

TR2TI1-Techniques Informatiques I

TOME 2 : Interface graphique (SWING) - Flux d'E/S
(Notes provisoires)

GIANNI TRICARICO

Haute Ecole en Hainaut
Campus Technique

AC 2013-2014

Organisation

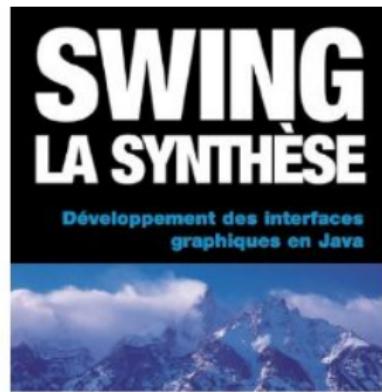
Objectifs

- Concevoir une interface graphique avec SWING
- Gérer les événements
- Introduire quelques Design Patterns
- Manipuler les flux E/S (fichiers)

Pré-requis

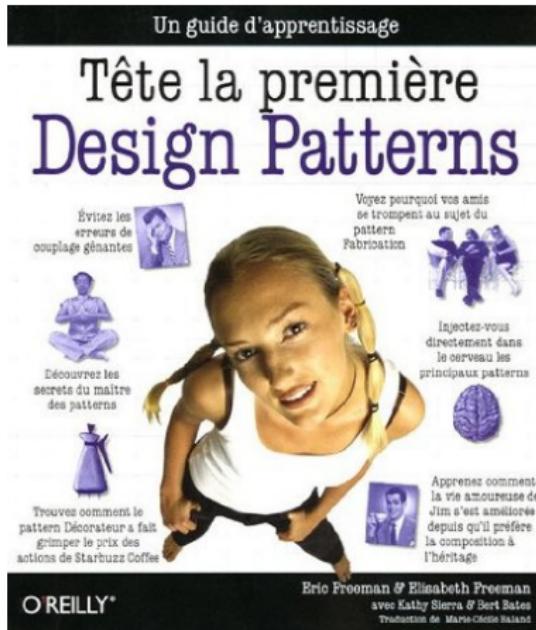
- Programmation orientée objet : héritage, polymorphisme, classe abstraite, interface.
- UML : diagramme de classes

Ouvrage de référence pour Swing(facultatif)



Valérie Berthié & Jean-Baptiste
Briaud, "*SWING LA
SYNTHESE*", Dunod, 2003

Ouvrage de référence pour les Design Patterns(facultatif)



Eric Freeman, Elisabeth Freeman,
Kathy Sierra, Bert Bates &
Marie-Cécile Baland *"Design
Patterns"*, O'Reilly Editions, 2005

Lecture pour approfondir ses connaissances

- Bruce Eckel, "*Thinking in Java, 3e*" Prentice Hall,2003,
<http://bruce-eckel.developpez.com/livres/java/traduction/tij2/>
- Cay S. Horstmann & Gary Cornell, "*Core Java, Volume 2 : Advanced Features (Java SE 6)*",Prentice Hall,2008
- Elliotte Rusty Harold, "*Java I/O*",O'Reilly,2010
- Elliotte Rusty Harold, "*Java Network Programming*",O'Reilly,2005
- Laurent Debrauwer, "*Design Patterns pour Java - Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java*",ENI,2009

Introduction aux design patterns

Définition du design pattern

- Un modèle de conception
 - parfois aussi "Motifs de conception" ou "Patrons de conception"
- Une organisation de classes utilisée pour résoudre un problème récurrent.
 - qui est conçue en tenant compte des bonnes pratiques de conception => favorisant l'extensibilité et la réutilisabilité
- Indépendant du langage de conception

Origine

- Domaine de l'architecture : C. Alexander "A Pattern Language : Towns, Buildings, Construction" [1977]
- Les principaux auteurs des design patterns GoF : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
 - Connus sous le nom GoF « Gang of Four »
 - Livre de référence : "Design patterns : Elements of reusable Object-Oriented Software" [1994]
- Design patterns GRASP (General Responsibility Assignment Software Patterns (or Principles) : Craig Larman
 - Livre : "Applying UML and Patterns" [1997]

Avantages

- Catalogue de solutions qui permet de bénéficier du savoir faire d'experts dans des contextes éprouvés.
- Facilite la conception.
- Ne pas réinventer la roue.
- Facilite la communication entre développeurs.
- Pour résoudre un problème.

Catalogues des DPs GoF

- Catalogue : "Design patterns : Elements of reusable Object-Oriented Software" [1994]
- 23 DPs
- Trois catégories
 - Les patterns de création
 - Les patterns de structure
 - Les patterns de comportement

Catégories des DPs GoF

- Les 5 patterns de création

- **But** : organiser la création d'objets
- *Factory method, AbstractFactory, Builder, Prototype, Singleton*

- Les 7 patterns de structure

- **But** : organiser la hiérarchie des classes et de leurs relations
- *Adapter, Bridge, Composite, Decorator, Interface, Flyweight, Proxy*

- Les 11 patterns de comportement

- **But** : organiser les interactions et répartir les traitements entre les objets
- *Interpreter, Template method, Chain of responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor*

Swing et les composants d'interface utilisateur (GUI)

Swing ?

Swing :

- est une bibliothèque permettant la création d'interfaces utilisateur en Java.
- fournit les classes pour la représentation des différents **éléments d'interfaces graphiques** : *boutons, fenêtres, menus, etc.*, et la gestion des **événements**.
- s'appuie sur la **bibliothèque AWT**(Abstract Windowing Toolkit) qui est la plus ancienne des deux.
- propose des éléments d'interfaces graphiques qui sont dessinés par la bibliothèque elle-même contrairement AWT qui délègue l'affichage à l'OS => même apparence sur toutes les plates-formes(améliore la portabilité) => **amélioration AWT**

Swing ?

Swing :

- "stocke" ces classes dans le paquetage **javax.swing** et ses sous-paquetages.
 - les noms de composants Swing commencent (souvent) par **J**

Interface graphique

L'interface graphique (GUI) :

- est la partie visible à l'écran d'une application
- permet à l'utilisateur d'interagir avec l'application

L'interface graphique est un moyen graphique permettant à l'utilisateur de "dialoguer" avec l'application.

Introduction

Réalisation d'une Interface graphique (GUI) :

- Construire GUI
 - Composants, containers et des fenêtres.
 - Gestionnaires d'agencement
- Gestion des actions des utilisateurs
- Séparation Vue/Modèle

Composants

Une interface graphique est constituée d'éléments graphiques qui sont nommés "composants" (ou "widgets"). Comme par exemple :

- boutons,
- labels,
- menus,
- champs de saisie, ...

Tour d'horizon des composants Swing

Nom : javax.swing.JLabel

Usage :

Permet d'afficher du texte **et/ou** une image.

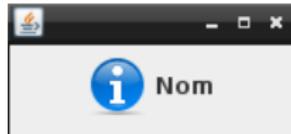
Méthodes

Ajouter du texte :

```
JLabel lbINom = new JLabel("Nom");  
//ou  
lbINom.setText("Nom");
```

La méthode setIcon permet d'ajouter une icône au texte.

```
lbINom.setIcon(new ImageIcon("information.png"));
```



Tour d'horizon des composants Swing

Nom : javax.swing.JTextField

Usage :

Permet d'afficher un champ de saisie.

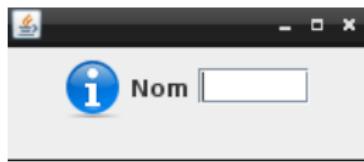
JPasswordField est une variante du JTextField, utilisé pour les mots de passe.

Méthodes

setText() : permet de spécifier un texte par défaut dans le JTextField

getText() : permet de récupérer le texte saisi

setColumns() : permet d'imposer le nombre de colonne



Tour d'horizon des composants Swing

Nom : javax.swing.JButton

Usage :

Permet d'afficher un bouton.

Méthodes

setIcon() permet d'ajouter une icône au bouton.

setMnemonic() : permet de définir un raccourci-clavier en tapant la combinaison de touches "Alt + mnemonic"



Tour d'horizon des composants Swing

Nom : javax.swing.JCheckBox et javax.swing.JRadioButton

Usage :

- JCheckBox : La case à cocher propose une option à l'utilisateur.
Mode de sélection : Les cases à cocher sont indépendantes les unes des autres, c'est-à-dire que plusieurs cases à cocher d'un même groupe peuvent être sélectionnées en même temps.
- JRadioButton : Le radio bouton propose une option à l'utilisateur.
Mode de sélection : par défaut, même comportement que les cases à cocher.

La classe ButtonGroup : permet de regrouper un ensemble de boutons. Cela permet qu'un seul bouton radio (ou case à cocher) du groupe peut être sélectionné à la fois à un moment donné.

Tour d'horizon des composants Swing



Tour d'horizon des composants Swing

Nom : javax.swing.JComboBox

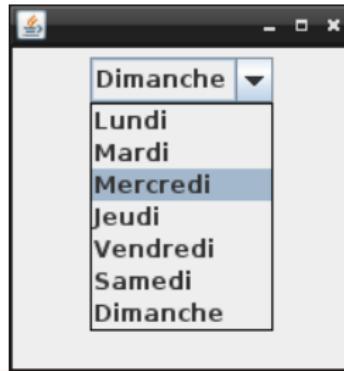
Usage :

Utilisé pour présenter une liste de valeurs possibles, au sein desquelles l'utilisateur effectue son choix.

Méthodes

`setMaximumRowCount()` : permet de définir le nombre d'items à afficher

`setEditable()` : permet de rendre la combo éditable



Propriétés communes

Activation et désactivation d'un composant

- Pour activer ou désactiver un composant :
`void setEnabled(true ou false)`
- Pour tester si un composant est activé :
`boolean isEnabled()`
- Pour rendre un composant visible ou non :
`void setVisible(true ou false)`
- Pour tester la visibilité :
`boolean isVisible()`

Propriétés communes

Couleurs :

- `setBackground (Color)` : permet de régler la couleur de fond
- `setForeground` : permet de régler la couleur de premier plan
- `Color` : classe `Color` se trouve dans le package `java.awt`.
- Pour construire une couleur, on peut utiliser le constructeur :
`public Color(int rouge,int vert,int bleu);`
- Ou les constantes prédéfinies :
`Color.white,Color.red, ...`

```
composant.setBackground(new Color(255,0,0));
composant.setBackground(Color.red);
```

Différentes fenêtres

- JFrame : Fenêtre principale avec bordure et barre de titre (non modale).
- JDialog : Fenêtre secondaire qui peut être modale (bloquant l'application) => boîte de dialogue

Une fenêtre B est modale par rapport à une fenêtre A si l'affichage de B empêche l'accès à la fenêtre A.

- JWindow : Fenêtre sans bordure ni barre de titre => splash screen (fenêtre d'accueil)

"Cette fenêtre incite l'utilisateur à patienter pendant le chargement et l'installation d'un logiciel tout en lui apportant diverses informations comme le nom du logiciel, le nom de l'éditeur, le logo de l'éditeur ou du logiciel, les droits d'auteur associés au logiciel, la version et l'état du chargement du logiciel." [Wikipédia]

Propriétés communes : JFrame & JDialog

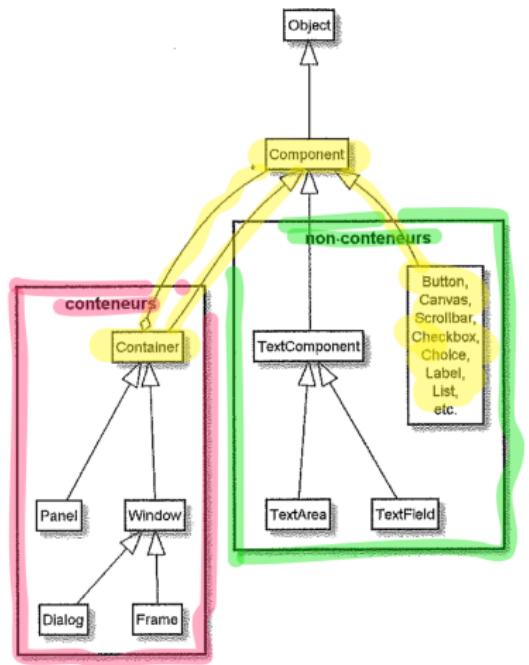
Méthodes

- void dispose() : permet de libérer la mémoire occupée par une fenêtre et tout ce qu'elle référence.
- setDefaultCloseOperation(int operation) : permet de définir un comportement lorsque l'utilisateur clique sur l'icône de fermeture. Le paramètre operation permet d'indiquer un comportement choisi parmi un ensemble prédéfini(constante).
- setResizable(boolean b) : permet de définir si une fenêtre est redimensionnable ou pas.
- void setTitle(String titre) : afficher du texte dans la barre de titre

