Interface Homme-Machine 2

Graphiques 2D et animations en Java

Éléments de base

Jacques Bapst

jacques.bapst@hefr.ch

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Quels outils sont disponibles en Java ?

- AWT / Swing
  - Icônes
  - Labels et Boutons
  - Bordures
  - Textes + Styles
    - ✓Label, Font
    - ✓JEditorPane
    - ✓JTextPane
- Java AWT Graphics
  - Assez rudimentaire
  - Formes, Textes, Images
- Java2D Graphics (dès SDK 1.2, fait partie de la plate-forme standard)
  - Étend et complète la librairie AWT Graphics
  - Nombreuses améliorations

Pourquoi utiliser la librairie Java2D ?

- **Personnaliser** l'apparence d'un composant
  - Au-delà de ce que permet la configuration de ses propriétés
- Créer son **propre composant** visuel *(widget)*
- **Dessiner** des courbes, des formes, des textures à l'écran
- Afficher des **textes**
  - Non horizontaux (orientation)
  - Avec un positionnement calculé (libellé d'une courbe par exemple)
  - Avec ligne de base quelconque (gestion de l'interligne)
- Créer une **animation**
- **Imprimer**
  - *Java Print Service* (JPS) est fortement basé sur *Graphics2D*

Que comprend Java2D ?

- La librairie Java2D comprend un ensemble de classes qui se trouvent pour la plupart dans les *packages* **`java.awt`**, **`java.awt.geom`** et **`java.awt.image`**.
- Certaines classes habituelles de *Swing* sont également utilisées dans Java2D, par exemple **`Color`** et **`Font`**.
- Parmi les classes principales de Java2D on peut mentionner :
  - **`Graphics`** / **`Graphics2D`**
  - **`Point2D`**
  - ***`Shape`*** : **`Line2D`**, **`Arc2D`**, **`Rectangle2D`**, **`Ellipse2D`**, **`GeneralPath`**, ...
  - ***`Stroke`*** : **`BasicStroke`**
  - ***`Paint`*** : **`GradientPaint`**, **`TexturePaint`**
  - ***`Composite`*** : **`AlphaComposite`**
  - **`AffineTransform`**
  - **`RenderingHints`**
  - ***`Image`*** : **`BufferedImage`**

Comment utiliser Java2D ?

- Par défaut, Java ne permet pas de dessiner directement sur un périphérique (on ne peut pas accéder directement à la mémoire écran par exemple).
- Le tracé graphique doit s'effectuer dans les limites d'un objet de type **`Component`** en utilisant son **contexte graphique** qui représente la surface de dessin du composant (un objet de type **`Graphics`** ou **`Graphics2D`**) .
- Par conséquent, pour réaliser des opérations graphiques, il faut disposer d'un composant capable de prendre en charge les opérations et le rendu du dessin.
- Le dessin est généralement effectué **en spécialisant un composant existant** (création d'une sous-classe) et en redéfinissant sa méthode de dessin (*painting*) qui dispose du contexte graphique de ce composant.

Painting

- Demander au système le dessin d'un composant :

  `repaint()`

  - Cette méthode est **appelée automatiquement** par le système lorsque c'est nécessaire (par exemple lorsque un composant masqué par une autre fenêtre redevient visible).
  - La méthode **`revalidate()`** qui se charge de recalculer (récursivement) la disposition des composants est appelée automatiquement, si nécessaire, avant l'appel à **`repaint()`**.

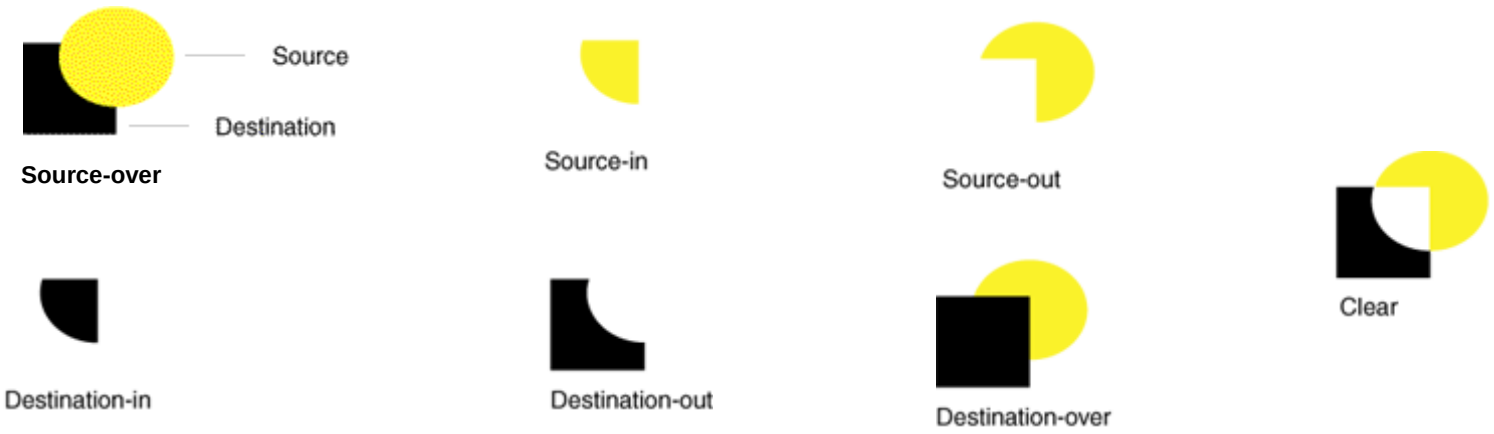
- Le dessin des composants s'effectue dans le *thread* de *Swing EDT (Event-Dispatching Thread).*
  - Le dessin des composants s'effectue après le traitement de tous les événements qui sont en attente dans la file *EventQueue.*
  - Les méthodes **`repaint()`** et **`revalidate()`** sont *thread safe* (elles peuvent être appelées depuis n'importe quel *thread*.

Opacity / Double buffering

- Les composants graphiques peuvent être **opaques** ou avoir une certaine transparence (*Alpha-Channel*).
- Les composants opaques nécessitent nettement moins de ressources système (temps de calcul) car il n'est pas nécessaire de dessiner les composants qui sont derrière lui en prenant en compte le mélange des couleurs (selon les règles de composition définies).
- Pour améliorer la qualité du rendu graphique (et notamment éviter le scintillement *(flickering)* de l'affichage), on utilise fréquemment la technique du **Double buffering** (appelé aussi *Back buffering*) qui consiste à dessiner d'abord dans une image interne *(Buffered Image)* et à copier ensuite cette image sur écran (en une seule opération rapide).

Règles de composition

- Les règles de composition qui s'appliquent lorsque l'on superpose plusieurs éléments graphiques sont définis par la propriété *composite* du contexte graphique.
- La classe **`AlphaComposite`** implémente les règles de base (celles définies par *Porter and Duff*) et permet de créer (avec **`getInstance`**`(`*rule*`)`) un objet définissant ces règles.
- Exemples (www.curious-creature.org/2006/09/20/new-blendings-modes-for-java2d)

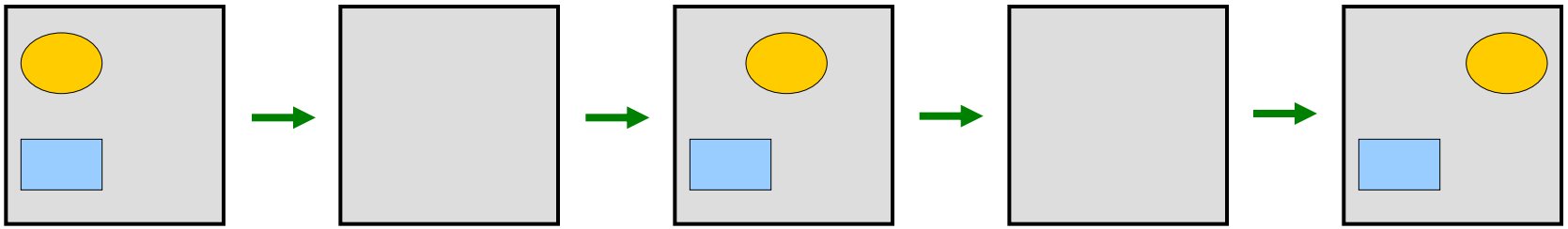


Scintillement / Animation

- Le scintillement *(flickering)* pose problème :
  - Si l'on a de nombreux objets à dessiner
  - Si le dessin des objets nécessite d'importants calculs intermédiaires
  - Si l'on veut créer une animation qui est généralement constituée par la répétition de deux activités :

Dessin de l'arrière-plan (ou effacement partiel)

