



Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

01 Introduction IHM

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Introduction

Interface Homme-Machine

- L'étude de l'**Interface Homme-Machine (IHM)** appelée également **Interaction Homme-Machine (Human Computer Interaction ou HCI)** est au confluence de **deux disciplines** :
 - Les sciences cognitives (psychologie, ergonomie cognitive, physiologie, ...)
 - Les techniques informatiques (affichage, capture et traitement des événements, ...)
- Il y a donc **deux vues** distinctes du sujet :
 - Les aspects psychologiques et physiologique de la communication entre un humain et une machine
 - Les aspects techniques de la réalisation (discipline informatique)
- Au sens large, une **interface** est un dispositif qui sert de **limite commune** à plusieurs entités qui communiquent.
- Dans le cas de l'interface homme-machine, la zone de communication (interface) se situe entre :
 - La représentation externe de l'état de la machine (image du système)
et
 - Les organes sensoriels (perception) et moteurs (actions) de l'utilisateur



Interface Homme-Machine 1 / Introduction

Éléments de l'interface homme-machine

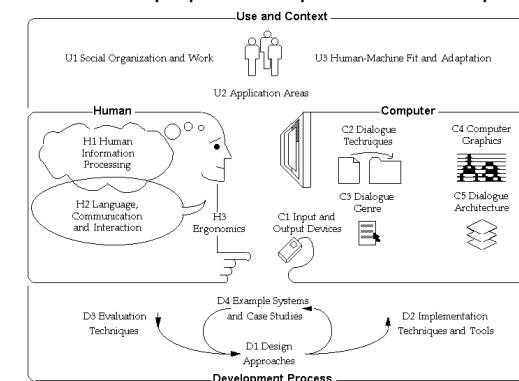
- Dans l'étude des interfaces homme-machine il y a donc trois éléments à considérer : l'**utilisateur** (l'homme), l'**ordinateur** (la machine) et leur **manière de communiquer** (l'interface).
- Une bonne connaissance des ces trois domaines est donc nécessaire à la réalisation d'une interface homme-machine réussie.
- En plus de ces trois éléments, il faut naturellement avoir une très bonne **connaissance du domaine** qu'est censé couvrir l'application informatique (espace du problème).
- L'interface homme-machine fait également intervenir une part de subjectivité car, en plus de la **composante technique**, il y a également une **composante artistique/esthétique** (la disposition des composants, le choix des couleurs, les sons, les images, etc) qui peut être appréciée de différentes manières selon les goûts et les préférences de chaque personne.
- Certaines **règles de base**, communément admises, sont cependant indépendantes de cet aspect subjectif.



Interface Homme-Machine 1 / Introduction

Éléments de l'interface homme-machine

- Une interface homme-machine s'inscrit toujours dans un certain **contexte d'utilisation** qui peut être plus ou moins spécialisé.



- Le développement d'une interface utilisateur est généralement un **processus itératif** avec création de **maquettes (prototypes)** qui sont soumises à l'appréciation de l'utilisateur puis corrigées/adaptées.





L'homme / L'utilisateur

- Un **élément essentiel** (prioritaire) du trio interface homme-machine (l'ordinateur est au service de l'Homme et non le contraire)
 - Quels sont ses besoins, ses attentes (phase d'analyse) ?
 - Quelles sont ses capacités physiques, ses limitations (handicaps) ?
 - Quelles sont ses capacités intellectuelles, ses connaissances du domaine traité (contexte d'utilisation) ?
- D'une manière générale, il s'attend à avoir une **rétroaction immédiate (feedback)** comme lorsqu'il utilise des objets et machines du monde réel (ascenseur, voiture, téléviseur, etc.)
- Il peut communiquer (percevoir et agir) de plusieurs manières à l'aide de ses différents sens (vue, toucher, mouvement, voix, regard, etc)
- Connaître certains éléments du système sensoriel humain est important pour concevoir une bonne interface.

Il faut savoir, par exemple, qu'un humain est très facilement distrait par les plus petits mouvements à la périphérie de son système de vision. Le clignotement et le mouvement doivent être réservés aux informations urgentes ou extrêmement importantes (sur le web, les publicitaires l'ont bien compris ;-).

