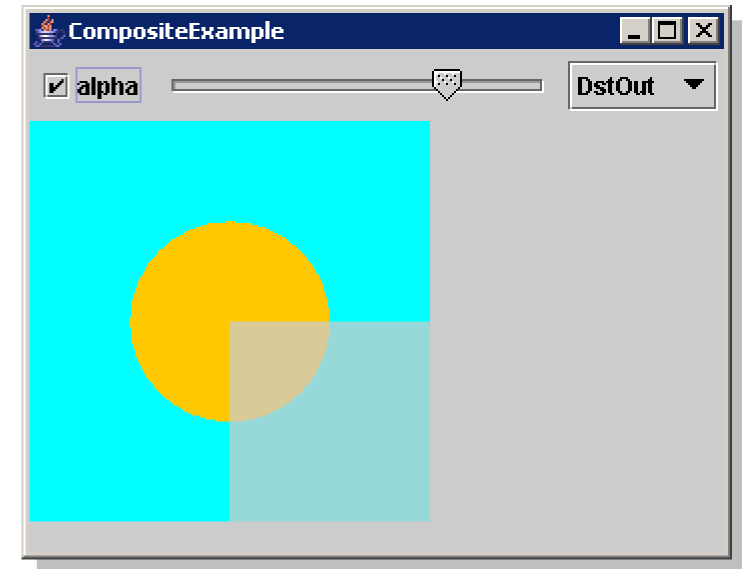
- La combo permet de choisir la règle, le slider l'alpha.

```
public class AlphaCompositionExample extends JComponent {
  protected void paintComponent(Graphics graphics) {
    super.paintComponent(graphics);
    Graphics2D g= (Graphics2D)graphics;
    g.clearRect(0, 0, getWidth(), getHeight());
    g.drawImage(image,0,0,null);
  }
  public void setComposite(Composite composite) {
    Graphics2D g2=(Graphics2D)image.getGraphics();
    g2.setColor(Color.CYAN);
    g2.fillRect(0,0,image.getWidth(),image.getHeight());
    g2.setColor(Color.ORANGE);
    g2.fill(ellipse);
    if (composite!=null)
      g2.setComposite(composite);
    g2.setColor(Color.RED);
    g2.fill(rectangle);
    g2.dispose();
    paintImmediately(0,0,getWidth(),getHeight());
  }
  private final BufferedImage image=
    new BufferedImage(200,200,BufferedImage.TYPE_INT_ARGB);
```



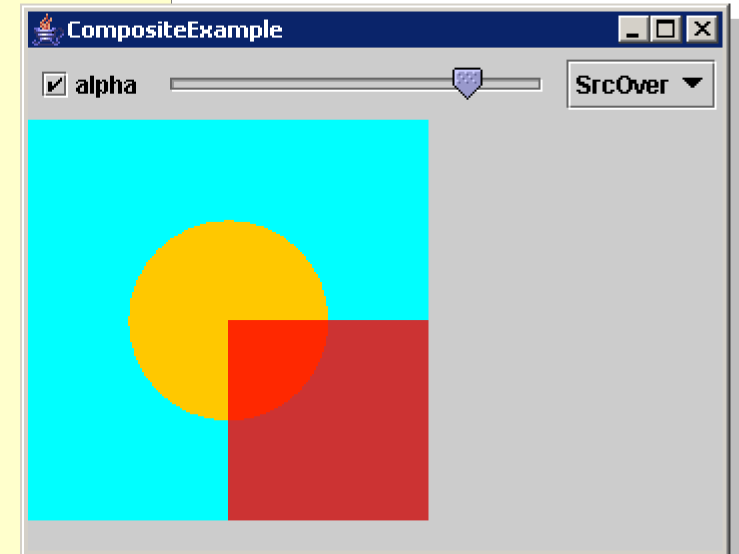
Exemple de Composition - 2

```
public class Aprivate static final Shape ellipse=
  new Ellipse2D.Float(50,50,100,100);
private static final Shape rectangle=
  new Rectangle(100,100,100,100);
static AlphaComposite getFieldValue(String name) {
  try {
    Field field=AlphaComposite.class.getField(name);
    return (AlphaComposite)field.get(null);
  } catch (NoSuchFieldException e) {
    throw new AssertionError(e);
  } catch (IllegalArgumentException e) {
    throw new AssertionError(e);
  } catch (IllegalAccessException e) {
    throw new AssertionError(e);
  }
}
void applyComposite(JCheckBox check,JSlider slider,
  JComboBox combo) {
  String rule=(String)combo.getSelectedItem();
  AlphaComposite composite=getFieldValue(rule);
  if (check.isSelected()) {
    float alpha=slider.getValue()/100.0f;
    composite=AlphaComposite.getInstance(
      composite.getRule(), alpha);
  }
  setComposite(composite); }
```