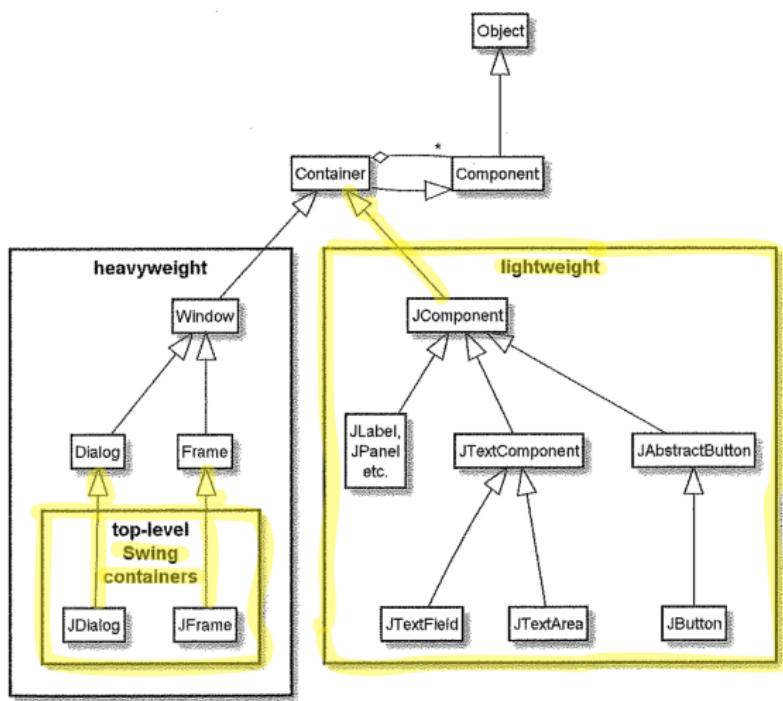
Introduction

- Un composant :
 - gère le rendu graphique
- Un conteneur permet de :
 - contenir d'autres **composants**
 - gérer l'agencement des composants contenus

Architecture AWT : Exemple tiré de Java (pattern Composite)



Architecture Swing : Exemple tiré de Java (pattern Composite)



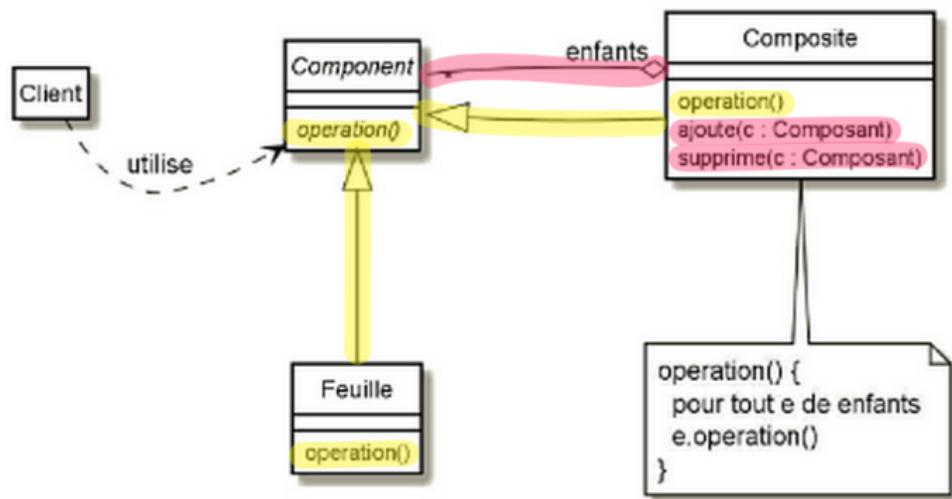
UML - Pattern Composite

"Le **pattern Composite** compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci." [Design Patterns - Tête la première : Juillet Aymeric]

Définition

Une interface graphique est une arborescence de composants dont les feuilles sont des composants graphiques du type JComponent et les nœuds des conteneurs. La racine est un conteneur particulier : une fenêtre (JFrame) ou une boîte de dialogue(JDialog).

Pattern Composite - Structure



Pattern Composite - Participants

- **Composant** est la classe abstraite qui introduit l'interface des objets de la composition, implante les méthodes communes et introduit la signature des méthodes qui gèrent la composition en ajoutant ou en supprimant des composants ;
- **Feuille** est la classe concrète qui décrit les feuilles de la composition (une feuille ne possède pas de composants) ;
- **Composé** est la classe concrète qui décrit les objets composés de la hiérarchie. Cette classe possède une association d'agrégation avec la classe **Composant** ;
- **Client** est la classe des objets qui accèdent aux objets de la composition et qui les manipulent.

Arbre de composants

GUI = arbre de composants graphiques

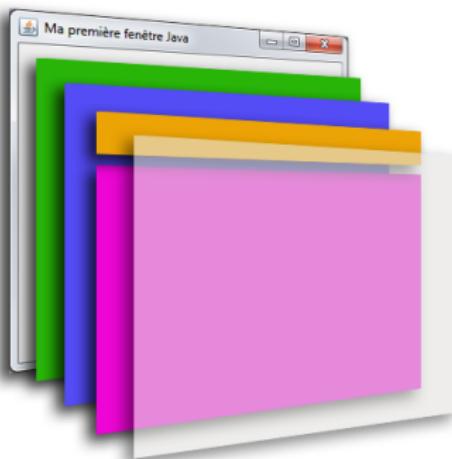
- **Racine de l'arbre** : la "fenêtre"
 - conteneur de top-level : *JFrame*, *JDialog* et *JWindow*
- **Nœuds de l'arbre** : les conteneurs intermédiaires
 - *JPanel* : Panneau vide utilisé pour la mise en page des composants à l'écran.
 - *JScrollPane* : Panneau à ascenseurs.
 - *JSplitPane* : Panneau partagé verticalement ou horizontalement.
- **Feuilles de l'arbre** : les composants graphiques dits "atomiques"
 - *JButton*, *JLabel*, ...

Conteneur "top-level" : JFrame

JFrame :

- possède une zone de contenu (content pane) destinée à accueillir les composants de l'interface
- peut contenir une barre de menu (JMenuBar)
- fournit l'API suivante :
 - accesseur/mutateur du ContentPane (get/setContentPane())
 - accesseur/mutateur de la JMenuBar (get/setMenuBar())

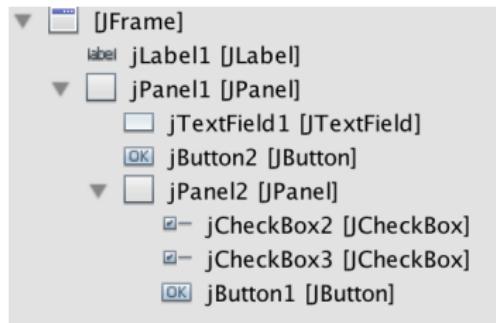
Conteneur "top-level" : JFrame



- **RootPane** : le conteneur principal qui contient les autres composants
- **LayeredPane** : forme juste un panneau composé du conteneur global et de la barre de menu
- **MenuBar** : la barre de menu, quand il y en a une ;
- **ContentPane** : c'est dans celui-ci que nous placerons nos composants ;
- **GlassPane (transparent)** : couche utilisée pour intercepter les actions de l'utilisateur avant qu'elles ne parviennent aux composants.

Réf. :<http://www.siteduzero.com/informatique/tutoriels/apprenez-a-programmer-en-java/1-objet-jframe>

Exemple GUI



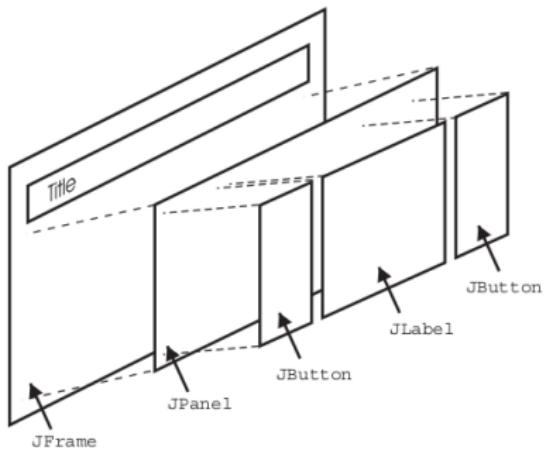
Méthodes communes aux containers

- Ajouter un composant dans un container

```
container.add(composant);
```

- Retirer un composant d'un container

```
container.remove(composant);
```

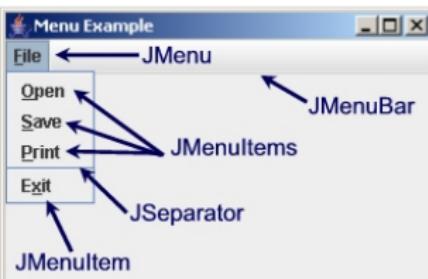


Fermeture d'une JFrame

- La fermeture d'une fenêtre(croix) ne met pas fin au programme, mais rend simplement la fenêtre invisible (comme si on appelait la méthode **setVisible(false)**).
- 4 modes de fermeture possibles, à définir avec **setDefaultCloseOperation** :
 - **DO NOTHING ON CLOSE** : ne fait rien
 - **HIDE ON CLOSE** : rend la fenêtre invisible (mode par défaut)
 - **DISPOSE ON CLOSE** : cache la fenêtre et libère toutes les ressources systèmes associées ; elles seront réallouées si la fenêtre redevient visible
 - **EXIT ON CLOSE** : termine le programme

Menus déroulants

Création d'un menu



- 1 Créer une barre de menus

```
JMenuBar menuBar=new JMenuBar();
```

- 2 Placer la barre de menus dans la fenêtre

```
frame.setJMenuBar(menuBar);
```

- 3 Créer un objet menu

```
JMenu editMenu=new JMenu("Edition");
```

- 4 Ajouter le menu à la barre de menus

```
menuBar.add(editMenu);
```

- 5 Ajouter les éléments au menu "Edition"

```
JMenuItem pasteItem;
pasteItem=new JMenuItem("Coller");
editMenu.add(pasteItem);
editMenu.addSeparator(); //ajoute un séparateur
JMenu optionsMenu=...; //un sous-menu
editMenu.add(optionsMenu);
```

Gestionnaires d'agencement

Introduction

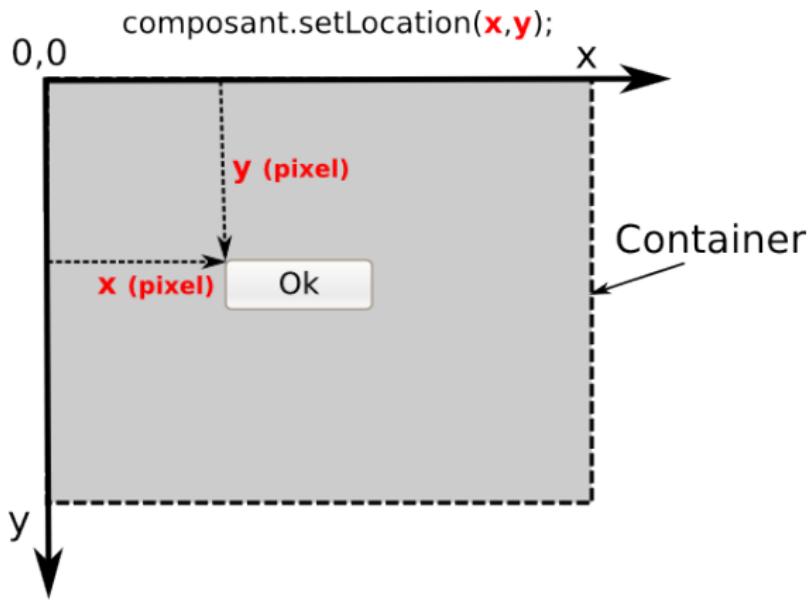
En Java, il existe deux méthodes pour l'agencement des composants graphiques, soit :

- **Positionnement absolu** : les composants sont placés de façon absolue dans un container, en fournissant les coordonnées et les dimensions de chacun d'entre eux.
- **Positionnement à l'aide de gestionnaires d'agencement** : les composants sont placés et dimensionnés automatiquement (même après redimensionnement de la fenêtre) par un *gestionnaire d'agencement*(layout managers).
Le gestionnaire d'agencement est un objet Java associé à un container.

Positionnement absolu

- `container.setLayout(null)` : permet de désactiver le gestionnaire d'agencement du container
- `composant.setSize(largeur, hauteur)` : Redimensionne le composant sur la largeur et la hauteur spécifiées
- `composant.setLocation(x,y)` : Déplace le composant vers un nouvel emplacement. Les coordonnées x et y utilisent les coordonnées du conteneur si le composant n'est pas de haut niveau ou les coordonnées de l'écran si le composant est de haut niveau (par exemple, à JFrame).
- `composant.setBounds(x,y, largeur, hauteur)` : Déplace et redimensionne le composant.

Positionnement absolu - Système de coordonnées



Problème du placement absolu

Problème : placement des composants au pixel près.



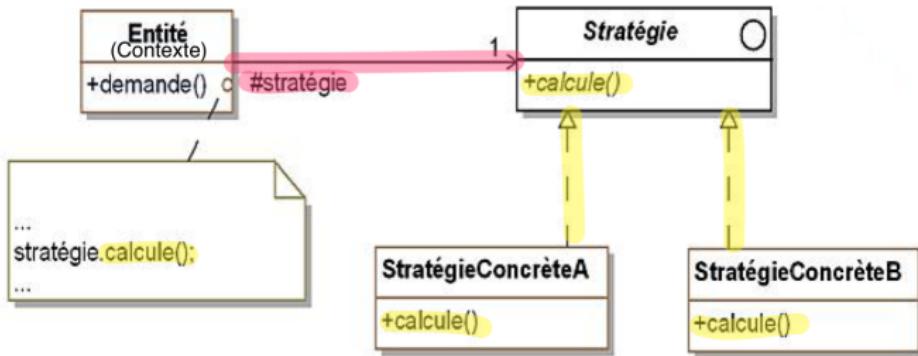
Solution : Mécanisme d'agencement des composants qui lors d'un agrandissement/réduction replace les composants au bon endroit.
Lier un objet "d'agencement" au composant conteneur et de disposer plusieurs stratégies (algorithmes) de placement.

