Look and Feel

Rémi Forax

Plan

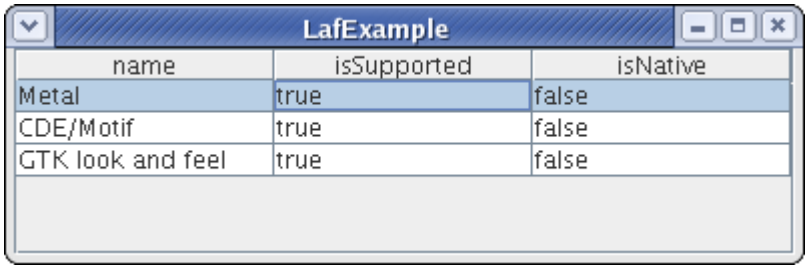
- Exemple Look and Feel
- Créer son L&F
- Metal & Thème
- Skin & Synth

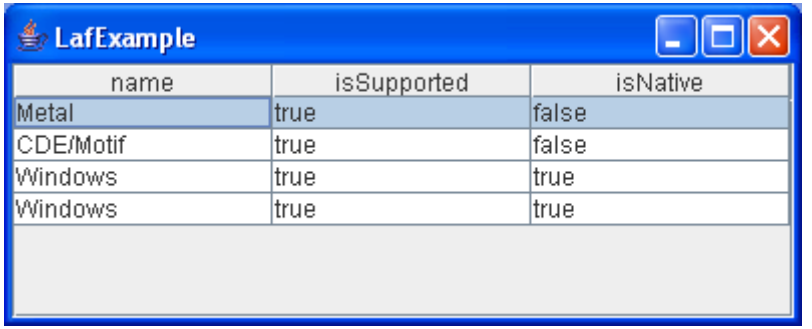
Look & Feel

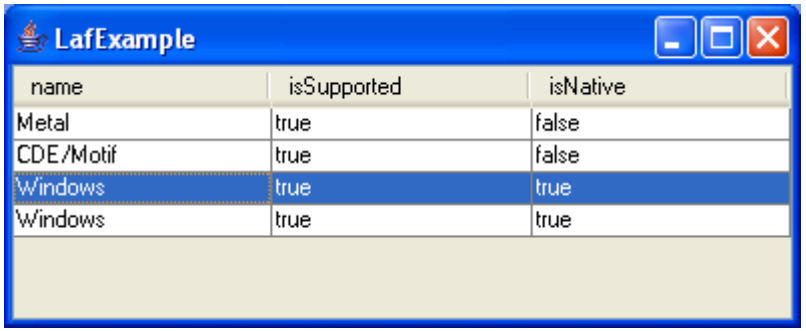
- Swing possède un mécanisme de Look & Feel qui fait qu'un composant n'est pas responsable de son rendu graphique
- Ceci permet
  - de choisir le look d'une application en fonction :
    - de la plateforme
    - d'un look personnel pour une suite d'application
  - De changer dynamiquement de look
- Mais dure de créer un L&F
  - Beaucoup de code à écrire (1 classe par composant)
  - Dure de débugger un L&F,
    aller-retour incessant entre le L&F et le composant

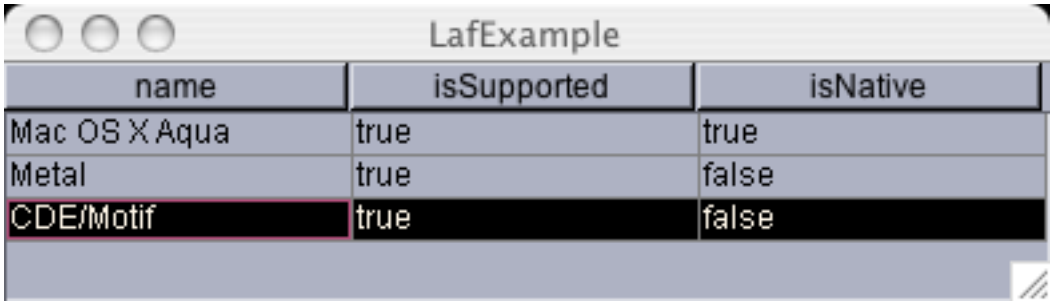
Exemple de différents L&F

- Différents Look and Feel









UIManager

- La classe UIManager gère les différents L&F plus un L&F courant
- Rechercher les L&F présent :
  - LookAndFeelInfo[] getInstalledLookAndFeels()

  LookAndFeelInfo correspond à un nom et une classe Java
- Installer un L&F
  - installLookAndFeel(String name, String className)
  - installLookAndFeel(LookAndFeelInfo info)

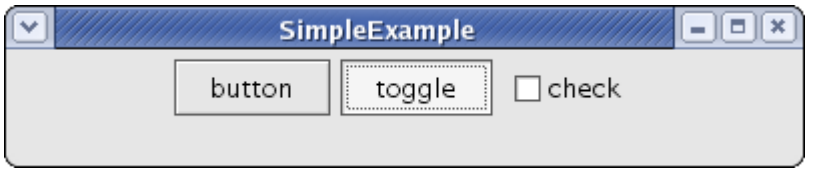
Changer de L&F

- Par défaut, le L&F utilisé est métal
- Si l'on souhaite changer le look au démarrage de l'application, il faut le faire avant l'appel à aux constructeurs des composants
  UIManager.setLookAndFeel(String className)
  UIManager.setLookAndFeel(LookAndFeel laf)
- Si l'on souhaite changer dynamiquement de L&F, il faut un recalcul pour chaque composant
  SwingUtilities.updateComponentTreeUI(frame);

Changer le L&F

- Changer le L&F avant la construction de l'interface graphique
  - **Windows** **com.sun.java.swing.plaf.windows.WindowsLookAndFeel**
  - **Motif** **com.sun.java.swing.plaf.motif.MotifLookAndFeel**
  - **Metal** **javax.swing.plaf.metal.MetalLookAndFeel**
  - **GTK** **com.sun.java.swing.plaf.gtk.GTKLookAndFeel**