EIA-FR / Jacques Bapst

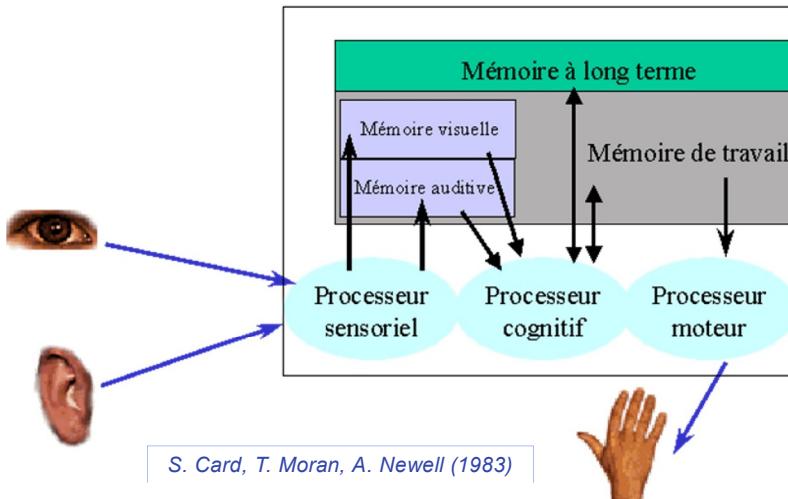


IHM1_01

5



Le modèle du processeur humain



EIA-FR / Jacques Bapst



IHM1_01

6



La machine / L'ordinateur

- Pour un humain, c'est une **boîte noire** :
 - Pas de *feedback* implicite (il faut le programmer explicitement)
 - Les informations contenues ne sont pas directement visibles (opacité)
 - Le déroulement des opérations ne peut pas, sans autre, être observé
- Le mode de fonctionnement n'est pas le même que le nôtre :
 - [+] Peut mémoriser un grand nombre d'informations et les retrouver rapidement et sans pertes
 - [+] Peut exécuter des instructions autant de fois que nécessaire sans erreurs (pas de notion de fatigue)
 - [-] Incapable de faire certaines choses qui nous paraissent triviales (tirer des conclusions, laisser tomber certains éléments négligeables au profit d'éléments plus importants, comprendre le langage naturel même des phrases simples, corrélérer des informations, etc.)
- Tout doit être programmé, même les actions les plus élémentaires.
 - Naturellement, avec un considérable effort de programmation, on peut améliorer la situation et tendre vers une certaine *intelligence artificielle* dont les objectifs visent à rapprocher la machine de l'humain.

EIA-FR / Jacques Bapst



IHM1_01

7



L'interface / L'interaction

- Connaissant les différences (importantes) entre l'homme et la machine comment trouver un **langage commun** pour communiquer au mieux ?
- C'est l'**interface utilisateur** qui est chargé de cette tâche.
- L'**interface utilisateur** est constituée par les parties matérielles et logicielles de l'ordinateur dont l'utilisateur se sert pour voir, entendre, toucher, parler, ... et ainsi communiquer (informations bidirectionnelles) avec la machine (et ses logiciels).
- C'est tout l'art du programmeur d'établir le lien entre ce que l'utilisateur souhaiterait et ce que les techniques informatiques peuvent offrir.
- Il faut trouver le bon compromis entre **ce qui peut être programmé** (dans un temps adapté et avec un coût raisonnable) et **ce que l'utilisateur souhaiterait** dans l'idéal.
- Dans de nombreux cas, c'est une **tâche difficile** mais très **importante** car **c'est souvent sur son l'interface utilisateur qu'un logiciel est jugé**.
- La qualité de l'interface utilisateur peut conditionner la réussite ou l'échec d'un logiciel (indépendamment de la qualité de ses fonctionnalités).

EIA-FR / Jacques Bapst



IHM1_01

8



Pérophériques d'entrée

- Les **pérophériques d'entrée** (*Input Devices*) sont des composants matériels qui sont utilisés pour "parler" à l'ordinateur.
- Les pérophériques d'entrée les plus courants sont :
 - Clavier
 - Souris (2D, 3D)
 - Touchpad / Trackstick
 - Écran tactile (*multi-touch*)
 - Boule de positionnement (*Trackball*)
 - Manette de jeu (*Joystick*)
 - Microphone
 - Tablette graphique (*Pen Pad*)
 - Manette *Wii, Kinect, ...*
- Le **clavier** et la **souris** (ainsi que ses substituts : *touchpad et autres*) constituent les pérophériques d'entrée principaux dans les applications informatiques classiques.



