# Les Images

Rémi Forax

# Plan

- Création d'image
- Chargement d'image
- Convertion d'image
- Traitement d'image

# Les différents implantations d'image

- Une image est un objet de la classe abstraite java.awt.Image
- Il existe plusieurs implantations :
  - java.awt.image.BufferedImage qui correspond à un tableau de pixel en mémoire
  - java.awt.image.VolatileImage qui correspond à une image stockée dans la carte video
  - sun.awt.image.ToolkitImage qui correspond à une image chargée de façon paresseuse par rapport à une source de donnée

# BufferedImage

- La classe **java.awt.image.BufferedImage** hérite de **java.awt.Image** et correspond à un tableau rectangulaire de pixels.
  - A chaque pixel est associé la couleur d'un point, dans une parmi plusieurs formes possibles appelées ***valeurs d'échantillonage***.
  - Ces valeurs sont interprétées selon le ***modèle de couleur*** de l'image (**ColorModel**).

# BufferedImage (2)

- Un **BufferedImage** est composé :
  - d'un **ColorModel** qui définit la façon d'interpréter les couleurs
  - d'un **WritableRaster**, donc d'un raster autorisé en écriture.
- Un **Raster** est composé :
  - d'un **DataBuffer** contenant les données brutes, dans un tableau
  - d'un **SampleModel** (modèle d'échantillonage) qui interprète les données brutes.
- On obtient le modèle et le raster par les méthodes **get**.

# Création d'un BufferedImage

- Création d'une image vide

```
BufferedImage(int width, int height, int typeImage)
```

- Les types d'images indiquent comment les couleurs des pixels sont codées.

| | |
|---|---|
| `TYPE_3BYTE_BGR` | bleu, vert, rouge, sur 8 bits chacun |
| `TYPE_4BYTE_ABGR` | alpha, bleu, vert, rouge, sur 8 bits chacun |
| `TYPE_4BYTE_ABGR_PRE` | alpha, bleu, vert, rouge, sur 8 bits chacun, les couleurs pondérées |
| `TYPE_BYTE_BINARY` | 1 bit par pixel, groupés en octets |
| `TYPE_BYTE_INDEXED` | 1 octet par pixel, indice dans une table de couleurs |
| `TYPE_BYTE_GRAY` | 1 octet par pixel, niveau de gris |
| `TYPE_USHORT_555_RGB` | rouge, vert, bleu, sur 5 bits, codés dans un short |
| `TYPE_USHORT_565_RGB` | rouge sur 5, vert sur 6, bleu sur 5 bits |
| `TYPE_USHORT_GRAY` | niveau de gris sur 16 bits |
| `TYPE_INT_RGB` | rouge, vert, bleu sur 8 bits chacun, dans un int |
| `TYPE_INT_BGR` | bleu, vert, rouge (Solaris) |
| `TYPE_INT_ARGB` | alpha, rouge, vert, bleu sur 8 bits, dans un int |
| `TYPE_INT_ARGB_PRE` | les couleurs déjà pondérées par alpha |

# Modèle de couleur de l'image

- Le **ColorModel** permet d'interpréter une couleur en fonction du type de l'image

```
ColorModel model= image.getColorModel();
```

- La méthode **getDataElements()** permet de décomposer une couleur en composantes

```
int argb = Color.BLACK.getRGB();
Object datas = model.getDataElements(argb, null);
```

datas correspond la plupart du temps à un tableau contenant les valeurs des composantes dans le bon ordre pour être stocké dans l'image

# A partir des pixels

- On accede aux pixels par un objet **WritableRaster**, par

```
BufferedImage image=new BufferedImage(500,400,
  BufferedImage.TYPE_AINT_RGB);
WritableRaster raster = image.getRaster();
```

- **Pour changer la valeur des pixels :**
  - la méthode générique **raster.setDataElement(int x,int y,Object datas)**, où datas est la valeur renvoyée par **getDataElement()**
  - une des méthode **raster.setPixel()**

```
Object datas = model.getDataElements(Color.BLUE.getRGB(), null);
raster.setDataElement(i,j,datas);
```

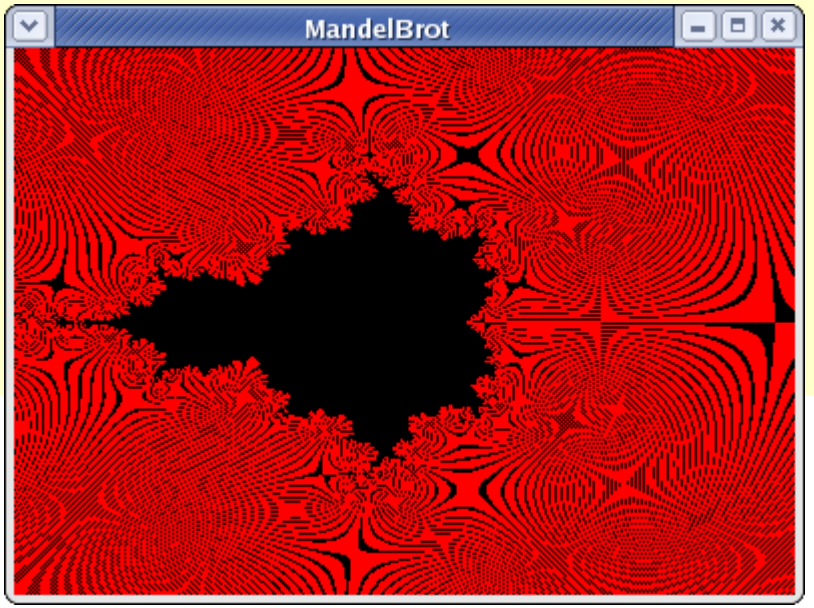
# A partir des pixels (2)

- **La méthode setPixel()** est surchargée pour chaque type d'image
- Pour le type **TYPE_INT_ARGB**,
  la méthode est **raster.setPixel(int i,int j,int[] datas)**
  le tableau doit contenir 4 entiers entre 0 et 255, pour les composantes alpha, rouge, vert, bleu :

```
int[] blue = {0,0,0,255};
raster.setPixel(i,j,blue);
```

