



---

# Interface Graphique en Java 1.6

## Look and Feel & Création de composants

Sébastien Paumier



# Look and Feel

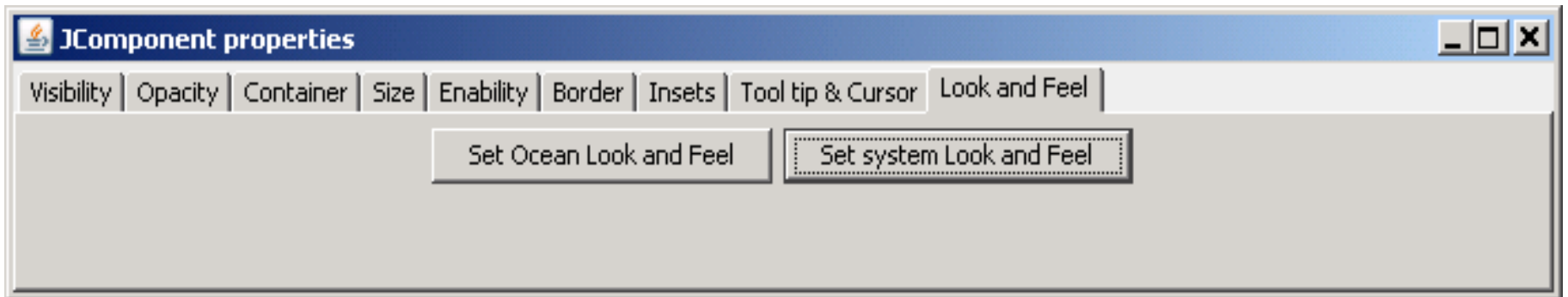
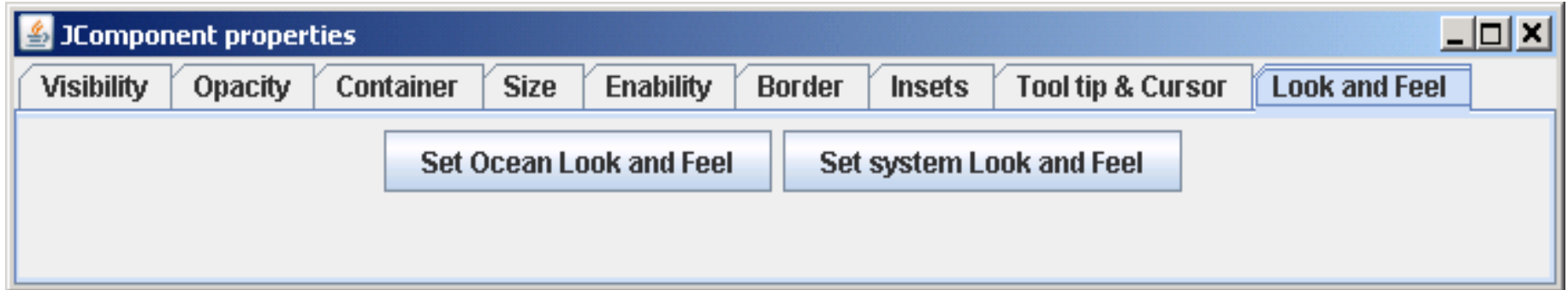
---

- mécanisme permettant de changer l'apparence des composants:
  - soit globalement, par exemple en adoptant le style du système
  - soit composant par composant
- possible, car en Swing, un composant n'est pas responsable de son rendu
- 1 objet Swing **JBinou**=1 objet **BinouUI**



# Look and Feel

- exemples de Look and Feel:





# UIManager

---

- **UIManager** gère les LnF disponibles, plus le LnF courant
- on obtient la liste des LnF disponibles avec **getInstalledLookAndFeels()** qui renvoie un tableau de **LookAndFeelInfo**:
  - **getName()** renvoie le nom du LnF
  - **getClassName()** renvoie le nom de la classe à charger avec **UIManager.setLookAndFeel**