Implémentation : *Pattern Stratégie*

Pattern Stratégie : Définition - Structure

Définition

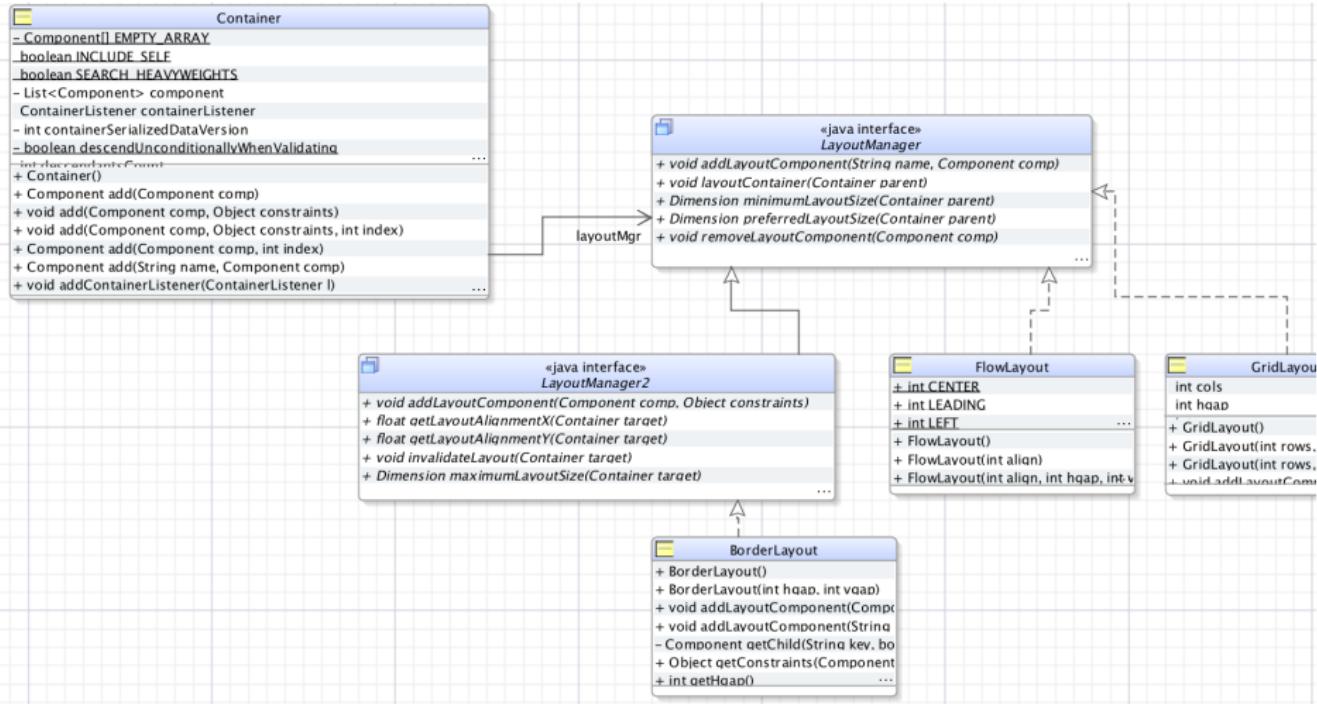
Le **pattern Stratégie** définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Stratégie permet à l'algorithme de varier indépendamment des clients qui l'utilisent.



Pattern Stratégie : Participants

- **Stratégie** est l'interface commune à tous les algorithmes. Cette interface est utilisée par Entité pour invoquer l'algorithme ;
- **StratégieConcrèteA** et **StratégieConcrèteB** sont les sous-classes concrètes qui implantent les différents algorithmes ;
- **Entité** est la classe utilisant un des algorithmes des classes d'implantation de Stratégie. En conséquence, elle possède une référence vers une instance de l'une de ces classes. Enfin, si nécessaire, elle peut exposer ses données internes aux classes d'implantation.

Pattern Stratégie : LayoutManager



Caractéristiques communes des gestionnaires d'agencement

- Les gestionnaires utilisent les tailles : **getPreferredSize()**, **getMaximalSize()** et **getMinimalSize()** de chaque composant pour les positionner dans le conteneur.
- La méthode **pack()** sur une JFrame demande à chaque conteneur sa taille préférée et redimensionne la fenêtre à sa taille préférée.
- Pour définir une autre taille, on redimensionne un composant avec : **setSize(largeur, hauteur)** ou **setPreferredSize(dimension)**

Caractéristiques communes des gestionnaires d'agencement

- Tout composant a un placement par rapport à son container, donné par son origine et ses dimensions.
 - La position est gérée par **get/setLocation()** et **get/setX()**, **get/setY()**
 - La taille gérée par **get/setSize()** et **get/setWidth()**, **get/setHeight()**
 - Le rectangle est géré par **get/setBounds()**.
- Le gestionnaire utilise ces méthodes pour placer un composant.

Gestionnaires d'agencement

- Il existe deux types de gestionnaire d'agencement :
 - LayoutManager : sans contrainte
 - Le **FlowLayout**
 - Le **GridLayout**
 - Le **BoxLayout**
 - LayoutManager2 : avec contraintes
 - Le **BorderLayout**
- Affecter un gestionnaire

```
JPanel panneau=new JPanel();
panneau.setLayout(new BorderLayout());
```

Gestionnaire FlowLayout

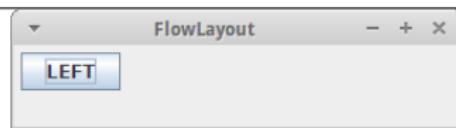
- C'est le gestionnaire d'agencement par défaut du JPanel
- Les éléments graphiques sont placés les uns à côté des autres, de gauche à droite.
- Le FlowLayout est paramétrable :
 - FlowLayout.LEFT : alignement des composants à gauche
 - FlowLayout.RIGHT : alignement des composants à droite
 - FlowLayout.CENTER (par défaut) : alignement des composants au centre

Remarque

Les gestionnaires d'agencement ignorent la méthode `setSize()` (taille réelle). Ils se basent sur la méthode `setPreferredSize()` (taille préférée).

Gestionnaire FlowLayout

- FlowLayout donne aux composants leur taille préférée dans les deux dimensions.
- Le constructeur du FlowLayout permet de définir le type d'alignement



```
container.setLayout(new FlowLayout(FlowLayout.LEFT));
```



```
container.setLayout(new FlowLayout(FlowLayout.RIGHT));
```



```
container.setLayout(new FlowLayout(FlowLayout.CENTER));
```

Gestionnaire GridLayout

- L'espace du container est découpé en une grille
- Les composants sont placés de gauche à droite et de haut en bas dans la grille
- Le nombre de **colonnes** et de **lignes** composant cette grille est fixé lors de l'instanciation de GridLayout.
- Tous les composants ont une taille identique



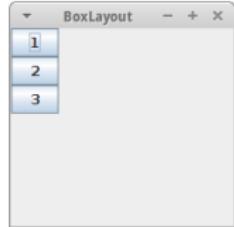
Gestionnaire GridLayout

```
GridLayout experimentLayout = new GridLayout(3,3);
compsToExperiment.setLayout(experimentLayout);
compsToExperiment.add(new JButton("1"));
compsToExperiment.add(new JButton("2"));
...
```

Gestionnaire BoxLayout

- Permet d'aligner les composants les uns à côté des autres, ou les uns en dessous des autres
- BoxLayout utilise la taille préférée des composants dans les deux dimensions
- Le constructeur du BoxLayout a besoin de savoir quel composant il agence et quel axe utiliser

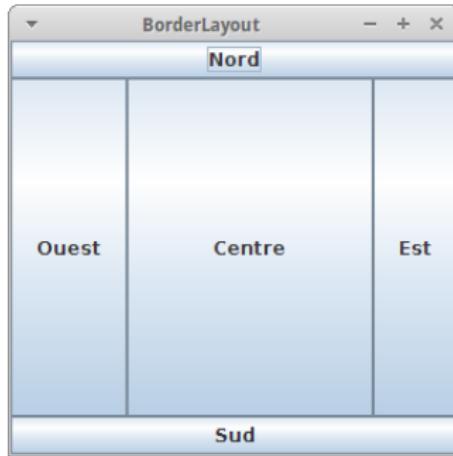
```
container.setLayout(new BoxLayout(container, BoxLayout.Y_AXIS));
```



Gestionnaire BorderLayout

- C'est le gestionnaire d'agencement par défaut du JFrame
- Divise un container en cinq régions
 - Région centrale => CENTER (placement par défaut)
 - Régions périphériques => SOUTH, NORTH, WEST et EAST
- Les composants situés dans les régions nord et sud possèdent leurs tailles préférées verticalement et occupent le maximum d'espace horizontalement
- Les composants situés dans les régions ouest et est possèdent leurs tailles préférées horizontalement et occupent le maximum d'espace verticalement
- Les composants situés au centre occupent toute l'espace restant.

Gestionnaire BorderLayout



```
pan.setLayout(new BorderLayout());
JButton bouton1=new JButton("Nord");
pan.add(bouton1, BorderLayout.NORTH);
JButton bouton2=new JButton("Sud");
pan.add(bouton2, BorderLayout.SOUTH);
JButton bouton3=new JButton("Ouest");
pan.add(bouton3, BorderLayout.WEST);
JButton bouton4=new JButton("Est");
pan.add(bouton4, BorderLayout.EAST);
JButton bouton5=new JButton("Centre");
pan.add(bouton5, BorderLayout.CENTER);
```

Gestion des événements

Événements

Toutes les interactions de l'utilisateur, avec l'interface graphique, se traduisent par des **événements**.

Exemples :

- Déplacement de la souris
- Obtention ou perte de focus
- Appui d'une touche du clavier
- Clic sur un bouton
- Sélection d'un item dans une liste déroulante ou dans un menu

Les classes d'événements

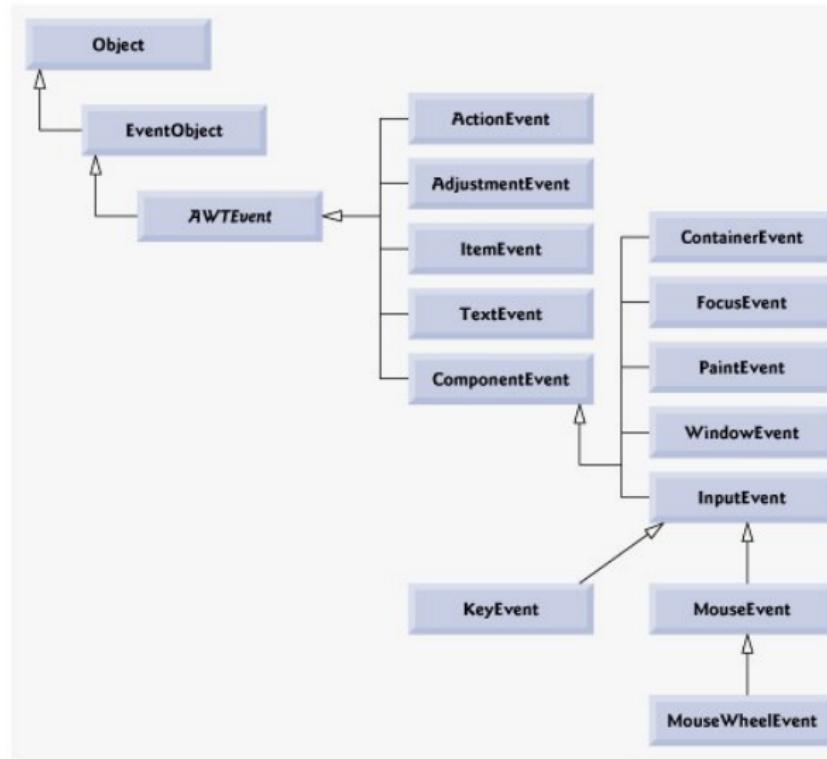
Les **événements** en Java sont représentés par des **objets** qui fournissent des informations sur l'événement lui-même et sur l'objet à l'origine de cet événement.