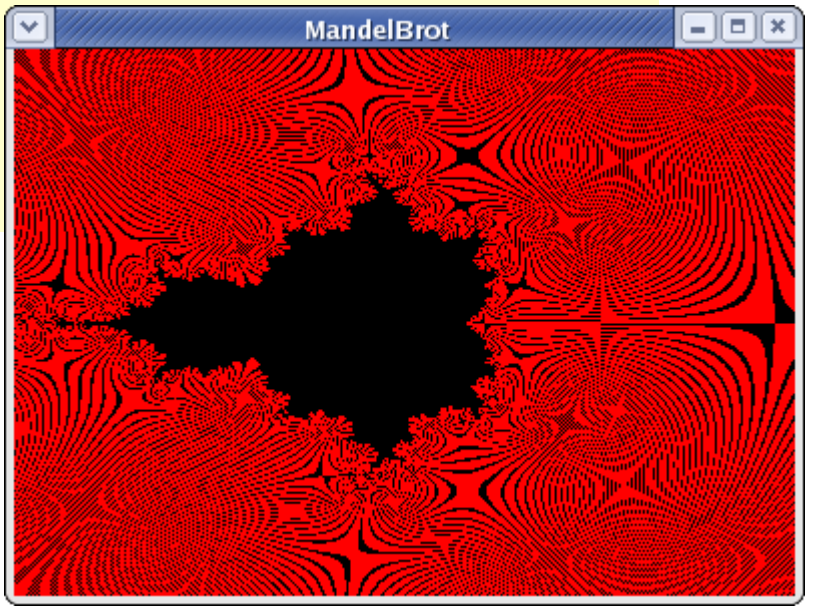
# Exemple de la fractal de mandelbrot

```
public class AsynchronousMandelBrot extends JComponent {
  BufferedImage image;
  private Thread thread;
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    int width=getWidth();
    int height=getHeight();
    if (image==null ||
        image.getWidth()!=width || image.getHeight()!=height) {
      if (thread!=null)
        thread.interrupt();

      image=new BufferedImage(width,height,BufferedImage.TYPE_INT_ARGB);
      thread=new Thread() {
        public void run() {
          computeMandelbrot(image);
        }
      };
      thread.start();
    }
    g.drawImage(image, 0, 0, null);
  }
```



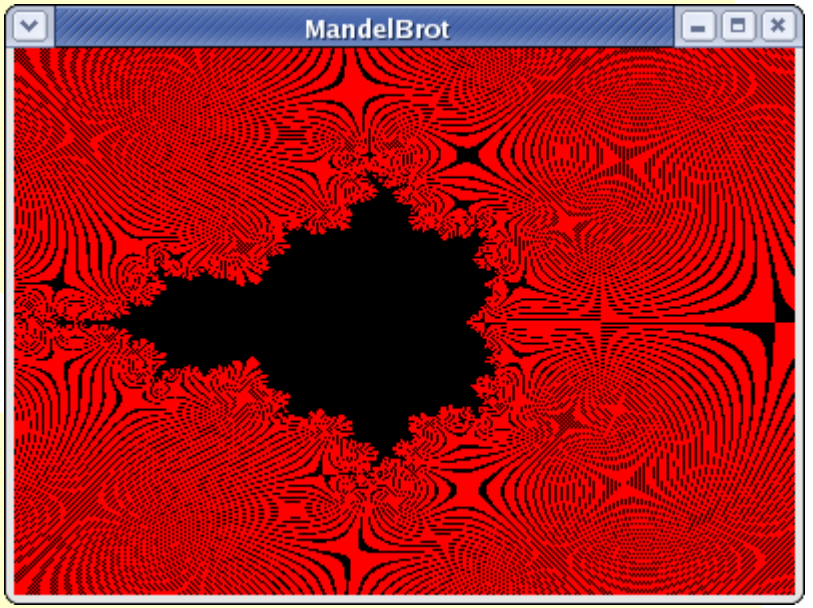
# Mandelbrot (2)

```
void computeMandelbrot(BufferedImage image) {
  int width = image.getWidth();
  int height = image.getHeight();
  WritableRaster raster = image.getRaster();
  ColorModel model = image.getColorModel();

  Object red = model.getDataElements(Color.RED.getRGB(), null);
  Object black = model.getDataElements(Color.BLACK.getRGB(), null);

  double[] xs=new double[width*height];
  double[] ys=new double[width*height];

  for(int iter=0;!Thread.currentThread().isInterrupted() &&
                 iter<MAX_ITERATIONS;iter++) {
    processOnePass(raster,width,height,
                   xs,ys,red,black);
    repaint();
  }
}
```



# Mandelbrot (3)

```
private void processOnePass(WritableRaster raster,int width,int height,
                            double[] xs,double[]ys, Object red,Object black) {
  for(int i=0;i<width;i++)
    for(int j=0;j< height;j++) {
      double a=XMIN+i*(XMAX-XMIN)/width;
      double b=YMIN+j*(YMAX-YMIN)/height;

      raster.setDataElements(i,j,
        diverge(xs,ys,i,j,width,a, b)?red:black);
    }
}
```

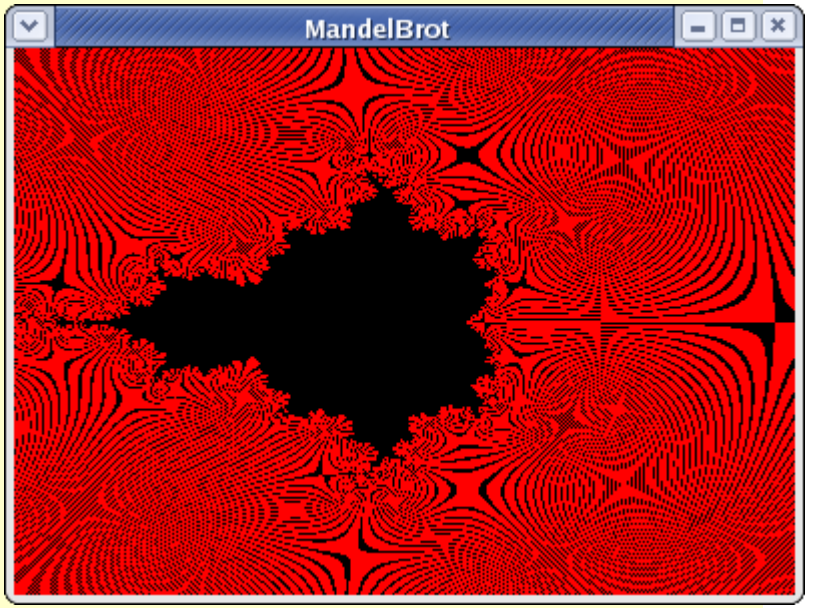



```
 public static void main(String[] args) {
   JFrame frame=new JFrame("MandelBrot");

   frame.setIgnoreRepaint(true);

   frame.setContentPane(new AsynchronousMandelBrot());
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setSize(400,300);
   frame.setVisible(true);
 }
```