# UIManager

---

- Lnf par défaut="Metal", thème "Ocean"
- 2 façons de changer le Lnf:
  - soit avant l'appel aux constructeurs des objets Swing
  - soit dynamiquement
- on utilise, au choix:

```
UIManager.setLookAndFeel(LookAndFeel l)
```

```
UIManager.setLookAndFeel(String class)
```



# UIManager

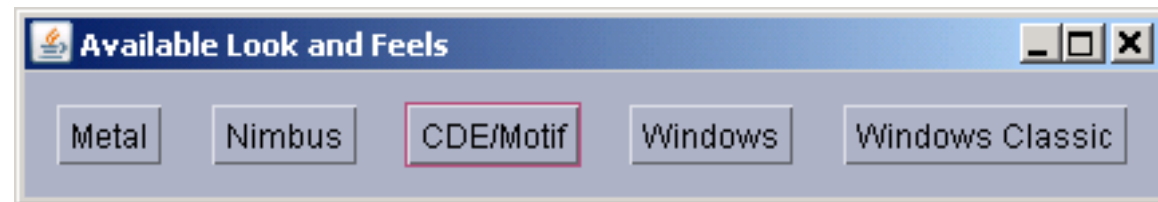
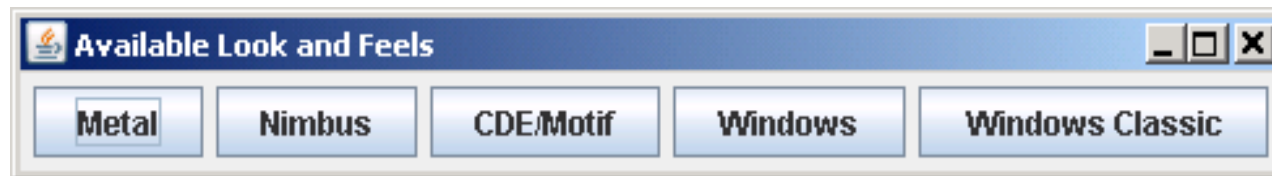
- quand on change dynamiquement le LnF, il faut demander à Swing de mettre à jour l'arborescence des composants concernés:

```
try {
    UIManager.setLookAndFeel(clazz);
} catch (ClassNotFoundException e1) {
    e1.printStackTrace();
} catch (InstantiationException e1) {
    e1.printStackTrace();
} catch (IllegalAccessException e1) {
    e1.printStackTrace();
} catch (UnsupportedLookAndFeelException e1) {
    e1.printStackTrace();
}
SwingUtilities.updateComponentTreeUI(f);
f.pack();
```



# UIManager

- examples:





# ComponentUI

---

- si l'on ne veut pas changer le Lnf de toute l'application, on peut le faire composant par composant avec les méthodes: **setUI** et **getUI**
- pour chaque composant Swing, on a une implémentation particulière de **ComponentUI: ButtonUI, TreeUI, etc**



# ComponentUI

---

- voici les méthodes de **ComponentUI**:
  - création de l'UI pour un composant:  
`static ComponentUI createUI (JComponent c)`
  - installation/désinstallation:  
`installUI (JComponent c)`  
`uninstallUI (JComponent c)`
  - tailles du composant (**null** si on délègue au  
LayoutManager qui gère le composant):  
`Dimension getMinimumSize (JComponent c)`  
`Dimension getMaximumSize (JComponent c)`  
`Dimension getPreferredSize (JComponent c)`



# ComponentUI

---

- 2 méthodes pour le dessin

```
void update(Graphics g, JComponent c)
```

```
void paint(Graphics g, JComponent c)
```

- **update** remplit le contrat d'opacité et appelle **paint**:

```
public void update(Graphics g, JComponent c) {  
    if (c.isOpaque()) {  
        g.setColor(c.getBackground());  
        g.fillRect(0, 0, c.getWidth(), c.getHeight());  
    }  
    paint(g, c);  
}
```

- inutile de la modifier: le rendu propre au composant doit être dans **paint**



# Le bouton miroir

- exemple de **ButtonUI** personnalisé, dérivé d'un **ButtonUI** existant:

```
public class MirrorButtonUI extends MetalButtonUI {  
  
    @Override public void paint(Graphics g, JComponent c) {  
        Graphics2D g2=(Graphics2D)g;  
        AffineTransform old=g2.getTransform();  
        g2.scale(-1,1);  
        g2.translate(-c.getWidth(),0);  
        super.paint(g2,c);  
        g2.setTransform(old);  
    }  
    ...  
}
```

```
JButton b=new JButton("Hello");  
b.setUI(new MirrorButtonUI());
```

on ne modifie qu'un bouton:





# Personnaliser le LnF

---

- lors qu'un **JComponent** est créé, il demande à l'**UIManager** de lui fournir le **...UI** qui lui correspond, via sa méthode **updateUI ()**
- exemple pour le **JButton**:

```
public void updateUI () {  
    setUI ( (ButtonUI) UIManager . getUI ( this ) ) ;  
}
```

- cela trouve l'objet **ButtonUI** à utiliser et invoque sa méthode **createUI**



# Personnaliser le LnF

---

- nous devons donc écrire la méthode **createUI** pour notre **MirrorButtonUI**:

```
/* All components can share the same MirrorButtonUI */  
private final static MirrorButtonUI mirrorUI=new MirrorButtonUI ();  
  
public static ComponentUI createUI (JComponent c) {  
    return mirrorUI;  
}
```



# Personnaliser le LnF

---

- l'**UIManager** délègue à la table de hachage **UIDefaults**
- on peut donc installer les **...UI** de son choix en modifiant **UIDefaults**

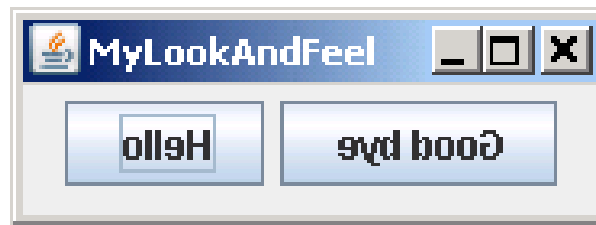
```
public class MyLookAndFeel extends MetalLookAndFeel {  
  
    @Override  
    public UIDefaults getDefaults() {  
        UIDefaults defaults=super.getDefaults();  
        defaults.put("ButtonUI", "fr.umlv.ig.lesson9.MirrorButtonUI");  
        return defaults;  
    }  
  
}
```



# Personnaliser le Lnf

- il ne reste plus qu'à installer notre Lnf pour changer l'aspect de tous les boutons:

```
try {
    UIManager.setLookAndFeel(new MyLookAndFeel());
} catch (UnsupportedLookAndFeelException e) {
    e.printStackTrace();
}
JFrame f=new JFrame("MyLookAndFeel");
JPanel p=new JPanel();
p.add(new JButton("Hello"));
p.add(new JButton("Good bye"));
```





# Créer un composant

---

- en mettant en application tout ce qui a déjà été vu, on peut créer de toutes pièces un nouveau composant:

le `JRadioSlider`



# Cahier des charges

---

- le **JRadioSlider** est un bouton rond que l'on tourne avec la souris
- il gère une valeur comprise dans un intervalle donné, en l'augmentant ou en la diminuant selon le sens de rotation du bouton, avec un *extent* donné
- on veut gérer le nombre d'augmentations de  $x$  par tour
  - exemple: un tour complet=4 augmentations de valeur  $x$



# Cahier des charges

---

- on veut une vue gérée en Look and Feel
- on veut un `RadioSliderUI` par défaut qui soit du style "Metal"
- on veut que l'interprétation du sens de rotation (+ ou -) dépende de l'orientation du composant (`ComponentOrientation`)



# Le modèle

---

- quel est le modèle de données ?
- le `BoundedRangeModel` correspond à ce que l'on veut faire, pas besoin de coder notre propre modèle
- on doit simplement gérer l'initialisation et le remplacement du modèle



# Le modèle

- on commence avec la gestion du modèle:

```
public class JRadioSlider extends JComponent {  
  
    private BoundedRangeModel model;  
    private int extentsPerTurn;  
  
    public JRadioSlider(int value, int extent, int min, int max, int extentsPerTurn) {  
        this(new DefaultBoundedRangeModel(value, extent, min, max), extentsPerTurn);  
    }  
  
    public JRadioSlider(BoundedRangeModel model, int extentsPerTurn) {  
        setModel(model);  
        this.extentsPerTurn=extentsPerTurn;  
        updateUI();  
    }  
  
    public BoundedRangeModel getModel() {  
        return model;  
    }  
  
    ...  
}
```