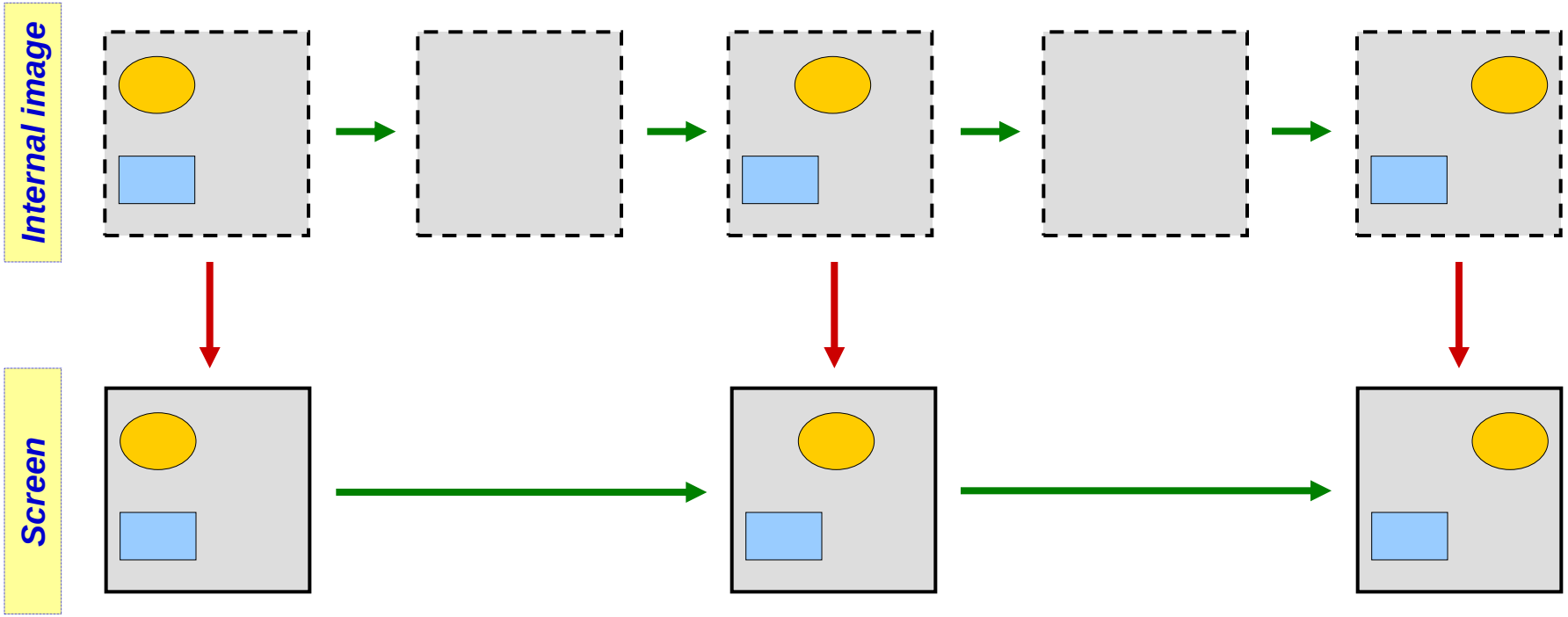
Dessin des composants à leur nouvelle position



Animation

Double buffering / Animation

- On peut réduire notablement le scintillement par l'utilisation du **double-buffering**.
- Dessin préalable dans une image interne puis transfert global (rapide) dans le contexte graphique de l'écran.



Visibilité et ordre de dessin

- La **visibilité** des composants est basée sur la hiérarchie de confinement des conteneurs. Certains conteneurs sont monocouche, d'autres sont multicouche *(Layered)*.
- Dans la librairie *Swing*, le **dessin** d'un composant (**`paint()`**) déclenche, dans cet ordre, les opérations suivantes :
  - Dessin de l'arrière plan (généralement dans la classe de base)
  - Dessin du composant lui-même **`paintComponent()`**
  - Dessin de la bordure du composant **`paintBorder()`**
  - Dessin des composants enfants **`paintChildren()`**
- Dans la librairie AWT, les algorithmes utilisés sont légèrement différents.
- Pour plus de détails : www.oracle.com/technetwork/java/painting-140037.html

Comment créer un graphique ?

- La technique la plus simple consiste à spécialiser le composant **`JPanel`** qui est un conteneur générique ne possédant aucun élément décoratif (c'est un composant opaque qui possède uniquement une couleur d'arrière-plan que l'on peut modifier en invoquant la méthode **`setBackground()`**).

- Il suffit donc créer une sous-classe de **`JPanel`** :

```
public class myGraphicPanel extends JPanel {
   . . .
}
```

Où placer les instructions de dessin ?

- Dans la sous-classe de **`JPanel`**, on redéfinira la méthode héritée **`paintComponent()`** dans laquelle on placera les instructions de dessin :

```
public class myGraphicPanel extends JPanel {

  protected void paintComponent(Graphics g) {
     . . . . .
     . . . . .     } Instructions de dessin du composant
     . . . . .
  }

}
```

Dessin de l'arrière-plan

- Avant de dessiner le composant proprement dit, il faut généralement appeler la méthode **`paintComponent()`** de la classe parente afin de redessiner l'arrière plan (et donc d'effacer le dessin courant, s'il y en a un).

```
public class myGraphicPanel extends JPanel {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    . . . . .
    . . . . .      } Instructions de dessin du composant
    . . . . .
  }
}
```

- Si l'on souhaite, par contre, créer un composant transparent, on invoquera la méthode **`setOpaque(false)`**.

Qui appelle *paintComponent()* ?

- La méthode **`paintComponent()`** ne devrait jamais être appelée explicitement par le programmeur car elle n'est pas *thread-safe*.
- Son exécution doit impérativement s'effectuer dans le *Event Dispatch Thread* de Swing.
- Elle sera automatiquement invoquée par le système lorsque c'est nécessaire. Son invocation n'est donc pas sous le contrôle du programmeur.
- On peut provoquer son exécution en invoquant la méthode **`repaint()`** du composant qui est, elle, *thread-safe*.
- Il existe une surcharge de la méthode **`repaint()`** qui permet de ne redessiner qu'une zone rectangulaire :

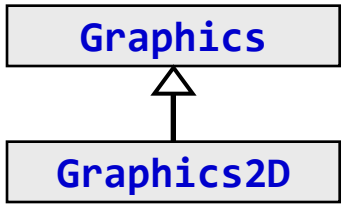
```
repaint(int leftx, int lefty, int width, int height)
```

Contexte graphique [1]

- La méthode **`paintComponent()`** reçoit en paramètre un objet de type **`Graphics`** qui représente le **contexte graphique** dans lequel sera dessiné le composant.
- Le **contexte graphique** est un objet qui :
  - **Représente la surface de dessin** qui peut être :
    - ✓Un composant visuel (sous-classe de **`Component`**)
    - ✓Une image hors-écran (sous-classe concrète de **`Image`**)
    - ✓Une zone destinée à être imprimée
  - **Mémorise l'état** de différents attributs graphiques (couleur, police de caractère, type et épaisseur du trait, type de jointure, région de découpage, etc.)
  - **Offre des méthodes** pour effectuer diverses opérations graphiques : traçage de lignes, traçage de formes, remplissage de surfaces, affichage de textes, copie d'images sur la surface de dessin, etc.

Contexte graphique [2]

- A partir de Java 2, tous les objets de type **Graphics** dans la librairie AWT sont en réalité des instances de **Graphics2D**.



- Il suffit donc d'un transtypage *(casting)* pour obtenir la référence de l'instance **Graphics2D** :