# Mandelbrot (4)

- Calcul de la divergence

```
private boolean diverge(double[] xs, double[] ys, int i,int j,
                        int width,double a, double b) {
    double xnew, x = xs[i+j*width];
    double ynew, y = ys[i+j*width];

    xnew = x * x - y * y + a;
    ynew = 2 * x * y + b;

    xs[i+j*width]=xnew;
    ys[i+j*width]=ynew;
    return x > 2 || y > 2;
  }

  private static final double XMIN = -2;
  private static final double XMAX = 2;
  private static final double YMIN = -2;
  private static final double YMAX = 2;
  private static final int MAX_ITERATIONS = 20;
}
```



# Changement du type d'une image

- Pour changer une image du type à un autre, le plus simple consiste à copier l'image dans une autre créer avec le bon type

```
private static BufferedImage copyImage(BufferedImage src, int imageType) {
  BufferedImage bufferedImage=new BufferedImage(
      src.getWidth(),src.getHeight(), imageType);
  Graphics2D g2 = bufferedImage.createGraphics();
  g2.drawImage(src, null, null);
  g2.dispose();
  return bufferedImage;
}
```

- Pour copier une image dans une autre, le plus facile est de passer par un **Graphics**

# Chargement d'image

- En Java, les images peuvent être chargées à partir :
  d'un nom de fichier, d'une URL, d'un flux (InputStream)

- Il existe deux façon de charger des images :
  - La classe java.awt.Toolkit (depuis la 1.0),
    permet uniquement de lire les images,
    utilise les décodeurs présent sur la plateforme,
    au moins GIF, JPG et PNG (depuis la 1.3)
  - La classe javax.imageio.ImageIO (depuis la 1.4),
    permet de lire et écrire,
    possède un mécanisme de SPI

# javax.imageio

- Issu du projet Java Advanced Imaging, ce sont les classes du paquetage **javax.imageio**.
- Propose des lecteurs (**ImageReader**) et ecrivain (**ImageWriter**) pour les types les plus classiques
- Format supportés par défaut sont :

  JPG (r/w), GIF (r), PNG(r/w) et BMP/WBMP(r/w) (1.5)
- Possède un mécanisme de SPI qui permet d'inclure de nouveau type en ajoutant des jars
  par ex: format DICOM imagerie médicale
  encoder payant de GIF, gif4j.com

# Utilisation simple

- ImageIO possède 4 méthode read qui renvoie un BufferedReader en fonction de d'une URL, d'un File, d'un InputStream ou d'un ImageInputStream
- ImageIO possède 3 méthode read qui prennent en paramétre un BufferedReader, un nom de format, et soit un File, un OutputStream ou un ImageOutputStream

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

public class Transcode {
  public static void main(String[] args) throws IOException {
    BufferedImage image=ImageIO.read(new File("lena.jpg"));
    ImageIO.write(image,"png",new File("lena.png"));
  }
}
```

# Connaître l'ensemble des décodeurs

- La méthode **ImageIO.getReaderFormatNames()** permet de connaître l'ensemble des format supportés et **getImageReadersByFormatName(String formatName)** renvoie un iterateur sur l'ensemble des ImageReader pour un format

```
public class AllReaders {
  public static void main(String[] args) throws IOException {
    for(final String format:ImageIO.getReaderFormatNames()) {
      for(ImageReader reader:new Iterable<ImageReader>() {
        public Iterator<ImageReader> iterator() {
          return ImageIO.getImageReadersByFormatName(format);
        }
      })
        System.out.println(format+" "+reader.getClass().getName());
    }
  }
}
```

# Obtenir la taille des images

- Il est possible d'obtenir la taille d'un image sans décoder toute l'image, en effet les readers ne décode que ce qui est nécessaire en fonction des demandes.

```
public static void main(String[] args) throws IOException {
  ImageInputStream input=ImageIO.createImageInputStream(
    new File("lena.jpg"));
  Iterator<ImageReader> readers=ImageIO.getImageReaders(input);
  if (!readers.hasNext())
    return;
  ImageReader reader=readers.next();
  reader.setInput(input);
  System.out.println(reader.getWidth(0)+" "+reader.getHeight(0));
}
```

- Pour un reader, une image peut être composer de plusieurs image. Les méthodes **getWith()** et **getHeight()** prennent donc un index en paramètre.

# Le Toolkit

- Chaque plateforme possède une implantation de la classe **java.awt.Toolkit**
  - **sun.awt.X11.XToolkit** (linux)
  - **sun.awt.motif.MToolkit** (motif, principalement solaris)
  - **sun.awt.windows.WindowsToolkit** (windows)
- Cette classe permet de:
  - créer les *peers* correspondant au composants heavyweight
  - voir les propriétés de la plateforme (fenêtre opaque, etc)
  - charger des images

# Toolkit & Chargement

- Il existe deux méthodes qui permette de charger des images en utilisant la classe java.awt.Toolkit
  - Image createImage(filename ou URL)
  - Image getImage(filename ou URL)
- L'image renvoyée ne contient pas de donnée (*proxy*)
- Le chargement à lieu lors du premier appel à drawImage
- celui-ci en fait en asynchrone par une thread créé, le drawImage() n'est donc pas bloquant
- L'affichage peux donc être progressif

# createImage()/getImage()

- **getImage()** par raport à **createImage()** maintient un cache de toutes les images chargées pour partager la même image entre plusieurs appels
- Le problème est que ce cache ne libère que rarement (voir jamais) les images chargées, donc **à ne pas utiliser**

```
public static Image getImage(String filename){
  Image image=cache.get(filename);
  if (image==null) {
    image=createImage(filename);
    cache.put(filename,image);
  }
  return image;
}
private final Map<String,Image> cache=...
```

# Affichage progressif

- **ImageObserver** est une interface que l'on doit implanter si l'on veut être averti de la progression du chargement de l'image
- Par défaut, l'ensemble des Component de l'AWT/Swing implante cette interface ce qui leur permet d'afficher l'image au fur et à mesure que les données sont décodés

```
public class ProgressiveDecode extends JComponent {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(image,0,0,observer);
  }
  private final Image image=Toolkit.getDefaultToolkit().createImage("lena.jpg");
  private final ImageObserver observer=new ImageObserver() {
    public boolean imageUpdate(Image img,int flags,int x,int y,int w,int h) {
      System.out.println(infoflags+" "+x+' '+y+' '+w+' '+h);
      return ProgressiveDecode.this.imageUpdate(img,flags,x,y,w,h);
    }
  };
  ...
```

# getScaledInstance()

- Un des intérêt d'effectuer le chargement au dernier moment est le fait que cela évite de stocker l'image originale en mémoire si l'on veut uniquement une version plus petite