Interaction utilisateur	Événement émis
Passage du focus à un composant	FocusEvent
Clic sur un JPanel	MouseEvent
Frappe d'une touche clavier sur un JPanel	KeyEvent
Clic sur un bouton	ActionEvent
Ajout d'une lettre dans un JTextField	DocumentEvent
Sélection d'un item dans une JList	ListSelectionEvent

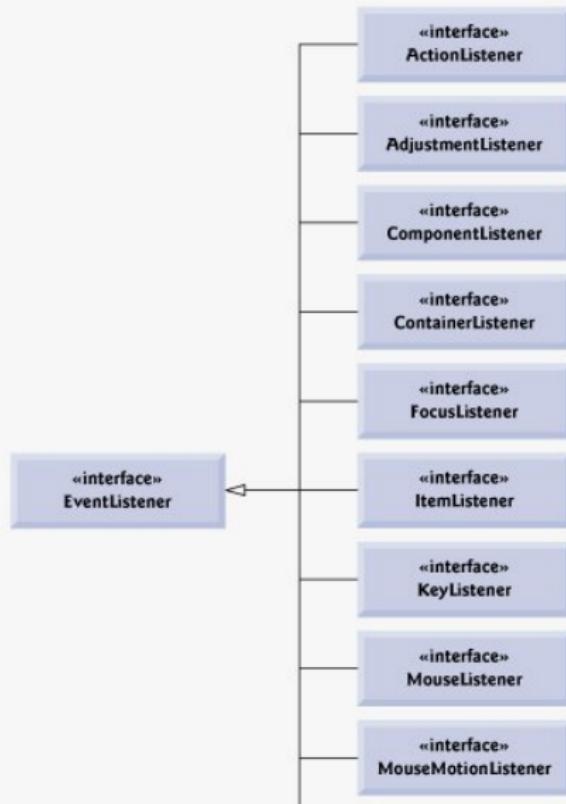
Gestion des événement : écouteurs

- Un écouteur est un objet destiné à recevoir et à gérer les événements générés par le système.
- Les écouteurs principaux se trouvent eux aussi dans le package **java.awt.event**

Les classes d'événements



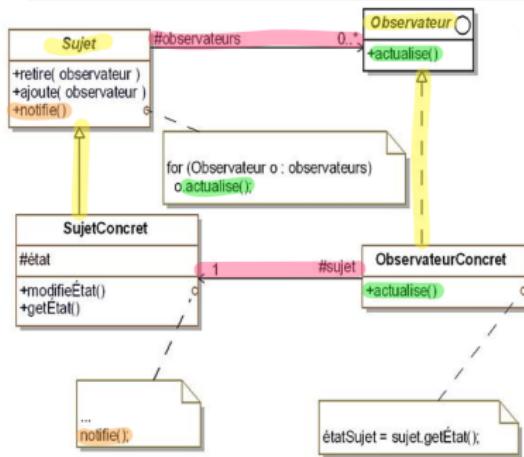
Les interfaces listener



Pattern Observateur - définition et structure

Définition

Le **pattern Observateur** définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

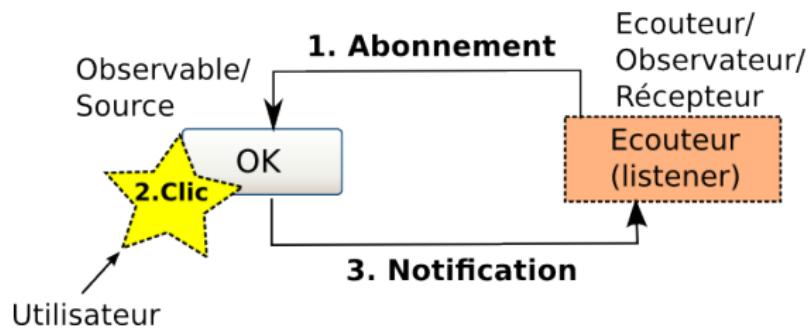


Souscription + diffusion = pattern Observateur

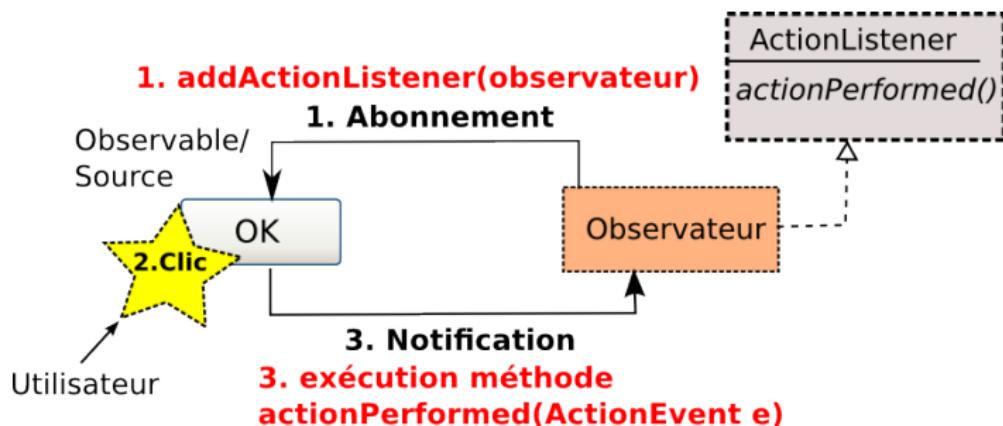
Design Observateur - Participants

- **Sujet(observé)** est la classe abstraite qui introduit l'association avec les observateurs ainsi que les méthodes pour ajouter ou retirer des observateurs ;
- **Observateur** est l'interface à implanter pour recevoir des notifications ;
- **SujetConcret** est une classe d'implantation d'un sujet. Un sujet envoie une notification quand son état est modifié ;
- **ObservateurConcret** est une classe d'implantation d'un observateur. Celui-ci maintient une référence vers son sujet et implante la méthode *actualise()*. Elle demande à son sujet des informations faisant partie de son état lors des mises à jour par invocation de la méthode *getEtat()*.

Principe



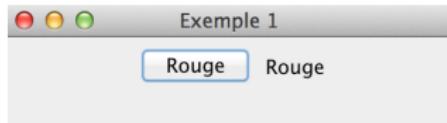
Pattern Observateur



- ① Souscription de l'écouteur auprès du composant graphique (sujet)
=> **Rq** : Création d'un écouteur(observateur) (implémenter une interface de type xxxListener)
- ② Déclenchement de l'événement + création d'objet de type événement (xxxEvent)
- ③ Notification de l'écouteur par le composant graphique

Exemple - Utilisation d'un écouteur externe

- Un clique sur le bouton "Rouge" affiche le texte "Rouge".
- Utilisation de l'interface **ActionListener** qui définit la méthode **actionPerformed()**. => écouteur
- Cette méthode sera appelée lorsque l'utilisateur réalisera un clique sur le bouton.



```
public class CouleurAction implements ActionListener{
    private JLabel lblCouleur;
    public CouleurAction(JLabel lblCouleur){
        this.lblCouleur=lblCouleur;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        lblCouleur.setText("Rouge");
    }
}
```

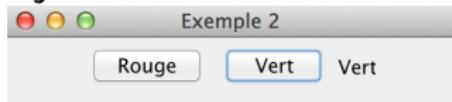
Exemple - Utilisation d'un écouteur externe (suite)

- Souscrire l'écouteur de type **CouleurAction** au bouton "btnBoutonR" à l'aide de la méthode *addActionListener(*)*.

```
public class FrmFenetre extends JFrame{  
  
    public frmFenetre(){  
        JPanel pnlPanneau=new JPanel();  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.getContentPane(pnlPanneau);  
        this.setSize(300,200);  
        this.setTitle("Exemple 1");  
        JButton btnBoutonR=new JButton("Rouge");  
        JLabel lblCouleur=new JLabel();  
        btnBoutonR.addActionListener(new CouleurAction(lblCouleur)); <-- (*)  
        pnlPanneau.add(btnBoutonR);  
        pnlPanneau.add(lblCouleur);  
    }  
  
    public static void main(String[] args) {  
        FrmFenetre maFenetre = new FrmFenetre();  
        maFenetre.setVisible(true);  
    }  
}
```

Exemple - 1 écouteur pour 2 boutons

- Ajoutons un deuxième bouton "Vert".



- getSource()* permet d'obtenir le composant ayant généré l'évènement (*)

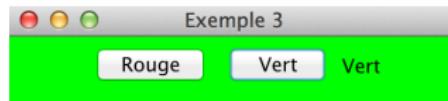
```
public class CouleurAction implements ActionListener{
    private JLabel lblCouleur;
    public CouleurAction(JLabel lblCouleur){
        this.lblCouleur=lblCouleur;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton btnSource=(JButton)e.getSource(); <--(*)
        lblCouleur.setText(btnSource.getText());
    }
}
```

Exemple - 1 écouteur pour 2 boutons(suite)

```
public class FrmFenetre extends JFrame{  
  
    public FrmFenetre(){  
        JPanel pnlPanneau=new JPanel();  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.getContentPane(pnlPanneau);  
        this.setSize(300,200);  
        this.setTitle("Exemple 2");  
        JButton btnBoutonR=new JButton("Rouge");  
        JButton btnBoutonV=new JButton("Vert");  
        JLabel lblCouleur=new JLabel();  
        CouleurAction couleurAction=new CouleurAction(lblCouleur);  
        btnBoutonR.addActionListener(couleurAction);  
        btnBoutonV.addActionListener(couleurAction);  
        pnlPanneau.add(btnBoutonR);  
        pnlPanneau.add(btnBoutonV);  
        pnlPanneau.add(lblCouleur);  
    }  
  
    public static void main(String [] args) {  
        FrmFenetre maFenetre = new FrmFenetre();  
        maFenetre.setVisible(true);  
    }  
}
```

Exemple - Code à éviter

- Ajoutons une fonctionnalité à notre application : la couleur de fond de la fenêtre est définie par les boutons.



```
public class CouleurAction implements ActionListener{
    private JLabel lblCouleur;
    private JPanel pnlPanneau;
    public CouleurAction(JLabel lblCouleur , JPanel pnlPanneau){
        this.lblCouleur=lblCouleur;
        this.pnlPanneau=pnlPanneau;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton btnSource=(JButton)e.getSource();
        lblCouleur.setText(btnSource.getText());

        if(btnSource.getActionCommand().equals("Rouge")){
            pnlPanneau.setBackground(Color.RED);
        }
        else{
            pnlPanneau.setBackground(Color.GREEN);
        }
    }
}
```

Exemple - Code souhaité

```
public class FrmFenetre extends JFrame{  
  
    public FrmFenetre(){  
        JPanel pnlPanneau=new JPanel();  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.getContentPane(pnlPanneau);  
        this.setSize(300,200);  
        this.setTitle("Exemple 3");  
        JButton btnBoutonR=new JButton("Rouge");  
        JButton btnBoutonV=new JButton("Vert");  
        JLabel lblCouleur=new JLabel();  
        btnBoutonR.addActionListener(new CouleurAction(lblCouleur,  
            pnlPanneau, Color.RED));  
        btnBoutonV.addActionListener(new CouleurAction(lblCouleur,  
            pnlPanneau, Color.GREEN));  
        pnlPanneau.add(btnBoutonR);  
        pnlPanneau.add(btnBoutonV);  
        pnlPanneau.add(lblCouleur);  
    }  
    public static void main(String [] args) {  
        FrmFenetre maFenetre = new FrmFenetre();  
        maFenetre.setVisible(true);  
    }  
}
```

Exemple - Amélioration du code (suite)

```
public class CouleurAction implements ActionListener{
    private JLabel lblCouleur;
    private JPanel pnlPanneau;
    private Color clrCouleur;
    public CouleurAction(JLabel lblCouleur, JPanel pnlPanneau, Color clrCouleur){
        this.lblCouleur=lblCouleur;
        this.pnlPanneau=pnlPanneau;
        this.clrCouleur=clrCouleur;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton btnSource=(JButton)e.getSource();
        lblCouleur.setText(btnSource.getText());
        pnlPanneau.setBackground(clrCouleur);
    }
}
```

Listener et classe anonyme

- Les classes anonymes permettent d'éviter
 - une multiplication des classes de listener.
 - de passer les composants à modifier en paramètre du constructeur de la classe de listener
- Limitation des classes anonymes :
 - ne peut pas implémenté deux interfaces
 - ne peut pas implémenté une classe et une interface

Listener et classe anonyme - Exemple

```
1 public class FrmFenetre extends JFrame{  
2     private JPanel pnlPanneau;  
3     private JLabel lblCouleur;  
4     public FrmFenetre(){  
5         pnlPanneau=new JPanel();  
6         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
7         this.getContentPane(pnlPanneau);  
8         this.setSize(300,200);  
9         this.setTitle("Exemple 5");  
10        JButton btnBoutonR=new JButton("Rouge");  
11        JButton btnBoutonV=new JButton("Vert");  
12        lblCouleur=new JLabel();  
13        btnBoutonR.addActionListener(new ActionListener() {  
14            @Override  
15            public void actionPerformed(ActionEvent e) {  
16                lblCouleur.setText("Rouge");  
17                pnlPanneau.setBackground(Color.RED);  
18            }  
19        });  
20        btnBoutonV.addActionListener(new ActionListener() {  
21            @Override  
22            public void actionPerformed(ActionEvent e) {  
23                lblCouleur.setText("Vert");  
24                pnlPanneau.setBackground(Color.GREEN);  
25            }  
26        });  
27        pnlPanneau.add(btnBoutonR);  
28        pnlPanneau.add(btnBoutonV);  
29        pnlPanneau.add(lblCouleur);  
30    }  
31  
32    public static void main(String[] args) {  
33        FrmFenetre maFenetre = new FrmFenetre();  
34        maFenetre.setVisible(true);  
35    }  
36}
```

Classes adaptateurs

Vous pouvez souhaiter afficher une boîte de dialogue lorsque l'utilisateur ferme une fenêtre et ne sortir qu'après confirmation de l'utilisateur.

Le listener de la fenêtre doit être un objet d'une classe implémentant l'interface `WindowListener`, qui possède **sept méthodes**.

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```

Classes adaptateurs

Toute classe qui implémente une interface doit implémenter toutes les méthodes de cette interface ; dans ce cas, cela signifie que les sept méthodes doivent être implémentées. Une seule nous intéresse en l'occurrence : la méthode `windowClosing()`.
=>six méthodes qui ne font rien !

Pour nous éviter d'avoir à implémenter six méthodes vides, Java fournit des classes d'implémentation par défaut des listeners. -> les classes adaptateurs

La classe écouteur hérite de la classe adaptateur et redéfini seulement la méthode nécessaire.