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  . . . . .
  g2d.…              // Instructions de dessin
  . . . . .
}
```

Primitives de dessin

- Le contexte graphique dispose notamment des deux principales méthodes de dessin :

`draw(`*Shape*`)`

`fill(`*Shape*`)`

- Le paramètre de ces méthodes, de type **`Shape`**, définit la forme à dessiner ou à remplir.
- De nombreuses méthodes (plus anciennes) existent :
  - `drawLine()`, `drawRect()`, `drawPolygon()`, `drawOval()`, `...`
  - `fillArc()`, `fillRect()`, `fillPolygon()`, `fillOval()`, `...`

Propriétés du contexte graphique

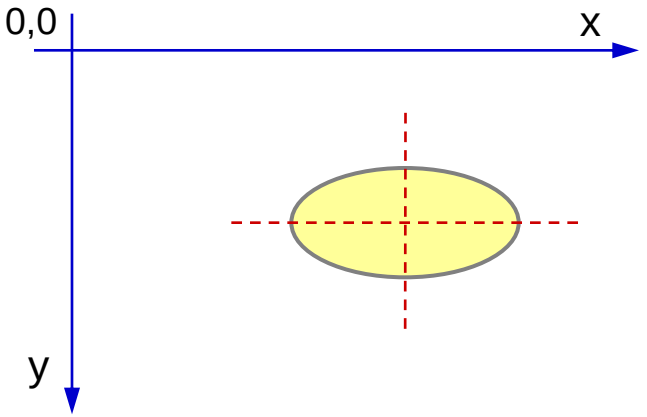
- Le contexte graphique (`Graphics2D`) dispose notamment des propriétés suivantes :

`paint`	Couleur, motif et texture qui seront utilisés par les méthodes `draw()` et `fill()`
`stroke`	Épaisseur style et type des lignes tracées par la méthode `draw()`
`transform`	Transformations géométriques (`AffineTransform`) appliquées avant le rendu du dessin (translation, rotation, ...)
`composite`	Méthode de composition du dessin courant avec ce qui est déjà dessiné (notamment gestion de la transparence des couleurs)
`clip`	Limite les opérations de dessin à une zone *(Clipping Area)*
`renderingHints`	Définition de différents paramètres de rendu du dessin. Gestion du compromis entre rapidité et qualité du rendu (*antialiasing*, *dithering*, *interpolation*, ...)
`font`	Police de caractères utilisée pour les textes `drawString()`

Systèmes de coordonnées

- Par défaut, le système de coordonnées de la surface de dessin est définit ainsi :

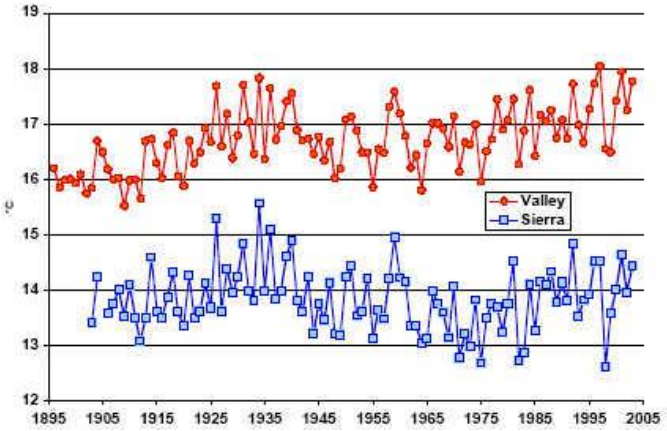
  (identique dans AWT/Swing)



- On distingue deux espaces de coordonnées :
  - Les coordonnées utilisateur *(**User coordinates**)*
  - Les coordonnées des périphériques *(**Device coordinates**)*
- Une **transformation** *(Affine transform)* permet de passer des coordonnées utilisateurs aux coordonnées des périphériques. La transformation par défaut garde la même origine, la même orientation et l'échelle vaut 1 (voir *Unités*).

Unités / Métrique

- La définition des unités utilisées dans les primitives de dessin (avec la transformation par défaut) diffère selon le type de périphérique utilisé pour la représentation.
- **Écran** et **images internes** :
  - 1 unité = 1 pixel
- **Imprimante** ou autres périphériques à *"haute résolution"* :
  - 1 unité = 1 point ($^{1}/_{72^{e}}$ de pouce ≈ 0.353 mm)
- Les unités peuvent être fractionnaires (`float` ou `double`)
- La définition d'une transformation permet de travailler avec les unités de l'espace utilisateur (espace du problème) ce qui rend le code plus lisible.



Transformations [1]

- Une **transformation affine** est utilisée pour passer des coordonnées utilisateur aux coordonnées des périphériques.
- Une matrice de transformation est appliquée à chaque point {x, y} du système de coordonnées utilisateur afin de calculer sa position {x', y'} dans le système de coordonnées des périphériques.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m00*x + m01*y + m02 \\ m10*x + m11*y + m12 \\ 1 \end{bmatrix}$$

Remarque : *après une telle transformation linéaire les droites restent des droites et le parallélisme est préservé.*

- Une transformation affine permet d'effectuer des translations, des rotations, des changements d'échelles, des cisaillements ainsi que des combinaisons de ces opérations.

Transformations [2]

- On peut créer une transformation générale en créant un objet de type **`AffineTransform`** qui dispose de différentes méthodes pour définir les valeurs de la matrice de transformation et en appelant ensuite la méthode **`transform()`** du contexte graphique (attention à ne pas utiliser **`setTransform()`**).
- Pour effectuer des transformations élémentaires, il est plus simple d'utiliser les méthodes mises à disposition par le contexte graphique :
  - **`translate(`***dx*`, `*dy*`)`** *Translation*
  - **`rotate(`***angle*`)`** *Rotation*
  - **`scale(`***fx*`, `*fy*`)`** *Facteur d'échelle (homothétie)*
  - **`shear(`***shx*`, `*shy*`)`** *Cisaillement*

Transformations [3]

- Exemple

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  AffineTransform saveAT = g2d.getTransform();
  AffineTransform at     = new AffineTransform();
  at.scale(0.5, 0.5);
  at.rotate(Math.toRadians(30));
  g2d.transform(at);           // ≠ setTransform() !
  . . . . .
  g2d.setPaint(…);             // Instructions de dessin
  g2d.draw(…);
  . . . . .
  g2d.setTransform(saveAT);    // Restaure la transformation
}
```

Copie du contexte graphique

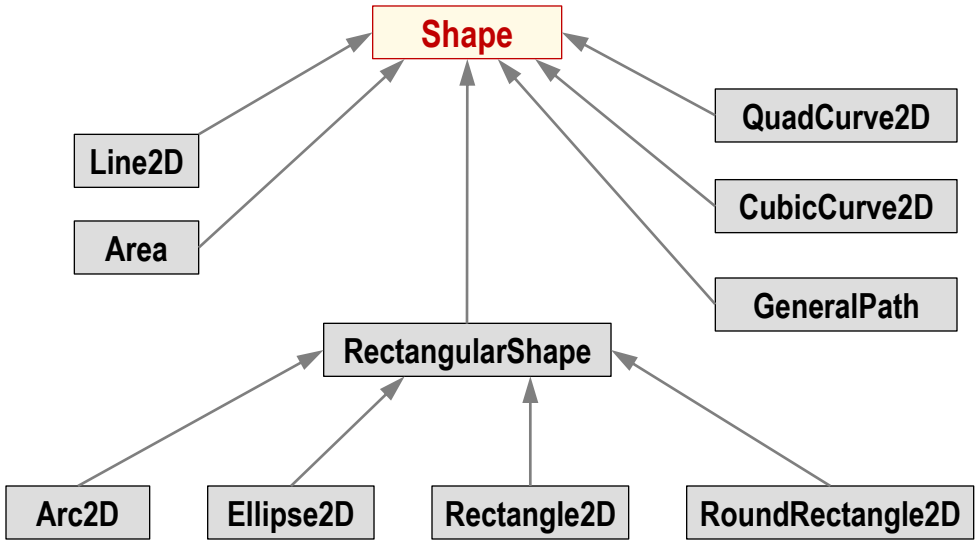
- La méthode **`create()`** (de la classe **`Graphics`**) permet de créer une copie d'un contexte graphique donné.
- Cela permet d'éviter de modifier (par exemple dans le contexte d'une méthode chargée de traiter une partie du graphique) les propriétés globales du contexte graphique et évite ainsi d'avoir à les remettre dans l'état initial.

```
public void paintCarSensors(Graphics2D g) {
  Graphics2D gCopy = (Graphics2D)g.create();
  . . . . .
  gCopy.setPaint(…);
  gCopy.setStroke(…);
  gCopy.draw(…);
  . . . . .
}
```

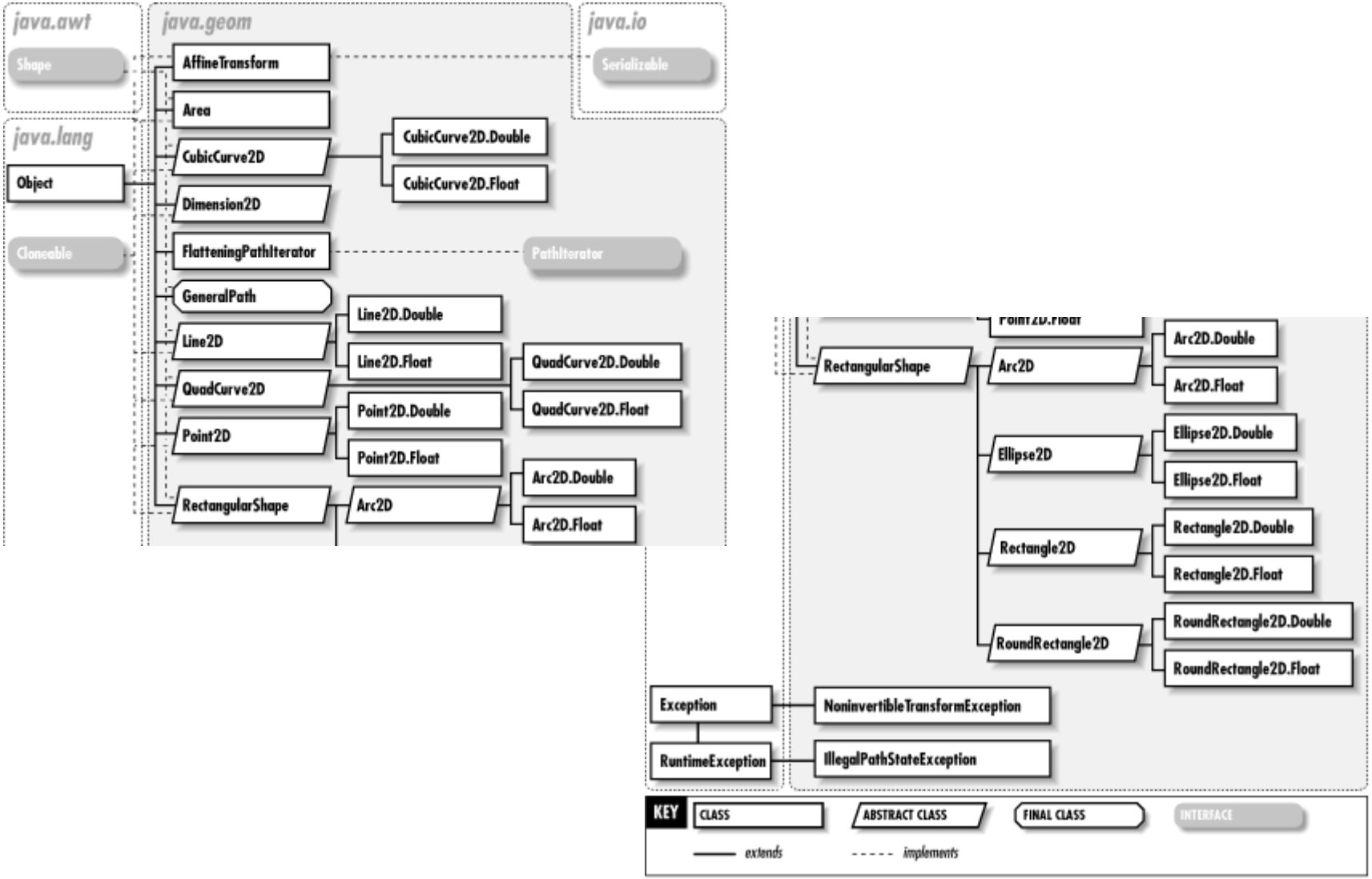
Formes (Shape)