```
public class ScaledDecode extends JComponent {
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(image,0,0,this);
  }
  private final Image image=Toolkit.getDefaultToolkit().
    createImage("lena.jpg").getScaledInstance(100,100,
                                              Image.SCALE_SMOOTH);
}
```

- Dans l'exemple, l'ensemble des données sur disque du fichier lena.jpg est lu mais seul un tableau de 100x100 pixels est utilisé en mémoire.

# MediaTracker

- Création avec le composant dans lequel l'image sera affichée (obligatoire null impossible ici)

```
MediaTracker tracker = new MediaTracker(this);
```

- Les images à pister sont ajoutées avec un numéro d'identification

- On attend la fin du chargement d'une image

```
Image image = toolkit.getImage(nom);
tracker.addImage(image,0);
```

et pour toutes les images

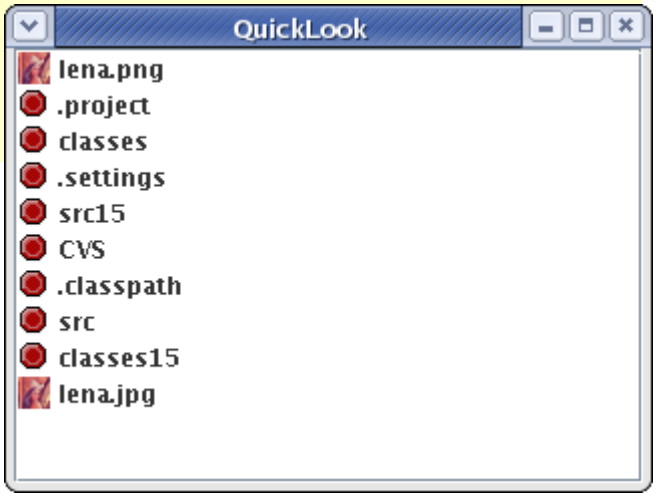
```
try { tracker.waitForID(0); }
catch(InterruptException e) {
```

```
try { tracker.waitForAll(); }
catch(InterruptException e) {
```

# Example utilisant le MediaTracker

- On souhaite afficher des fichiers avec un petit icône

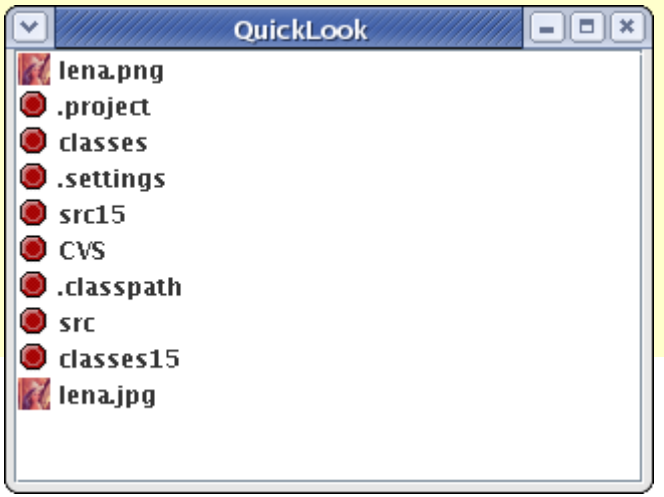
```
public class QuickLookModel extends AbstractListModel {
  public QuickLookModel(File directory) {
    this.files=directory.listFiles();
    tracker=new MediaTracker(new JPanel());
    Toolkit toolkit=Toolkit.getDefaultToolkit();
    images=new Image[files.length];
    for(int i=0;i<images.length;i++) {
      Image image=toolkit.createImage(files[i].getPath());
      image=image.getScaledInstance(16, 16,Image.SCALE_SMOOTH);
      tracker.addImage(image,i);
      images[i]=image;
    }
    ...
  }
```



# le MediaTracker (2)

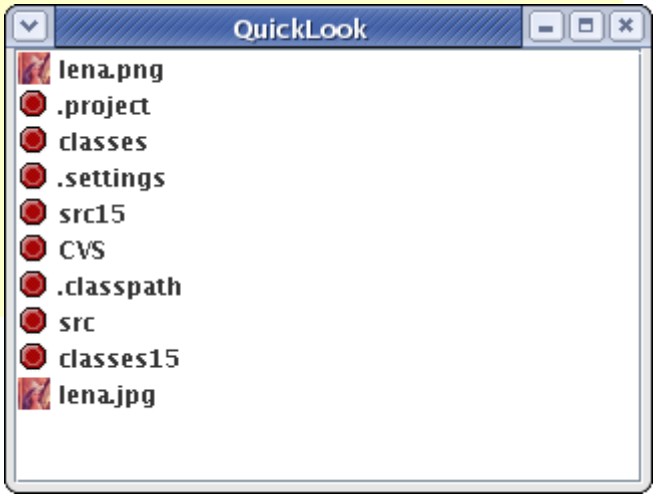
- La thread du pisteur d'images

```
public class QuickLookModel extends AbstractListModel {
  public QuickLookModel(File directory) {
    ...
    new Thread() {
      public void run() {
        for(int i=0;i<images.length;i++) {
          try {
            final int id=i;
            tracker.waitForID(id);
            EventQueue.invokeLater(new Runnable() {
              public void run() {
                fireContentsChanged(QuickLookModel.this, id, id);
              }
            });
          } catch(InterruptedException e) {
          }
        }
      }
    }.start();
  }
```



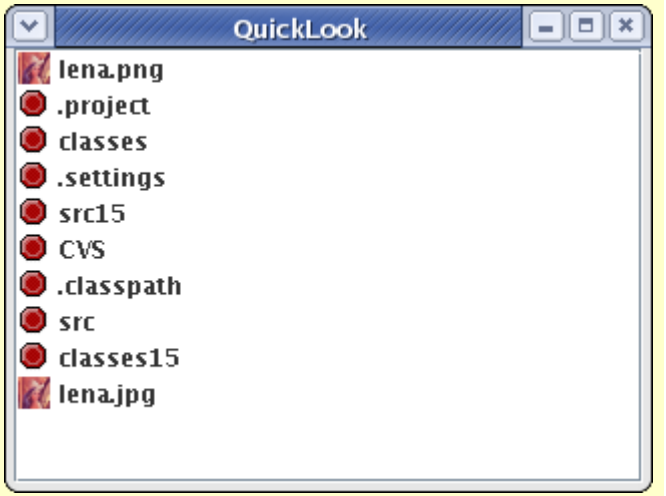
# le MediaTracker (3)

