

# Interfaces Graphiques

I see a red door and I want it painted black

No colors anymore I want them to turn black

Jagger/Richards

Jean-Baptiste.Yunes@univ-paris-diderot.fr

Université Paris Diderot

©2014

- Quand un composant AWT est-il dessiné ?
  - le système (*system triggered*) peut avoir besoin de forcer le dessin d'un composant (première apparition à l'écran, retaille, obscurcissement...)
  - l'application (*application triggered*) peut avoir besoin de forcer le dessin (clic pour dessiner l'effet d'appui, dessin personnalisé...)

- Comment un composant est-il dessiné ?
- chaque composant AWT possède une méthode  
`void paint(Graphics g);`
- dont le code est exécuté sous le contrôle du thread principal de l'interface après que l'on ait réclamé la mise à jour du composant (*system triggered*)
- et une méthode  
`void update(Graphics g);`
- lorsque c'est *application triggered* (permet le dessin incrémental)

- Les composants ne décident pas du moment où le dessin doit se faire, mais doivent être prêt à dessiner lorsque l'interface le décide (inversion de contrôle)
- si l'on souhaite forcer la réalisation du dessin d'un composant, il faut lui envoyer une requête :

`composant.repaint();`

- même dans ce cas il y a inversion de contrôle. Ceci a pour effet de demander (poliment) au thread de l'interface que l'on désire mettre à jour le composant
  - asynchronisme...

- Pour Swing, le dessin est obtenu par appel successif à trois méthodes (depuis `paint(Graphics)`)

`void paintComponent(Graphics g);`

- doit être redéfinie pour personnaliser le dessin

`void paintBorder(Graphics g);`

- très rarement redéfinie

`void paintChildren(Graphics g);`

- très rarement redéfinie

- en Swing, `update()` n'est pas utilisée...

- What's that Graphics object?
- Un objet Graphics est ce que l'on désigne couramment par contexte graphique et représente à la fois :
  - la *feuille* sur laquelle le dessin sera effectué (plus spécifiquement une *partie* de la *feuille*),
  - le *crayon* qui sera employé,
  - les *formes* de base qui peuvent être dessinées

- Les contextes graphiques ne sont jamais construits par le code utilisateur
- ils sont obtenus indirectement :
  - en paramètre à une méthode de dessin, ex :  
`paint`
  - par clonage d'un contexte existant
  - par obtention d'un contexte associé à un composant (s'il est visible!)

- Ainsi pour dessiner à l'écran, on peut :
  - récupérer le contexte graphique d'un composant visible
    - dessiner dedans
- cette technique possède un inconvénient majeur
  - le composant affecté ne sait rien des modifications graphiques apportées...
    - elles sont vite perdues...



- Pour dessiner on peut aussi :
  - sous-classer un composant existant (si possible un composant léger) et redéfinir la méthode de dessin adéquate

`Component : paint (Graphics )`

- `Canvas : paint (Graphics )`, plus habituel en AWT
- `JComponent : paintComponent (Graphics )`
- `JPanel : paintComponent (Graphics )`, plus habituel en Swing (car opaques par défaut)

- ces méthodes reçoivent à chaque appel un nouveau contexte graphique
- il ne faut pas tenter de retenir un contexte graphique d'un appel à l'autre...
- le contexte graphique reçu est positionné avec des valeurs d'attributs par défaut (épaisseur 1, etc)

- Java emploie désormais des `Graphics2D`
- les méthodes recevant des `Graphics` comme `paint(Graphics)` reçoivent maintenant des `Graphics2D`
- on peut donc utiliser l’idiome suivant :

```
public void paintComponent(Graphics g)
    Graphics2D g2d = (Graphics2D)g;
    . . .
```

$(0,0)$

- Pour désigner un pixel de l'espace rectangulaire, de largeur  $l$  et hauteur  $h$ , associé à un composant, la convention suivante est utilisée :
- le pixel en haut à gauche est  $(0,0)$
- en haut à droite est  $(l-1,0)$
- en bas à gauche  $(0,h-1)$
- en bas à droite  $(l-1,h-1)$

- Attention : l'espace associé à un composant peut varier car un composant peut-être retaillé...
- nécessité de consulter sa taille avant de dessiner quoi que ce soit...

`Dimension getSi e();`

- `Dimension : height, width`

- Qu'est ce qu'on peut faire avec des Graphics ?