- Un ensemble de classes implémentant l'interface **Shape** se trouvent dans le package `java.awt.geom`.
- Pour chaque genre de forme, on trouve généralement une classe abstraite comportant deux classes internes (concrètes) appelées **Float** et **Double** selon la précision des coordonnées associées.
- Exemples :

```
Line2D.Float
Line2D.Double

Rectangle2D.Float
Rectangle2D.Double

Ellipse2D.Float
Ellipse2D.Double
```

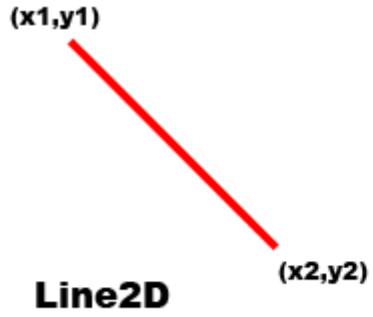


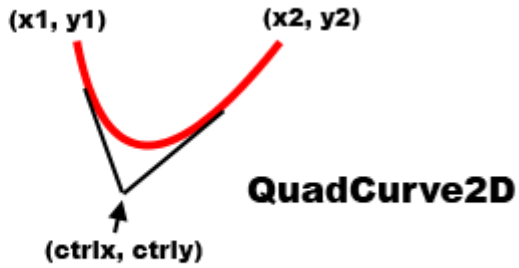
Formes (Shape)

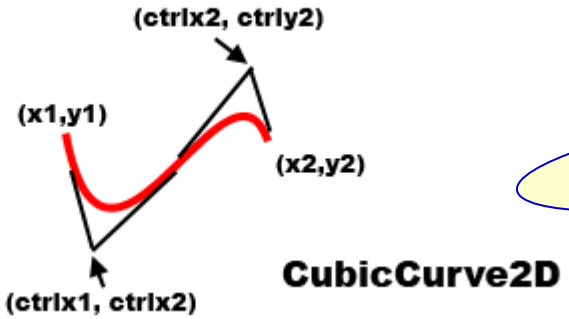


Points et Lignes

- Les **points** [x, y] sont définis par des instances de la classe **`Point2D`** (**`Point2D.Float`** ou **`Point2D.Double`**).
- Plusieurs types de **lignes** sont disponibles :



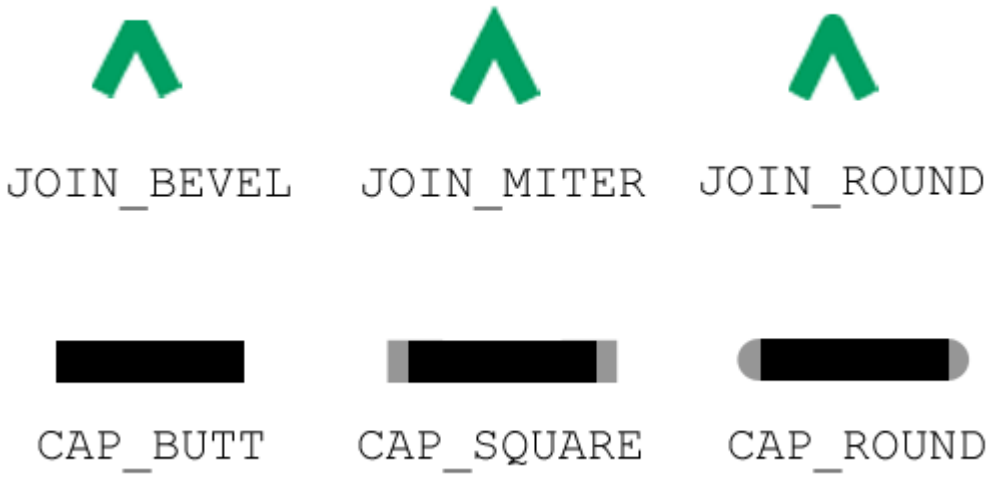




Courbes de Bézier

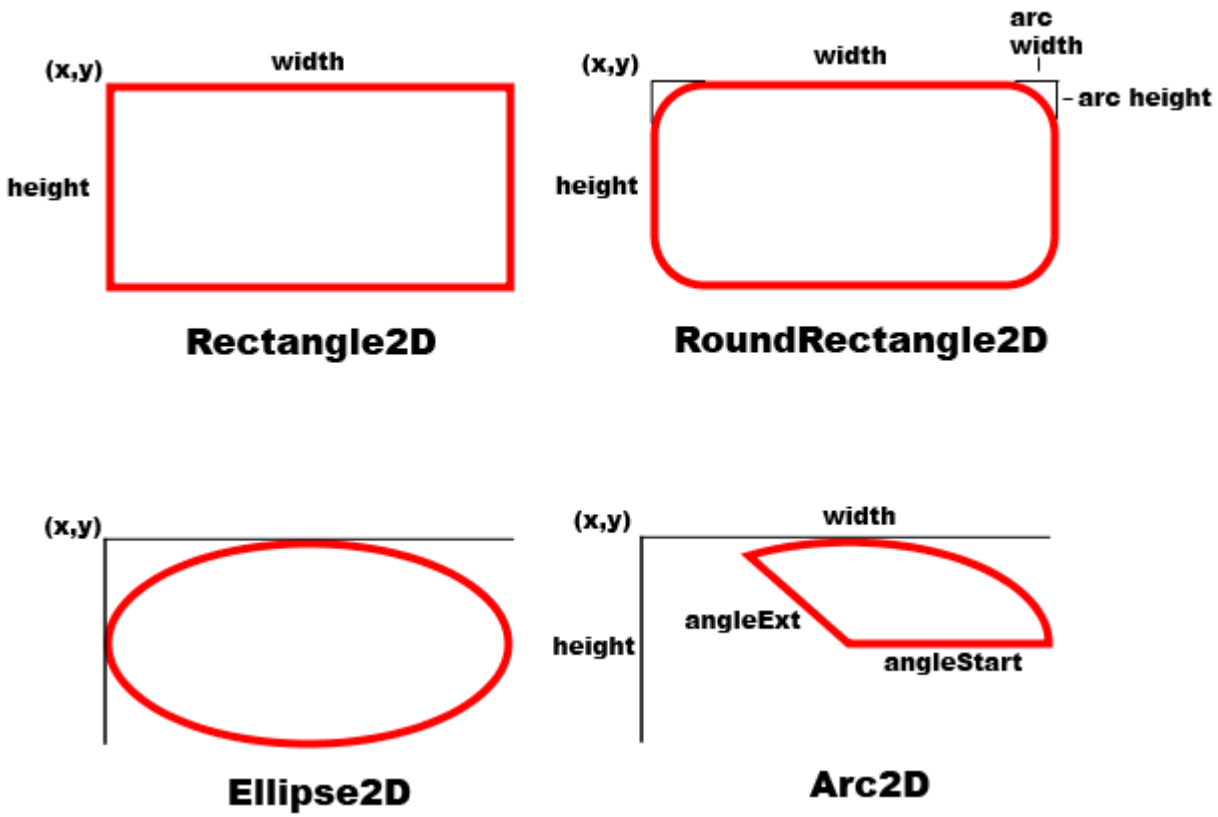
Jointure et terminaison des lignes

- Le type de jointure et de terminaison des lignes peut être défini lors de la création d'un objet de type **`Stroke`**.
- On utilise généralement le constructeur de la classe **`BasicStroke`** (qui implémente l'interface **`Stroke`**) pour définir un type de ligne.