# Le modèle

- on interdit les modèles **null**
- s'il y avait déjà un modèle, on enlève le listener qu'on avait mis
- on rafraîchit l'UI, au cas où il y ait des choses en cache

```
public void setModel(BoundedRangeModel model) {
    if (model==null)
        throw new IllegalArgumentException("Invalid null model");
    if (this.model!=null) {
        this.model.removeChangeListener(changeListener);
    }
    this.model=model;
    model.addChangeListener(changeListener);
    /* When the model changes, we flush
     * everything the UI may have in cache */
    updateUI();
}
```



# Le modèle

- on va relayer les événements du modèle avec un listener créé une fois pour toutes:

```
private final ArrayList<ChangeListener> changeListeners=  
    new ArrayList<ChangeListener>();  
  
private boolean firing=false;  
  
public void addChangeListener(ChangeListener l) {  
    changeListeners.add(l);  
}  
  
public void removeChangeListener(ChangeListener l) {  
    if (firing)  
        throw new IllegalStateException(  
            "Cannot remove listeners while fire... is using them");  
    changeListeners.remove(l);  
}  
  
protected void fireStateChanged() {  
    ...  
}
```



# Le modèle

- pour plaire à l'utilisateur, on va ajouter des raccourcis pour accéder aux données du modèle:

```
public void setMinimum(int m) {  
    model.setMinimum(m);  
}  
  
public int getMinimum() {  
    return model.getMinimum();  
}  
  
public void setMaximum(int m) {  
    model.setMaximum(m);  
}  
  
public int getMaximum() {  
    return model.getMaximum();  
}
```

```
public void setValue(int v) {  
    model.setValue(v);  
}  
  
public int getValue() {  
    return model.getValue();  
}  
  
public void setExtent(int e) {  
    model.setExtent(e);  
}  
  
public int getExtent() {  
    return model.getExtent();  
}
```



# Le modèle

---

- on va aussi relayer l'information **valueIsAdjusting**, afin de permettre à l'utilisateur de ne pas essayer de modifier le **JRadioSlider** s'il est déjà en cours de modification, pour des raisons de rafraîchissement efficace

```
public void setValueIsAdjusting(boolean b) {  
    model.setValueIsAdjusting(b);  
}  
  
public boolean getValueIsAdjusting() {  
    return model.getValueIsAdjusting();  
}
```



# Le modèle

---

- on doit également gérer le paramètre **extentsPerTurn** qui contrôle le nombre d'augmentations par tour:
  - exemple: si **extentsPerTurn** vaut 4, chaque quart de tour correspondra à une augmentation (ou diminution) de valeur **extent**

```
public void setExtentsPerTurn(int e) {
    int old=extentsPerTurn;
    extentsPerTurn=e;
    /* We tell the world that this property has changed */
    firePropertyChange("extentsPerTurn",old,e);
}

public int getExtentsPerTurn() {
    return extentsPerTurn;
}
```



# La vue

---

- nous devons maintenant gérer la vue en Look and Feel
- commençons par définir un type abstrait, qui devra être étendu par toutes les implémentations:

```
public abstract class RadioSliderUI extends ComponentUI {  
    /* Abstract ancestor of all RadioSliderUIs */  
}
```



# La vue

---

- créons ensuite une implémentation correspondant au style "Metal":

```
public class MetalRadioSliderUI extends RadioSliderUI {  
}
```

- et indiquons au **JRadioSlider** comment mettre à jour son UI:

```
@Override public void updateUI() {  
    setUI (UIManager.getUI(this));  
}
```