Interfaces graphiques (GUI)

- Dans la majorité des applications, la partie la plus importante de l'interaction homme-machine fait appel aux pérophériques de base qui sont : l'**écran**, le **clavier** et la **souris**.
- Progressivement, les interfaces en **mode caractères** (lignes de commandes entrées au clavier) ont été substituées par des interfaces utilisant le mode graphique (**Graphical User Interface** ou **GUI**).
- Les **interfaces graphiques** utilisent différents **éléments visuels** appelés **Composants graphiques** ou **Widgets** ou **Controls** et qui, en combinaison avec l'utilisation de la souris rendent les choses plus simples et plus intuitives pour l'utilisateur.
- Parmi ces **éléments visuels** on peut citer :
 - Les **boutons** (cases à cocher, bouton radio, etc)
 - Les **menus** (menus déroulants, menus contextuels, etc)
 - Les **listes à choix** (listes simples, listes déroulantes)
 - Les **champs de texte** (champs simples, champ multilignes, éditeurs, etc)
 - Les **barres de défilement** (ascenseurs)
 - ...



Les pérophériques de sortie

- Les **pérophériques de sortie** (*Output Devices*) sont des composants matériels que l'ordinateur utilise pour communiquer avec l'utilisateur (pour l'informer).
- Les principaux pérophériques de sortie sont les suivants :
 - Écran
 - Imprimante
 - Haut-parleurs (carte audio)
 - Écouteurs (carte audio)
 - Manette de jeu (*Joystick*) avec rétroaction (retour de force)
 - ...
- L'**écran** et l'**imprimante** (sortie "papier", documents imprimés) constituent les pérophériques de sortie principaux dans les applications informatiques classiques.



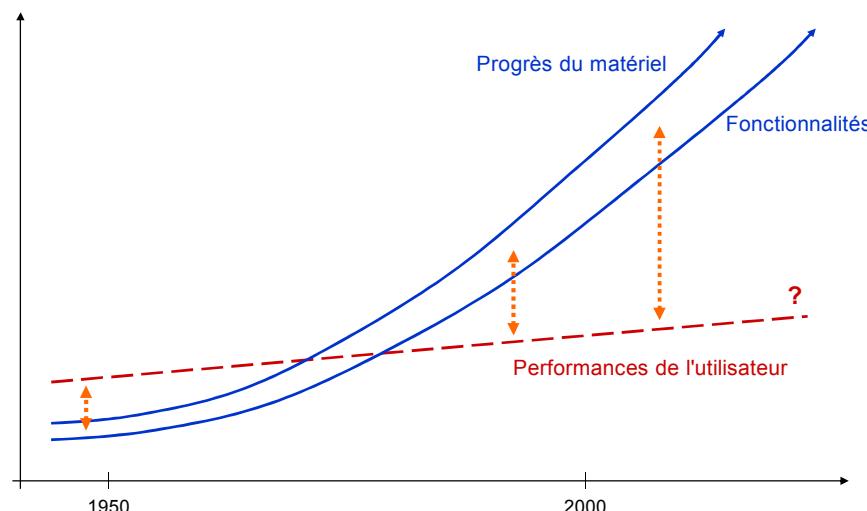
WIMP

- Les interfaces utilisateur qui intègrent ces éléments graphiques (composants visuels / *Widgets* / *Controls*) sont parfois désignées par l'acronyme **WIMP** qui est l'abréviation de :
 - **Window** : Notion de "fenêtre" (zone d'interaction indépendante) (c'est une notion importante qui a même donné son nom à un système d'exploitation !)
 - **Icon** : Éléments graphiques visuels (images, boutons, champs de texte, bulles d'aide, etc.)
 - **Menu** : Choix d'actions parmi une liste proposée (barres de menu, menus déroulants, menus contextuels, rubans, etc.)
 - **Pointer** : Curseur/Pointeur manipulé par la souris, et qui permet d'interagir avec les composants visuels (pointage, sélection, tracé, drag & drop)
- Avec le clavier, la souris est l'élément d'entrée le plus important.