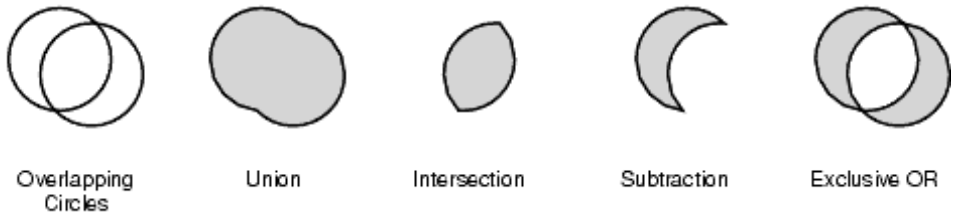
Formes de base

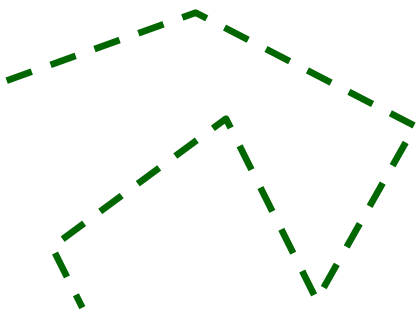
- Les différentes **formes de base** que l'on trouve dans le package `awt.java.geom` sont paramétrables.

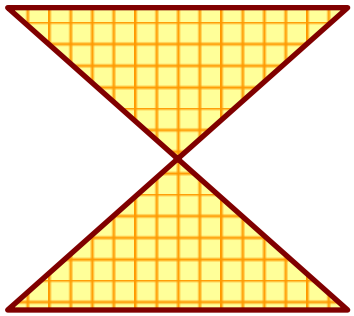


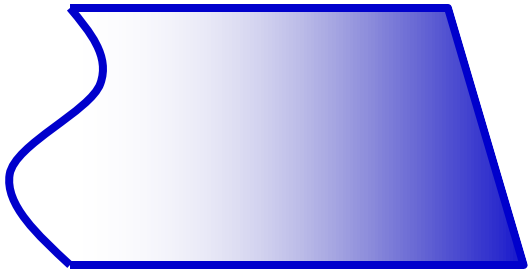
Formes quelconques

- Des **formes quelconques** peuvent être créées
  - En utilisant la classe **`GeneralPath`** et les méthodes **`moveTo()`**, **`lineTo()`**, **`quadTo()`**, **`curveTo()`**, **`closePath()`**)
  - Ou en combinant plusieurs formes différentes (**`Area`**) à l'aide d'opérateurs géométriques : *union*, *intersection* *soustraction*, *XOR*



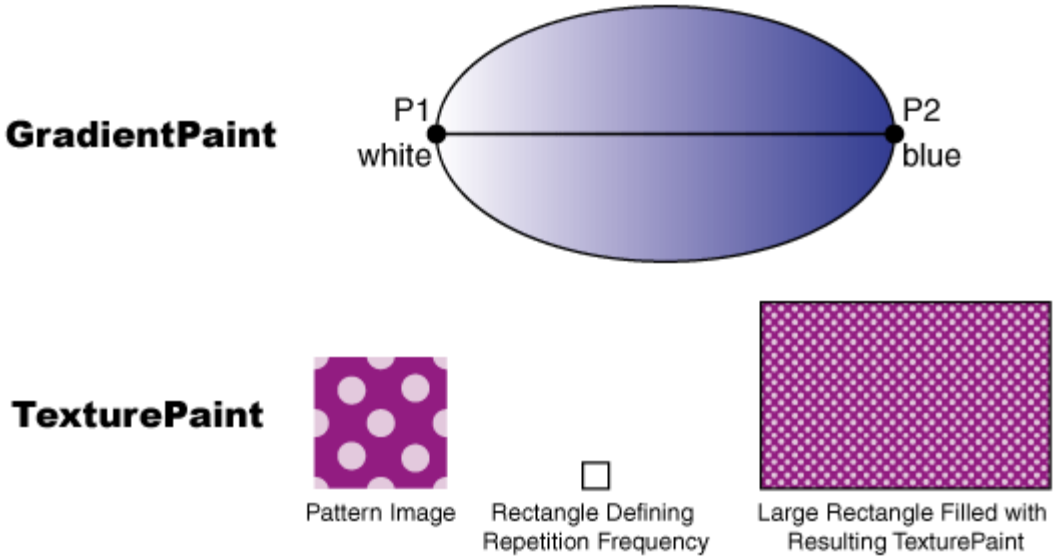






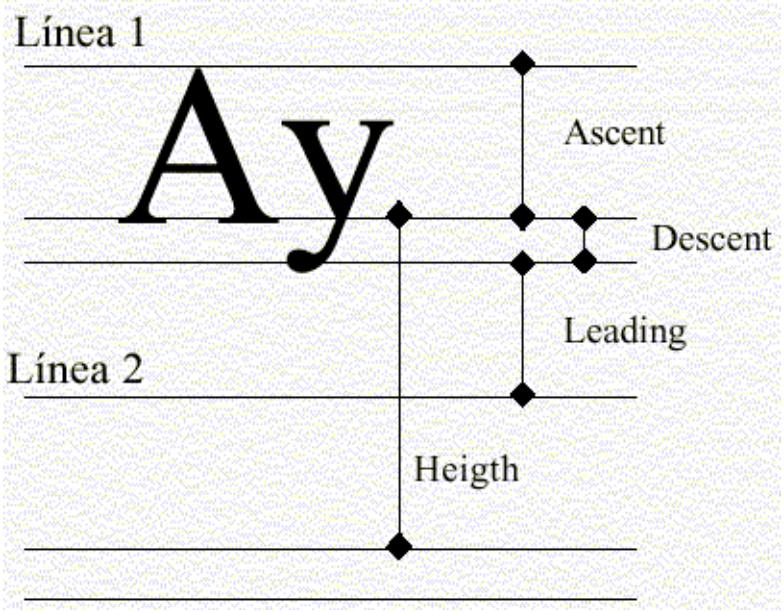
Dégradés de couleur / Motifs

- Un objet de type **`Paint`** peut représenter une couleur simple, un dégradé de couleur ou un motif répétitif (texture).
- Les classes **`GradientPaint`** et **`TexturePaint`** sont utilisées pour définir des remplissages avec des dégradés de couleur et des motifs.



Textes [1]

- On peut placer des textes dans un graphique en utilisant la méthode **`drawString()`** qui dessine les caractères (*glyphes*) selon la police courante du contexte graphique.
- Les dimensions de l'image d'un texte peuvent être obtenues en utilisant la classe **`FontMetrics`** ou la classe **`LineMetrics`** (*font*`.getLineMetrics()`)



- Exemple :

```
Graphics2D g2d = ...;
g2d.setFont(...);
FontMetrics fontMetrics = g2d.getFontMetrics();
int longueur = fontMetrics.stringWidth("Un texte");
int hauteur  = fontMetrics.getHeight();
```

Textes [2]

- On peut également manipuler du texte en utilisant la classe **`TextLayout`** (package `java.awt.font`) qui enregistre la représentation graphique d'un texte avec ses différents attributs (police, style, ...).
- Cette classe utilise un objet de type **`FontRenderContext`** qui gère les métriques du texte (dimensions) dans le contexte graphique donné (contrairement à **`FontMetrics`**, les dimensions peuvent être des nombres fractionnaires [type **`double`**]).
- Les dimensions d'un texte (le rectangle circonscrit) peuvent être consultées en invoquant la méthode **`getBounds()`** sur l'objet de type **`TextLayout`**.
- Pour l'affichage du texte, on utilisera la méthode **`draw()`** de l'objet **`TextLayout`**.

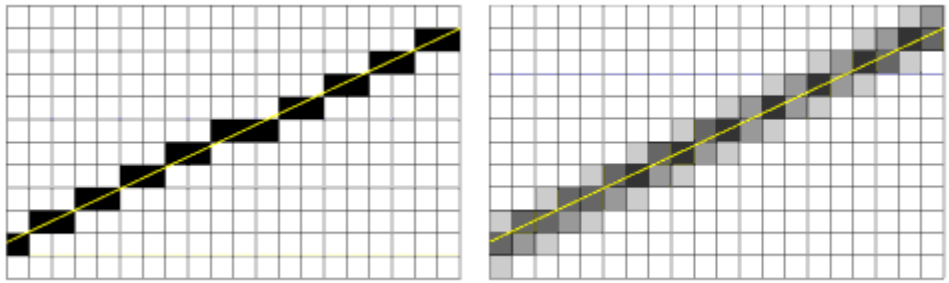
Textes [3]

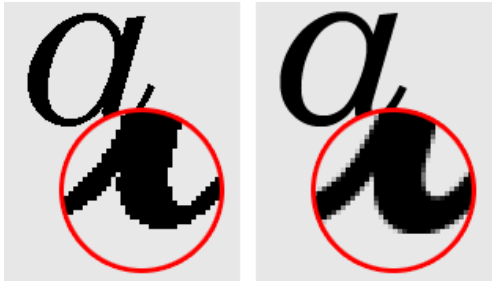
- Exemple avec **TextLayout** :

```
. . .
Graphics2D g2d = ...;
FontRenderContext frc = g2d.getFontRenderContext();
Font fnt = new Font("SanSerif", Font.PLAIN, 16);
TextLayout tLayout = new TextLayout("Un texte", fnt, frc);

Rectangle2D bounds = tLayout.getBounds();
double tWidth  = bounds.getWidth();
double tHeight = bounds.getHeight();

float posX = ...;
float posY = ...;
tLayout.draw(g2d, posX, posY);
. . .
```

RenderingHints [1]

- La qualité du rendu des dessins peut être améliorée en invoquant la méthode **`setRenderingHint()`** du contexte graphique qui permet de paramétrer différentes options comme l'*antialiasing*, le rendu des couleurs, le *dithering, ...*
- L'**antialiasing** (ou anti-crénelage) a pour but d'atténuer les effets crénelés d'une image ou d'un texte grâce au lissage des pixels en bordure des zones contrastées (par interpolation des couleurs). On crée ainsi des zones de transition plus douces qui améliorent le rendu visuel du graphique.