Pattern composé :

Modèle-Vue-Contrôleur

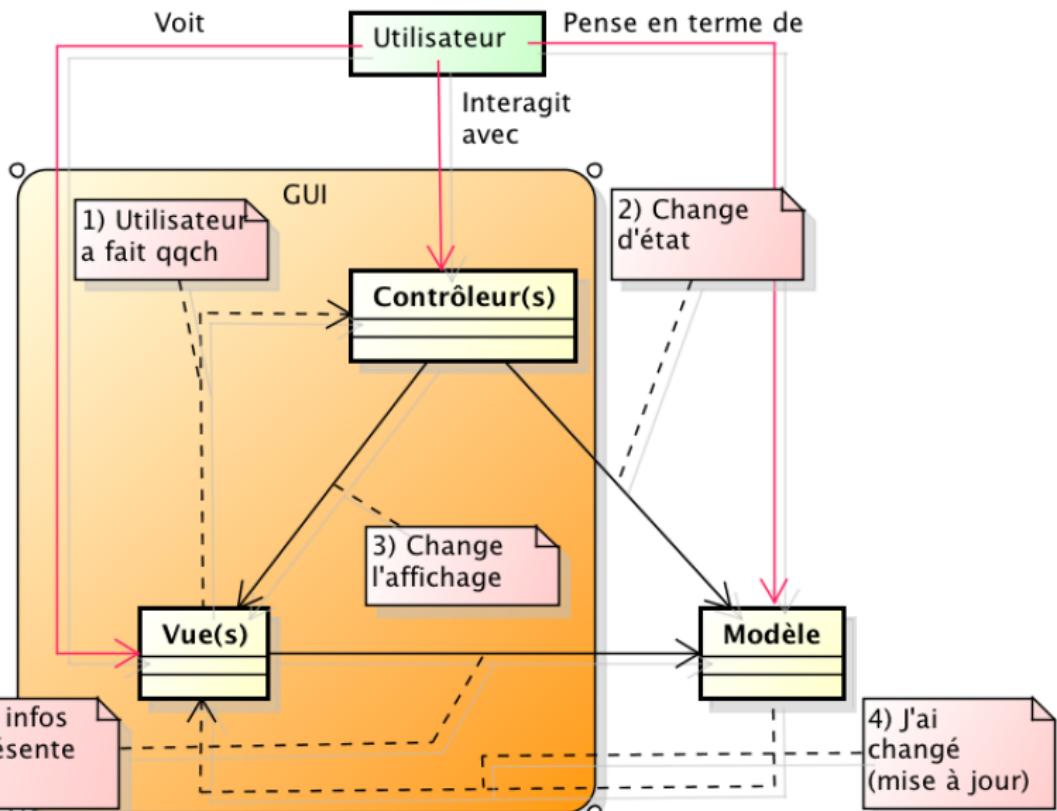
- Une architecture logicielle => une manière de structurer une application
- Un ensemble de patterns qui collaborent dans une même conception

But : faciliter l'évolution des IHM

Décomposition des responsabilités

- **Modèle** : Défini les aspects fonctionnels : les états, les données et la logique applicative
- **Vues** : Représentation graphique de données du Modèle
- **Contrôleurs** : Associés aux vues, ils font évoluer l'état du Modèle

Structure MVC



Le modèle

Le modèle :

- Représente les **données**
- Fournit les **accès aux données** (getXXX et setXXX)
- Fournit les **traitements** applicables aux données (Logique)
- Expose les **fonctionnalités** de l'application (méthodes publiques)
- Peut fonctionner sans interface graphique

La vue

La vue :

- Représente graphiquement l'état du Modèle
- Défini la structure de l'interface
- Agit sur les propriétés de ses composants graphiques
- N'effectue aucun traitement.

Le contrôleur

Le contrôleur :

- Fournit la traduction des actions de l'utilisateur en actions sur le modèle
- Fournit la vue appropriée par rapport aux actions de l'utilisateur et des réactions du modèle

Convention de nommage

- **Packages :**

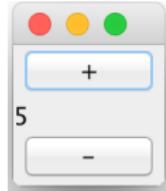
- **Contrôleur** : be.heh.projet.controleurs
- **Vue** : be.heh.projet.vues
- **Modèle** : be.heh.projet.modele

- **Classes**

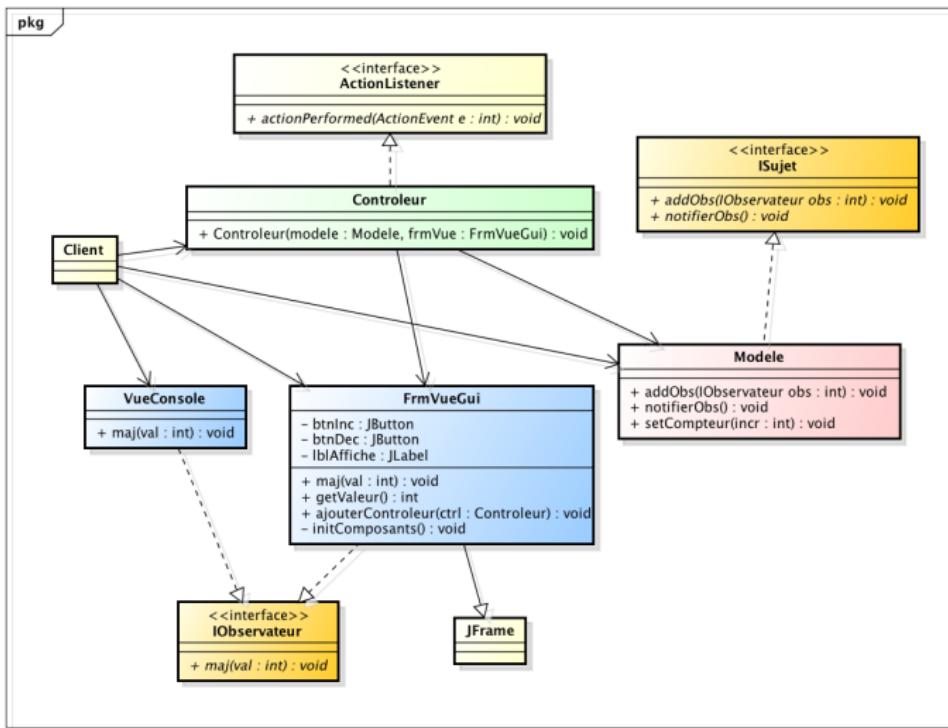
- **Contrôleur** : ControleurNomDeClasse.java
- **Vue** : VueNomDeClasse.java
- **Modèle** : ModeleNomDeClasse.java

Exemple : Compteur

```
run:  
Valeur initiale 0  
Nouvelle valeur 1  
Nouvelle valeur 2  
Nouvelle valeur 3  
Nouvelle valeur 4  
Nouvelle valeur 5
```



Exemple : Compteur



powered by Astah

Flux Entrées/Sorties

Flux entrées/sorties

But : lire/écrire des données à partir d'un fichier, de la mémoire, du réseau...

- Deux API d'entrées/sorties
 - Entrées/sorties bloquantes (`java.io.*`)
 - Entrées/sorties non bloquantes (`java.nio.*`) – non abordées
- Deux mode d'entrée/sortie
 - Mode binaire : écriture et lecture de données bruts
 - Mode caractère : écriture et lecture de caractères...
- Deux mode d'accès :
 - Séquentiel :
 - Direct (ou aléatoire) :

La classe File

Un objet *File* représente le nom et le chemin d'un fichier ou d'un répertoire sur le disque. Mais il ne représente PAS les données contenues dans le fichier.

- **Constante** : `File.separatorChar`

Retourne '/' sous Mac/Linux et '\sous Windows.

- **Constructeur** : `File(String path)`

Construit un objet de type File correspondant au chemin 'path'

- **Constructeur** : `File(String path, String name)` :

Construit un objet de type File correspondant au chemin obtenu à partir du répertoire 'path' et du nom 'name'

```
File f=new File("monProjet"+File.separatorChar+"fichier");
File f=new File("monProjet","fichier");
```

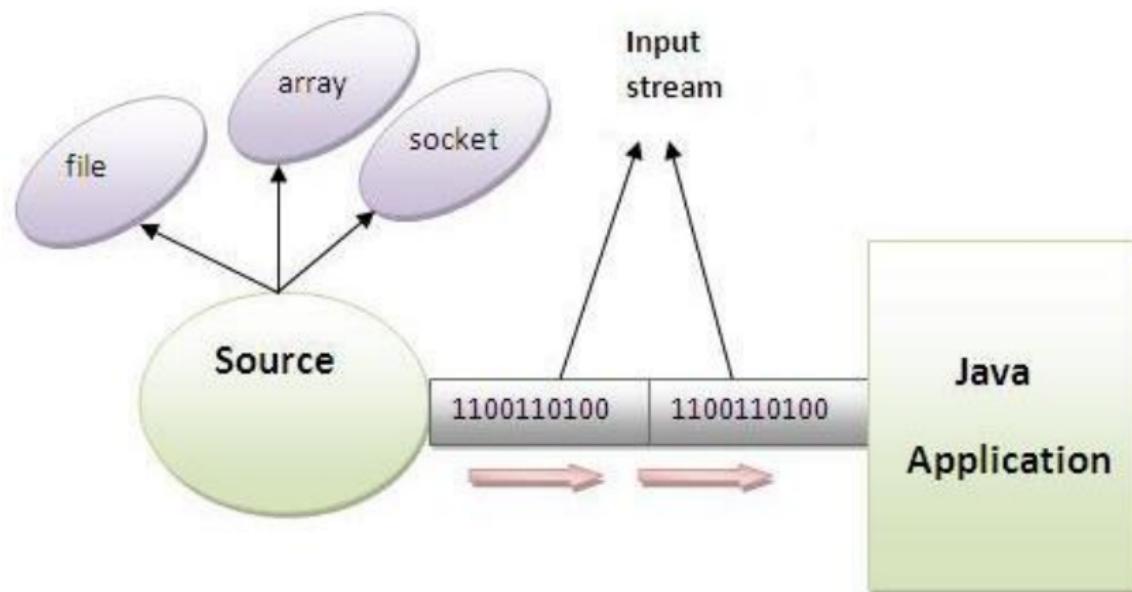
Classe File - Méthodes

- boolean exists() : le fichier existe-t-il ?
- boolean isDirectory() : est-ce un répertoire
- boolean isFile() : est-ce un fichier ?
- boolean canRead() : autorisation de lecture ?
- boolean canWrite() : autorisation de d'écriture ?
- String getAbsolutePath() : retourne le chemin absolu
- String[] list() : retourne la liste des fichiers si c'est un répertoire
- boolean delete() : supprime le fichier
- boolean renameTo(File dest) : renomme le fichier
- void mkdir() : créer le répertoire correspondant à ce File

Flux entrées/sorties

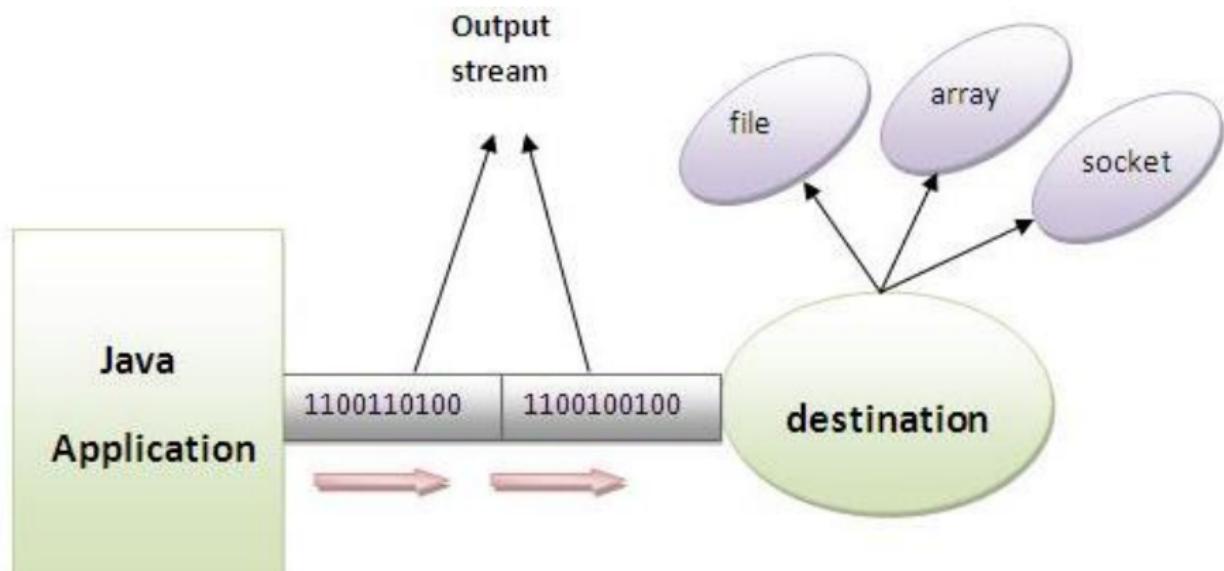
- Un flux est une séquence ordonnée d'**octets** de longueur indéterminée
- **Flux d'entrée** déplace des octets de données d'une source (fichier, clavier, socket, ..) vers un programme Java(**InputStream**).
- **Flux de sortie** déplacer des octets de données venant d'un programme Java vers une cible(**OutputStream**).

Flux d'entrée : InputStream



//r4cps.com/vpjava/WebRoot/io.html

Flux de sortie : OutputStream



http:

//r4cps.com/vpjjava/WebRoot/io.html

Flux d'entrée : java.io.InputStream

InputStream(classe abstraite) est la super-classe de toutes les classes de **lecture de flux d'octets**.

- **public abstract int read()** : méthode abstraite de lecture d'un octet
- **public int read(byte b[])** : lecture de plusieurs octets par appel répété à *read()*. Cette méthode retourne le nombre d'octets lus.
- **public void close()** : méthode de fermeture du flux.