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.gtk.GTKLookAndFeel");

JPanel panel=new JPanel();
panel.add(new JButton("button"));
panel.add(new JToggleButton("toggle"));
panel.add(new JCheckBox("check"));
```

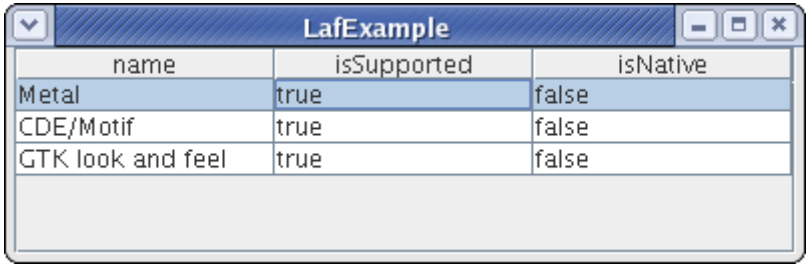


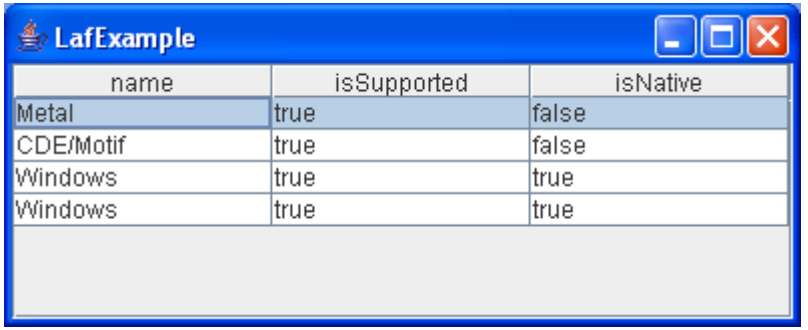
X-Plateforme et L&F natif

- UIManager possède des méthodes permettant d'obtenir
  - Le L&F Courant :
    - getLookAndFeel()
  - Le L&F de la plateforme
    - getSystemLookAndFeelClassName()
  - Le L&F indépendant de la plateforme (métal)
    - getCrossPlatformLookAndFeelClassName()

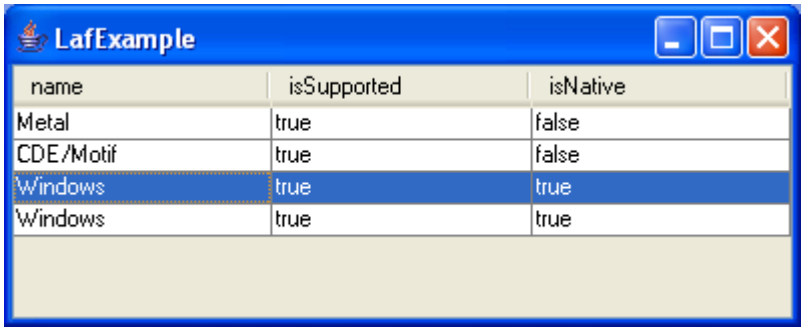
X-Plateforme et L&F natif (2)

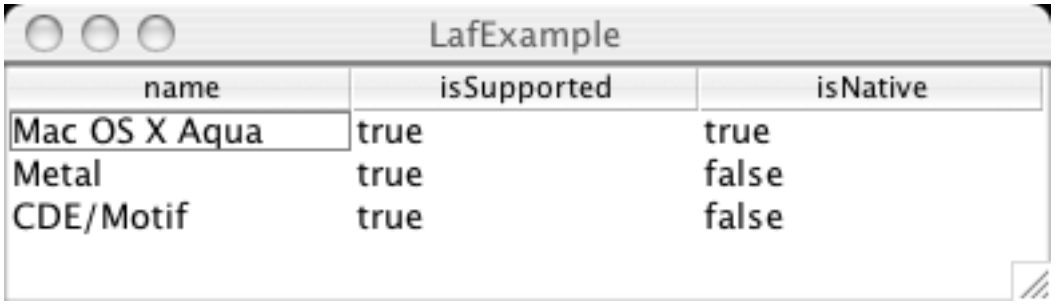
- Cross Plateform Look and Feel





- Look and Feel de la plateforme
  (ne sont supportés que sur 1 plateforme GTK/Windows,/Aqua)





Exemple de L&F

- On utilise un modèle de table à partir d'une liste de LookAndFeel
- Il est possible d'éviter le switch !! (code pas top objet)

```
public class LookAndFeelTableModel extends AbstractTableModel {
  private final ArrayList<LookAndFeel> lafs;
  public LookAndFeelTableModel(ArrayList<LookAndFeel> lafs) {
    this.lafs=lafs;
  }
  public int getRowCount() {
    return lafs.size();
  }
  public int getColumnCount() {
    return COLUMNS.length;
  }
  public String getColumnName(int column) {
    return COLUMNS[column].name();
  }
  public Object getValueAt(int row, int column) {
    LookAndFeel laf=lafs.get(row);
    switch(column) {
      ...
    }
    ...
  }
}
```

Exemple de L&F (2)

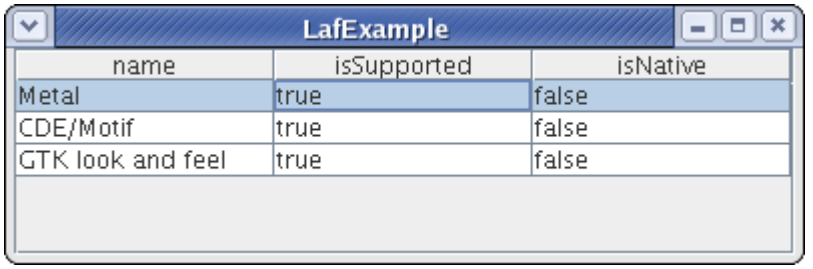
- Remplacer par un appel polymorphe sur la colonne
- On utilise un enum pour calculer valeur de la cellule

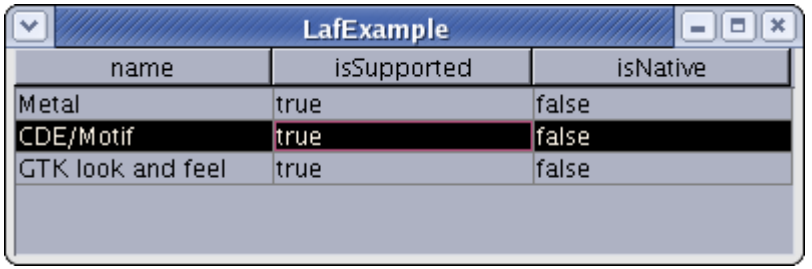
```
public class LookAndFeelTableModel extends AbstractTableModel {
  ...
  public Object getValueAt(int row, int column) {
    return COLUMNS[column].getValue(lafs.get(row));
  }
  static final Column[] COLUMNS=Column.values();
  private enum Column {
    name {
      String getValue(LookAndFeel laf) {
       return laf.getName();
      }
    }, isSupported {
      Boolean getValue(LookAndFeel laf) {
        return laf.isSupportedLookAndFeel();
      }
    }, isNative {
      Boolean getValue(LookAndFeel laf) {
        return laf.isNativeLookAndFeel();
      }
    };