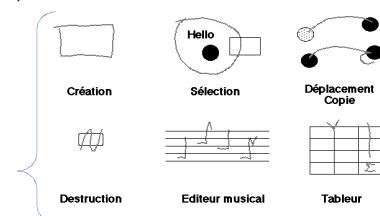


Évolution de l'interactivité



Styles d'interaction

- Différents **styles d'interaction** (ou **paradigmes d'interaction**) sont envisageables pour la communication entre l'homme et la machine .
- Ils font appel aux différents canaux d'entrée-sortie dont nous disposons
 - En entrée : nos cinq sens (vue, ouïe, toucher, odorat, goût)
 - En sortie : nos *actuateurs* (membres, doigts, yeux, tête, voix, ...)
- Quelques styles d'interaction classiques :
 - Style conversationnel (langage de commandes, *shell-scripts*, ...)
 - Interaction par formulaires ou tableurs
 - Sélection par menus (plus nécessaire de se souvenir des commandes)
 - Navigation hyper-textuelle (point-and-click)
 - WIMP (ensemble de styles d'interaction)
 - Manipulation directe (édition WYSIWYG, interaction iconique)
 - Langage naturel écrit ou parlé (*query language*, commande vocale)
 - Interaction gestuelle
 - ...



Manipulation directe

- Le style d'interaction appelé "**Manipulation directe**" caractérise un ensemble de propriétés qui donne à l'utilisateur le sentiment d'un engagement direct dans l'action qu'il effectue.
- Trois principes directeurs :
 - Représentation permanente** des objets (ceux qui ont un sens pour l'utilisateur)
 - Utilisation d'**actions physiques** pour effectuer les opérations (via un dispositif de pointage comme la souris)
 - Opérations incrémentales, réversibles et à effet visible
- Ben Schneiderman (1983) :

"Point and click instead of remember and type"
- L'inconvénient principal de ce style d'interaction est qu'il ne permet pas facilement la programmation des actions par l'utilisateur.
- On parle parfois de "**Manipulation indirecte**" lorsque les actions sont déclenchées en passant par des menus et des boîtes de dialogue.



Métaphores

- L'ordinateur ne fonctionne pas de la même manière que les humains et ses données, ses actions ne sont pas directement visibles.
- Il est donc important que l'utilisateur puisse se faire une **représentation mentale** (un **modèle**) du fonctionnement de la machine et des logiciels qu'il utilise.
- Un bon moyen d'offrir une interface qui permette à l'utilisateur de se forger une représentation mentale (un modèle du comportement) est de tenter d'**imiter le fonctionnement et les réactions de certains objets du monde réel** (que l'utilisateur est sensé bien connaître).
- On parle alors de **métaphore**, c'est-à-dire qu'on crée des éléments graphiques qui ont une **analogie** (présentation, comportement) **avec des objets du monde réel**.
- La **métaphore du bureau** est largement utilisée dans les interfaces graphiques. On représente et manipule des objets qui imitent le travail au bureau (dossiers, documents, poubelles, boîtes aux lettres, carnets de notes, horloges, etc.).





Look and Feel

- La notion de "**Look and Feel**" est caractérisée par un ensemble cohérent de propriétés de l'interface graphique. Il inclut notamment :
 - La manière de présenter les composants graphiques (**aspect**) :
 - ✓ Forme
 - ✓ Couleur
 - ✓ Texture
 - La manière dont les composants réagissent (**comportement**) :
 - ✓ Sélection
 - ✓ Exécution
 - ✓ Déplacement
 - ✓ Menus contextuels
 - ✓ Bulles d'aides
 - ✓ Sons
- Le **Look and Feel** regroupe donc un ensemble de caractéristiques qui peuvent parfois être définies par l'utilisateur (choix de **thèmes**, de **peaux** [**skin**], etc.) mais dont certaines sont déterminées par le système d'exploitation et son sous-système de fenêtrage (*Windows, MacOS, Motif, ...*).