# La vue

- la méthode `updateUI()` doit être redéfinie dans chaque objet vue, car `UIManager.getUI()` doit connaître le type de la vue, ce qu'elle obtient en invoquant `getClass()` sur son paramètre:

```
@Override public void updateUI() {  
    setUI(UIManager.getUI(this));  
}
```

permet de savoir qu'on est dans un `JRadioSlider`



# La vue

---

- la méthode `updateUI ()` n'a pas à se préoccuper de l'existence de l'UI, c'est l'`UIManager` qui gère:

```
UIDefaults.getUI() failed: no ComponentUI class for:  
fr.uml.v.ig.lesson9.JRadioSlider[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0  
.0,border=,flags=0,maximumSize=,minimumSize=,preferredSize=]  
java.lang.Error  
    at javax.swing.UIDefaults.getUIError(UIDefaults.java:711)  
    at javax.swing.MultiUIDefaults.getUIError(MultiUIDefaults.java:133)  
    at javax.swing.UIDefaults.getUI(UIDefaults.java:741)  
    at javax.swing.UIManager.getUI(UIManager.java:1016)  
    at fr.uml.v.ig.lesson9.JRadioSlider.updateUI(JRadioSlider.java:131)  
    at fr.uml.v.ig.lesson9.JRadioSlider.setModel(JRadioSlider.java:36)  
    at fr.uml.v.ig.lesson9.JRadioSlider.<init>(JRadioSlider.java:21)  
    at fr.uml.v.ig.lesson9.JRadioSliderDemo.main(JRadioSliderDemo.java:17)
```



# La vue

---

- nous devons donc créer un Look and Feel incluant notre UI, qu'il faudra ensuite installer au début de toute application utilisant un **JRadioSlider**:

```
public class MyMetalLnF extends MetalLookAndFeel {  
  
    @Override  
    public UIDefaults getDefaults() {  
        UIDefaults defaults=super.getDefaults();  
        defaults.put("RadioSliderUI", "fr.umlv.ig.lesson9.MetalRadioSliderUI");  
        return defaults;  
    }  
  
}
```



# La vue

---

- si on essaie maintenant de créer un **JRadioSlider**, on obtient toujours la même exception:

```
UIDefaults.getUI() failed: no ComponentUI class for:  
fr.umlv.ig.lesson9.JRadioSlider[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0  
.0,border=,flags=0,maximumSize=,minimumSize=,preferredSize=]
```

- pourquoi ?
- parce que l'**UIManager** se sait pas qu'à un **JRadioSlider** correspond un **RadioSliderUI**



# La vue

---

- on résout le problème en redéfinissant la méthode `getUIClassID` de `JRadioSlider`:

```
@Override
public String getUIClassID() {
    return "RadioSliderUI";
}
```

- on obtient alors une nouvelle exception:

```
UIDefaults.getUI() failed: createUI() failed for
fr.umlv.ig.lesson9.JRadioSlider[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0
.0,border=,flags=0,maximumSize=,minimumSize=,preferredSize=]
java.lang.reflect.InvocationTargetException
```



# La vue

---

- pourquoi ?
- parce qu'il manque la méthode `createUI`:  
à notre `MetalRadioSliderUI`:

```
public static ComponentUI createUI(JComponent c) {  
    return new MetalRadioSliderUI(c);  
}
```

- il est maintenant temps de s'occuper du rendu



# Le MetalRadioSliderUI

---

- commençons par créer un peintre qui reprenne les couleurs des boutons "Metal":

```
private Paint getBackgroundPaint(int w,int r,int radius) {
    java.util.List<?> gradient=(java.util.List<?>)UIManager.get("Button.gradient");
    if (gradient==null) {
        return Color.WHITE;
    } else {
        return new RadialGradientPaint(w/2,h/2,radius,new float[]{0.5f,.98f,1f},
            new Color[]{(Color) gradient.get(4),(Color) gradient.get(2),
                (Color) gradient.get(3)});
    }
}
```



# Le MetalRadioSliderUI

- on peut maintenant dessiner le fond du bouton:

```
@Override
public void paint(Graphics g, JComponent c) {
    Graphics2D g2=(Graphics2D)g.create();
    try {
        int w=c.getWidth();
        int h=c.getHeight();
        int radius=(w<h)?w/2-5:h/2-5;
        if (radius<1) {
            /* radius must be greater than 0 */
            radius=1;
        }
        g2.setPaint(getBackgroundPaint(w,h,radius));
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        g2.fillOval(w/2-radius,h/2-radius,radius*2,radius*2);
        g2.setColor(MetalLookAndFeel.getFocusColor());
        g2.drawOval(w/2-radius,h/2-radius,radius*2,radius*2);
    } finally {
        g2.dispose();
    }
}
```