    abstract Object getValue(LookAndFeel laf);
  }
```

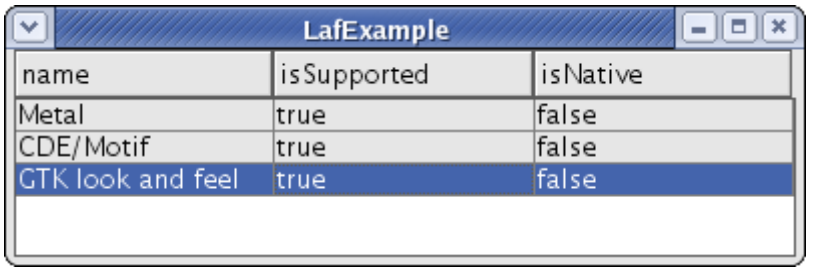
Exemple de L&F (2)

```
private static LookAndFeel loadLaf(String className)
  throws InstantiationException, IllegalAccessException, ClassNotFoundException {
  Class<LookAndFeel> clazz=(Class<LookAndFeel>)Class.forName(className);
  return clazz.newInstance();
}
public static void main(String[] args) throws Throwable {
  LookAndFeelInfo[] infos=UIManager.getInstalledLookAndFeels();
  final ArrayList<LookAndFeel> lafs=new ArrayList<LookAndFeel>(infos.length);
  for(LookAndFeelInfo info:infos)
      lafs.add(loadLaf(info.getClassName()));
  TableModel model=new LookAndFeelTableModel(lafs);
  final JFrame frame=new JFrame("LafExample");
  final JTable table=new JTable(model);
  table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
  table.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent event) {
      final int row=table.getSelectedRow();
      if (row==-1)
        return;
      EventQueue.invokeLater(new Runnable() { // obligatoire car bug !!
        public void run() {
          try {
            UIManager.setLookAndFeel(lafs.get(row));
          } catch(UnsupportedLookAndFeelException e) {
          }
          SwingUtilities.updateComponentTreeUI(frame);
        }
      });
    }
  });
```

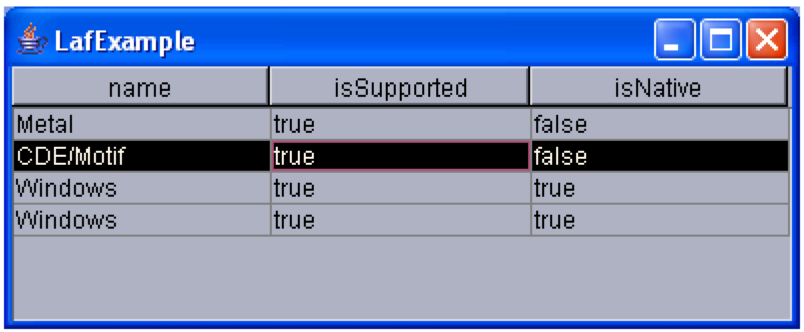
L&F sous linux

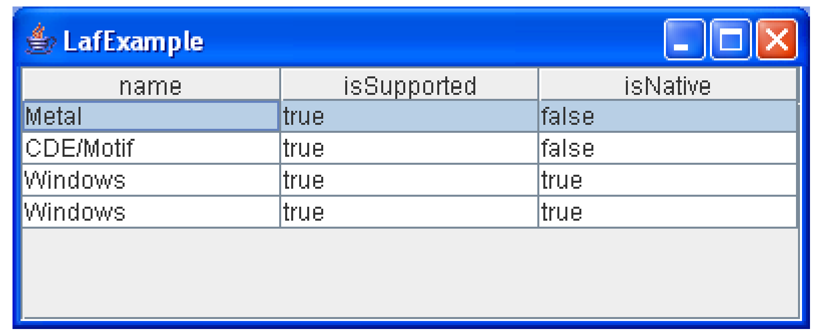


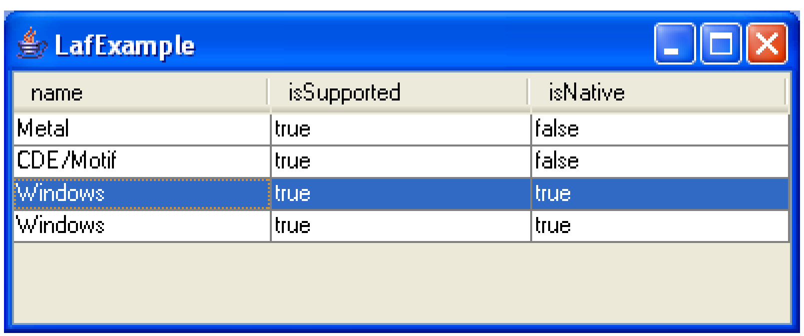


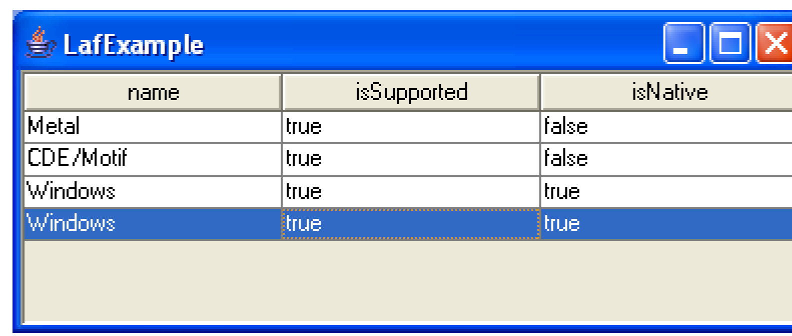


L&F sous windows

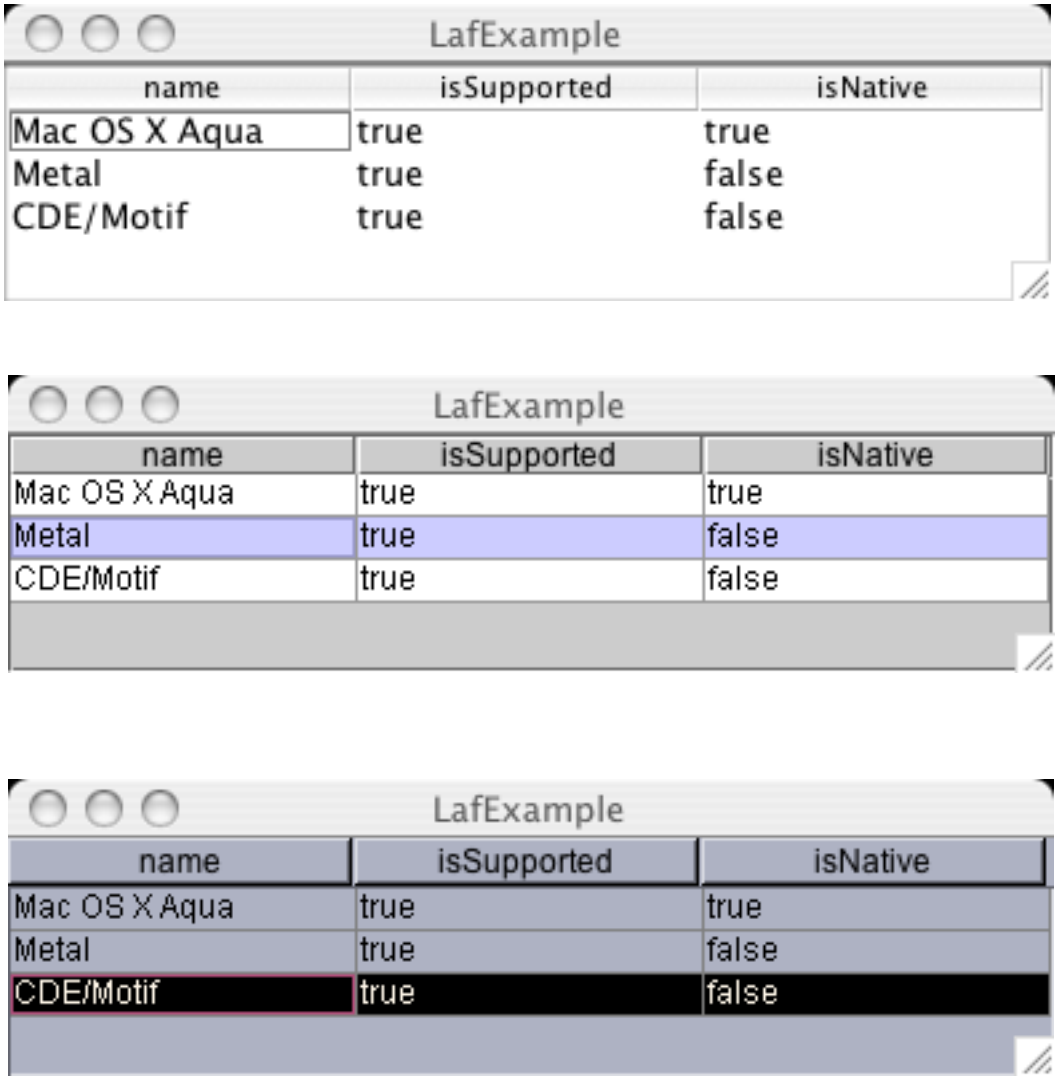








L&F sous MacOS



ComponentUI (1)

- Le rendu graphique d'un JComponent est délégué à un **javax.swing.plaf.ComponentUI**