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

02

Introduction aux librairies AWT et Swing Palette de composants / Propriétés

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Introduction AWT et Swing

Plate-forme Java

- La **plate-forme Java** est composée d'un ensemble de librairies et d'outils qui constituent les briques de base (*Core Libraries*) sur lesquelles on construit les applications. Ces librairies sont disponibles dans toutes les implémentations des machines virtuelles Java (indépendamment du système d'exploitation et du type de machine cible).
- Certaines de ces librairies regroupées sous le nom **User Interface Toolkits** (anciennement nommées **JFC** pour **Java Foundation Classes**) sont destinées à la gestion des interfaces utilisateur graphiques (GUI), aux graphiques 2D, à l'impression, ainsi qu'à différentes tâches associées (*Cut&Paste*, accessibilité, etc.).
- Ces librairies contiennent les éléments de base qui servent à écrire des applications Java qui interagissent avec l'utilisateur (applications de type *client* ou *desktop* par opposition aux applications de type *serveur*).
- La gestion des interfaces utilisateur graphiques repose principalement sur deux librairies de classes : **AWT** et **Swing**.

EIA-FR / Jacques Bapst



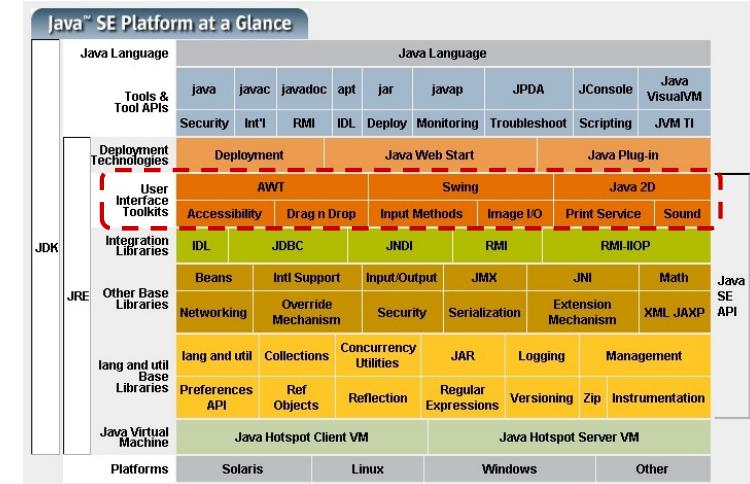
IHM1_02

2



Interface Homme-Machine 1 / Introduction AWT et Swing

Architecture de la plate-forme Java



EIA-FR / Jacques Bapst



IHM1_02

3



Interface Homme-Machine 1 / Introduction AWT et Swing

Librairie AWT

- La librairie **AWT** (*Abstract Window Toolkit*) a été introduite dès les premières versions de Java (1.0, 1.1).
- AWT** offre des fonctionnalités assez rudimentaires en terme d'interface utilisateur.
- La librairie **AWT** constitue malgré tout le socle de base sur lequel ont été élaborées les nouvelles fonctionnalités introduites avec la plate-forme Java 2 (versions 1.2 et suivantes).
- L'écriture d'**Applets** fait encore fréquemment appel à la librairie **AWT** car les navigateurs ne supportent souvent pas les librairies plus récentes (par exemple Swing) sans installation de *Plug-In*.
- La plate-forme Java *Micro-Edition* (J2ME) fait également appel aux composants **AWT** pour la gestion de l'interface utilisateur des assistants personnels (PDA), smartphones, micro-contrôleur, ...
- La classe **Graphics2D** qui comprend toutes les classes permettant de gérer des graphiques bidimensionnels (2D) fait également partie de la librairie **AWT**.

EIA-FR / Jacques Bapst



IHM1_02

4



Librairie Swing

- La librairie **Swing** constitue un kit de développement d'interfaces utilisateur graphiques.
- La librairie **Swing** repose sur la librairie AWT mais offre un grand nombre de **nouveaux composants** pour la création et la gestion de l'interface utilisateur (plus proche des applications natives).
- Swing** offre la possibilité de configurer la représentation visuelle et le comportement des composants (*Pluggable Look-and-Feel*).
- L'application peut ainsi prendre un look purement Java (indépendant de la plate-forme cible) ou alors mimer l'aspect et le comportement du système de fenêtrage de la plate-forme d'exécution (par ex. *Windows*, *Macintosh*, *Motif*, etc.).
- Bien que les classes et sous-paquets de **Swing** se trouvent dans la branche `javax.swing`..., la librairie **Swing** fait partie intégrante des librairies de base (*core libraries*) de la plate-forme Java.