RenderingHints [2]

- Différentes constantes de la classe **RenderingHints** (des paires clés/valeurs de propriétés) sont à utiliser pour configurer la qualité du rendu.

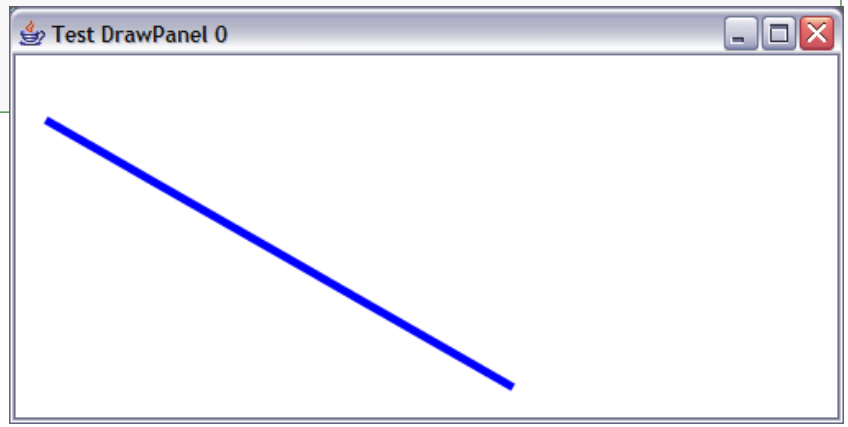
```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                       RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

  . . .
  . . .
}
```

Attention : Lorsque l'on active l'*antialiasing*, l'objet dessiné dépasse de quelques pixels sa taille théorique (en raison du lissage de la bordure). Il faut en tenir compte lorsque l'on souhaite effacer sélectivement l'objet en redessinant la couleur de fond.

Autrement dit, la gomme doit être légèrement plus grande que l'objet à effacer !

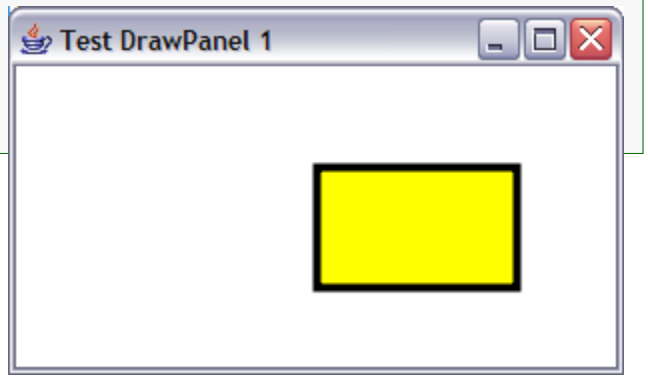
Exemple 0

```
public class DrawPanel0 extends JPanel {

  public DrawPanel0() {
    setPreferredSize(new Dimension(500, 220));
    setBackground(Color.WHITE);
  }

  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;

    //--- Ligne ----------------------------------------------------
    Line2D.Float line = new Line2D.Float(20, 40, 300, 200);
    g2d.setPaint(Color.BLUE);
    g2d.setStroke(new BasicStroke(5));
    g2d.draw(line);
  }
}
```



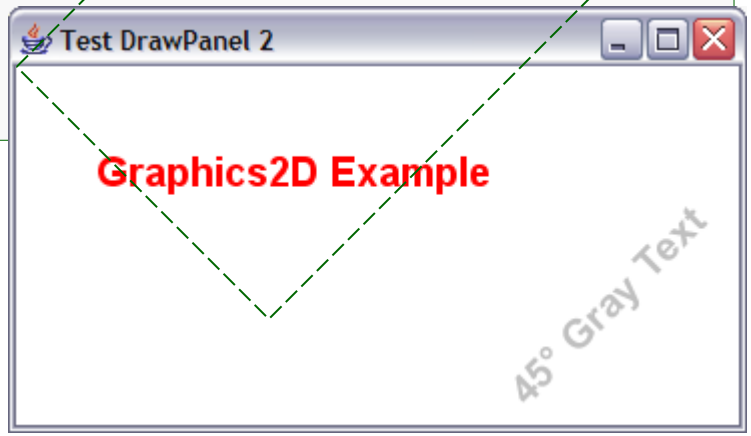
Exemple 1

```
public class DrawPanel1 extends JPanel {

  public DrawPanel1() {
    setPreferredSize(new Dimension(300, 150));
    setBackground(Color.WHITE);
  }

  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;

    //--- Rectangle -------------------------------------------------
    Rectangle2D.Float rect = new Rectangle2D.Float(150, 50, 100, 60);
    g2d.setPaint(Color.YELLOW);
    g2d.fill(rect);
    g2d.setPaint(Color.BLACK);
    g2d.setStroke(new BasicStroke(4));
    g2d.draw(rect);
  }
}
```



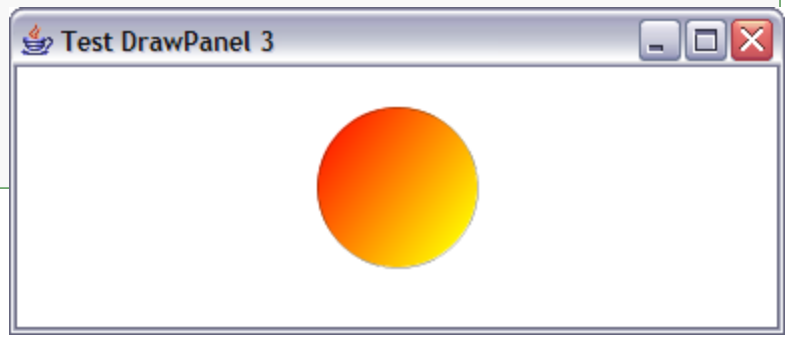
Exemple 2

```
public class DrawPanel2 extends JPanel {
  public DrawPanel2() {
    setPreferredSize(new Dimension(400, 200));
    setBackground(Color.WHITE);
  }
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;
    //--- Texte ----------------------------------------------------
    g2d.setFont(new Font("SanSerif", Font.BOLD, 20));
    g2d.setPaint(Color.RED);
    g2d.drawString("Graphics2D Example", 40, 60);
    g2d.rotate(-Math.PI/4);
    g2d.setPaint(Color.LIGHT_GRAY);
    g2d.drawString("45° Gray Text", 60, 300);
    g2d.rotate(Math.PI/4);
  }
}
```



Exemple 3

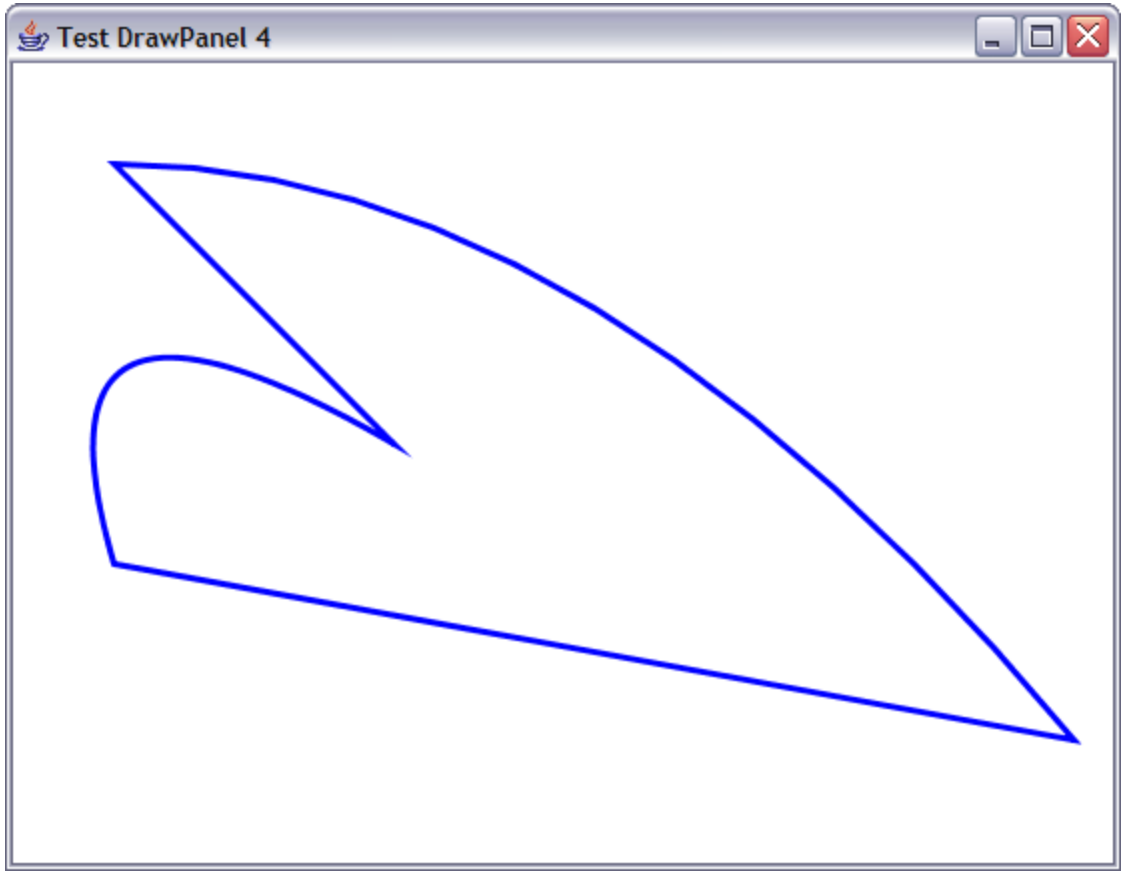
```
public class DrawPanel3 extends JPanel {

  public DrawPanel3() {
    setPreferredSize(new Dimension(380, 130));
    setBackground(Color.WHITE);
  }

  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D)g;

    //--- Cercle ------------------------------------------------
    Ellipse2D.Float cercle = new Ellipse2D.Float(150, 20, 80, 80);
    g2d.setStroke(new BasicStroke(1));
    g2d.setPaint(new GradientPaint(150, 20, Color.RED,
                                   230, 80, Color.YELLOW));
    g2d.fill(cercle);
    g2d.setStroke(new BasicStroke(0));
    g2d.setPaint(Color.BLACK);
    g2d.draw(cercle);
  }
}
```