- Il existe un CompoentUI par type de JComponent
  - ButtonUI pour un JButton
  - TreeUI pour un JTree, etc.
- Il y a deux façon d'installer un ComponentUI
  - localement sur un JComponent avec **get/setUI(ComponentUI)**
  - de façon global en indiquant au L&F quel ComponentUI créer pour un JComponent

ComponentUI (2)

- Les méthodes de ComponentUI
  - creation par *réflexion*, appel la *factory* :
    - **static ComponentUI createUI(JComponent c)**
  - install/uninstall pour un composant
    - installUI(JComponent)
    - uninstallUI(JComponent)

    peut ne rien faire
- Permet de choisir si l'on veut un ComponentUI pour plusieurs JComponent ou pas

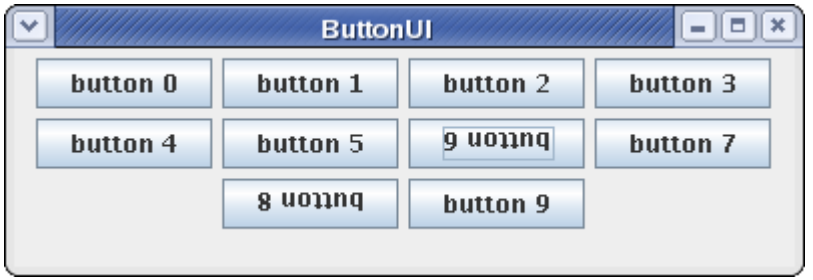
ComponentUI (3)

- Les méthodes de ComponentUI (suite)
  - Taille du composant
    - **getMaximum/minimum/preferredSize(JComponent c)**
    - **null** correspond à laisser le gestionnaire de géométrie faire
  - Dessin
    - dessin de l'arrière plan
      - **update(Graphics g, JComponent c)**
        doit appeler **paint**(g,c)
    - dessin de l'avant plan
      - **paint**(Graphics g, JComponent c)

Exemple de ButtonUI

- exemple sans gestion de la création :
  changer le l'avant plan d'un bouton, en fait le texte ici.

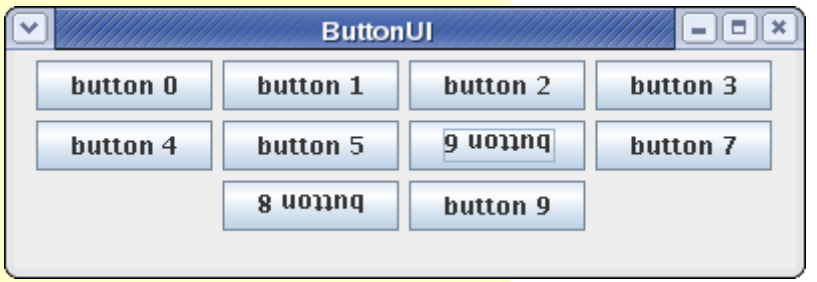
```
public class MyButtonUI extends MetalButtonUI {
  public @Override void paint(Graphics g, JComponent c) {
    Graphics2D g2=(Graphics2D)g.create();
    int w=c.getWidth();
    int h=c.getHeight();
    g2.translate(w/2,h/2);
    g2.scale(-1, -1);
    g2.translate(-w/2,-h/2);
    super.paint(g2, c);
    g2.dispose();
  }
  ...
```



Exemple de ButtonUI (2)

- Lorsque l'on clique sur un bouton, celui-ci change d'objet responsable de son look