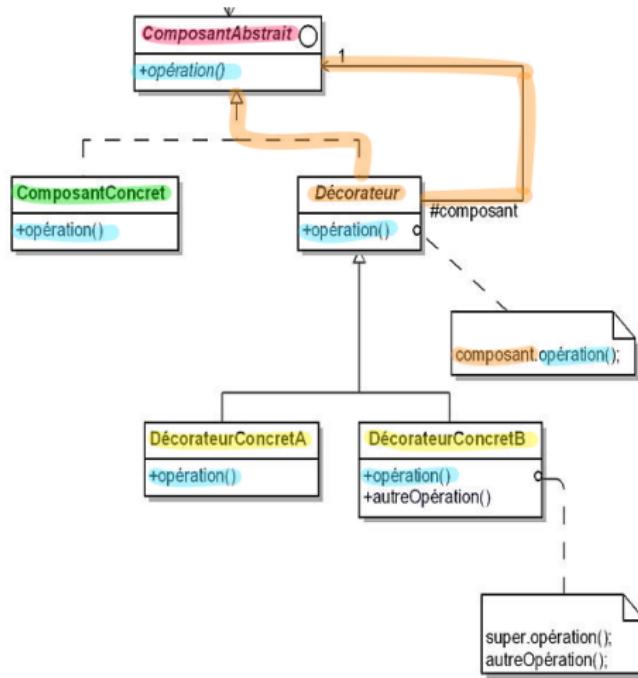
Le package `java.io` est largement basé sur le pattern Décorateur. en effet, la classe *InputStream* et ses sous-classes sont structurées suivant ce pattern.

Pattern Décorateur/Enveloppe (Decorator ou Wrapper) - Définition

Définition

Le pattern **Décorateur** attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.

Pattern Décorateur/Enveloppe (Decorator ou Wrapper)



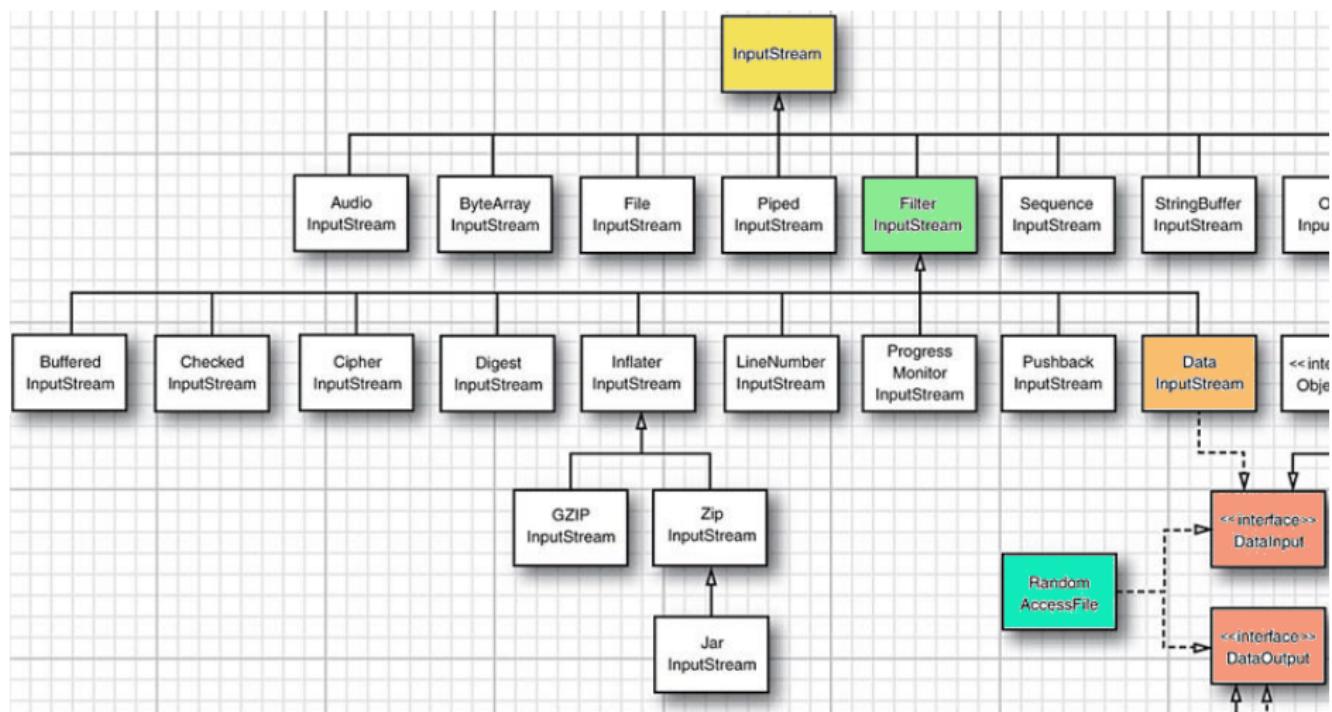
Pattern Décorateur/Enveloppe (Decorator ou Wrapper) - Participants

- **ComposantAbstrait** est l'interface commune au composant et aux décorateurs
- **ComposantConcret** est l'objet initial auquel de nouvelles fonctionnalités doivent être ajoutées
- **Décorateur** est une classe abstraite qui détient une référence vers un composant
- **DécorateurConcretA** et **DécorateurConcretB** sont des sous-classes concrètes de Décorateur qui ont pour but l'implantation des fonctionnalités ajoutées au composant.

Hiérarchie d'InputStream-pattern Décorateur



Hiérarchie d'InputStream



Sous-classes InputStream

Elles représentent les classes concrètes qui produisent une entrée depuis différentes sources.

- `FileInputStream` (fichier) : lire des données depuis un fichier
- `ByteArrayInputStream` (tableau) : lire des données à partir d'un tableau d'octets.
- `StringBufferInputStream` (chaîne de caractères) : lire une chaîne comme un flux de données

Classe FileInputStream - Composant concret

Permet de lire des octets et des tableaux d'octets dans un fichier.

- Constructeur : `FileInputStream(File fichier)`
Construction d'un flux d'entrée en provenance d'un fichier
- Méthode : `int read()`
- Méthode : `int read(byte b[])`

Classe FileInputStream - Exemple

```
File f=new File("test.txt");
try {
    FileInputStream fis = new FileInputStream(f);
    //lecture du premier byte
    int premierByte=fis.read();
    //lecture du deuxième byte
    int deuxiemeByte = fis.read();
    //Création d'un tableau de 1000 bytes
    byte tabByte[] = new byte[1000];
    //lecture et stockage de 1000 bytes dans le tableau
    fis.read(tabByte);
    //fermeture du flux
    fis.close();

} catch (FileNotFoundException e) {
} catch (IOException e) {
}
```

Classe FilterXXX - Décorateur Abstrait

- La classe abstraite "FilterXXX" est la classe de base pour tous les décorateurs.
- Pour les `InputStream` et `OutputStream`, les flux sont adaptés à des usages particuliers en utilisant des sous-classes « décoratives » de `FilterInputStream` et `FilterOutputStream`
- Deux sous-classes de `FilterInputStream`
 - `BufferedInputStream`
 - `DataInputStream`

La classe `BufferedInputStream` - Décorateur concret

- La classe `BufferedInputStream` permet de lire des octets ou des tableaux d'octets dans un fichier en évitant des accès disques ou réseaux trop nombreux.
- Cette classe se charge de lire sur le flux de données un grand nombre d'octets qu'elle garde dans un tampon (mémoire tampon). Tous les appels à la méthode `read()` ultérieurs sur une instance de cette classe renvoient des données provenant de ce tampon tant qu'il y reste des octets à lire
- Le tampon est rechargeé à partir du fichier quand on a épuisé toutes les informations qui s'y trouvent.

La classe BufferedInputStream - Décorateur concret

- constructeur : `BufferedInputStream(InputStream in)`
Le tampon a la taille 8192 par défaut.
- Constructeur : `BufferedInputStream(InputStream in, int taille)`
Le tampon a la taille *taille*

La classe BufferedInputStream - Exemple

```
public class TestFichierBuf {  
    public static void main(String[] args) {  
        File f=new File("test.txt");  
        try {  
            FileInputStream fis = new FileInputStream(f);  
  
            BufferedInputStream bis=new BufferedInputStream(fis,512);  
  
            for(int i=0;i<1000;i++){  
                System.out.println(bis.read());  
            }  
  
            bis.close();  
  
        } catch (FileNotFoundException e) {  
        } catch (IOException e) {  
        }  
    }  
}
```

La classe `DataInputStream` - Décorateur concret

La classe `DataInputStream` permet de lire des valeurs de **types primatifs** et des **String** en provenance d'un flux d'entrée.

- Constructeur : `DataInputStream (InputStream in)`
- Cette classe implémente l'interface `DataInput`, ce qui fournit les méthodes suivantes :
boolean readBoolean()
byte readByte()
char readChar()
double readDouble, ...

Sous-classes InputStream

Classe	Fonction	Arguments du Constructeur
Mode d'emploi		
ByteArray-InputStream	Autorise un tampon en mémoire pour être utilisé comme InputStream	Le tampon depuis lequel extraire les bytes.
		Comme une source de données. Connectez le à un objet FilterInputStream pour fournir une interface pratique.
StringBuffer-InputStream	Convertit un String en un InputStream	Un String . L'implémentation fondamentale utilise actuellement un StringBuffer .
		Comme une source de données. Connectez le à un objet FilterInputStream pour fournir une interface pratique.
File-InputStream	Pour lire les informations depuis un fichier.	Un String représentant le nom du fichier, ou un objet File ou FileDescriptor .
		Comme une source de données. Connectez le à un objet FilterInputStream pour fournir une interface pratique.

Sous-classe InputStream

Piped-InputStream	Produit la donnée qui sera écrite vers le PipedOutput-Stream associé. Applique le concept de « tuyauterie ».	PipedOutputStream
Sequence-InputStream	Convertit deux ou plusieurs objets InputStream dans seul InputStream .	Deux objets InputStream ou une Énumération pour un récipient d'objets InputStream .
Filter-InputStream	Classe abstraite qui est une interface pour les décorateurs lesquels fournissent une fonctionnalité profitable aux autres classe InputStream . Voir Tableau 11-3.	Voir Tableau 11-3. Voir Tableau 11-3.

Sous-classes de FilterInputStream

classe	Fonction	Arguments du constructeur
		Mode d'emploi
Data-InputStream	Employé de concert avec DataOutputStream , afin de lire des primitives (int , char , long , etc.) depuis un flux de manière portable.	InputStream Contient une interface complète vous permettant de lire les types de primitives.
Buffered-InputStream	Utilisez ceci pour empêcher une lecture physique chaque fois que vous désirez plus de données. Cela dit « Utiliser un tampon. »	InputStream , avec en option la taille du tampon. Ceci ne fournit pas une interface en soi, mais une condition permettant d'employer le tampon.
LineNumber-InputStream	Garde trace des numéros de ligne dans le flux d'entrée; vous pouvez appeler getLineNumber() et setLineNumber(int) .	InputStream Cela n'ajoute que la numérotation des lignes, de cette façon on attachera certainement un objet interface.
Pushback-InputStream	Possède un tampon retour-chariot d'un byte permettant de pousser le dernier caractère lu en arrière.	InputStream Généralement employé dans le scanner pour un compilateur et probablement inclus parce qu'il était nécessaire au compilateur Java. Vous ne l'utiliserez probablement pas.

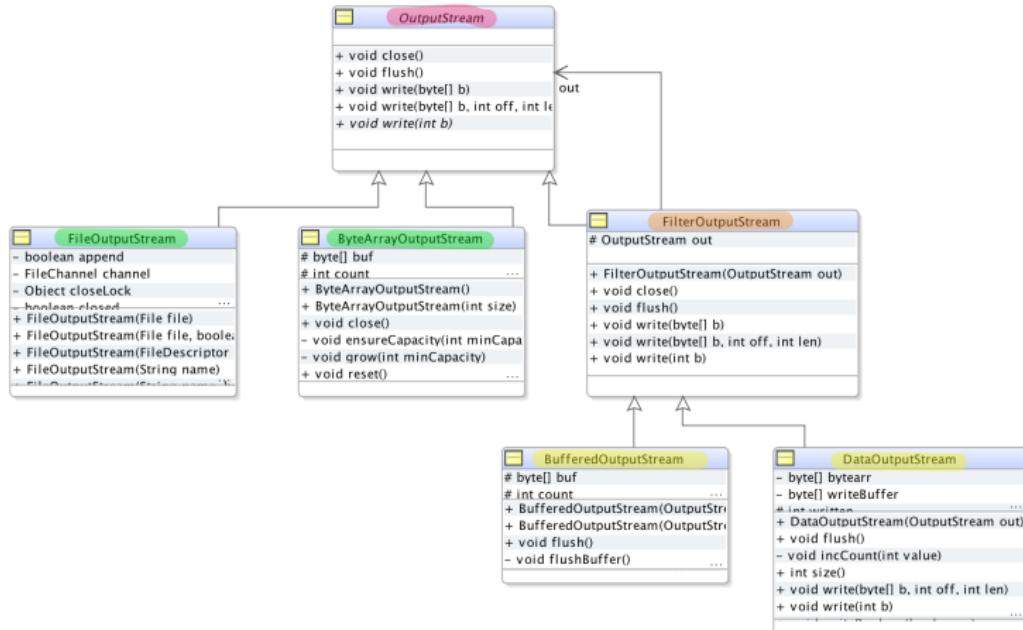
Flux de sortie : `java.io.OutputStream`

OutputStream(classe abstraite) est la super-classe de toutes les classes d' **écriture de flux d'octets**.