Exemple 4

```
protected void paintComponent(Graphics g) {
   super.paintComponent(g);
   Graphics2D g2d = (Graphics2D)g;

   g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);

   //--- Forme quelconque (GeneralPath) ---------------------------
   GeneralPath path = new GeneralPath();

   final int offset = 50;
   path.moveTo(offset, offset);     // Point de départ

   for (int i=0; i<=120; i=i+10) {
     path.lineTo(offset+(4*i), offset+(i*i)/50);
   }

   path.lineTo(offset, offset+200);
   path.quadTo(0, 80, offset+140, offset+140);
   path.closePath();                // Ferme la forme avec une ligne

   g2d.setPaint(Color.BLUE);
   g2d.setStroke(new BasicStroke(3));
   g2d.draw(path);
  }
}
```

Exemple 4 / Résultat



Double-buffering [1]

- Pour utiliser la technique du ***Double-buffering*** on crée une image interne de type **`BufferedImage`** dans laquelle on dessine.
- Le principe est le suivant :

```
// Création d'une image interne de type BufferedImage à partir du
// conteneur dans lequel on souhaite dessiner (panel)
BufferedImage buffer = (BufferedImage)panel.createImage(panel.getWidth(),
                                                         panel.getHeight());

// Création d'un contexte graphique de type Graphics2D qui sera utilisé
// pour dessiner dans l'image interne
Graphics2D bg2d = buffer.createGraphics();

// Dessin dans image interne en utilisant le contexte graphique de
// l'image interne (bg2d)
Rectangle2D.Float rect = new Rectangle2D.Float(150, 50, 100, 60);
bg2d.setPaint(Color.YELLOW);
bg2d.fill(rect);
. . .
```

Double-buffering [2]

- Finalement, on copie l'image interne dans le contexte graphique de l'écran avec la méthode **`drawImage()`**.

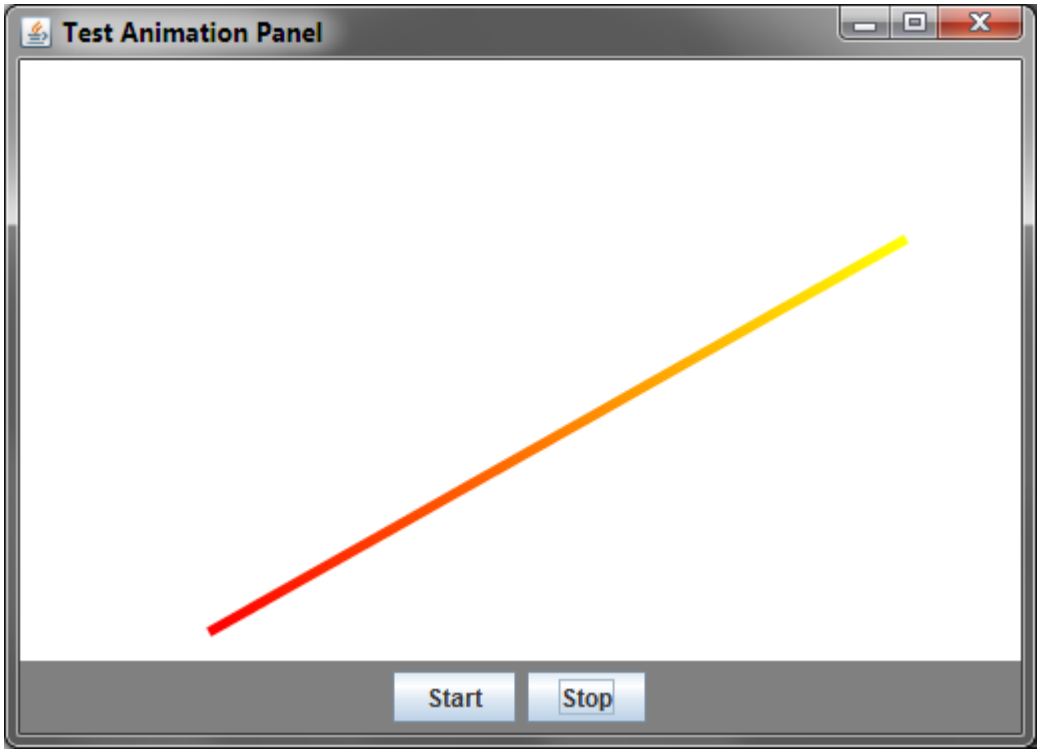
```
// Copie de l'image interne (buffer) dans le contexte graphique du
// conteneur JPanel affiché à l'écran
loaded = screenPanelContext.drawImage(buffer, 0, 0, iObserver);
```

Indicateur si entièrement copié	Contexte graphique du conteneur (JPanel)	Image interne (BufferedImage)	Coordonnées x, y où sera copiée l'image	Par ex. conteneur de type JPanel

- Si l'on souhaite que l'image interne soit transparente, on créera une image de type *AlphaRGB* (avec canal *alpha*).

```
BufferedImage buffer = new BufferedImage(panel.getWidth(),
                                         panel.getHeight(),
                                         BufferedImage.TYPE_INT_ARGB);
```

Exemple d'animation [1]



Important : Un *thread* indépendant doit être créé pour mettre à jour l'état du graphique (modèle).

Exemple d'animation [2]

- Programme principal :

```
public class TestAnimPanel {

  public static void main(String args[]) {
    JFrame frame = new AnimPanelFrame();
    frame.setVisible(true);
  }
}
```

Exemple d'animation [3]

```
public class AnimPanelFrame extends JFrame {

  private AnimModel   animModel = new AnimModel(20, 20, 40, 50);
  private AnimPanel   animPanel = new AnimPanel(animModel);
  private JButton     btStart   = new JButton("Start");
  private JButton     btStop    = new JButton("Stop");
  private JPanel      btnPanel  = new JPanel();

  //-------------------------------------------------------------------
  public AnimPanelFrame() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("Test Animation Panel");

    btStart.setName("START");
    btStop.setName("STOP");

    btStart.addActionListener(new AnimPanelButtonCtrl(animPanel));
    btStop.addActionListener(new AnimPanelButtonCtrl(animPanel));

    btnPanel.setBackground(Color.GRAY);
    btnPanel.add(btStart);
    btnPanel.add(btStop);
```

Exemple d'animation [4]

Suite AnimPanelFrame

```
    getContentPane().add(animPanel, BorderLayout.CENTER);
    getContentPane().add(btnPanel,  BorderLayout.SOUTH);

    pack();
    setLocationRelativeTo(getParent());  // Center window
  }
}
```

Exemple d'animation [5]

```
public class AnimModel {

  private Point2D.Float   p1;          // First  line extremity
  private Point2D.Float   p2;          // Second line extremity

  //-------------------------------------------------------------------------
  public AnimModel(float initialX1, float initialY1,
                   float initialX2, float initialY2) {

    p1 = new Point2D.Float(initialX1, initialY1);
    p2 = new Point2D.Float(initialX2, initialY2);
  }

  //-------------------------------------------------------------------------
  public synchronized Point2D getPoint1() {
    return (Point2D)(p1.clone());
  }

  //-------------------------------------------------------------------------
  public synchronized Point2D getPoint2() {
    return (Point2D)(p2.clone());
  }
```

Exemple d'animation [6]