```
public class QuickLookModel extends AbstractListModel {
  public int getSize() {
    return files.length;
  }
  public File getElementAt(int index) {
    return files[index];
  }
  public Image getImageAt(int index) {
    int status=tracker.statusID(index, false);
    if ((status & MediaTracker.COMPLETE)!=0)
      return images[index];
    if ((status & (MediaTracker.ABORTED|MediaTracker.ERRORED))!=0)
      return NO_IMAGE;
    return null;
  }
  final MediaTracker tracker;
  private final File[] files;
  final Image[] images;

  private static final Image NO_IMAGE=
    new ImageIcon(QuickLookModel.class.
      getResource("Stop16.gif")).getImage();
```



# le MediaTracker (4)

```
public class QuickLookModel extends AbstractListModel {
  public static void main(String[] args) {
    final QuickLookModel model=new QuickLookModel(new File("."));
    JList list=new JList(model);

    list.setCellRenderer(new DefaultListCellRenderer() {
      public Component getListCellRendererComponent(
        JList list,Object value,int index,boolean isSelected,
        boolean focus) {

        super.getListCellRendererComponent(list,value,index,isSelected,focus);
        Icon icon=icons[index];
        if (icon==null) {
          Image image=model.getImageAt(index);
          if (image!=null) {
            icon=icons[index]=new ImageIcon(image);
            System.out.println(icon);
          }
        }
        setIcon(icon);
        setText(((File)value).getName());
        return this;
      }
      private final Icon[] icons=new Icon[model.getSize()];
    });
```



# ImageIcon et Swing

- La classe **javax.swing.ImageIcon** est une implémentation de l'interface **javax.swing.Icon.**
- Les constructeurs

```
Icon icon = new ImageIcon(filename);
Icon icon2 = new ImageIcon(url);
```

utilisent un MediaTracker, l'image est chargée si ce n'est pas déjà fait avant de retourner.

- La méthode **getImage()** retourne l'image représentée par l'icône.

# VolatileImage

- Correspond à une image stockée dans la mémoire de la carte video donc ça va plus vite à afficher
- Voici les différentes factory possible :
  - Component.createVolatileImage(int width, int height)
  - GraphicsConfiguration.createCompatibleVolatileImage( int width, int height)
  - GraphicsConfiguration.createCompatibleVolatileImage( int width, int height, int transparency) (pas super supporté par les cartes :)
- GraphicsDevice.getAvailableAcceleratedMemory() renvoie la mémoire totale de la carte graphique

# ImageCapabilities

- Correspond aux capacité de la VolatileImage : volatileImage.getCapabilities()
- Certaine carte video ont une mémoire très rapide mais qui peut être effacer sans préavis.
- Il y a donc deux stades d'accélérations :
  - l'image réside dans la mémoire de la carte : capabilities.isAccelerated()
  - l'image peut-être effacer à tout instant : capabilities.isTrueVolatile()

# VolatileImage et problèmes

- Il y a deux problèmes qui peuvent arriver lorsque l'on utilise des VolatileImage
- Le contenue de l'image peut être perdu :
  tester **volatileImage.contentLost()**
  il faut dans ce cas redessiner le contenue de l'image
- La configuration graphique à pu changer à chaud
  (par exemple : Ctrl Alt + sous Linux), le type de l'image peut alors être incompatible :
  tester **volatileImage.validate(getGraphicsConfiguration())**
- il faudra recréer une image :

# Afficher une VolatileImage

```
private VolatileImage image;
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  if (image==null)
    image=createVolatileImage(getWidth(), getHeight());
  do {
    int code=image.validate(getGraphicsConfiguration());
    switch(code) {
      case VolatileImage.IMAGE_INCOMPATIBLE:
        image = createVolatileImage(getWidth(), getHeight());
      case VolatileImage.IMAGE_RESTORED:
        renderOffscreen();      // restore contents
    }
    if (image.validate(getGraphicsConfiguration())
        ==VolatileImage.IMAGE_OK)
      g.drawImage(image, 0, 0, this);
  } while (image.contentsLost());
}
void renderOffscreen() {
  int w=getWidth();
  int h=getHeight();
  Graphics2D g=image.createGraphics();
  for(int i=0;i<1000;i++) {
    g.drawLine(r.nextInt(w),r.nextInt(h),r.nextInt(w),r.nextInt(h));
  }
  g.dispose();
}
```

# ImageIO vs Toolkit

- Il y a une différence dans le type des images renvoyé par ImageIO et Toolkit
  - ImageIO renvoie l'image dans le type le plus facile pour décoder l'image
  - Toolkit renvoie l'image dans le type par défaut de l'écran
- Il est donc important lorsque l'on cherche la meilleur performance de copier les image issue de ImageIO pour les mettre dans le format de l'écran

# Opérations sur les images

- Java propose des opérations de traitement d'images à base de filtres implantant l'interface **BufferedImageOp**.
- Il existe cinq classes qui implémentent **BufferedImageOp**:
  - Transformation affine **`AffineTransformOp`**
  - Rescale de couleur **`RescaleOp`**
  - Changement de couleur **`LookUpOp`**
  - Convertion de couleur **`ColorConvertOp`**
  - Convolution **`ConvolveOp`**

# Filtrer une image

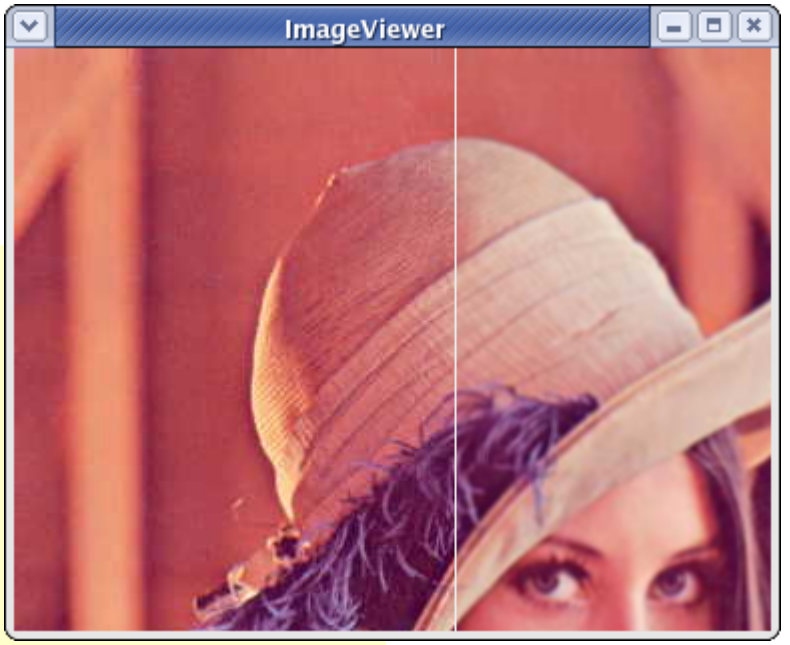
- Appliquer une opération de filtrage revient à appeler la méthode filter(BufferedImage src,BufferedImage dst) sur un BufferedImageOp

```
BufferedImageOp op=new ConvolveOp(...);

BufferedImage
right=op.createCompatibleDestImage(left,left.getColorModel())
;
op.filter(left,right);
```

- Le paramètre destination peut être null dans ce cas, un BufferedImage destination est créé et renvoyé en valeur de retour

```
BufferedImageOp op=new ConvolveOp(...);

BufferedImage right=op.filter(left,null);
```

# Visualisateur d'image




```
public class ImageViewer extends JComponent {
  public ImageViewer(Image left,Image right) {
    this.left=left;
    this.right=right;

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        update(e.getX());
      }
    });
    addMouseMotionListener(new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        update(e.getX());
      }
      public void mouseMoved(MouseEvent e) {
      }
    });
  }
```

# Visualisateur d'image(2)

```
void update(int newSplit) {
  int x,width;
  if (split<newSplit) {
    x=split;
    width=newSplit-split;
  } else {
    x=newSplit;
    width=split-newSplit;
  }
  split=newSplit;
  paintImmediately(x-1,0,width+2,getHeight());
}
protected @Override void paintComponent(Graphics g) {
  super.paintComponent(g);
  int width=getWidth();
  int height=getHeight();
  g.setClip(0,0,split-1,height);
  g.drawImage(left,0,0,this);
  g.setClip(split,0,width-split,height);
  g.drawImage(right,0,0,this);
}
private int split;
private final Image left;
private final Image right;
```

# Convolution

- Une convolution calcule la valeur pour chaque pixel en pondérant les couleurs des 9 pixels (3x3) de son entourage.
- Les valeurs sont données dans une matrice de flottant appelé *noyau* (kernel en anglais).
- Les plus spectaculaires sont le flou (*blur*) et la détection de contour
- Les matrices de covolutions habituelles :
  - Pour le flou, les 9 pixels ont même poids.
  - Pour la détection de contour, les voisins ont un poids nul ou négatif.

# ConvolveOp

- Une convolution est définie par :
  - un noyau (kernel)
  - un comportement sur les bords
  - éventuellement des indications sur le rendu (**RenderingHints**)
- Deux constructeurs :

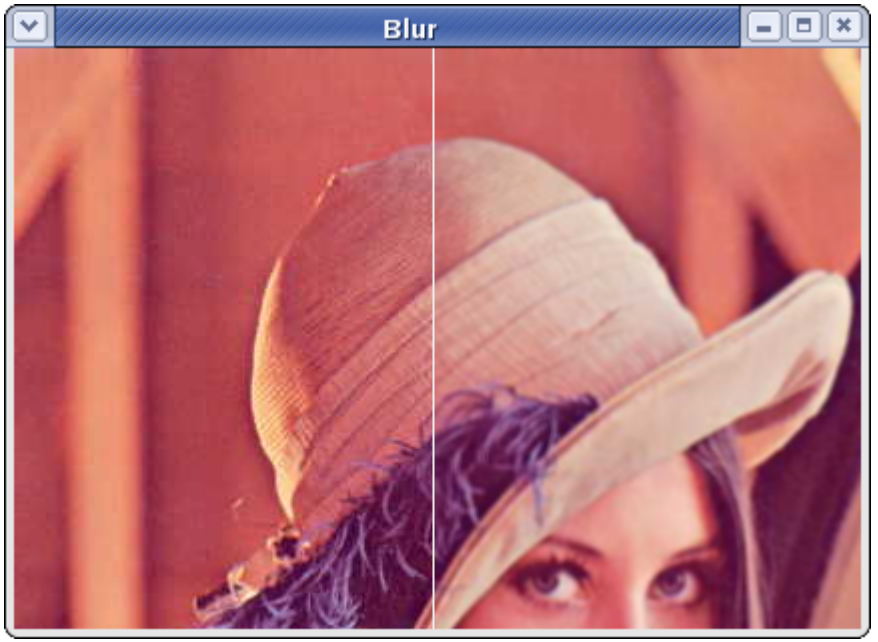
```
ConvolveOp(Kernel kernel)
ConvolveOp(Kernel kernel,int edgeCondition,RenderingHints hints);
```

- les conditions au bords sont :
  - **EDGE_ZERO_FILL** les pixels aux bords sont traités comme s'ils entourés de zéros (défaut)
  - **EDGE_NO_OP** les pixels aux bords sont copiés sans modification

# Matrice de convolution

- Pour le flou :

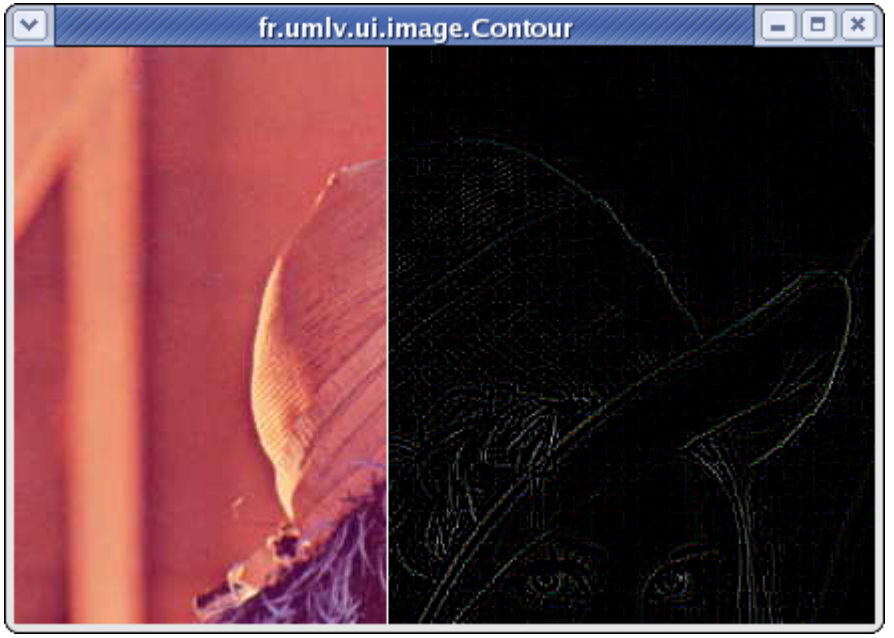
```
float[] blur =new float[] {
   1f/9f, 1f/9f, 1f/9f,
   1f/9f, 1f/9f, 1f/9f,
   1f/9f, 1f/9f, 1f/9f
};
Kernel kernel = new Kernel(3, 3, blur);
ConvolveOp op = new ConvolveOp(kernel,ConvolveOp.EDGE_NO_OP,null);
BufferedImage dst = op.filter(src,null);
```



# Matrice de convolution (2)

- La détection de contour :

```
float[] contour=new float[]{
  0f, -1f, 0f,
  -1f, 4f, -1f,
  0f, -1f, 0f
};
```





- L'accentuation des couleurs :

```
float[] sharpen=new float[]{
  0f, -1f, 0f,
  -1f, 5f, -1f,
  0f, -1f, 0f
};
```

# Rescale

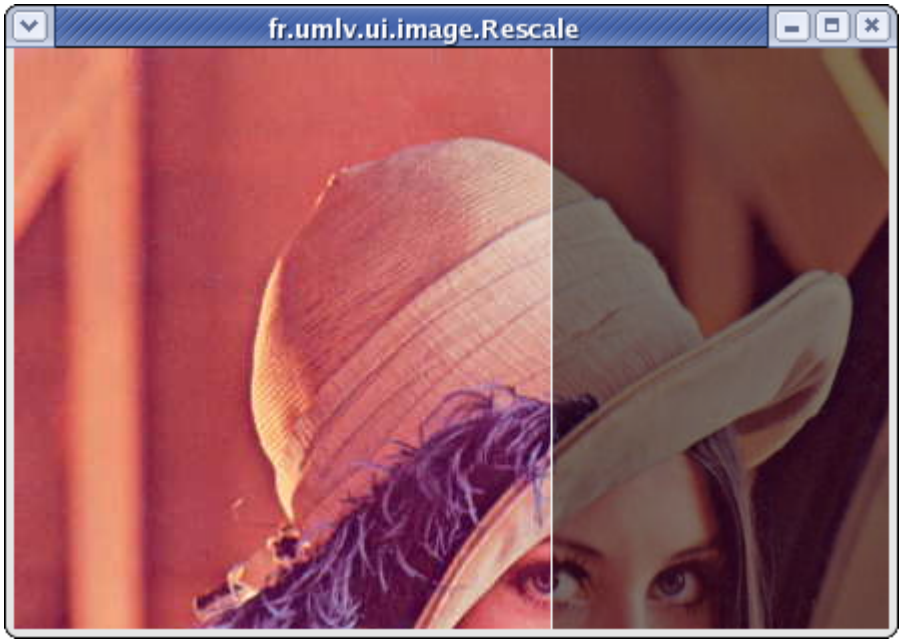
- Les opérations de “rescaling” modifient les intensités des composantes RGB par deux paramètres :
  - un facteur multiplication ***m***
  - un décalage ***d***
- La nouvelle intensité est ***i’ = i * m + d***
- Ainsi,
  - si ***m*** < 1, l’image est assombrie
  - si ***m*** > 1, l’image est plus brillante
  - ***d*** est compris entre 0 et 256 et ajoute un éclairement supplémentaire
- Exemples:

```
op = new RescaleOp(.5f, 0, null)
op = new RescaleOp(.5f, 64, null)
op = new RescaleOp(1.2f, 0, null)
op = new RescaleOp(1.5f, 0, null)
```

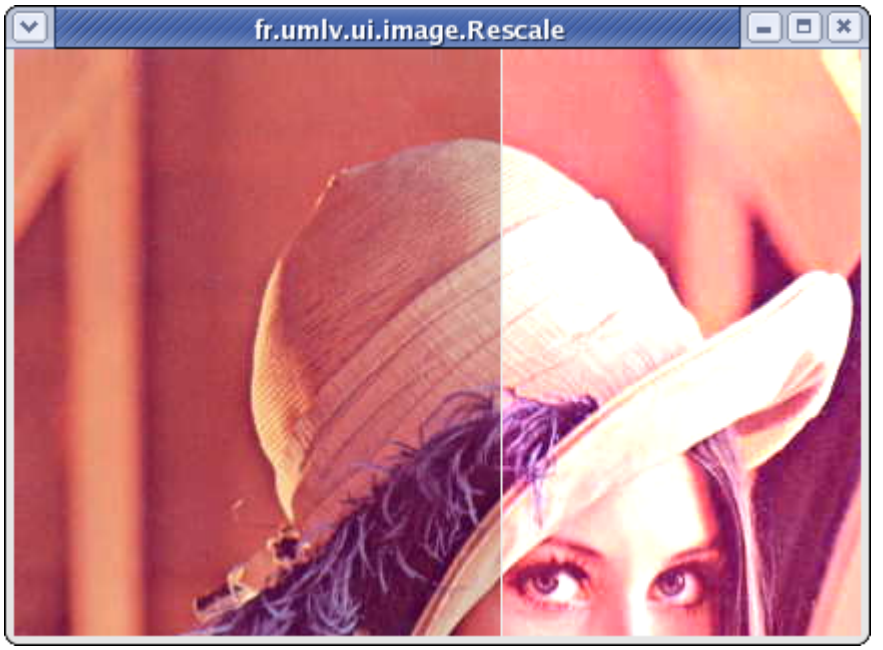
plus sombre
plus sombre avec éclairement
plus brillant
encore plus brillant

# Exemple de rescale

- Plus sombre :
  BufferedImageOp op =
    new RescaleOp(0.5f,0f,null);

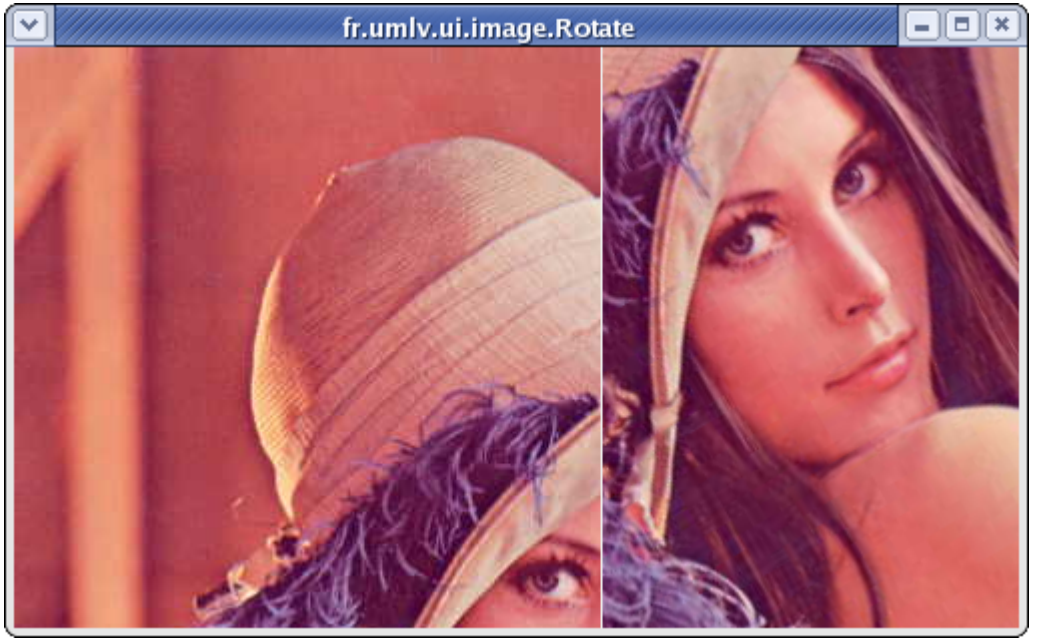



- Plus brillant :
  BufferedImageOp op =
    new RescaleOp(1.5f,0f,null);



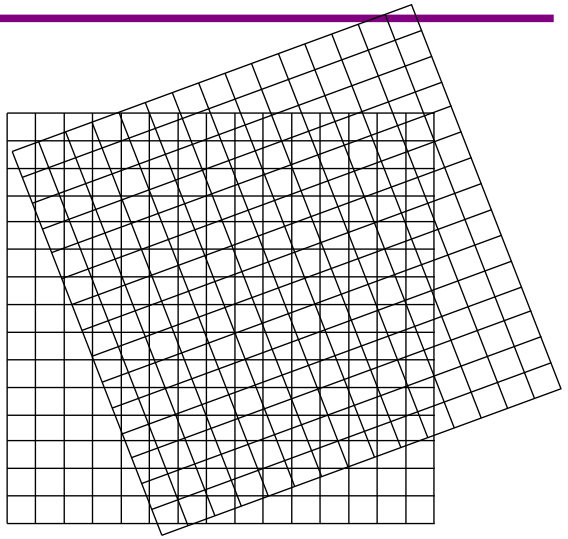
# Opérations affines

- Il est possible d'appliquer des transformations affines à une image :
  - rotation
  - translation (peu d'intérêt)
  - étirement
  - cisaillement
- Exemple :

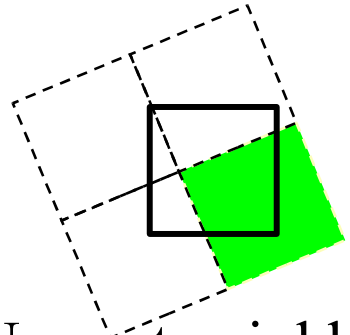