Exemple de Composition - 3

```
  public static void main(String[] args) {
    final AlphaCompositionExample component=
      new AlphaCompositionExample();
     String[] rules=new String[]{
      "Clear", "Dst",    "DstAtop", "DstIn",
      "DstOut","DstOver", "Src",      "SrcAtop",
      "SrcIn", "SrcOut",  "SrcOver", "Xor"
     };
    final JSlider slider=new JSlider(0,100);
    final JCheckBox checkBox=new JCheckBox("alpha");
    checkBox.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        slider.setEnabled(checkBox.isSelected());
      }
    });
    final JComboBox comboBox=new JComboBox(rules);
    ActionListener listener=new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        component.applyComposite(checkBox, slider,
comboBox);
    } };
    checkBox.addActionListener(listener);
    comboBox.addActionListener(listener);
    slider.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        component.applyComposite(checkBox, slider,
comboBox);
    } });
```



Transformations Affines

- Les transformations affines servent à modifier les coordonnées utilisateur avant affichage
- Par exemple, le repère peut être centré au milieu de la zone de dessin.
- Les transformations sont :
  - ***Rotation (rotate)***
  - ***Translation (translate)***
  - ***Dilatation (scale)***
  - ***Cisaillement (shear)***
- La classe **`AffineTransform`** permet de créer et de composer des transformations affines.

Transformations Affines

- Mathématiquement, une transformation affine est représentée par une matrice 3 x 3 dite “***matrice en coordonnées homogènes***” dont la dernière ligne est toujours (0 0 1).

- Seuls les 6 autres coefficients sont conservés. On peut donner ces coefficients explicitement, ou les faire calculer en fonction de la nature de l’opération recherchée.

- La Création s'effectue par l'intermédaire de méthode statique (*factory*).

Transformations Affines - Création

- Création de rotation :

```
AffineTransform.getRotateInstance(double theta)
```

```
AffineTransform.getRotateInstance(double theta, double x, double y)
```

- Création de translation :

```
AffineTransform.getTranslateInstance(double tx, double ty)
```

- Création de dilatation :

```
AffineTransform.getScaleInstance(double sx, double sy)
```

- Création de cisaillement :

```
AffineTransform.getShearInstance(double shx, double shy)
```

Transformations Affines - Composition

- Il est possible de composer les transformations en appliquant une transformation à une transformation existante.

- Rotation

```
transform.rotate(double theta)
```

- Translation

```
transform.translate(double tx, double ty)
```

- Dilatation

```
transform.scale(double sx, double sy)
```

- Cisaillement

```
transform.shear(double shx, double shy)
```

Transformations affines implicites

- L'affichage, lors de l'exécution d'un **`paintComponent()`**, est optimisé. Seule la zone qui doit être rafraîchie l'est vraiment, et cela dépend bien sûr de l'événement qui a provoqué l'affichage.

- Le contexte graphique maintient une transformation affine qui contient la translation du composant d'affichage par rapport au rectangle de réaffichage. Cette transformation *implicite* ne doit pas être ignorée, mais utilisée.

- Il faudra donc penser à conserver cette transformation.

Utiliser les Transformations Affines

- Lorsque l'on applique une transformation sur le **Graphics**, il faut penser à conserver les transformations antérieures.

- Composer la transformation affine avec celle existante :

```
AffineTransform transform=...
g2.transform(transform);
```

- Récupérer la transformation existante :

```
AffineTransform transform=g2.getTransform();
transform.rotate(...);
g2.setTransform(transform);
```

getTransform(), effectue une copie.

Contrôle du rendu

- Le rendu peut-être amélioré (aux dépens de la rapidité) par un ensemble de “hints” (conseils).
- Chaque conseil concerne un aspect et indique un souhait (peut ne pas être exhaucer).
- Un conseil se présente donc comme un couple : clé d’une propriété et valeur de cette propriété.

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
g2.setRenderingHint(RenderingHints.KEY_RENDERING,
  RenderingHints.VALUE_RENDER_QUALITY);
```

- On peut aussi écrire :

```
RenderingHints hints=...
g2.setRenderingHints(hints);
```

Interpolation des images

- Demande une interpolation bicubique sur l'image.

```
protected void paintComponent(Graphics graphics) {
  super.paintComponent(graphics);
  Graphics2D g=(Graphics2D)graphics;
  if (interpolation) {
    g.setRenderingHint(
      RenderingHints.KEY_INTERPOLATION,
      RenderingHints.VALUE_INTERPOLATION_BICUBIC);
  }
  g.drawImage(image, 0, 0, getWidth(), getHeight(), this);
}
public Dimension getPreferredSize() {
  return new Dimension(
    image.getWidth(this),
    image.getHeight(this));
}
public void setInterpolation(boolean interpolation) {
  this.interpolation=interpolation;
  paintImmediately(0,0,getWidth(),getHeight());
}
private boolean interpolation;
private final Image image=
  new ImageIcon("moon.jpg").getImage();
```