Suite AnimModel

```
  //-------------------------------------------------------------------------
  public synchronized void newPositionPoint1(float newX, float newY) {
    p1.x = newX;   p1.y = newY;
  }

  //-------------------------------------------------------------------------
  public synchronized void newPositionPoint2(float newX, float newY) {
    p2.x = newX;   p2.y = newY;
  }
}
```

Exemple d'animation [7]

```
public class AnimPanel extends JPanel {

  private static float SPEED_X1=3, SPEED_Y1=2, SPEED_X2=4, SPEED_Y2=2;

  private AnimModel animModel;
  private boolean   animRunning = false;

  //-------------------------------------------------------------------------
  public AnimPanel(AnimModel animModel) {
    this.animModel = animModel;
    setPreferredSize(new Dimension(500, 300));
    setBackground(Color.WHITE);
  }
```

Exemple d'animation [8]

Suite AnimPanel

```
//-----------------------------------------------------------------------
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;

  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                       RenderingHints.VALUE_ANTIALIAS_ON);

  Line2D.Float line = new Line2D.Float(animModel.getPoint1(),
                                       animModel.getPoint2());
  g2d.setPaint(new GradientPaint(animModel.getPoint1(), Color.RED,
                                 animModel.getPoint2(), Color.YELLOW));
  g2d.setStroke(new BasicStroke(5));
  g2d.draw(line);
}
```

Exemple d'animation [9]

Suite AnimPanel

```
  //-----------------------------------------------------------------------
  protected boolean isAnimRunning() {
    return animRunning;
  }

  //-----------------------------------------------------------------------
  protected void setAnimRunning(boolean running) {
    animRunning = running;
  }

  //-----------------------------------------------------------------------
  public void startAnimThread() {
    if (isAnimRunning()) return;  // Don't start a second thread
    setAnimRunning(true);
    new Thread(new AnimThread(animModel, this, SPEED_X1, SPEED_X2,
                                               SPEED_X2, SPEED_Y2)).start();
  }
}
```

Exemple d'animation [10]

```
public class AnimThread implements Runnable {

  private AnimModel  animModel;
  private AnimPanel  animPanel;
  private float      dx1, dy1, dx2, dy2;

  //-------------------------------------------------------------------------
  public AnimThread(AnimModel animModel, AnimPanel animPanel,
                    float     speedX1,   float     speedY1,
                    float     speedX2,   float     speedY2    ) {
    this.animModel = animModel;
    this.animPanel = animPanel;
    this.dx1       = speedX1;     this.dy1 = speedY1;
    this.dx2       = speedX2;     this.dy2 = speedY2;
  }
```

Exemple d'animation [11]

Suite AnimThread

```
//-----------------------------------------------------------------------
public void run() {
  while (animPanel.isAnimRunning()) {
    float newX = incrX1Pos((float)(animModel.getPoint1().getX()),
                                    animPanel.getSize().width);
    float newY = incrY1Pos((float)(animModel.getPoint1().getY()),
                                    animPanel.getSize().height);
    animModel.newPositionPoint1(newX, newY);

    newX = incrX2Pos((float)(animModel.getPoint2().getX()),
                               animPanel.getSize().width);
    newY = incrY2Pos((float)(animModel.getPoint2().getY()),
                               animPanel.getSize().height);
    animModel.newPositionPoint2(newX, newY);

    animPanel.repaint();
    try {
      Thread.sleep(20);
    }
    catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple d'animation [12]

Suite AnimThread

```
//-----------------------------------------------------------------------
private float incrX1Pos(float startPos, float limit) {
  float newPos = (float)(animModel.getPoint1().getX()) + dx1;
  if (newPos<0) {
    dx1 = -dx1;   return 0;
  }
  if (newPos>limit) {
    dx1 = -dx1;   return limit;
  }
  return newPos;
}

//-----------------------------------------------------------------------
private float incrY1Pos(float startPos, float limit) {
  float newPos = (float)(animModel.getPoint1().getY()) + dy1;
  if (newPos<0) {
    dy1 = -dy1;   return 0;
  }
  if (newPos>limit) {
    dy1 = -dy1;   return limit;
  }
  return newPos;
}
```

Exemple d'animation [13]

Suite AnimThread

```
  //-----------------------------------------------------------------------
  private float incrX2Pos(float startPos, float limit) {
    float newPos = (float)(animModel.getPoint2().getX()) + dx2;
    if (newPos<0) {
      dx2 = -dx2;   return 0;
    }
    if (newPos>limit) {
      dx2 = -dx2;   return limit;
    }
    return newPos;
  }

  //-----------------------------------------------------------------------
  private float incrY2Pos(float startPos, float limit) {
    float newPos = (float)(animModel.getPoint2().getY()) + dy2;
    if (newPos<0) {
      dy2 = -dy2;   return 0;
    }
    if (newPos>limit) {
      dy2 = -dy2;   return limit;
    }
    return newPos;
  }
}
```

Exemple d'animation [14]

```
public class AnimPanelButtonCtrl implements ActionListener {

  private AnimPanel animPanel;

  //-------------------------------------------------------------------------
  public AnimPanelButtonCtrl(AnimPanel animPanel) {
    this.animPanel = animPanel;
  }

  //-------------------------------------------------------------------------
  public void actionPerformed(ActionEvent event) {
    if (((JButton)(event.getSource())).getName().equals("STOP")) {
      animPanel.setAnimRunning(false);
    }
    else if (((JButton)(event.getSource())).getName().equals("START")) {
      animPanel.startAnimThread();
    }
  }
}
```

Animation

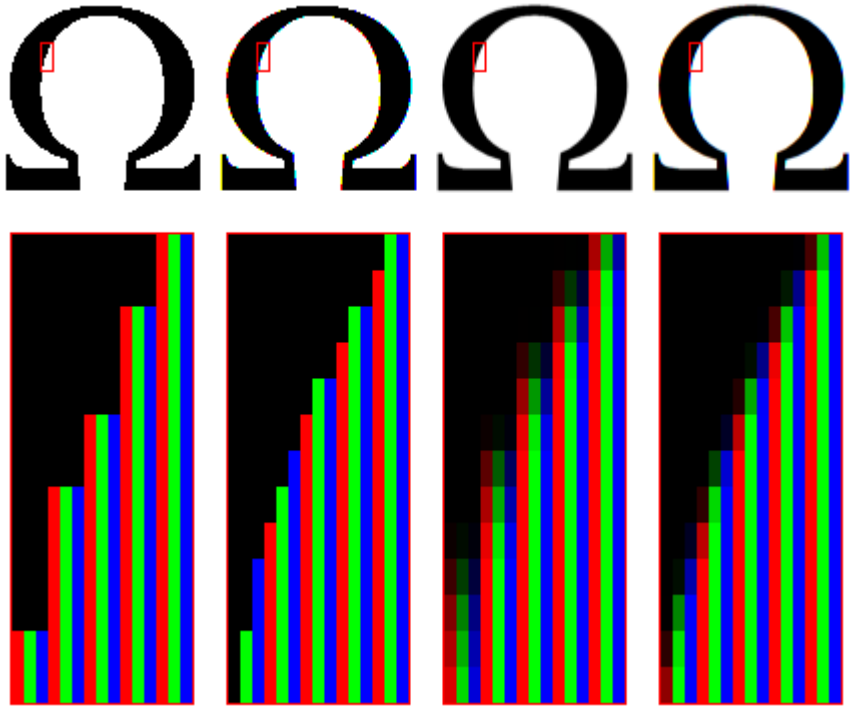
- L'exemple précédent illustre le principe de la création d'une application comportant une animation graphique.
- Dans une application réelle, les données représentant l'état du graphique pourraient provenir d'un système d'information (température d'une cuve, cours de la bourse, etc.).
- Si l'animation présente une certaine complexité graphique (ce qui n'est pas le cas ici), il est préférable d'utiliser la technique du *double-buffering* pour fluidifier l'animation et éviter un scintillement désagréable.
- Il faut se souvenir que la méthode `paintComponent()` peut être invoquée à n'importe quel moment par le système (et pas seulement lors de l'appel à `repaint()`). Si l'image interne est créée dans un *thread* indépendant, il faut s'assurer, lors de la copie à l'écran, qu'elle se trouve dans un état cohérent.

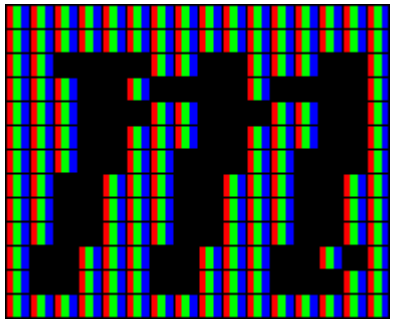
Graphiques 2D

Annexes

Subpixel Rendering [1]

- Le **subpixel rendering** exploite l'affichage sélectif des trois sous-pixels *rouge, vert, bleu* pour affiner le tracé.
- Cette technique est applicable principalement avec les écrans LCD et peut être combinée avec l'antialiasing.





Subpixel Rendering [2]

- Depuis la version 1.6 du SDK, la librairie *Swing* exploite le **subpixel rendering** du système d'exploitation pour afficher les textes.