Menus et barre d'outils

- Les menus
- Les menus contextuels
- Les barres d'outils
- Les actions

Les Menus

- **`JMenu`** est un menu a des entrées qui sont **`JMenuItem`**, **`JCheckBoxMenuItem`**, **`JRadioButtonMenuItem`**, **`JSeparator`**, et **`Jmenu`**.

- **`JMenubar`** sert de barre de stockage pour les menus Utilise un **`BoxLayout`** vertical. Ses composants peuvent donc être espacés ou groupés.

- On utilise la méthode **`setJMenuBar()`** pour installer le menu sur la fenêtre.

Exemple de Menu

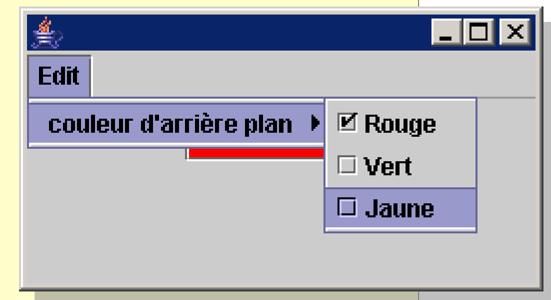
- Exemple de menu avec sous-menu.

```
public static void main(String[] args) {
  JMenu edit=new JMenu("Edit");
  edit.add(createBackgroundColorMenu(button));

  JMenuBar bar=new JMenuBar();
  bar.add(edit);

  JButton button=new JButton("Hello Menu");
  JPanel panel=new JPanel();
  panel.add(button);

  JFrame frame=new JFrame();
  frame.setJMenuBar(bar);
  frame.setContentPane(panel);
  frame.setSize(400,300);
  frame.show();
}
```



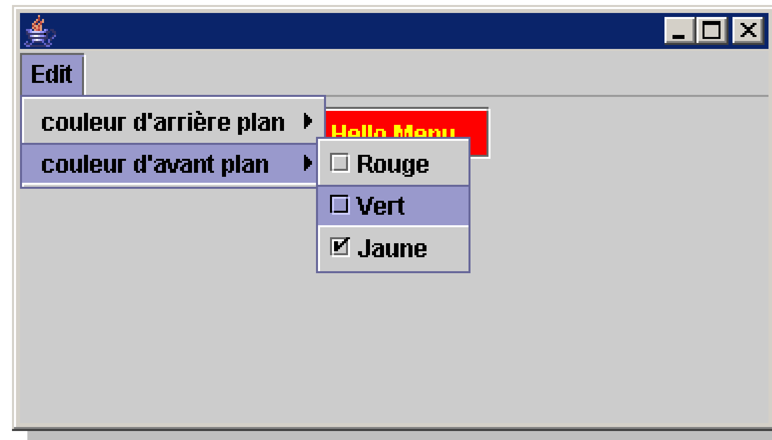
Exemple de Menu (2)

- Exemple suite.

```
enum Item {
  Rouge(Color.RED), Vert(Color.GREEN), Bleue(Color.BLUE);
  MyColor(Color color) {
    this.color=color;
  }
  Color color;
}
private static JMenu createBackgroundColorMenu(
  final JComponent component) {
  JMenu menu=new JMenu("couleur d'arrière plan");
  ButtonGroup group=new ButtonGroup();
  for(final Item item:Item.values()) {
    JCheckBoxMenuItem item=new JCheckBoxMenuItem(item.name());
    item.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        component.setBackground(item.color);
      }
    });
    menu.add(item);
    group.add(item);
  }
  return menu;
}
```

Ajout d'un second menu

- On souhaite modifier l'exemple précédent pour ajouter un second menu ayant les mêmes couleurs mais permettant de changer la couleur d'avant plan.



Ajout d'un second menu (2)

```
interface ColorChanger {
  void changeColor(JComponent component,Color color);
}

private static JMenu createColorMenu(final JComponent component,
  String title,final ColorChanger changer) {
  JMenu menu=new JMenu(title);
  ButtonGroup group=new ButtonGroup();
  for(final Item item:Item.values()) {
    JCheckBoxMenuItem item=new JCheckBoxMenuItem(item.name());
    item.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        changer.changeColor(component,item.color);
      }
    });
    menu.add(item);
    group.add(item);
  }
  return menu;
}
```

Ajout d'un second menu (3)

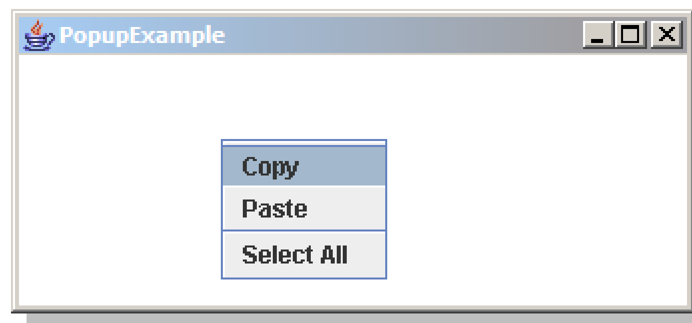
- Le code d'appel devient alors celui-ci :

```
public static void main(String[] args) {
  JMenu edit=new JMenu("Edit");
  edit.add(createColorMenu(button,"arrière plan",new ColorChanger {
    public void changeColor(JComponent component,Color color) {
      component.setBackground(color);
    }
  }));

  edit.add(createColorMenu(button,"avant plan",new ColorChanger {
    public void changeColor(JComponent component,Color color) {
      component.setForeground(color);
    }
  }));
  ...
}
```

- On utilise des classes anonymes pour implanter l'interface **ColorChanger**

- La classe **JPopupMenu** est un conteneur de **JMenuItem** qui implante les menus contextuels
- Marche de la même façon que **JMenu**
- On utilise la méthode **show**(Component composant,int x,int y) pour faire apparaître le menu sur le composant à la position (x,y)



A la main

- Si l'on veut tout contrôler :
  - On place un **MouseListener** sur le composant
  - On ecoute **mousePressed**() **ET** **mouseReleased**()
  - Si **isPopupTrigger**() est vrai pour l'évènement
  - On affiche le menu déroulant grâce à **show**(...)

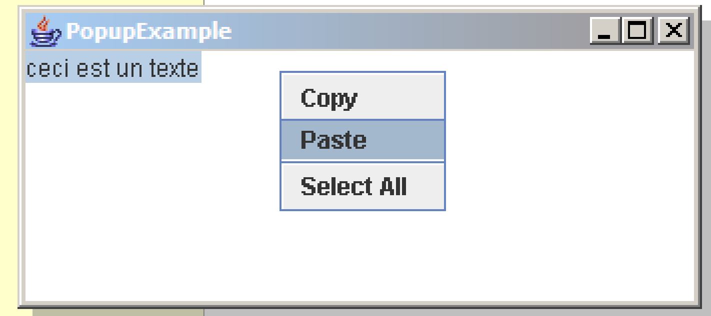
```
public static void main(String[] args) {
  final JTextArea area=new JTextArea();
  JMenuItem copy=new JMenuItem("Copy");
  copy.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      area.copy();
    }
  });
  JMenuItem paste=new JMenuItem("Paste");
  paste.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      area.paste();
    }
  });
  JMenuItem selectAll=...
```

```
  final JPopupMenu menu=new JPopupMenu();
  menu.add(copy);
  menu.add(paste);
  menu.add(new JSeparator());
  menu.add(selectAll);
```

A la main(suite)

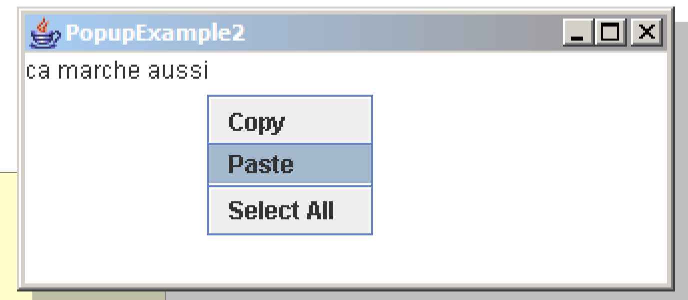
```
  area.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
      mouseChanged(e);
    }
    public void mouseReleased(MouseEvent e) {
      mouseChanged(e);
    }
    private void mouseChanged(MouseEvent e) {
      if (e.isPopupTrigger())
        menu.show(area,e.getX(),e.getY());
    }
  });

  JFrame frame=new JFrame("PopupExample");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setContentPane(area);
  frame.setSize(400,300);
  frame.setVisible(true);
}
```



Le menu contextuel en automatique

- A partir de la version 1.5, sur JComponent :
- **get/setPopupMenu**(JPopupMenu menu) installe/demande un menu contextuel sur le composant
- **get/setInheritsPopupMenu**(boolean) indique que le menu est contenu dans le composant parent



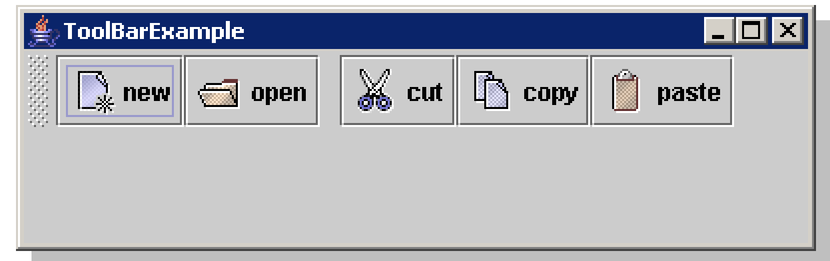

```
  area.setComponentPopupMenu(menu);

  JFrame frame=new JFrame("PopupExample");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setContentPane(area);
  frame.setSize(400,300);
  frame.setVisible(true);
}
```

La barre d'outils

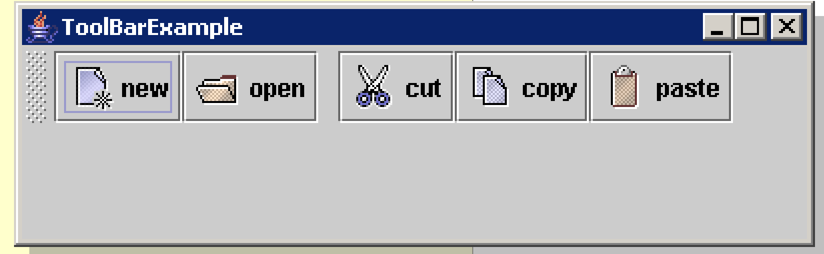
- **JToolBar** conteneur général, qui change d'orientation et peut être flottant. Il utilise un **BoxLayout**.

- Une barre d'outil contient des boutons classique que l'on ajoute avec add().



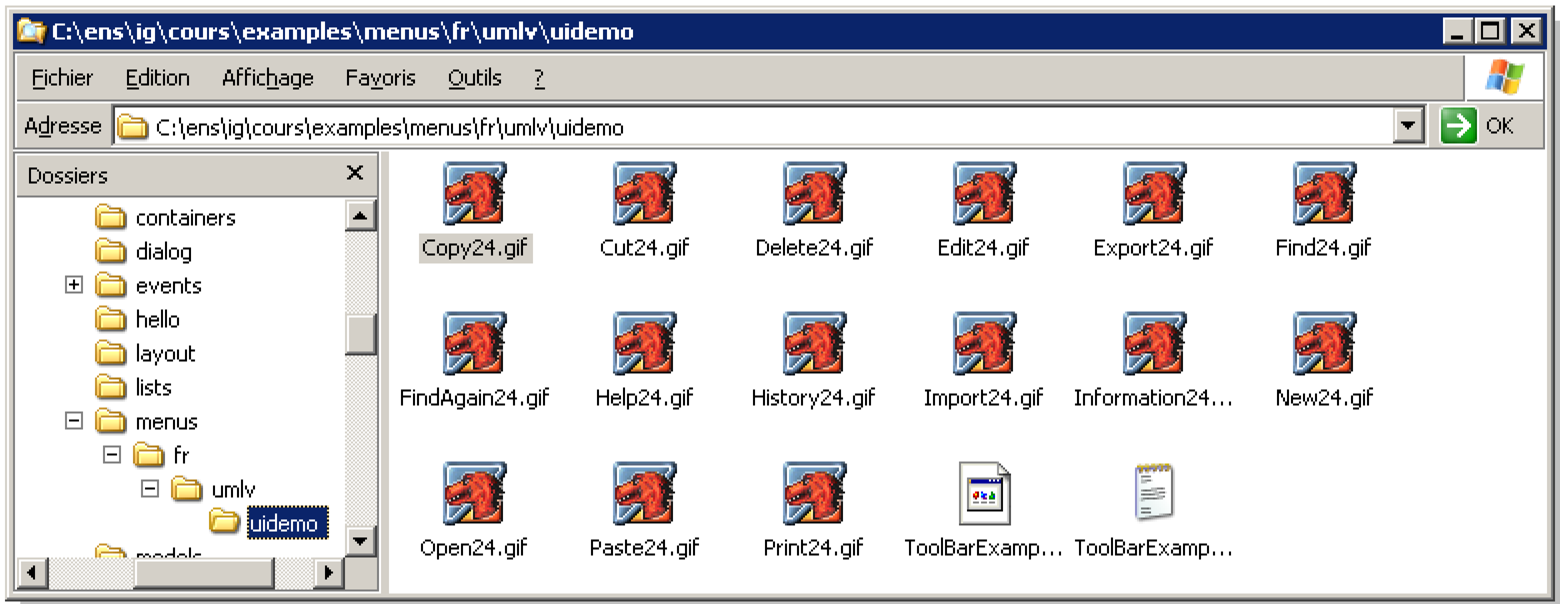
- La barre d'outil se place sur une des quatres bord d'un container possèdant un **BorderLayout** (pas au centre).

La barre d'outils (2)




```
private Icon getIcon(String iconFileName) {
  URL url=getClass().getResource(iconFileName);
  if (url==null)
    return null;
  return new ImageIcon(url);
}
private JButton createButton(String text,String iconFileName) {
  return new JButton(text,getIcon(iconFileName));
}
private JToolBar createToolBar() {
  JToolBar bar=new JToolBar();
  bar.add(createButton("new","New24.gif","nouveau"));
  bar.add(createButton("open","Open24.gif","ouvrir"));
  bar.add(Box.createHorizontalStrut(10));
  bar.add(createButton("cut","Cut24.gif","couper"));
  bar.add(createButton("copy","Copy24.gif","copier"));
  bar.add(createButton("paste","Paste24.gif","coller"));
  return bar;
}
public static void main(String[] args) {
  ToolBarExample toolBar=new ToolBarExample();
  JFrame frame=new JFrame("ToolBarExample");
  frame.getContentPane().add(
    toolBar.createToolBar(),BorderLayout.NORTH);
  frame.setSize(400,300);
  frame.show();
}
```

- La méthode **getResource()** sur une classe permet d'obtenir les images de façon relative au package de cette classe.




```
private Icon getIcon(String iconFileName) {
  URL url=getClass().getResource(iconFileName);
  if (url==null)
    return null;
  return new ImageIcon(url);
}
```

Les Actions

- Une `Action` est une interface représentant un objet qui contient :
  - Un raccourci clavier (*accelerator*)
  - Une lettre de navigation (*mnemonic*)
  - Un nom
  - Une description courte (*tooltip*)
  - Un icône
  - Et le code de l'action

- De plus, une action possède un état activé ou non (*enabled*).

Les Actions (2)

- Les conteneurs `JMenu`, `JPopupMenu` et `JToolBar` honorent les actions:
  - un même objet d'une classe implémentant `Action` peut être "ajouté" à plusieurs de ces conteneurs.
  - les diverses instances ***opèrent de concert***.
  - par exemple, un objet ajouté à un menu et à une barre d'outils est activé ou désactivé simultanément dans les deux.

- Les classes dérivées de `AbstractAction` sont utiles quand une ***même*** action peut être déclenchée de ***plusieurs*** manières.

- Une `Action` hérite de interface `ActionListener` et doit donc implanté aussi la méthode *actionPerformed()*.

AbstractAction

- La classe **AbstractAction** est :
  - un bean ;
  - une classe abstraite ;
  - elle implémente l'interface **Action** ;
  - la seule méthode à écrire est **actionPerformed()**.
- Constructeurs :

```
AbstractAction(String name,Icon icon)
```

AbstractAction (2)

- La classe **`AbstractAction`** implémente le nom, l'icône etc. sous forme de propriétés :
  - ensemble des propriétés **`getKeys()`**
  - Obtenir la valeur d'une proriété **`getValue(String key)`**
  - Changer la valeur d'un proriété **`putValue(String key, Object newValue)`**
  - Listener de propriétés : **`addPropertyChangeListener(...)`** **`removePropertyChangeListener(...)`**

- Les propriétés sont : ACCELERATOR_KEY, MNEMONIC_KEY, NAME, SHORT_DESCRIPTION, SMALL_ICON

Utilisation des actions

- Création d'une classe qui étend **AbstractAction**

```
final JTextArea area=new JTextArea();
final Action cutAction=new AbstractAction(
  "cut",getIcon("Cut24.gif")) {
  public void actionPerformed(ActionEvent e) {
    area.cut();
  }
};
cutAction.putValue(Action.SHORT_DESCRIPTION, "couper");
cutAction.setEnabled(false);
```

- Ne pas utiliser une **Action** comme **ActionListener**, cela ne sert à rien.

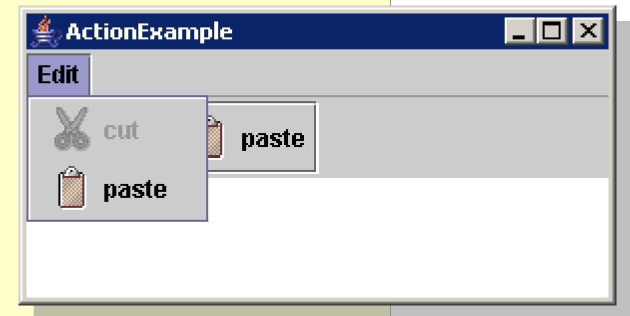
Utilisation des action (2)

- Utilisation dans un menu *et* dans une barre d'outils

```
Action cutAction=...

JToolbar bar=new JToolbar();
bar.add(new JButton(cutAction));
bar.add(cutAction);

JMenu menu=new JMenu("edit");
menu.add(new JMenuItem(cutAction));
menu.add(cutAction);
```



- add(**Action**) est une erreur de *design*.

Utilisation des action (3)

- Lorsque l'on désactive une action, celle-ci désactive les menus et bouttons contenant l'action.

```
final Action cutAction=...
final JTextArea bar=new JTextArea();
...
area.addCaretListener(new CaretListener() {
  public void caretUpdate(CaretEvent e) {
   String text=area.getSelectedText();
   cutAction.setEnabled(text!=null);
  }
});
```