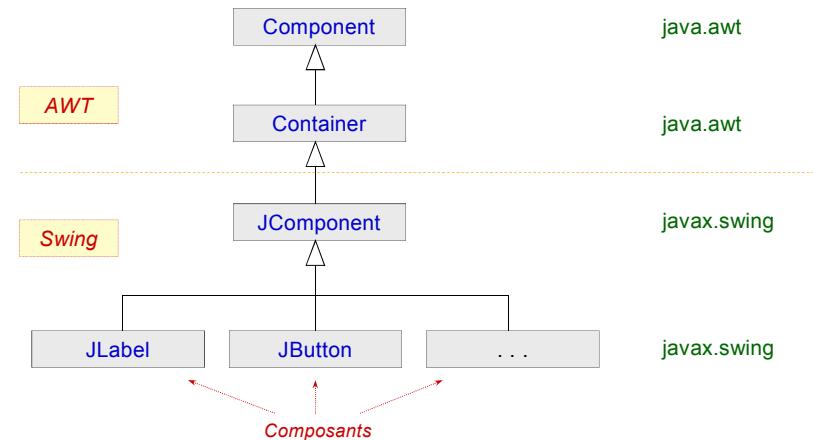
Composants

- Une interface utilisateur graphique (GUI) est composée **d'éléments de base** comme des boutons, des ascenseurs, des menus déroulants, des champs de texte, etc.
- Ces éléments de base sont appelés "**Composants**" en Java. Dans certains environnements, ces éléments sont appelés "**Widgets**" ou "**Controls**".
- La liste des composants à disposition constitue une caractéristique importante d'un kit de développement d'interfaces graphiques.
- En Java, on distingue des **composants légers** (*Lightweight*) et des **composants lourds** (*Heavyweight*).
 - Les composants lourds sont basés sur du code natif de la plate-forme d'exécution (*Peer Component*).
 - Les composants légers sont entièrement écrits en Java et sont donc indépendants de la plate-forme d'exécution.
- Excepté quelques composants de haut-niveau, pratiquement tous les composants Swing sont des composants légers (contrairement à AWT).



Classes de base des composants

- Pratiquement tous les composants Swing héritent de la classe **JComponent** qui elle-même hérite des classes AWT **Container** et **Component**.



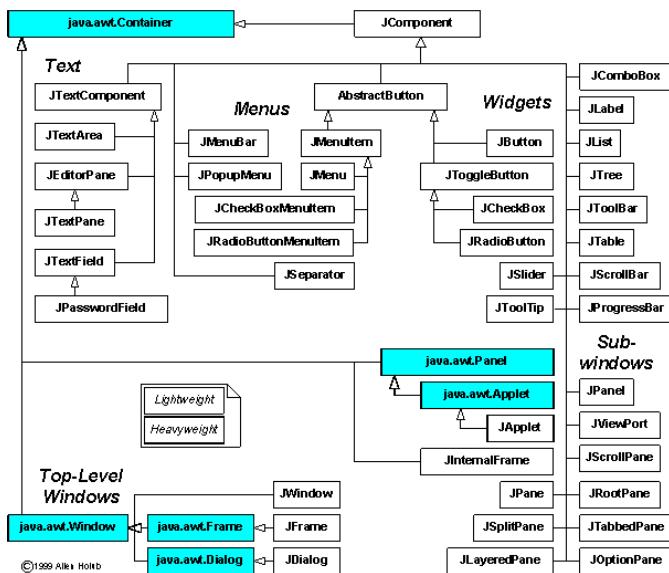
Classes de base des composants

- Par héritage, les composants Swing sont donc également des composants AWT (car la classe **Component** est une super-classe de **JComponent**).
- Les classes de la plupart des composants Swing commencent par la lettre '**J**' et se distinguent ainsi des composants AWT correspondants :
 - Composant '**Bouton**' AWT : **Button**
 - Composant '**Bouton**' Swing : **JButton**
- Il est possible de **créer de nouveaux composants** visuels ou d'étendre (spécialiser) ceux qui existent. Ce travail nécessite naturellement une bonne connaissance de l'architecture de la librairie Swing y compris de certaines classes de bas-niveau afin que les nouveaux composants soient créés dans le même esprit et s'intègrent avec les autres (interopérabilité).
- Les pages qui suivent donnent un bref aperçu (tour d'horizon) des principaux composants qui sont disponibles dans la librairie Swing.