# Le MetalRadioSliderUI

---

- pour positionner le point de saisie, il faut calculer son angle:

```
double valueToAngle(int v) {  
    int maxPerTurn=slider.getExtent()*slider.getExtentsPerTurn();  
    int value=v%maxPerTurn;  
    return value*2*Math.PI/maxPerTurn;  
}
```

- mais, il faut tenir compte de l'orientation du composant:
  - symétrie axiale si on est en `ComponentOrientation.RIGHT_TO_LEFT`



# Le MetalRadioSliderUI

---

- on va créer une transformation qui rend compte de l'angle et de la symétrie, si besoin est:

```
AffineTransform createTransform(int w, int h) {  
    AffineTransform t=AffineTransform.getTranslateInstance(w/2,h/2);  
    if (slider.getComponentOrientation()==ComponentOrientation.RIGHT_TO_LEFT) {  
        t.scale(-1,1);  
    }  
    angle=valueToAngle(slider.getValue());  
    t.rotate(angle);  
    return t;  
}
```



# Le MetalRadioSliderUI

- il faut ensuite utiliser cette transformation pour dessiner le point de saisie, avec des couleurs différentes pour indiquer le mouvement:

```
@Override
public void paint(Graphics g, JComponent c) {
    ...
    if (slider.getValueIsAdjusting()) {
        g2.setPaint(MetalLookAndFeel.getAcceleratorSelectedForeground());
    } else {
        g2.setPaint(MetalLookAndFeel.getAcceleratorForeground());
    }
    int smallRadius=radius/6;
    g2.transform(createTransform(w,h));
    g2.fillOval(radius-3*smallRadius,-smallRadius,2*smallRadius,2*smallRadius);
    ...
}
```



# Le MetalRadioSliderUI

- il faut se préoccuper du rafraîchissement en plaçant des listeners sur la vue et le modèle
- c'est le travail de `installUI`:

```
final ChangeListener changeListener=new ChangeListener() {
    @Override public void stateChanged(ChangeEvent e) {
        slider.paintImmediately(0,0,slider.getWidth(),slider.getHeight());
    }
};

final PropertyChangeListener orientationListener=new PropertyChangeListener() {
    @Override public void propertyChange(PropertyChangeEvent evt) {
        slider.paintImmediately(0,0,slider.getWidth(),slider.getHeight());
    }
};

@Override public void installUI(JComponent c) {
    slider.getModel().addChangeListener(changeListener);
    slider.addPropertyChangeListener("componentOrientation",orientationListener);
}
```



# Le MetalRadioSliderUI

---

- on stocke les listeners, car il faudra les retirer proprement dans **uninstallUI**:

```
@Override
public void uninstallUI(JComponent c) {
    slider.getModel().removeChangeListener(changeListener);
    slider.removePropertyChangeListener("componentOrientation", orientationListener);
}
```



# Le MetalRadioSliderUI

- on peut maintenant tester notre vue en utilisant un **BoundedRangeModel** commun avec un **JSlider**:

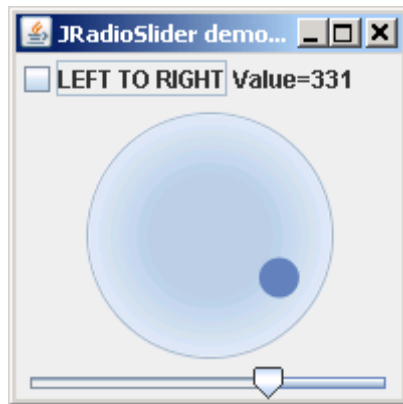
```
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(new MyMetalLnF());
    } catch (UnsupportedLookAndFeelException e) {
        e.printStackTrace();
    }
    JFrame frame = new JFrame("JRadioSlider demonstration");
    final BoundedRangeModel model=new DefaultBoundedRangeModel(30,5,0,1000);
    final JRadioSlider radioSlider = new JRadioSlider(model,100);
    frame.getContentPane().add(radioSlider);
    final JSlider slider = new JSlider(model);
    frame.getContentPane().add(slider, BorderLayout.SOUTH);
    ...
}
```