`draw*`



`fill*`



`setComposite`



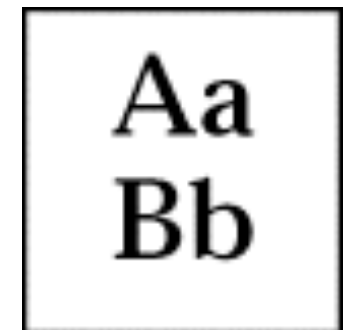
`shear`



`setClip`



`setRenderingHints`



`setFont`

- dans les exemples précédents on a pu observer l'emploi de différentes méthodes sur l'objet `Graphics` afin d'obtenir des dessins

```
setColor(Color);
```

```
drawLine(int x0,int y0,int x1,int y1);
```

```
fillRect(int x0,int y0,int l,int h);
```

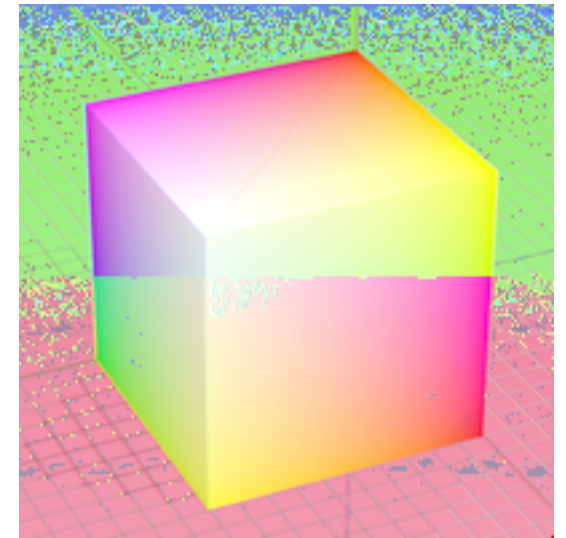
- Le trait est, par défaut, continu et d'épaisseur 1px
- La couleur est en général par défaut : noir

- La classe Couleur (immutable) représente les couleurs. Deux espaces de représentation :

- en RGB

```
Color(int r, int g, int b);
```

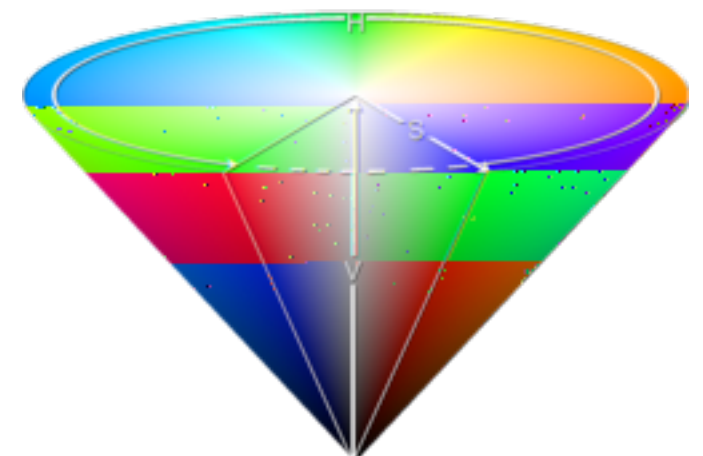
```
Color(float r, float g,  
float b);
```



extrait de Wikipédia

- en HSB (Hue, Saturation, Brightness aka Teinte, Saturation, Valeur)