Architecture de la librairie Swing



EIA-FR / Jacques Baps



Composants Swing [1]

| | |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JComponent | La racine de la hiérarchie de composants de Swing. Ajoute des fonctionnalités communes à Swing comme par exemple les bulles d'aide (<i>Tooltips</i>). |
| JLabel | Un composant simple (en affichage seul) qui affiche du texte, une image ou les deux. |
| JTextField | Un composant pour afficher, saisir et éditer une ligne de texte simple. Basé sur JTextComponent . |
| JPasswordField | Un champ de saisie de texte pour des données confidentielles, comme des mots de passe. N'affiche pas le texte durant la saisie. Basé sur JTextField . |
| JFormattedTextField | Un champ de texte formaté. Permet la saisie d'informations qui doivent respecter un certain format (dates, nombres, n° d'article, etc.) Différents formateurs peuvent être associés. Basé sur JTextField . |

EJA-FR / Jacques Baps



Composants Swing [2]

| | |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JButton | Un bouton-poussoir qui peut afficher du texte, des images ou les deux. Sert principalement à déclencher des actions. La classe AbstractButton est la classe de base de tous les boutons. |
| JToggleButton | Bouton à deux états (bistable). Composant père des composants JCheckBox et de JRadioButton . |
| JRadioButton | Un bouton à cocher (bouton radio) servant à afficher des choix mutuellement exclusifs. La classe ButtonGroup permet de grouper des boutons radio. |
| JRadioButtonMenuItem | Un bouton radio à utiliser dans les menus. |
| JCheckBox | Une case à cocher servant à afficher des choix non mutuellement exclusifs. |
| JCheckBoxMenuItem | Une case à cocher à utiliser dans les menus. |

EIA-FR / Jacques Baps



Composants Swing [3]

| | |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JMenuBar | Un composant qui affiche un ensemble de menus déroulants. |
| JMenu | Un menu déroulant dans un JMenuBar ou comme sous-menu. |
| JPopupMenu | Une fenêtre qui s'ouvre pour afficher une menu. Employé par JMenu et pour les menus contextuels indépendants. |
| JMenuItem | Un élément sélectionnable dans un menu (option). |
| JList | Un composant qui affiche une liste de choix sélectionnables. Les choix sont généralement représentés par des chaînes de caractères ou des images mais d'autres objets sont possibles. |
| JComboBox | Une combinaison de champs de saisie de texte et d'une liste déroulante de choix. L'utilisateur peut entrer une sélection ou choisir un élément dans la liste déroulante. |
| JSpinner | Un champ de saisie d'informations ordonnées permettant de passer en revue les valeurs possibles à l'aide de deux boutons de défilement (ou à l'aide du clavier). Affichage sur une seule ligne (contrairement à JComboBox). |

EIA-FR / Jacques Baps





Composants Swing [4]

| | |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JToolTip | Une fenêtre légère qui affiche une documentation simple ou une astuce quant le pointeur de souris reste au-dessus d'un composant (bulle d'aide). |
| JTextComponent | Le composant racine d'un système d'affichage et d'édition de texte puissant et extrêmement personnalisable. Fait partie du paquetage <code>javax.swing.text</code> . |
| JTextArea | Un composant servant à afficher et à éditer plusieurs lignes de texte simple. Basé sur <code>JTextComponent</code> . |
| JEditorPane | Un éditeur de texte puissant, configurable via un objet <code>EditorKit</code> , avec des instances prédéfinies pour afficher et éditer du texte au format HTML et RTF. |
| JTextPane | Une sous-classe de <code>JEditorPane</code> servant à afficher et éditer un texte formaté qui n'est pas au format RTF ou HTML. Permet d'ajouter une fonctionnalité de traitement de texte simple à une application. |
| JScrollBar | Un ascenseur de défilement horizontal ou vertical. |
| JProgressBar | Un composant qui affiche la progression d'une opération. |



