

Guide du graphiste, Graphics2D¹

Lors de l'écriture de composants, nous sommes parfois amenés à vouloir faire des graphiques (dessins) dans le composant. Pour ce faire, considérons les deux classes *Graphics* et *Graphics2D* de l'API Java².

1 LE CONTEXTE GRAPHIQUE

Chaque composant³ permet d'accéder à un objet de type *Graphics2D* qu'il contient. C'est cet objet qui permet de manipuler le graphisme au sein d'un composant.

Cet objet manipule les propriétés suivantes:

- x le composant dans lequel dessiner,
- x un système de coordonnées et une transformation affine éventuelle (*affin transformation*),
- x la zone de découpe courante (*clip*),
- x le trait (*stroke*),
- x la couleur courante (*color*),
- x la police de caractère courante ainsi que ses attributs (*font*),
- x l'opération logique courante pour les pixels; dessiner en effaçant le fond (*paint*) ou faire un ou logique exclusif (*xor*),
- x ...⁴

Le contexte graphique s'obtient de manière implicite⁵ via la réécriture de la méthode `paintComponent(Graphics)` dans un cadre SWING et la méthode `paint` dans un contexte AWT. Notons que la méthode `paint` existe encore avec SWING mais qu'elle ne doit pas être redéfinie.

Pour que tout se passe bien et pour pouvoir profiter pleinement des fonctionnalités graphiques de Java, il faut faire un appel à la méthode `paintComponent` de la classe parent et convertir l'objet *Graphics* reçu en un objet *Graphics2D* puisque dans un contexte Swing, s'en est toujours un.

Nous pouvons donc écrire,

```
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    ...
}
```

¹ Ce document a été rédigé par *pbt* en octobre 2008.

² Les images et cette note est basée sur la Javadoc et sur l'article [Learning Java 2D \(sun\)](#)

³ À partir de la classe *Container*

⁴ Éventuellement d'autres propriétés que je n'ai pas investiguées

⁵ Il est possible de l'obtenir explicitement, mais nous n'en parlerons pas ici.

La méthode `paintComponent` est appelée chaque fois que le composant nécessite d'être redessiné. Pour forcer la mise à jour du composant, il suffit de faire appel à sa méthode `repaint`. Notez que l'appel à la méthode `repaint` fait appel à la méthode `update` du composant qui fera appel –au moment opportun– à la méthode `paintComponent` du composant.

2 FORMES GÉOMÉTRIQUES, MÉTHODES DE DESSIN

Un contexte graphique définit des méthodes permettant de dessiner des formes géométriques. Il est possible de dessiner des formes vides (**draw**) ou des formes pleines (**fill**). Ces deux méthodes `draw(Shape)` et `fill(Shape)` prennent en paramètres des formes géométriques. Ces formes géométriques sont celles de base telles que *ligne*, *ellipse*, *rectangle*, ... (voir figure). On peut également dessiner un **point** via les classes `Point2D` (`Point2D.Float` et `Point2D.Double`).

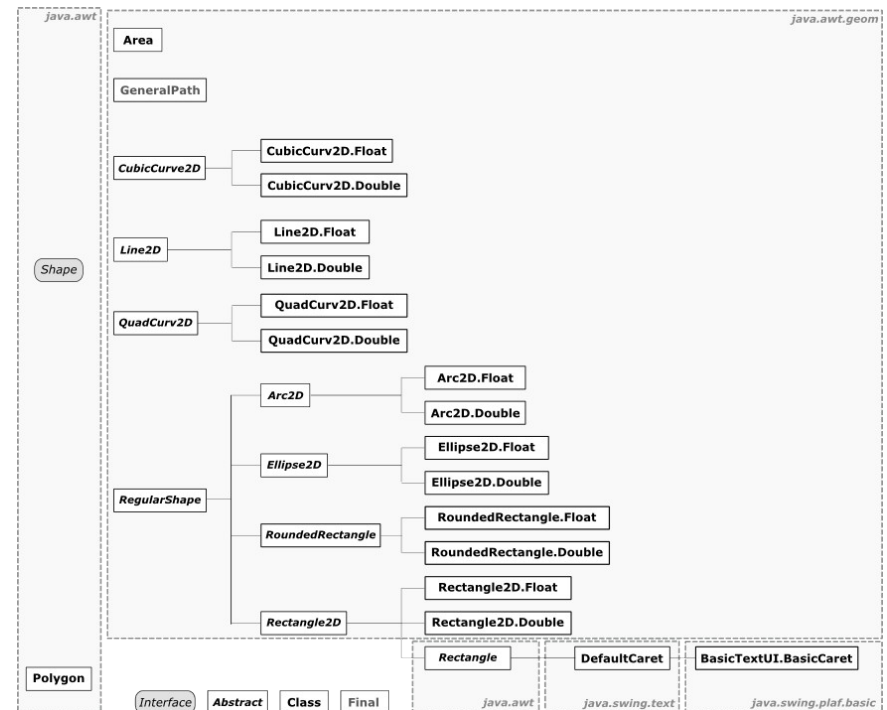


Figure 1 - Graphics shape hierarchy

Il existe également des méthodes `draw<Shape>` permettant de dessiner “directement” une forme, par exemple `drawLine()`. Pour plus d'info, consultez la documentation de la classe *Graphics*.

Dessiner une croix en plein milieu d'un panel se fait donc comme suit:

```
Shape line1 = new Line2D.Double(0,0,getWidth(),getHeight());
Shape line2 = new Line2D.Double(0,getHeight(),getWidth(),0);
g2.draw(line1);
g2.draw(line2);
```

ou bien sans utiliser de “Shape” mais directement les méthodes de dessin de formes:

```
g2.drawLine(0,0,getWidth(),getHeight());
g2.drawLine(0,getHeight(),getWidth(),0);
```

Nous laissons le lecteur intéressé se renseigner sur l'utilisation des différentes formes.

3 LE TEXTE, FONT

Écrire du texte est aussi simple que d'y dessiner une forme, il suffit d'utiliser la méthode `drawString`, comme suit:

```
g2.drawString("Graphics2D", 0, getHeight());
```

Dans ce cas, le texte sera écrit avec la police (*font*) et la couleur par défaut. Il est possible d'en changer.

Pour définir une police, utilisez le constructeur de `Font(String name, int style, int size)`,

- x le nom de la police peut être une famille de police (*family name*) tel que **Dialog**, **DialogInput**, **Monospaced**, **Serif**, ou **SansSerif** ou un nom de police,
- x le style peut être –parmi les constantes de la classe `Font`— `PLAIN`, `BOLD`, `ITALIC` ou un ou bit à bit de `BOLD` et d'`ITALIC`,
- x la taille est exprimée en points

```
Font font = new Font("SansSerif", Font.BOLD, 14);
```

4 LE TRAIT, STROKE

4.1 La couleur du trait

Vous pouvez préciser la couleur du trait simplement par la méthode `setColor`, un peu comme suit,

```
g2.setColor(Color.MAGENTA);
```

4.2 La forme du trait

Vous pouvez préciser la forme du trait (*stroke*) qui est utilisé pour chaque méthode `draw` et `drawShape`, en assignant un nouveau trait au graphique. Pour ce faire vous définirez un trait via la classe `BasicStroke` implémentant l'interface `Stroke`.

Un trait (*basic stroke*) a les attributs suivants (incomplet) :

- x une épaisseur (*width*),
- x une « forme de bouts » (*end caps*) pour les chemins non fermés, cette forme peut être; `CAP_BUTT`, `CAP_ROUND` ou `CAP_SQUARE`,

- x une forme de jointure de lignes (*line joins*) pour les intersections de chemins et les segments non fermés, cette jointure peut être; `JOIN_BEVEL`, `JOIN_MITER` ou `JOIN_ROUND`.

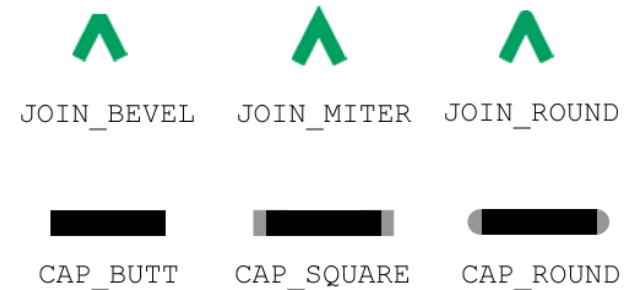


Figure 2 : Type de bouts et de jointures (Source <http://java.sun.com>)

On pourrait donc définir un trait comme suit (après avoir éventuellement sauvegardé le trait par défaut en utilisant le *getter* associé),

```
Stroke stroke = new BasicStroke(
    15f,
    BasicStroke.CAP_ROUND,
    BasicStroke.JOIN_MITER);
g2.setStroke(stroke);
```

5 LA COULEUR DE REMPLISSAGE, PAINT

Vous pouvez définir comment vous allez « remplir » vos formes via l'interface `Paint`. Il existe trois types de remplissages:

- x une couleur uniforme,
- x un dégradé de couleurs, ou
- x une texture

Pour remplir par une **couleur uniforme**, il suffit d'utiliser la classe `Color` qui implémente `Paint`, ainsi on écrira par exemple,

```
g2.setPaint(Color.BLUE);
RoundRectangle2D rectangle = new RoundRectangle2D.Double(
    getWidth()/4,
    getHeight()/4,
    getWidth()/2,
    getHeight()/2,
```

```
getWidth()/8,
getHeight()/8);
g2.fill(rectangle);
```

Pour remplir par un **dégradé de couleur**, on utilisera la classe **GradientPaint** qui permet de définir un point et une couleur de départ ainsi qu'un point et une couleur d'arrivée pour le dégradé. On peut lui ajouter un booléen précisant si le dégradé se répète ou pas. Par exemple,

```
g2.setPaint(new GradientPaint(0, getHeight()/2, Color.RED,
getWidth(), getHeight()/2, Color.CYAN));
```

Pour remplir avec une **texture**, on utilisera la classe **TexturePaint** qui permet sur base d'une **BufferedImage** et d'un **Rectangle2D** de mapper l'un sur l'autre et de remplir la forme. D'après la documentation de Sun un code à l'allure suivante devrait faire l'affaire⁶,

```
URL url = Images.class.getResource("image.png");
Image img = getToolkit().createImage(url);
int iw = img.getWidth(this);
int ih = img.getHeight(this);
BufferedImage bi = new BufferedImage(
iw, ih, BufferedImage.TYPE_INT_RGB);
Graphics2D tG2 = bi.createGraphics();
tG2.drawImage(img, 0, 0, this);
Rectangle r = new Rectangle(0,0,iw/2,ih/2);
g2.setPaint(new TexturePaint(bi,r));
```

6 INSERTION D'IMAGE

Il est également possible de dessiner une image en utilisant la méthode `drawImage()` qui nécessite d'avoir créé une **Image** sur base d'un fichier *png*, *jpg* ou autre.

Pour créer une image, c'est simple vous pouvez vous contenter d'une instruction de la forme

```
Image image = getToolkit().createImage("maBelleImage.png");
```

Dans ce cas, vous verrez bien votre image lorsque vous travaillerez dans votre environnement de développement, vous la verrez bien aussi si **vous la placez au bon endroit au bon moment** mais elle ne sera pas intégrée à votre fichier *jar*.

Pour ce faire, créez un package dans votre projet intitulé *images* (par exemple) et ajoutez-y une classe vide intitulée *Images* (par exemple).

Ajoutez vos images dans le répertoire du package (probablement *src/images*) et créez vos images avec des instructions de la forme:

```
private final URL URL_IMG =
Images.class.getResource("maBelleImage.png");
private final Image IMG =
getToolkit().createImage(URL_IMG);
```

⁶ todo Ce code a l'image qui clignote, je dois encore régler ce problème.

Le fait de préciser que le fichier image est une **ressource** de la classe l'inclura dans votre fichier *jar*.

7 DÉFINITION DU CONTEXTE GRAPHIQUE, RENDERING

Il est possible d'indiquer des valeurs concernant le rendu des graphismes affichés. Ces indications sont renseignées sous la forme de paires *clé/valeur*. Ces clés sont définies dans la classe **RenderingHints**.

Par exemple on pourra préciser pour la clé **KEY_RENDERING** une des valeurs **VALUE_RENDER_DEFAULT**, **VALUE_RENDER_QUALITY** ou **VALUE_RENDER_SPEED** qui sera positionnée grâce à la méthode `setRenderingHint ...` et le code aura l'allure suivante,

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

8 SYSTÈME DE COORDONNÉES, TRANSFORMATION

Le contexte graphique contient une transformation affine permettant de passer du système de coordonnées de l'utilisateur (l'image à afficher par exemple) au système de coordonnées du graphique. Par défaut, cette transformation affine est l'identité.

Il est possible de modifier cette transformation pour centrer l'origine du repère ou choisir une autre transformation; translation (*translation*), dilatation (*dilatation*), rotation (*rotation*), cisaillement (*shear*), ...

On pourra par exemple demander au contexte graphique d'appliquer une transformation en utilisant l'instruction suivante où *af* est une transformation affine.

```
g2.transform(af);
```

Cette transformation peut-être définie sur base d'une matrice 3x3 ou bien, pour les transformations habituelles, demandée aux méthodes de classe de la classe **AffineTransform**. Pour une rotation d'un angle de *theta* **radians** autour du centre (du composant), on peut écrire:

```
AffineTransform af = AffineTransform.getRotateInstance(
theta,
this.getWidth()/2,
this.getHeight()/2);
```

9 REMARQUES

Pour être complet, il faudrait aborder les notions de **composition** (*compose*) et de **découpage** (*clipping*)... mais c'est suffisant pour une première approche.

10 BORD D'UN COMPOSANT, BORDER

Avertissement:

Ce qui suit n'est en rien particulier aux graphiques 2D (Graphics2D)

Chaque composant Swing (JComponent) peut être affublé d'un bord⁷. Bien que n'étant pas des composants, les bords peuvent dessiner une bordure à un composant. Pas seulement pour “faire beau” mais également pour imposer un espace vide ou donner un titre à un composant.

Il existe un mutateur permettant de modifier un bord et une “usine à bords⁸” (*border factory*) permettant de produire tout une série de bords habituels. Ce qui donne pour tracer une simple ligne autour d'un panel,

```
JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createLineBorder(Color.black));
```

ou pour tracer un bord sur base de deux autres bords,

```
setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createBevelBorder(BevelBorder.RAISED),
    BorderFactory.createBevelBorder(BevelBorder.LOWERED)));
```

Remarque: Lorsque Java dessine un bord, il utilise les valeurs du contexte graphique –la forme de son pinceau par exemple – il est donc bon que la méthode `paintComponent` lorsqu'elle modifie les attributs du contexte graphique les **restaure** en fin de dessin.

⁷ Ceci est extrait du tutorial [How to use Border \(sun\)](#) où vous trouverez des screenshots des différents bords.

⁸ Il faudrait trouver le terme adéquat en français.