# Le MetalRadioSliderUI



redimensionnement  
du composant



orientation du  
composant



la phase d'ajustement est  
caractérisée par le point noir



# Une vue active

---

- on veut maintenant rendre la vue manipulable à la souris
- on met un **MouseListener** pour savoir quand l'utilisateur clique sans relâcher sur le point de saisie
- quand c'est le cas, on installe un **MouseMotionListener** qui va gérer le déplacement de la souris
- ces listeners seront gérés par **installUI** et **uninstallUI**



# Une vue active

- voici le **MouseListener**:

```
final MouseAdapter mouseManager=new MouseAdapter() {
    @Override public void mousePressed(MouseEvent e) {
        int w=slider.getWidth();
        int h=slider.getHeight();
        AffineTransform t=createTransform(w,h);
        int radius = (w < h) ? w / 2 - 5 : h / 2 - 5;
        int smallRadius=radius/6;
        Ellipse2D point=new Ellipse2D.Float(radius-3*smallRadius,-smallRadius,
                                           2*smallRadius,2*smallRadius);
        Shape tmp=t.createTransformedShape(point);
        if (tmp.contains(e.getX(),e.getY())) {
            slider.getModel().setValueIsAdjusting(true);
            slider.addMouseMotionListener(mouseMotionManager);
        }
    }

    @Override public void mouseReleased(MouseEvent e) {
        slider.getModel().setValueIsAdjusting(false);
        slider.removeMouseMotionListener(mouseMotionManager);
    }
};
```



# Une vue active

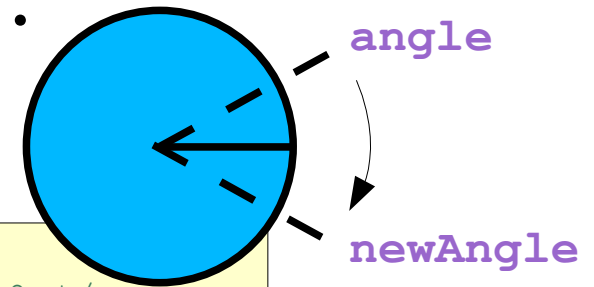
- et voici le **MouseEventListener**:

```
final MotionEventAdapter mouseMotionManager=new MotionEventAdapter() {
    @Override public void mouseDragged(MouseEvent e) {
        int x2=slider.getWidth()/2;
        int y2=slider.getHeight()/2;
        int dx=e.getX()-x2;
        int dy=e.getY()-y2;
        if (slider.getComponentOrientation()==ComponentOrientation.RIGHT_TO_LEFT) {
            dx=-dx;
        }
        double newAngle=Math.atan2(dy,dx);
        /* We want newAngle to be in [0;2.PI] */
        if (newAngle<0) newAngle=newAngle+2*Math.PI;
        int maxPerTurn=slider.getExtent()*slider.getExtentsPerTurn();
        int value=slider.getValue();
        int currentBaseValue=value-value%maxPerTurn;
        /* Case 1: increasing value */
        if (newAngle>angle && newAngle-angle<Math.PI) {
            int newValue=currentBaseValue+angleToValue(newAngle);
            angle=newAngle;
            slider.setValue(newValue);
            return;
        }
        ...
    }
};
```



# Une vue active

- attention au cas où le pointeur de souris a franchi la limite d'un tour:



```
...
/* Case 2: increasing value, but angle<2.PI and newAngle>0 */
if (newAngle<Math.PI/4 && angle>1.5*Math.PI) {
    int newValue=currentBaseValue+maxPerTurn+angleToValue(newAngle);
    if (newValue+slider.getExtent()>slider.getMaximum()) {
        /* The BoundedRangeModel does not support a value does not
        * respect the constraint value+extent <= maximum */
        newValue=slider.getMaximum()-slider.getExtent();
        /* We adapt the angle to the corrected value */
        newAngle=valueToAngle(newValue);
    }
    angle=newAngle;
    slider.setValue(newValue);
    return;
}
...
```