```
AffineTransform at=AffineTransform.getRotateInstance(Math.PI/6);
RenderingHints hints=new RenderingHints(
  RenderingHints.KEY_INTERPOLATION,
  RenderingHints.VALUE_INTERPOLATION_BILINEAR);
BufferedImageOp op=new AffineTransformOp(at,hints));
```

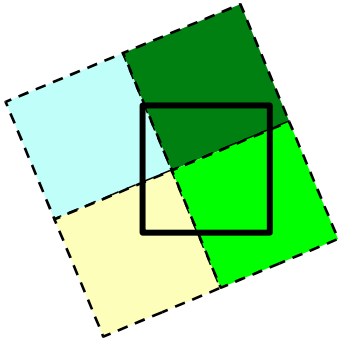
# Rotations



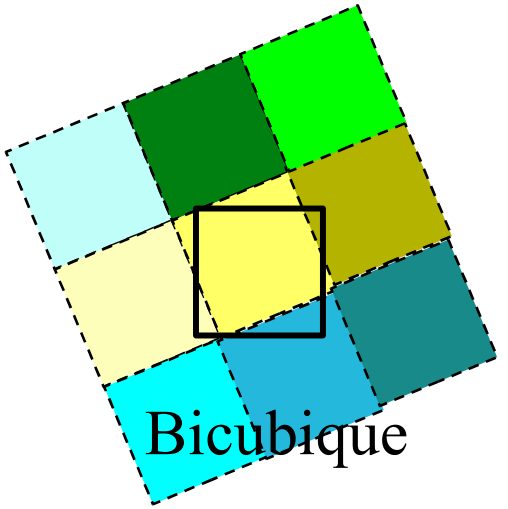
- Après rotation, une grille de points doit être affichée dans une autre grille.
  Quelle est la couleur d'un pixel ?
- Trois algorithmes, la couleur est celle :
  - ***nearest neighbor*** : du pixel le plus proche
  - ***bilinear*** : combinaison des quatre pixels source
  - ***bicubic*** : combinaison des neufs pixels source



Nearest neighbor



Bilinéaire



Bicubique

# Manipulation par “Lookup”

- Une opération de “Lookup” remplace les intensités de couleurs par inspection (lookup) dans des tables.
- Les tables d’inspections peuvent être de type
  **`ByteLookupTable`** OU **`ShortLookupTable`**
- Dans les constructeurs,
  - le premier argument est un décalage dans la table,
  - le deuxième un tableau (d’entiers courts ou d’octets) à une ou deux dimensions.
    - une dimension : les mêmes changements pour les trois couleurs
    - deux dimensions : chaque composante RGB modifiée de façon indépendante

# Exemple de lookup

- Supprimer les rouges

```
short[] ident=new short[256];
short[] zero=new short[256];
for(int i=0;i<256;i++)
  ident[i]=(short)i;
short[][] noRed={zero,ident,ident};
LookupTable table=
   new ShortLookupTable(0,noRed);
BufferedImageOp op=
   new LookupOp(table,null);
```

- Inverser les couleurs


```
short[] inverse=new short[256];
for (int i=0;i<256;i++)
  inverse[i]=(short)(255-i);
LookupTable table=new ShortLookupTable(0,inverse);
BufferedImageOp op = new LookupOp(table, null);
```

# Niveau de gris

- Transformation qui permet de passer d'une image en couleur à une image en niveau de gris en gardant la même intensité lumineuse pour chaque point

- Pour passer en niveau de gris, on utilise une opération de conversion d'espace de couleurs **ColorConvertOp**

- Le constructeur le plus utilisé :
  **ColorConvertOp(ColorSpace space,RenderHints hints)**

# Exemple de niveau de gris

- Un exemple de construction :

```
new ColorConvertOp(ColorSpace.getInstance(
  ColorSpace.CS_GRAY), null));
```