```
public static void main(String[] args) {
  ActionListener listener=new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      JButton button=(JButton)e.getSource();
      if (button.getUI() instanceof MyButtonUI)
        button.setUI(new MetalButtonUI());
      else
        button.setUI(new MyButtonUI());
    }
  };

  JPanel panel=new JPanel();
  for(int i=0;i<10;i++) {
    JButton button=new JButton("button "+i);
    button.addActionListener(listener);
    panel.add(button);
  }
  ...
```



Créer son propre LookAndFeel

- Lors de la création d'un JComponent, celui-ci demande à l'**UIManager** le ComponentUI correspondant à lui-même

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

- L'**UIManager** délègue l'appel **getUI()** à l'objet **UIDefaults** du **LookAndFeel** courant
- L'**UIDefaults** demande au composant quel est son **UIClassID** (**getUIClassID()**) et cherche dans sa table de hachage quelle est le nom de la classe correspondant
- La classe est chargée en mémoire et la méthode **createUI()** est appelée par *réflexion*

Créer son propre LookAndFeel

- Créer son propre L&F consiste à remplir dans **UIDefaults** la table de hachage en indiquant pour chaque **UIClassID** quelle est la classe dérivant de **ComponentUI** correspondante
- Pour aider, le paquetage :
  - **javax.swing.plaf.basic** contient une implantation abstraite de ces classes
  - **javax.swing.plag.multi** contient une implantation qui permet de multiplexer les L&F
  - **javax.swing.plaf.metal** contient le L&F Metal

Exemple simple

- En créant un L&F dérivant de metal, ici, tous les **JButton** se partage le même **MirorButtonUI**

```
public class MirrorLookAndFeel extends MetalLookAndFeel {
  public UIDefaults getDefaults() {
    UIDefaults uiDefaults=super.getDefaults();
    uiDefaults.put("ButtonUI", "fr.umlv.ui.plaf.MirrorButtonUI");
    return uiDefaults;
  }
}
public class MirrorButtonUI extends MetalButtonUI {
  public @Override void paint(Graphics g, JComponent c) {
    Graphics2D g2=(Graphics2D)g.create();
    int w=c.getWidth();
    int h=c.getHeight();
    g2.translate(w/2,h/2);
    g2.scale(-1, -1);
    g2.translate(-w/2,-h/2);
    super.paint(g2, c);
    g2.dispose();
  }
  public static ComponentUI createUI(JComponent c) {
    return mirorButtonUI;
  }
  private static final MirrorButtonUI mirorButtonUI=
    new MirrorButtonUI();
}
```

Metal & Thème

- Le L&F Métal possède des thèmes, utilisant la classe abstraite MetalTheme, qui permettent de personnaliser les couleurs et les fonts
- Le thème par défaut est :
  - steel pour JDK <1.5
    DefaultMetalTheme hérite de MetalTheme
  - ocean pour JDK 1.5
    OceanMetalTheme hérite de DefaultMetalTheme

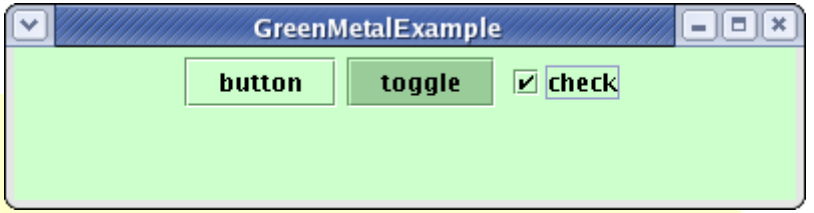
Océan

Steel

Changer le thème

- Obtenir le thème courant :
  - static MetalTheme MetalLookAndFeel.getCurrentTheme()
- Changer le thème
  - static void MetalLookAndFeel.setCurrentTheme(
    MetalTheme theme)
  - par ligne de commande
    java -Dswing.metalTheme=steel MaClasse
    java -Dswing.metalTheme=ocean MaClasse

Créer son propre thème



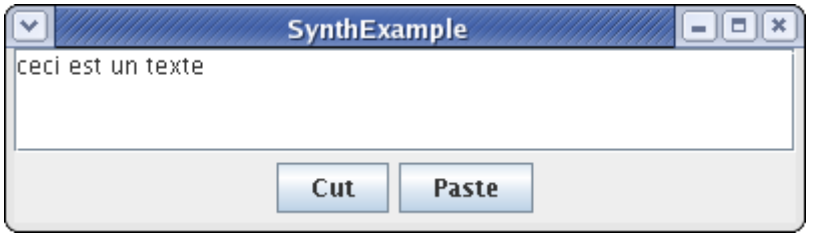

```
public class GreenMetalTheme extends DefaultMetalTheme
  public String getName() {
    return "Green";
  }
  private final ColorUIResource primary1=new ColorUIResource(102, 102, 153);
  private final ColorUIResource primary2=new ColorUIResource(153,153, 204);
  private final ColorUIResource primary3=new ColorUIResource(204, 204, 255);
  private final ColorUIResource secondary1=new ColorUIResource(102, 153, 102);
  private final ColorUIResource secondary2=new ColorUIResource(153, 204, 153);
  private final ColorUIResource secondary3=new ColorUIResource(204, 255, 204);

  protected ColorUIResource getPrimary1() { return primary1; }
  protected ColorUIResource getPrimary2() { return primary2; }
  protected ColorUIResource getPrimary3() { return primary3; }

  protected ColorUIResource getSecondary1() { return secondary1; }
  protected ColorUIResource getSecondary2() { return secondary2; }
  protected ColorUIResource getSecondary3() { return secondary3; }

  public static void main(String[] args) throws UnsupportedLookAndFeelException {
    MetalLookAndFeel laf=new MetalLookAndFeel();
    MetalLookAndFeel.setCurrentTheme(new GreenMetalTheme());
    UIManager.setLookAndFeel(laf);
```

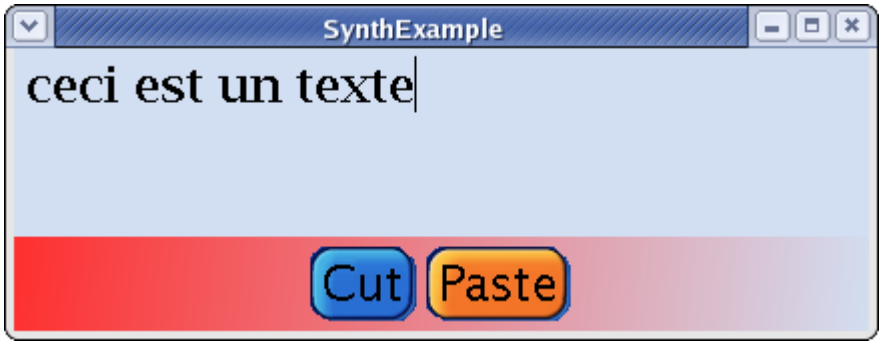
Le L&F Synth

- Permet d'indiquer par l'intermédiaire d'un fichier XML le L&F de l'application
- Il est possible de :
  - changer les couleurs
  - changer les fontes
  - mapper des images sur des composants graphiques
  - indiquer un rendu spécifique (painter)
- Synth sert de base pour les looks GTK et Windows XP
- Synth se trouve dans le paquetage **javax.swing.plaf.synth**

Skin XML

- Synth utilise un fichier XML pour specifier la 'peau' (skin) de l'application
- Chargement du fichier XML,
  on utilise la méthode **synthLookAndFeel.load()**

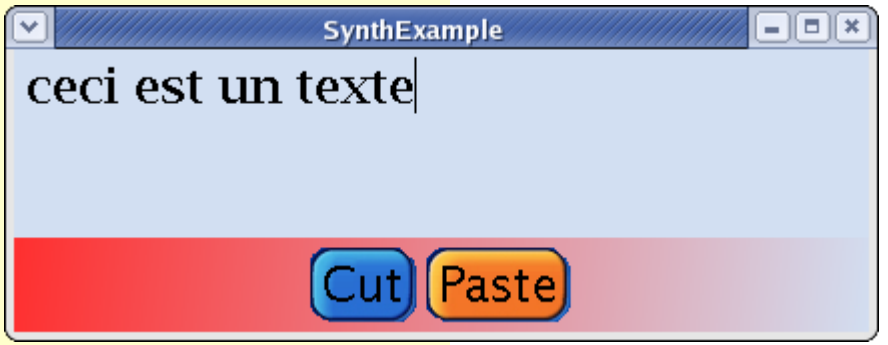




Un exemple simple

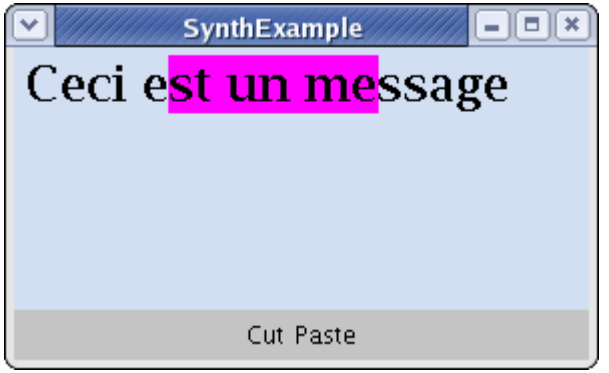
- Exemple d'une interface simple

```
SynthLookAndFeel synth = new SynthLookAndFeel();
synth.load(SynthExample.class.
    getResourceAsStream("simple.xml"), SynthExample.class);
UIManager.setLookAndFeel(synth);
final JTextArea area=new  JTextArea();
Action cut=new AbstractAction("Cut") {
  public void actionPerformed(ActionEvent e) {
    area.cut();
  }
};
Action paste=new AbstractAction("Paste") {
  public void actionPerformed(ActionEvent e) {
    area.paste();
  }
};

JPanel panel=new JPanel();
panel.add(new JButton(cut));
panel.add(new JButton(paste));

JFrame frame=new JFrame("SynthExample");
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.getContentPane().add(new JScrollPane(area));
frame.getContentPane().add(panel,BorderLayout.SOUTH);
frame.setSize(400,300);
frame.setVisible(true);
```



Skin : Le format du XML

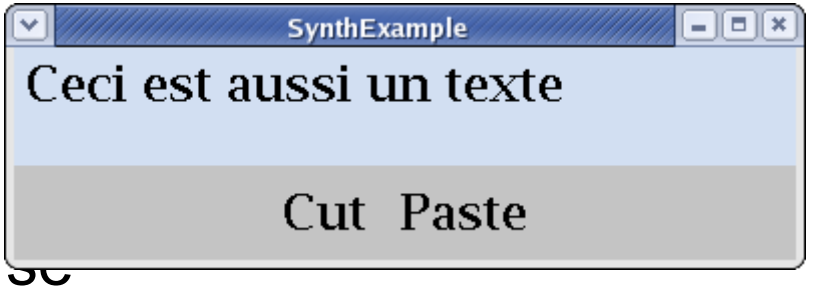
- Le fichier XML permet de :
  - définir des styles (états, fontes, insets)
  - d'associer (bind) un style à un type de composant




```
<?xml version="1.0" encoding="iso-8859-1"?>
<synth>
  <style id="text-style">
    <state>
       <color value="#D2DFF2" type="BACKGROUND"/>
       <color value="#000000" type="TEXT_FOREGROUND"/>
       <color value="#FF00FF" type="TEXT_BACKGROUND"/>
    </state>
    <font name="Serif" size="24" style="BOLD"/>
    <insets top="4" left="6" bottom="4" right="6"/>
  </style>
  <bind style="text-style" type="region" key="TextArea"/>
</synth>
```

Style et bindings

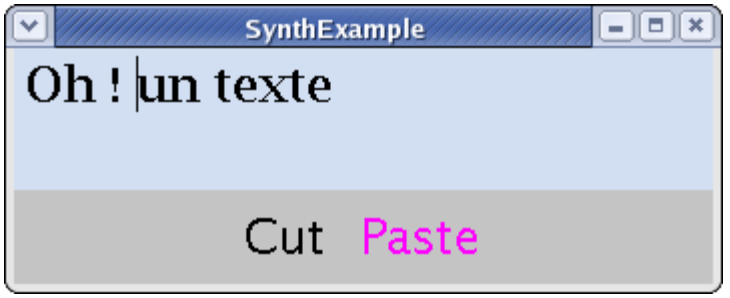
- Il est possible d'associer un même style à plusieurs types de composant
- Les boutons et la zone de texte se partage le même style




```
...
 <style id="text-style">
   <state>
      <color value="#D2DFF2" type="BACKGROUND"/>
      <color value="#000000" type="TEXT_FOREGROUND"/>
      <color value="#FF00FF" type="TEXT_BACKGROUND"/>
   </state>
   <font name="Serif" size="24" style="BOLD"/>
   <insets top="4" left="6" bottom="4" right="6"/>
 </style>
 <bind style="text-style" type="region" key="TextArea"/>
 <bind style="text-style" type="region" key="Button"/>
...
```

Skin et Etat

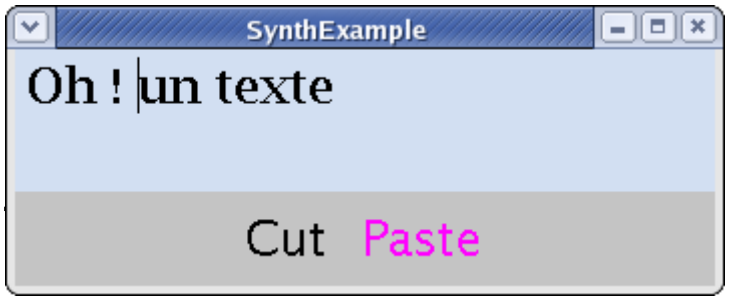
- Le format XML de Synth permet de décrire des couleurs ou des images différentes en fonction d'un état d'un composant




```
...
  <style id="button-style">
    <state>
       <color value="#000000" type="TEXT_FOREGROUND"/>
    </state>
    <state value="MOUSE_OVER">
       <color value="#FF00FF" type="TEXT_FOREGROUND"/>
    </state>
    <font name="SansSerif" size="24"/>
    <insets top="4" left="6" bottom="4" right="6"/>
  </style>
  <bind style="button-style" type="region" key="Button"/>
...
```

Etats possibles

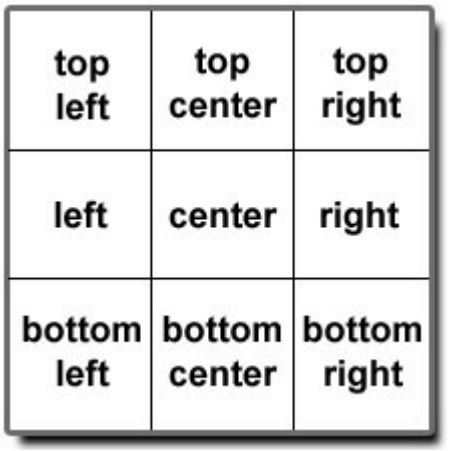
- Les états possibles sont regroupés dans la classe SynthConstants :
  DEFAULT, DISABLED, ENABLED,
  FOCUSED, MOUSE_OVER,
  PRESSED, SELECTED
- Tous les composants ne supporte pas
- Il est possible d'avoir plusieurs états en même temps




```
...
  <style id="button-style">
    <state value="DEFAULT">
       <color value="#000000" type="TEXT_FOREGROUND"/>
    </state>
    <state value="MOUSE_OVER">
       <color value="#FF00FF" type="TEXT_FOREGROUND"/>
    </state>
    <font name="SansSerif" size="24"/>
    <insets top="4" left="6" bottom="4" right="6"/>
  </style>
  <bind style="button-style" type="region" key="Button"/>
...
```

Style et Image

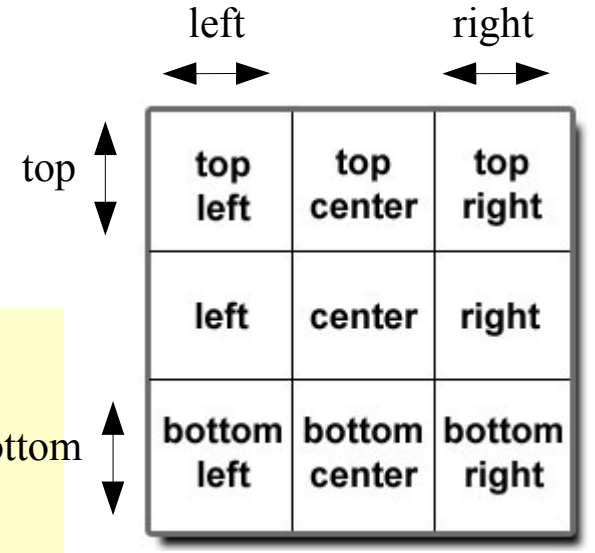
- Il est possible de plaquer une image sur des composants
- L'image est découpée en neuf régions :
  - top-left, top-right, bottom-left et bottom-right ne s'agrandisse pas
  - top-center et bottom-center s'agrandissent uniquement horizontalement
  - left et right s'agrandissent uniquement verticalement
  - center qui s'agrandit verticalement et horizontalement



- Ressemble à un BorderLayout

ImagePainter

- Pour plaquer l'image, on utilise un imagePainter en spécifiant les décalages (sourceInsets) top/left/bottom/right
- Image utilisée :




```
...
  <style id="button-style">
   <state>
     <imagePainter
       method="buttonBackground"
       path="button.png"
       sourceInsets="9 10 9 12"
       paintCenter="true"
       stretch="true"/>
      <color value="#D2DFF2" type="BACKGROUND"/>
      <color value="#000000" type="TEXT_FOREGROUND"/>
   </state>
   <font name="SansSerif" size="24"/>
   <insets top="4" left="6" bottom="4" right="6"/>
 </style>
 <bind style="button-style" type="region" key="Button"/>
...
```

Les paramétres de l'ImagePainter

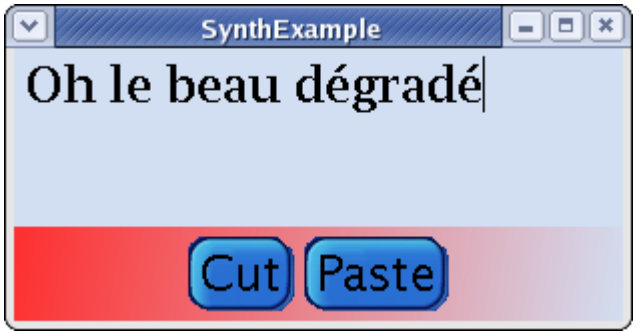
- Les paramétres :
  - **methode** correspond à la méthode en Java qui sera appelée
  - **path** au chemin par rapport à la classe spécifiée lors du load
  - **sourceInsets** déjà vu
  - **paintCenter**, doit on dessiner le centre ?
  - **strech**, doit on redimensionner ou tuiler ?


```
...
     <imagePainter
       method="buttonBackground"
       path="button.png"
       sourceInsets="9 10 9 12"
       paintCenter="false"
       stretch="true"/>
...
```

center=false

Définir son propre painter

- Synth permet au lieu d'utiliser un **ImagePainter** de définir son propre *painter* en héritant de la classe **SynthPainter**




```
public class GradientPainter extends SynthPainter {
   public void paintPanelBackground(SynthContext context,
                                    Graphics g, int x, int y,
                                    int w, int h) {

      Color start=context.getStyle().getColor(context, ColorType.FOREGROUND);
      Color end=context.getStyle().getColor(context, ColorType.BACKGROUND);

      Graphics2D g2 = (Graphics2D)g;
      GradientPaint grPaint = new GradientPaint(x, y,start,x+w, y+h,end);
      g2.setPaint(grPaint);
      g2.fillRect(x, y, w, h);
   }
}
```

Définir son propre painter (2)

- Dans le XML, la balise **objet** permet d'instancier le **GradientPainter** qui doit avoir un constructeur sans argument
- La balise **painter** possède un attribut **id-ref** pour faire référence à un *painter* particulier

```
<?xml version="1.0" encoding="iso-8859-1"?>
<synth>
  <object id="gradientPainter" class="fr.umlv.ui.plaf.GradientPainter"/>
  <style id="panel-style">
     <state>
       <color value="#FF3030" type="FOREGROUND"/>
       <color value="#D2DFF2" type="BACKGROUND"/>
    </state>
    <painter method="panelBackground" idref="gradientPainter"/>
  </style>
  <bind style="panel-style" type="region" key="Panel"/>
  ...
</synth>
```

SynthContext

- L'objet **SynthContext** contient le context courant d'exécution
- Il possède les méthodes :
  - le composant à afficher
    - JComponent getComponent()
  - les états du composant (constante de SynthConstant)
    - int getComponentState()
  - la région (Button, TextArea etc.)
    - Region getRegion()
  - le style associé
    - SynthStyle getStyle()

SynthStyle

- objet reflétant les valeurs définies dans le XML
  - renvoie la couleur en fonction du type (BACKGROUNG, FOREGROUND...)
    - Color getColor(SynthContext context, ColorType type)
  - renvoie la fonte, l'icone, les insets ou le painter
    - Font getFont(SynthContext context)
    - Icon getIcon(SynthContext context, Object key)
    - Insets getInsets(SynthContext context, Insets insets)
    - SynthPainter getPainter(SynthContext context)
- Toutes les méthodes prennent un contexte mais seule getColor() et getFont() utilise l'état du context