# La bordure

---

- un composant Swing poli doit respecter sa bordure
- or, ce n'est pas le cas:





# La bordure

- il suffit de gérer les marges:

```
@Override
public void paint(Graphics g, JComponent c) {
    Graphics2D g2 = (Graphics2D) g.create();
    try {
        Insets insets=c.getInsets();
        int w = c.getWidth()-insets.left-insets.right;
        int h = c.getHeight()-insets.bottom-insets.top;
        /* We adjust the axis on the real drawing area */
        g2.translate(insets.left,insets.top);
    }
    ...
}
```



évite de devoir systématiquement ajouter `insets.left` et `insets.top` à toutes les coordonnées



# Actif/inactif

- quand le composant n'est pas actif, il faut bloquer les événements souris:

```
final PropertyChangeListener enableListener=new PropertyChangeListener() {
    @Override public void propertyChange(PropertyChangeEvent evt) {
        if (slider.isEnabled()) {
            slider.addMouseListener(mouseManager);
        } else {
            slider.removeMouseListener(mouseManager);
        }
        slider.paintImmediately(0,0,slider.getWidth(),slider.getHeight());
    }
};

@Override public void installUI(JComponent c) {
    slider.getModel().addChangeListener(changeListener);
    slider.addPropertyChangeListener("componentOrientation",orientationListener);
    slider.addPropertyChangeListener("enabled",enableListener);
    slider.addMouseListener(mouseManager);
}
```

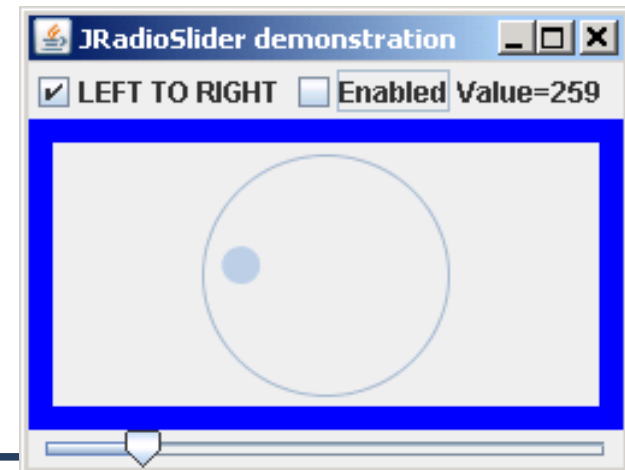


# Actif/inactif

- il est recommandé de changer de rendu, en choisissant des couleurs particulières quand le composant est inactif:

```
if (!slider.isEnabled()) {  
    g2.setPaint(MetalLookAndFeel.getControlDisabled());  
} else if (slider.getValueIsAdjusting()) {  
    g2.setPaint(MetalLookAndFeel.getAcceleratorSelectedForeground());  
} else {  
    g2.setPaint(MetalLookAndFeel.getAcceleratorForeground());  
}
```

```
private Paint getBackgroundPaint(int w, int h, int radius) {  
    if (!slider.isEnabled()) {  
        return slider.getBackground();  
    }  
    ...  
}
```





# contains

- si le composant est transparent, il faut que **contains** tienne compte de la forme réelle du composant:

```
@Override public boolean contains(JComponent c, int x, int y) {
    if (c.isOpaque()) {
        return super.contains(c, x, y);
    }
    /* If the component has no background paint, contains
     * must return true only if (x,y) is in the circle area */
    Insets insets=c.getInsets();
    int w = c.getWidth()-insets.left-insets.right;
    int h = c.getHeight()-insets.bottom-insets.top;
    int radius = (w < h) ? w / 2 - 5 : h / 2 - 5;
    if (radius<1) {
        /* radius must be greater than 0 */
        radius=1;
    }
    x=x-insets.left-w/2;
    y=y-insets.top-h/2;
    return x*x+y*y<radius*radius;
}
```



# Tada!!

- et voici notre composant fin prêt, avec un bouton cliquable en arrière-plan:

