

# Undo - Redo

- ❑ Généralités
- ❑ Transactions annulables
- ❑ Séquences de transactions
- ❑ Evénements d'édition
- ❑ Textes
- ❑ Etats

# Généralités

---

- Le “undo” (annuler) et “redo” (refaire) sont parmi les opérations les plus appréciées dans les interfaces ergonomiques.
- Ce sont des opérations difficiles à implémenter.
- Questions:
  - quelles sont les transactions annulables ?
  - quelle partie de l’environnement doit être sauvegardée pour pouvoir le reconstituer ?
  - vaut-il mieux conserver l’opération, ou son inverse ?
- Java fournit un cadre surtout adapté aux opérations sur les textes.
- Plusieurs variantes existent, mais il reste du travail au programmeur.

# UndoableEdit

---

- L'interface de base est **UndoableEdit**. Une implémentation par défaut est **AbstractUndoableEdit**
- “Edit” est synonyme de transaction ou opération, terme emprunté aux éditeurs de textes.
- Les méthodes sont

<code>boolean canUndo()</code>	indique que la transaction peut être annulée
<code>boolean canRedo()</code>	indique que la transaction peut être refaite
<code>void die()</code>	la transaction ne peut plus être annulée ni répétée
<code>void redo() throws CannotRedoException</code>	refait la transaction
<code>void undo() throws CannotUndoException</code>	annule la transaction

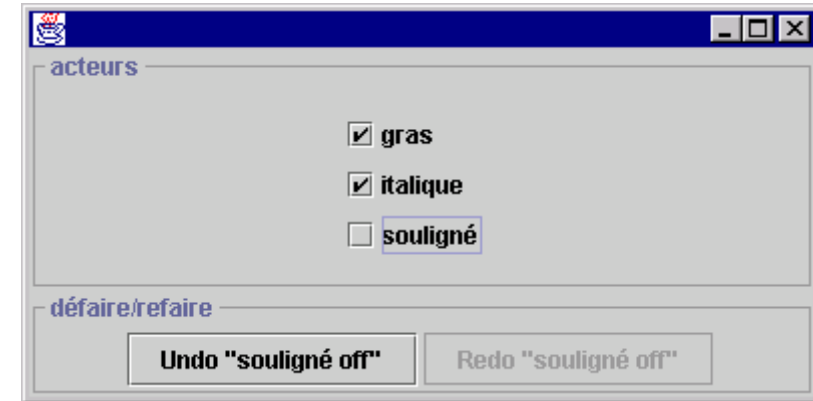
# AbstractUndoableEdit

---

- C'est l'implémentation par défaut de **UndoableEdit**
- Elle maintient deux booléens internes *alive* et *done* qui gèrent correctement le **canUndo()** et **canRedo()**.
- On sous-classe cette classe en redéfinissant **undo()** et **redo()**
- On utilise la sur-classe en appelant **super.undo()**, **super.redo()**.

# Exemple des boutons à cocher

- L'opération de coche ou décoche peut être annulée ou refaite, à partir d'un autre composant (paire de boutons, plus souvent entrée de menu ou boutons d'une barre d'outils)



Démarche:

- Chaque action sur le bouton génère un objet d'une classe **ToggleEdit** dérivant de **AbstractUndoableEdit**. L'objet contient
  - le bouton concerné
  - l'état du bouton
- La classe **ToggleEdit** redéfinit les méthodes **undo()** et **redo()**.
- L'opération d'annulation ou répétition est lancée en appelant la méthode **undo()** ou **redo()** sur l'objet créé.

# ToggleEdit



TestUndoableToggle.bat

```
import javax.swing.undo.*;

public class ToggleEdit extends AbstractUndoableEdit {
    private final JToggleButton bouton;
    private final boolean selectionne;

    public ToggleEdit(JToggleButton bouton) {
        this.bouton = bouton;
        selectionne = bouton.isSelected();
    }

    public void redo() throws CannotRedoException {
        super.redo();
        bouton.setSelected(selectionne);
    }

    public void undo() throws CannotUndoException {
        super.undo();
        bouton.setSelected(!selectionne);
    }
}
```

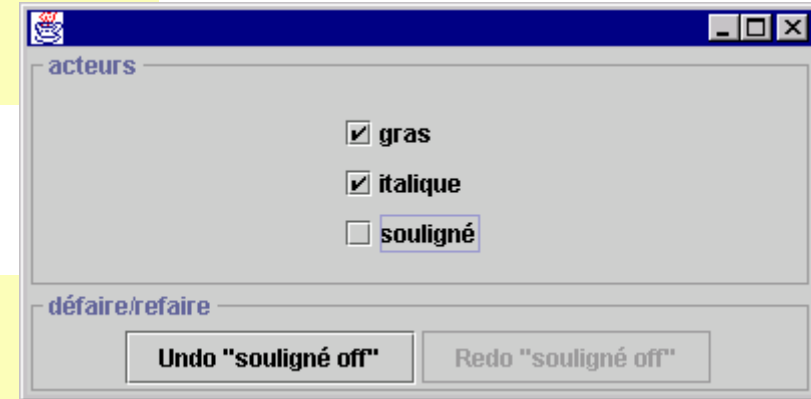
# Le panneau

- Le panneau est composé de trois boutons à cocher

```
JCheckBox gras = new JCheckBox("gras");
JCheckBox ital = new JCheckBox("italique");
JCheckBox soul = new JCheckBox("souligné");
```

- et de deux boutons d'annulation et répétition:

```
JButton undoButton = new JButton("Undo");
JButton redoButton = new JButton("Redo");
undoButton.addActionListener(new UndoIt());
redoButton.addActionListener(new RedoIt());
```



- chaque bouton à cocher (**JCheckBox**) a un écouteur dont la méthode **actionPerformed** est :

```
public void actionPerformed(ActionEvent ev) {
    JToggleButton b = (JToggleButton) ev.getSource();
    edit = new ToggleEdit(b);
    updateButtons(); // voir page suivante
}
```

# Les écouteurs

```
class UndoIt implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        try {
            edit.undo();
        } catch (CannotUndoException ex) {}
        finally {
            updateButtons();
        }
    }
}
```

```
class RedoIt implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        try {
            edit.redo();
        } catch (CannotRedoException ex) {}
        finally {
            updateButtons();
        }
    }
}
```

```
private void updateButtons() {
    undoButton.setText(edit.getUndoPresentationName());
    redoButton.setText(edit.getRedoPresentationName());
    undoButton.setEnabled(edit.canUndo());
    redoButton.setEnabled(edit.canRedo());
}
```

# Complément

- L'interface **UndoableEdit** a une méthode **getPresentationName** qui retourne une chaîne de caractère façonnable en fonction de la transaction
- Les méthodes **getUndoPresentationName** et **getRedoPresentationName** concatènent le préfix Undo et Redo avec la chaîne fournie par **getPresentationName**

```
class ToggleEdit {  
    private final JToggleButton bouton;  
    private final boolean selectionne;  
  
    ...  
    public String getPresentationName() {  
        return "\"" + bouton.getText()  
            + (selectionne ? " on" : " off") + "\"";  
    }  
    ...  
}
```

# Séquences de transactions

- Pour se “souvenir” d’une séquence de transactions, et pouvoir revenir en arrière arbitrairement loin, on utilise un gestionnaire de transactions (**UndoManager**).
- Un **UndoManager** gère les transactions (**Edit**). Il permet de reculer (undo) et d’avancer (redo) tantque possible.
- Une transaction à inscrire dans un gestionnaire doit lui être notifiée,
  - soit directement, par **addEdit(UndoableEdit edit)**
  - soit en utilisant le fait qu’un **UndoManager** implémente un **UndoableEditListener**. On enregistre le gestionnaire dans la liste des auditeurs.



# Implémentation simple



TestManagerToggle.bat

- Un **UndoManager** étend **CompoundEdit** qui lui étend **AbstractUndoableEdit**
- On remplace simplement
  - la variable **UndoEdit edit** par **UndoManager manager**
  - et on modifie **actionPerformed()** en conséquence

# Modifications

```

class TogglePanel extends JPanel
  implements ActionListener {
  private UndoableEdit edit;
  private JButton undoButton, ...;

  public void actionPerformed(ActionEvent e) {
    JToggleButton b
      = (JToggleButton) e.getSource();
    edit = new ToggleEdit(b);
    updateButtons();
  }

  class UndoIt implements ActionListener {
    public void actionPerformed(ActionEvent e){
      try {
        edit.undo();
      } ...
    }
  }

  private void updateButtons() {
    undoButton.setText(
      edit.getUndoPresentationName());
    ...
  }

```

```

class TogglePanel extends JPanel
  implements ActionListener {
  private UndoManager manager
    = new UndoManager();
  private JButton undoButton, ...;

  public void actionPerformed(ActionEvent e) {
    JToggleButton b
      = (JToggleButton) e.getSource();
    manager.addEdit(new ToggleEdit(b));
    updateButtons();
  }

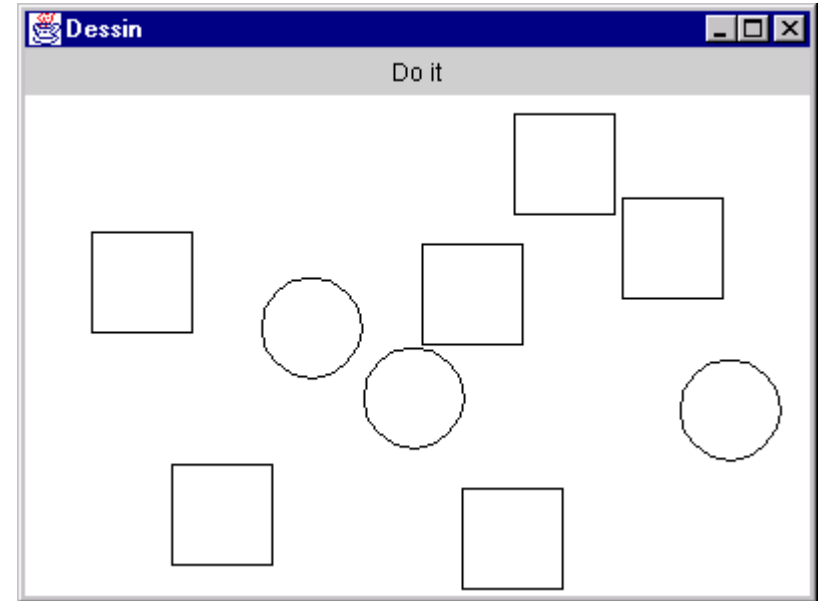
  class UndoIt implements ActionListener {
    public void actionPerformed(ActionEvent e){
      try {
        manager.undo();
      } ...
    }
  }

  private void updateButtons() {
    undoButton.setText(
      manager.getUndoPresentationName());
    ...
  }

```

# Un deuxième exemple

- Le programme *de départ* affiche, au clic de souris, un carré ou un cercle, selon que la touche majuscule n'est pas ou est enfoncé.
- La séquence des formes engendrées est enregistrée dans un vecteur en vue d'un affichage facile.
- *Comment l'adapter au undo/redo ?*



```
class SimplePaint extends JPanel {
    protected Vector formes = new Vector();
    protected PaintCanvas canvas = new PaintCanvas(formes);
    protected int width = 50;
    protected int height = 50;

    public SimplePaint() {
        setLayout(new BorderLayout());
        add(new Label("Do it", Label.CENTER), BorderLayout.NORTH);
        add(canvas, BorderLayout.CENTER);
        canvas.addMouseListener(new AjouterForme());
    }
    ...
}
```

# Les formes et le canevas

```
...
class AjouterForme extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        Shape shape;
        if (e.isShiftDown())
            shape = new Ellipse2D.Double(e.getX(), e.getY(), width, height);
        else
            shape = new Rectangle2D.Double(e.getX(), e.getY(), width, height);
        formes.addElement(shape);
        canvas.repaint();
    }
}
```

```
class PaintCanvas extends JPanel {
    Vector formes;
    ...
    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        super.paintComponent(g2);
        g2.setColor(Color.black);
        Enumeration enum = formes.elements();
        while(enum.hasMoreElements()) {
            Shape shape = (Shape) enum.nextElement();
            g2.draw(shape);
        }
    }
}
```

# Ajouter undo/redo



UndoRedoPaintApp.bat

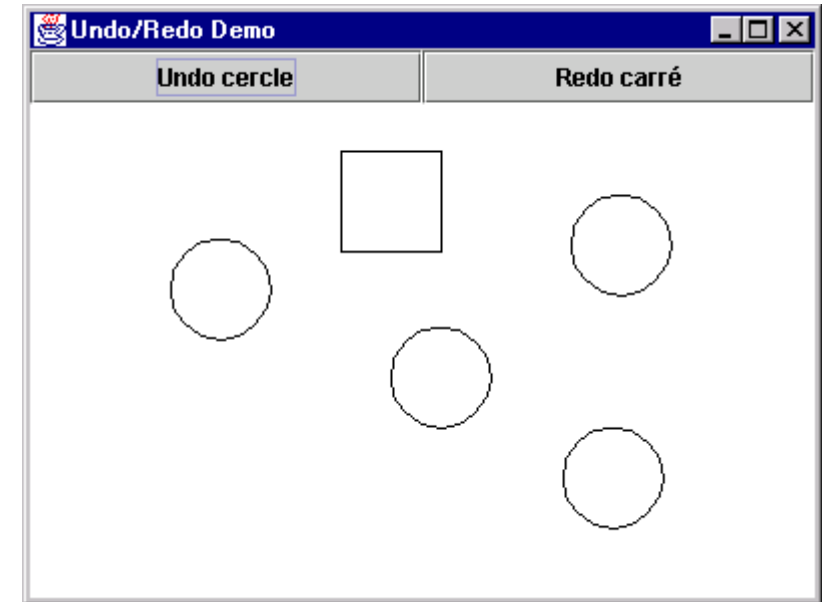
- Comme pour l'exemple précédent
  - deux boutons “undo” et “redo”
  - deux auditeurs d'actions, un sur chaque bouton
- Création d'une classe **FormeEdit** pour les transactions, et de deux classes dérivées.

```
class FormeEdit extends AbstractUndoableEdit {
    protected Vector formes;
    protected Shape shape;

    public FormeEdit(Shape shape, Vector formes) {
        this.formes = formes;
        this.shape = shape;
    }
    public void undo() {
        super.undo();
        formes.remove(shape);
    }

    public void redo() {
        super.redo();
        formes.add(shape);
    }
}
```

```
class CarreEdit extends FormeEdit {
    public CarreEdit(Shape shape, Vector formes) {
        super(shape, formes);
    }
    public String getPresentationName() {
        return "carré";
    }
}
```



# L'auditeur AjouterForme

```
public void mousePressed(MouseEvent e) {
    Shape shape;
    if (e.isShiftDown())
        shape = new Ellipse2D.Double(e.getX(), e.getY(), width, height);
    else
        shape = new Rectangle2D.Double(e.getX(), e.getY(), width, height);
    formes.addElement(shape);
    canvas.repaint();
}
```

■ Avant

```
public void mousePressed(MouseEvent e) {
    Shape shape;
    UndoableEdit edit;
    if (e.isShiftDown()) {
        shape = new Ellipse2D.Double(e.getX(), e.getY(), width, height);
        edit = new CercleEdit(shape, formes);
    }
    else {
        shape = new Rectangle2D.Double(e.getX(), e.getY(), width, height);
        edit = new CarreEdit(shape, formes);
    }
    formes.addElement(shape);
    manager.addEdit(edit);
    canvas.repaint();
    updateButtons();
}
```

■ Après

# Événements

- Il existe une classe spécifique **UndoableEditEvent**
- Un tel événement comporte une *source*, et un **UndoableEdit**
- Les auditeurs sont de l'interface **UndoableEditListener**, avec la méthode **undoableEditHappened(UndoableEditEvent e)**.
- **UndoManager** implémente **UndoableEditListener**, avec la méthode

```
public void undoableEditHappened(UndoableEditEvent e) {
    addEdit(e.getEdit())
}
```

- Usage :

```
...
UndoableEdit edit;
...
edit = new CercleEdit(shape, formes);
...
manager.addEdit(edit);
...
```

```
...
UndoableEdit edit;
...
edit = new CercleEdit(shape, formes);
UndoableEditEvent ue;
ue = new UndoableEditEvent(this, edit);
...
manager.undoableEditHappened(ue);
...
```

# mais...

- Il manque un objet qui lance des **UndoableEditEvent**'s.  
L'implémentation précédente fait comme si, la présente le fait.
- Seuls les documents de textes sont capables, pour l'instant, d'en lancer. Lançons cela pour **JPanel** :

```
class PaintCanvas extends JPanel {
    ...
    public void addUndoableEditListener(UndoableEditListener listener) {
        listenerList.add(UndoableEditListener.class, listener);
    }

    public void removeUndoableEditListener(UndoableEditListener listener) {
        listenerList.remove(UndoableEditListener.class, listener);
    }

    public void fireUndoableEditUpdate(UndoableEditEvent e) {
        Object[] listeners = listenerList.getListenerList();
        for (int i = listeners.length-2; i>=0; i-=2) {
            if (listeners[i] == UndoableEditListener.class)
                ((UndoableEditListener)listeners[i+1]).undoableEditHappened(e);
        }
    }
}
```

# ...et donc

- Un UndoManager est un listener parfait

```
public FireUndo() {  
    ...  
    canvas.addMouseListener(new AjouterForme());  
    canvas.addUndoableEditListener(manager);  
}
```

- Il ne reste plus qu'à lancer les événements

```
public void mousePressed(MouseEvent e) {  
    Shape shape;  
    UndoableEdit edit;  
    shape = ...  
    edit = ...  
    formes.addElement(shape);  
    UndoableEditEvent ue = new UndoableEditEvent(this, edit);  
    canvas.fireUndoableEditUpdate(ue);  
    canvas.repaint();  
    updateButtons();  
}
```

# UndoableEditSupport

- Pour faciliter la vie aux programmeurs (en attendant que les choses se simplifient), Java propose une classe utilitaire de gestion de listeners et d'envoi d'événements, les **UndoableEditSupport**.
- Ils réalisent pour l'essentiel ce qui a été programmé en dur.

```
class PaintCanvas extends JPanel {
    UndoableEditSupport support = new UndoableEditSupport();
    ...
    public void addUndoableEditListener(UndoableEditListener listener) {
        support.addUndoableEditListener(listener);
    }

    public void removeUndoableEditListener(UndoableEditListener listener) {
        support.removeUndoableEditListener(listener);
    }

    public void postEdit(UndoableEdit e) { // le fireUndoableEditUpdate....
        support.postEdit(e);
    }
}
```

# UndoableEditSupport (fin)

- Au lieu de lancer les événements, on poste les Edit:

```
public void mousePressed(MouseEvent e) {
    UndoableEdit edit;
    edit = ...
    UndoableEditEvent ue = new UndoableEditEvent(this, edit);
    canvas.fireUndoableEditUpdate(ue);
    ...
}
```

- devient

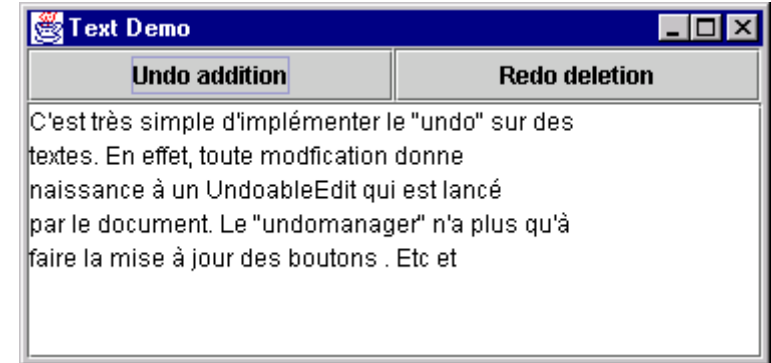
```
public void mousePressed(MouseEvent e) {
    UndoableEdit edit;
    edit = ...
    canvas.postEdit(edit);
    ...
}
```

# Dans les textes



UndoRedoTextApp.bat

- Dans les textes, ça va tout seul.



```

JTextArea editor = new JTextArea();

public UndoRedoText() {
    editor.getDocument().addUndoableEditListener(new ManageIt());
    undoButton.addActionListener(new UndoIt());
    redoButton.addActionListener(new RedoIt());
}

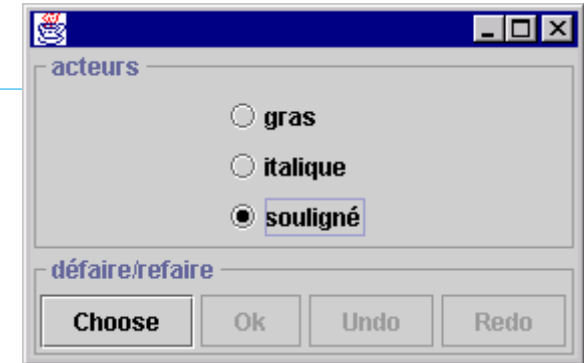
class ManageIt implements UndoableEditListener {
    public void undoableEditHappened(UndoableEditEvent e) {
        manager.undoableEditHappened(e);
        updateButtons();
    }
}

```



TestStateEditToggle.bat

- Dans les exemples précédents, on conservait explicitement l'état après modification. Un tel procédé n'est pas suffisant dans de nombreuses situations, comme dans un groupe de boutons radio.
- Java propose une forme générale d'état appelé **StateEdit**. Tout objet dont la classe implémente l'interface **StateEditable** peut sauvegarder son état avant et après modification dans l'état, et ainsi le récupérer.
- Mieux, la prise en compte de l'état de départ et de l'état d'arrivée peut être programmée, permettant ainsi de cumuler des modifications.



# Etats : description

---

- Un **StateEdit** est créé par

```
StateEdit etat = new StateEdit(unObjet);
```

- La classe de **unObjet** implémente l'interface **StateEditable**.
- Un **StateEdit** contient en interne une table de hachage (en fait deux). Les méthodes

```
public void storeState(Hashtable h);  
public void restoreState(Hashtable h);
```

- de **StateEditable** permettent de sauvegarder et de récupérer les données à conserver.
- La sauvegarde débute à la création, et s'arrête par la méthode **end()** de **StateEdit**.

# Etats: structure de l'exemple

```
class TogglePanel extends JPanel implements ActionListener, StateEditable {
    StateEdit etat;
    JButton undoButton, redoButton, chooseButton, endButton;
    JRadioButton gras, ital, soul;
    ButtonGroup polices;

    public TogglePanel() {
        //installer les composants;
        chooseButton.addActionListener(new ChooseIt());
        endButton.addActionListener(new EndIt());
        undoButton.addActionListener(new UndoIt());
        redoButton.addActionListener(new RedoIt());
    }

    public void storeState(Hashtable h) {...}
    public void restoreState(Hashtable h) {...}
    class ChooseIt implements ActionListener {...}
    class EndIt implements ActionListener {...}
    class UndoIt implements ActionListener {...}
    class RedoIt implements ActionListener {...}
}
```

# Etats : fin de l'exemple

- Trois boutons radio sont donnés. A partir de **Choose**, on démarre l'enregistrement des modifications, jusqu'à l'activation du bouton **Ok**.
- Un **Undo** restitue l'état *initial*, et un **Redo** revient à l'état *final*.

```
class ChooseIt implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
        etat = new StateEdit(TogglePanel.this);
        updateButtons();
    }
}
```

```
public void storeState(Hashtable h) {
    h.put(polices, polices.getSelection());
}

public void restoreState(Hashtable h) {
    ButtonModel b = (ButtonModel) h.get(polices);
    b.setSelected (true);
}
```

```
class UndoIt implements ActionListener {
    public void actionPerformed(
        ActionEvent ev) {
        try { etat.undo(); }
        catch (CannotUndoException ex) {}
        updateB();
    }
}
```