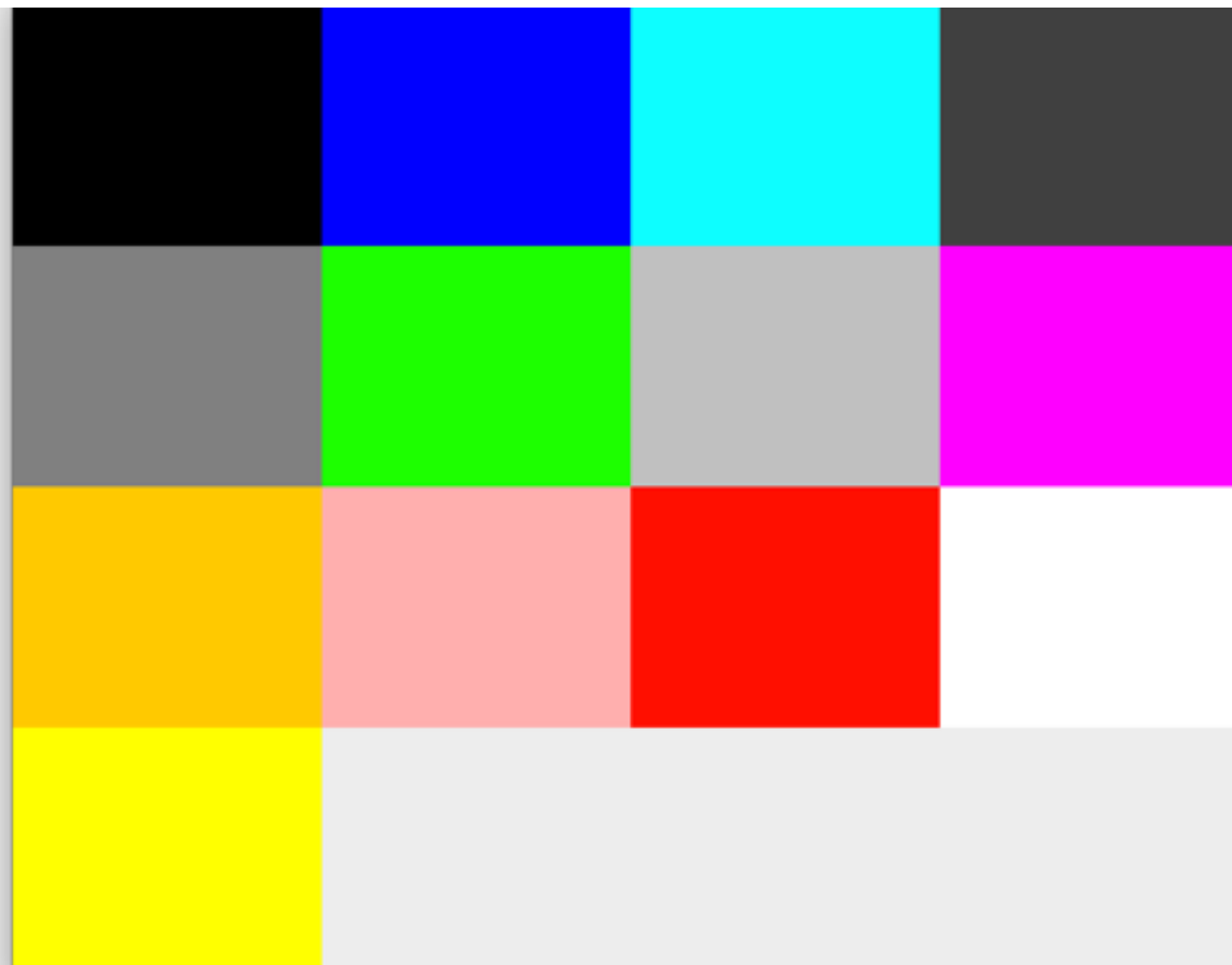
- *via* `Color.getHSB(float h,  
float s, float b);`



extrait de Wikipédia



- Utile : la classe `Color` propose diverses couleurs de base déjà construites :
  - `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`



- À savoir : les composants sont tous dotés de deux couleurs modifiables
  - `Color getBackground / setBackground(Color)`
  - `Color getForeground / setForeground(Color)`
- la plupart des composants utilisent
  - la couleur de fond comme couleur de remplissage
    - le `JLabel` est transparent...
  - la couleur d'avant-plan comme couleur par défaut des contextes graphiques pour le composant

- La modification de la couleur du contexte graphique  
`g.setColor(Color);`
- les `Graphics2D` utilisent aussi une couleur de fond (utilisé par `clearRect()`)  
`g.setBackground(Color);`
- valeur par défaut celle du composant

- Important :
  - il existe une quatrième composante
  - la transparence
  - RGBA ou HSVA

8								8								8								8							
Alpha								Red								Green								Blue							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

`Color(r,g,b,a)`

extrait de Wikipédia

`Color(int rgba,boolean hasAlpha)`

- Pour les effets de transparence il faut utiliser des compositions, *i.e.* mixer la couleur à afficher avec celle déjà existante sur le *papier*

`setComposite ( Composite )`

`AlphaComposite`

- modes de mélange standardisées : Porter-Duff  
Alpha Blending
- la description est assez complexe, consultez internet pour de nombreux exemples...

- Shape est une interface de `java.awt`
- il existe un package `java.awt.geom` contenant de nombreuses formes prédéfinies comme
  - arcs, rectangles, cubiques, quadratiques, polygones, etc
- la forme la plus générale est `GeneralPath`
  - composition de chemins (paths) composés de lignes, cubiques, quadratiques...

- la classe `BasicStroke` permet de créer toute sorte de traits (pointillés, etc)

`BasicStrokeExemple.java`

- On peut créer ses propres Strokes

`CustomStrokeExemple.java`

- La classe `Font` est utilisée pour représenter une police de caractères
- qui correspond à ce qui est nécessaire pour afficher du texte à l'écran
  - soit : convertir un caractère (un symbole) en dessin (le glyphe associé)



- Il existe de catégories de polices :
  - les polices physiques
    - essentiellement celles fournies par l'OS
  - les polices logiques
    - sont garanties d'usage, mais leur apparence peut varier d'une plateforme à l'autre...
    - cinq :
      - Serif, SansSerif, Monospaced, Dialog, DialogInput
      - `Font.SERIF`, `Font.SANS_SERIF`, etc.

- Chaque police peut être désirée dans un style (une variante particulière) :

PLAIN

BOLD

ITALIC

BOLD      ITALIC

- Attention : ce style est combiné avec la police désirée (*i.e.* Helvetica Bold avec ITALIC)
- ici Helvetica Bold est une fonte (différente de Helvetica en variante Bold!)

```
Font(String nom, int style, int taille)
```

```
Font( SansSerif ,Font.BOLD,24 );
```

- GraphicsEnvironment :

```
Font [] getAllFonts()
```

```
String [] getAvailableFontFamilyNames()
```

```
String [] getAvailableFontFamilyNames(Locale)
```

- `g` est un `Graphics` ou un `Graphics2D`
- La modification de la police courante s'effectue avec  
`g.setFont(Font);`
- le rendu d'une chaîne de caractères s'effectue avec  
`g.drawString( );`

- Le rendu d'un texte dans une police donnée nécessite la connaissance des caractéristiques dimensionnelles de la police

## FontMetrics

ascent, maxAscent, descent, maxDescent,  
height, leading, maxAdvance



- Il existe des méthodes utilitaires pour calculer les dimensions de chaînes de caractères

- pour obtenir une métrique *police*

`LineMetrics getLineMetrics(...)`

- pour obtenir une métrique *géométrique*

`Rectangle2D getStringBounds(...)`

- La manipulation des images est délicate
- leur représentation/codage est très variable
  - la classe est `java.awt.Image`
  - la classe la plus fréquemment utilisée est `java.awt.image.BufferedImage`
- Son affichage peut être obtenu par l'une des nombreuses variantes de `g.drawImage( ) ;`

- La lecture ou l'écriture **synchrone** d'une image s'effectue par l'intermédiaire de la classe

`javax.imageio.ImageIO`

- méthodes statiques

`read(File), read(URL)`

`write(RenderedImage i,String f,File o);`

- Les formats supportés sont, au moins, jpeg, png, gif, bmp, wbmp.



- La lecture asynchrone d'une image s'effectue par l'intermédiaire de la classe

`java.awt.Toolkit`

- méthode statique

`getDefaultToolkit()`

- puis

`getImage(URL)`

- Les formats supportés sont, au moins, jpeg, png, gif.
- `Toolkit` est la classe qui fait le pont entre le monde Java et le monde natif

`ImageAsyncLoadingExample.java`

- Comment gérer l'asynchronisme ?

`java.awt.MediaTracker`  
`ImageLoadingExample.java`

- suivi de chargement de médias

`java.awt.image.ImageObserver`  
`ImageAsyncObserverExample.java`

- suivi fin

- les Graphics2D ont la méthode  
`drawImage (BufferedImage  
i, BufferedImageOp o, int x, int y) ;`
- qui permet d'opérer des transformations
  - comme la détection de bord par exemple

- les Images peuvent être créées :
- via les constructeurs adéquats de `BufferedImage`
  - le problème est le choix du type...
- via un composant ou la configuration système
  - `Component.createImage`
  - `GraphicsConfiguration.createCompatibleImage`

- on notera que les Images sont des supports de dessin

`Image.getGraphics()`

- très utile pour le double buffering...

`java.awt.image.Raster`

- permettent la manipulation directe des données de l'image
  - très difficiles à utiliser

`java.awt.image.PixelGrabber`

- permettent de récupérer les données de l'image en RGB
- pour les petites lectures/modifications
  - `getRGB/setRGB` de `BufferedImage`

- Les applications sont souvent distribuées sous la forme d'une archive
- Java ARchive
  - une archive compressée
  - contenant tout ce qui est nécessaire à l'exécution d'une application Java

- Commande jar

`jar cfm archive manifest fichiers`

- si on encapsule des fichiers de données (par exemple des images), comment faire dans le code ?
- méthode URL `getResource(String nom)` du `ClassLoader` d'une classe chargée depuis le jar