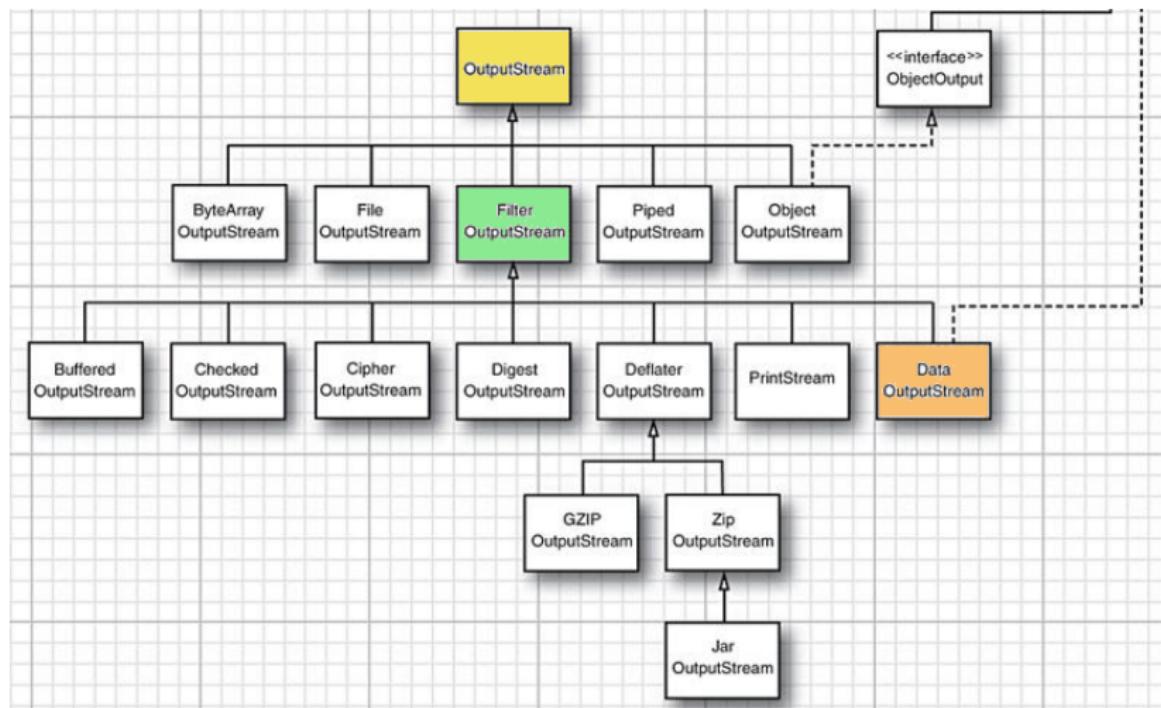
- **public abstract void write(int b)** : méthode abstraite d'écriture d'un octet, Cette méthode prend un paramètre de type *int*, seuls les **8 bits de poids faible** du paramètre sont pris en compte.
- **public void write(byte b[])** : écriture de plusieurs octets par appel répété de *write()*. Cette méthode retourne le nombre d'octets lus.
- **public void close()** : méthode de fermeture du flux.

La classe *OutputStream* et ses sous-classes sont structurées suivant le pattern Décorateur.

Hiérarchie d'OutputStream-pattern Décorateur



Hiérarchie OutputStream



Classe OutputStream

- `FileOutputStream` (fichier) : écrire des données depuis un fichier
- `ByteArrayOutputStream` (tableau) : écrire des données vers un tableau d'octets.
- `StringBufferOutputStream` (chaîne de caractères) : écrire une chaîne comme un flux de données

Sous-classes OutputStream

Classe	Fonction	Arguments du constructeur
		Mode d'emploi
ByteArray-OutputStream	Crée un tampon en mémoire. Toutes les données que vous envoyez vers le flux sont placées dans ce tampon.	En option la taille initiale du tampon. Pour désigner la destination de vos données. Connectez le à un objet FilterOutputSteam pour fournir une interface pratique.
File-OutputStream	Pour envoyer les informations à un fichier.	Un String représentant le nom d'un fichier, ou un objet File ou FileDescriptor . Pour désigner la destination de vos données. Connectez le à un objet FilterOutputSteam pour fournir une interface pratique.
Piped-OutputStream	N'importe quelle information que vous écrivez vers celui-ci se termine automatiquement comme une entrée du PipedInput-Stream associé. Applique le concept de « tuyauterie. »	PipedInputStream Pour indiquer la destination de vos données pour une exécution multiple [multithreading]. Connectez le à un objet FilterOutputSteam pour fournir une interface pratique.

Sous-classes OutputStream

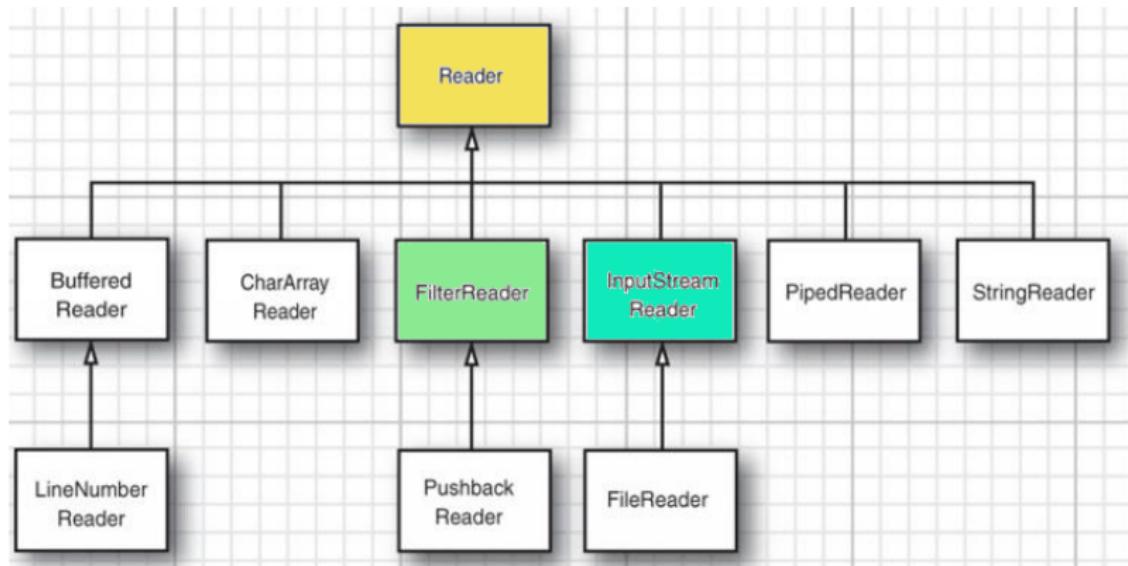
Filter-OutputStream	Classe abstraite qui est une interface pour les décorateurs qui fournissent des fonctionnalités pratiques aux autres classes d' OutputStream . Voir Tableau 11-4.	Voir Tableau 11-4. Voir Tableau 11-4.
----------------------------	---	--

Classe Reader et Writer

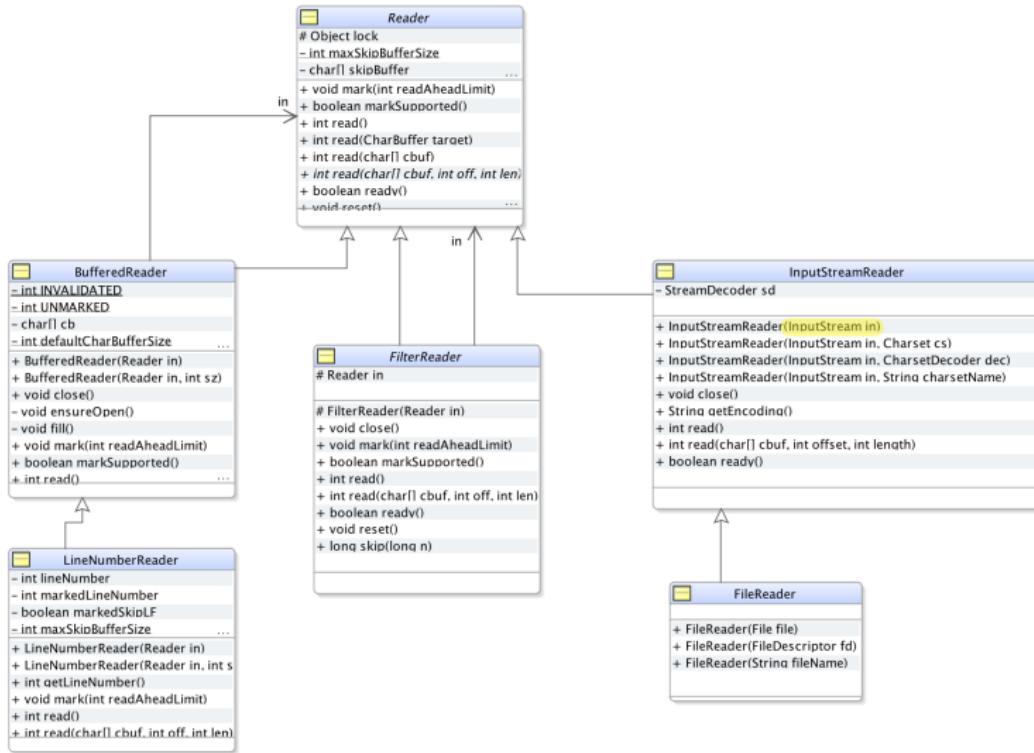
- Les Reader et les Writer sont des flux de caractères(char) (InputStreams/OutputStreams flux d'octets)
- Supporte nativement l'**Unicode (16 bits)** dans toutes les opérations d'E/S
- Opérations plus rapides que InputStream/OutputStream
- La classe Reader déclare une méthode abstraite de lecture de caractères. Retourne le nombre de caractères lus, ou -1 si la fin du flux est atteinte.
- La classe Writer déclare une méthode abstraite d'écriture de caractères.

La classe *Reader/Writer* et ses sous-classes sont structurées suivant le pattern Décorateur.

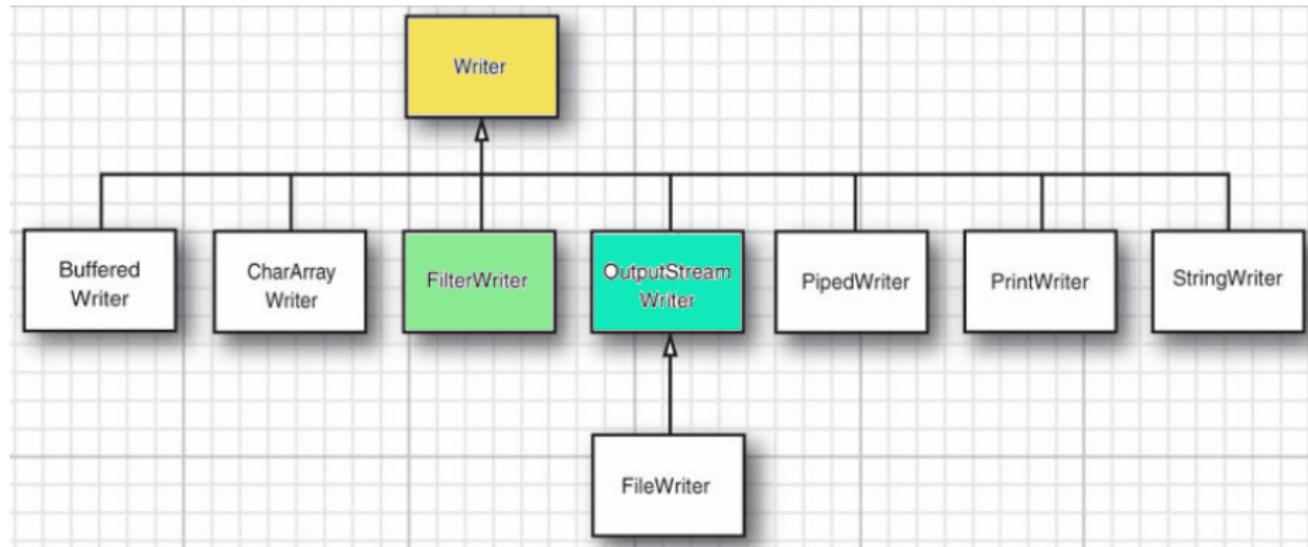
Hiérarchie Reader



Hiérarchie Reader



Hiérarchie Writer



Correspondance InputStream/OutputStream – Reader/Writer

Sources & Récipients: Classe Java 1.0	Classe Java 1.1 correspondante
InputStream	Reader convertisseur : InputStreamReader
OutputStream	Writer convertisseur : OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(pas de classe correspondante)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Modifier le comportement du flux Reader/Writer

Filtres : classe Java 1.0	Classe correspondante en Java 1.1
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (classe abstract sans sous-classe)
BufferedInputStream	BufferedReader (a aussi <code>readLine()</code>)
BufferedOutputStream	BufferedWriter
DataInputStream	Utilise DataInputStream (sauf quand vous voulez utiliser <code>readLine()</code> , alors vous devez utiliser un BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer (utilise un constructeur qui prend un Reader à la place)
PushBackInputStream	PushBackReader

Classes "passerelles"

Les classes « passerelles » :

- InputStreamReader convertit un InputStream en un Reader
- OutputStreamWriter convertit un OutputStream en un Writer

Classes FileReader et FileWriter

Les classes `java.io.FileReader` et `java.io.FileWriter` permettent respectivement la lecture et l'écriture d'un fichier sous la forme d'un flux de caractères.

Pour écrire un *String*, il faut écrire chaque caractère les uns après les autres !

- `BufferedReader/BufferedWriter` : permet d'optimiser la lecture/l'écriture dans un fichier.
- `PrintWriter` : formate les données afin qu'elles soient plus lisibles (écriture de *String*).

L'accès aléatoire aux fichiers (RandomAccessFile)

- L'abstraction fournie par les flux est utile lorsqu'un programme doit lire de façon séquentielle la totalité d'un fichier.
- Certaines applications nécessitent de pouvoir aller lire à n'importe quel endroit dans un fichier ou nécessitent de pouvoir retourner en arrière. On parle d'accès aléatoire à un fichier (random access). L'utilisation de flux n'est pas appropriée pour ces applications.
- Pas de lien avec les classes InputStream/OutputStream et Reader/Writer

Classe RandomAccessFile

- Pour construire une instance de la classe RandomAccessFile, il faut un fichier (classe File) ou un nom de fichier ainsi qu'un mode d'accès.
- Le mode d'accès est soit la lecture uniquement (« r ») soit la lecture et l'écriture (« rw »).
- La classe RandomAccessFile fournit des méthodes permettant la lecture et l'écriture, de la même façon que les flux (basé sur les interfaces DataInput et DataOutput).

boolean readBoolean() ou void writeBoolean(boolean v)

byte readByte() ou void writeByte(int v)

char readChar() ou void writeChar(int v)

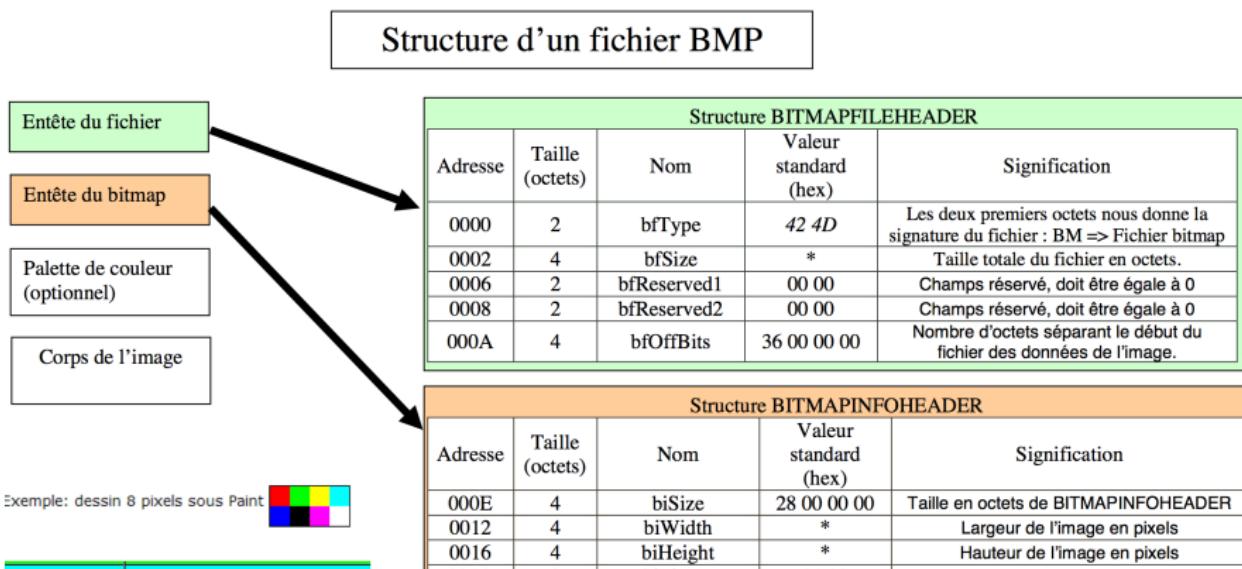
double readDouble ou void writeDouble(double v), ...

Classe RandomAccessFile - Méthodes

La classe `RandomAccessFile` fournit également des méthodes pour déplacer le pointeur de fichier. Le pointeur de fichier maintient la position du prochain octet à lire ou écrire.

- `seek()` : permet de se déplacer d'un enregistrement à un autre dans le fichier
- `getFilePointer()` : permet de connaître où on se trouve dans le fichier
- `length()` : permet de déterminer la taille maximum du fichier.

Classe RandomAccessFile - Exemple



Classe RandomAccessFile - Exemple

```
public static void main(String[] args){  
    try {  
        RandomAccessFile raf = new RandomAccessFile("chat.bmp", "r");  
        raf.seek(2);  
        long tailleFichier=readUInt(raf);  
        System.out.println("Taille du fichier = "+tailleFichier+" octets");  
  
        raf.seek(raf.getFilePointer()+12);  
        long largeur=readUInt(raf);  
        System.out.println("Largeur = "+largeur+" pixels");  
  
        long hauteur=readUInt(raf);  
        System.out.println("Hauteur = "+hauteur+" pixels");  
    } catch (FileNotFoundException e) {  
    } catch (IOException e) {  
    }  
}  
  
}  
  
public static long readUInt(RandomAccessFile raf){  
    long intNonSigne=0L;  
    try {  
        long byte0 = raf.read();  
        long byte1=raf.read();  
        long byte2=raf.read();  
        long byte3=raf.read();  
        intNonSigne=(byte3<<24)+(byte2<<16)+(byte1<<8)+(byte0<<0);  
  
    } catch (IOException e) {  
    }  
    return intNonSigne;  
}
```

Usage

typique des flux d'E/S

Usage typique

① Lecture/écriture d'un fichier texte

- Lecture : FileReader + BufferedReader (décorateur)
- Ecriture : FileWriter + BufferedWriter (décorateur) + PrintWriter (décorateur)

② Stocker et récupérer des données : DataInputStream / DataOutputStream

- Lecture : FileInputStream +
BufferedInputStream (décorateur) +
DataInputStream (décorateur)
- Ecriture : FileOutputStream + BufferedOutputStream
(décorateur) + DataOutputStream (décorateur)

③ Accès aléatoire en lecture et écriture : RandomAccessFile

Multithreading

Utilité des threads

- Méthodes d'E/S (flux fichier, mémoire, clavier, réseau, ...)
 - Utilisation de la méthode *System.in.read()* : tant que l'utilisateur n'a pas saisi un caractère, le programme ne passe pas à l'instruction suivante. => instruction I/O bloquante !
 - permet pour un serveur de gérer plusieurs clients
- Gestion temporiseurs (Timers)
- Tâches longues et consommatrices de temps ou des tâches répétitives.
 - exemple : animation, gros calculs, ...
- Tâches indépendantes

Pocessus (Tâche)-Définition

Une des fonctionnalités de l'OS est la **gestion des processus**.

Un processus peut être défini comme " une instance de programme en cours d'exécution" :

- Le programme est un **code statique** unique (une suite d'instructions) qui est chargé en mémoire centrale à partir du moment où il doit être exécuté ;
- L'exécution d'un programme correspond à **l'exécution séquentielle de ses instructions**(fil d'exécution=thread) ; on utilise un compteur ordinal (instruction pointer) pour mémoriser, lors de l'exécution du programme, l'adresse de la prochaine instruction à exécuter ;

Processus (Tâche)-Définition (suite)

- Un processus désigne une exécution particulière d'un programme : le même programme peut être exécuté par plusieurs processus "en même temps" ; Par contre leurs **segments de données sont séparés** parce que deux utilisateurs ne manipulent pas le même texte ni les mêmes commandes du traitement de texte.
- Chaque exécution correspond à une instance différente et donc à un processus différent.

Processus (Tâche)

Un processus regroupe principalement :

- le **code** du programme : un espace mémoire contenant les instructions du programme ;
- un espace mémoire pour les **données** de travail (variables, pile, tas) ;

L'OS attribue à chaque processus un identifiant (PID, Process IDentifier en anglais).

Multitâche

Un système d'exploitation multitâche réattribue périodiquement (quantum de temps de l'ordre du centième de seconde) au CPU une tâche (processus) différente dans le but de faire progresser l'exécution de plusieurs programmes à la fois.

- l'**ordonnanceur**(scheduler) **de tâche** est un composant de l'OS choisissant l'ordre d'exécution des processus sur le processeur d'un ordinateur.
=> chef d'orchestre + arbitre
- L'utilisateur a l'**impression** que plusieurs programmes sont exécutés "simultanément".

Processus vs Thread

- Inconvénients des processus :
 - Le passage d'un processus à un autre est beaucoup trop lent.
 - La communication entre les processus nécessite de recourir à des techniques sophistiquées (échange de données).
- Les threads (fil/flux/ chemin d'exécution)
 - Basculement d'un thread à un autre est beaucoup plus rapide.
 - Étant donné que les threads partagent leur mémoire :
=>la communication entre threads dans un même processus est plus efficace.

Processus multithread (multithreading)

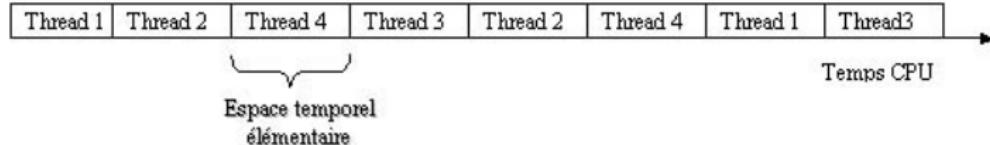
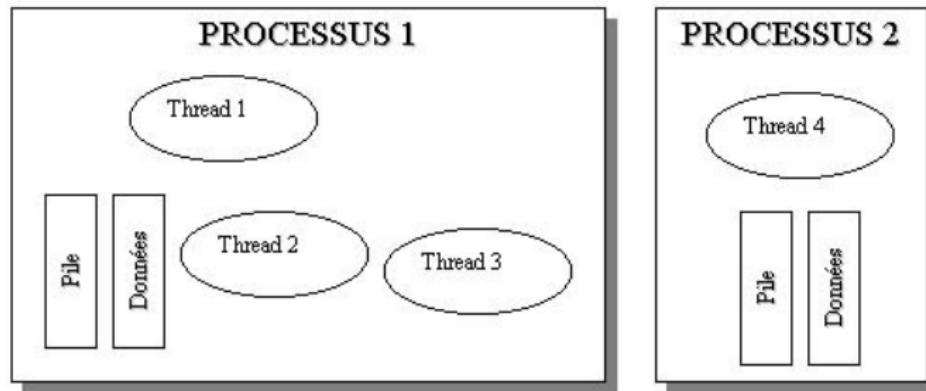
Multithreading : exécution à l'intérieur d'un processus de plusieurs sous-tâches (exécution concurrente de threads)

- Un processus peut posséder plusieurs threads.
- Les ressources allouées à un processus (temps processeur, mémoire) sont partagées entre les threads qui le composent.
 - lorsqu'un thread modifie une variable (non locale à lui), tous les autres threads voient la modification
 - un fichier ouvert par un thread est accessible aux autres threads (du même processus)
- Un processus possède au moins un thread (qui exécute le programme principal, habituellement la méthode *main()*).

Processus multithread (multithreading)

Sur un ordinateur (monoprocesseur), à un instant donné, il n'y a qu'**UN SEUL THREAD** en cours d'exécution, et éventuellement d'autres threads en attente d'exécution.

Processus à un thread et à plusieurs threads



Exemple

Le processus MS-Word peut impliquer plusieurs threads :

- Interaction avec le clavier.
- Rangement de caractères sur la page.
- Sauvegarde régulière du travail fait.
- Contrôle orthographe, etc.

Création de Threads

En Java, les **threads** font partie intégrante du langage (**pas d'import**).

Deux méthodes :

- **Dériver une classe de *Thread***

=> quand on n'a pas besoin d'hériter d'une autre classe.

Méthode *run()* à définir : code qui sera exécuté dans le thread

Le seul objet capable de lancer un thread !

- **Implémenter l'interface *Runnable***

=> quand on souhaite hériter d'une autre classe.

Méthode (abstraite) public void *run()* : code qui sera exécuté dans le thread passer l'objet Runnable au constructeur du Thread

Exécution d'un Thread

- *Runnable* est à un thread ce qu'un travail est à un travailleur.
- Méthode *start()* : démarre le fil d'exécution

Création de Threads : classe Thread

Créer une classe qui hérite de la classe Thread.

```
class MonThread extends Thread {  
    public void run {  
        ... mettre ici le code que le thread doit executer...  
    }}  
}
```

L'utiliser :

```
MonThread th = new MonThread(); -> creer un nouveau thread  
th.start(); -> demarrer le thread (etat executable)
```

Création de Threads : interface Runnable

Créer une classe qui hérite de la classe Thread.

```
class MaClasse implements Runnable {  
    public void run {  
        ... mettre ici le code que le thread doit executer...  
    } }
```

L'utiliser :

```
MaClasse mc = new MaClasse();  
Thread th= new Thread(mc);  
th.start();
```

Terminaison d'un thread

- **Arrêt naturel :**

Un thread s'arrête lorsqu'il atteint la fin de sa méthode *run()* et libère les ressources qui lui ont été allouées.

- **Arrêt prématué :**

Utilisation de la méthode *stop()* : elle est vigoureusement désapprouvée (*deprecated*), car elle laisse dans un état indéterminé certains des objets qui dépendent du thread stoppé.
La manière propre et fiable de terminer un thread consiste à modifier la valeur d'une variable(flag) que le thread consulte régulièrement.