Composants Swing [5]

| | |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JToolBar | Un composant qui affiche un ensemble d'outils ou d'actions sélectionnables par l'utilisateur. |
| JSeparator | Un composant simple qui affiche une ligne horizontale ou verticale. Utilisé pour diviser visuellement les interfaces complexes en sections. |
| JSlider | Un composant qui simule un curseur que l'utilisateur peut manipuler pour saisir et gérer une valeur numérique. |
| JOptionPane | Un composant servant à afficher des boîtes de dialogue simples. Comporte des méthodes statiques utiles pour afficher des types de dialogues communs (confirmation, avertissement, etc). |
| JColorChooser | Un composant complexe qui permet à l'utilisateur de sélectionner une couleur depuis un ou plusieurs espaces de couleurs. Employé en conjonction avec le paquetage <code>javax.swing.colorchooser</code> . |
| JFileChooser | Un composant complexe qui permet à l'utilisateur de sélectionner un fichier ou un répertoire avec possibilité de filtrage. Employé en conjonction avec le paquetage <code>javax.swing.filechooser</code> . |



Composants Swing [6]

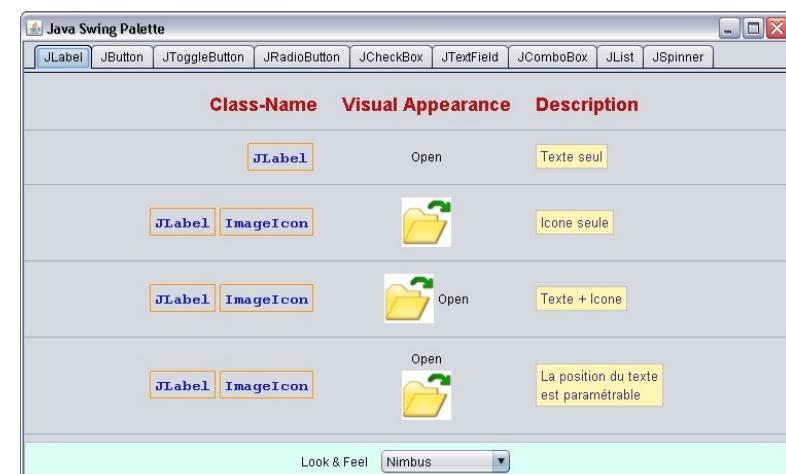
| | |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JTable | Un composant complexe et puissant servant à afficher des tableaux et à éditer leur contenu. Typiquement utilisé pour afficher des chaînes de caractères, mais peut être personnalisé pour accepter, dans ses cellules, n'importe quel type de données. Utilisé en conjonction avec le paquetage <code>javax.swing.table</code> . |
| JTree | Un composant complexe et puissant servant à l'affichage de données structurées en arborescence. Les valeurs des données sont typiquement des chaînes de caractères, mais le composant peut être personnalisé pour afficher n'importe quel type de données. Utilisé en conjonction avec le paquetage <code>javax.swing.tree</code> . |

Remarque : Les composants `JMenuBar` et `JPopupMenu` peuvent être considérés comme des conteneurs (que nous verrons plus loin). En raison de leur usage spécialisé, ils ont malgré tout été placés dans la liste des composants visuels élémentaires (tabelles précédentes).



Exemples de composants Swing [1]

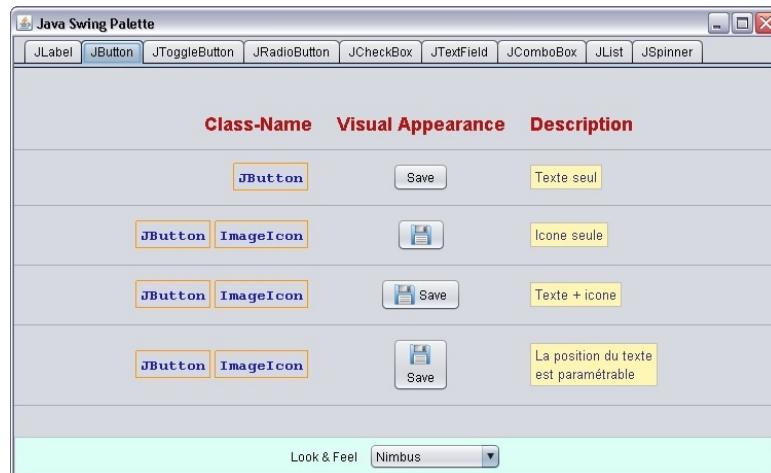
- Libellés : **JLabel**





Exemples de composants Swing [2]

- Boutons : JButton



EIA-FR / Jacques Bapst



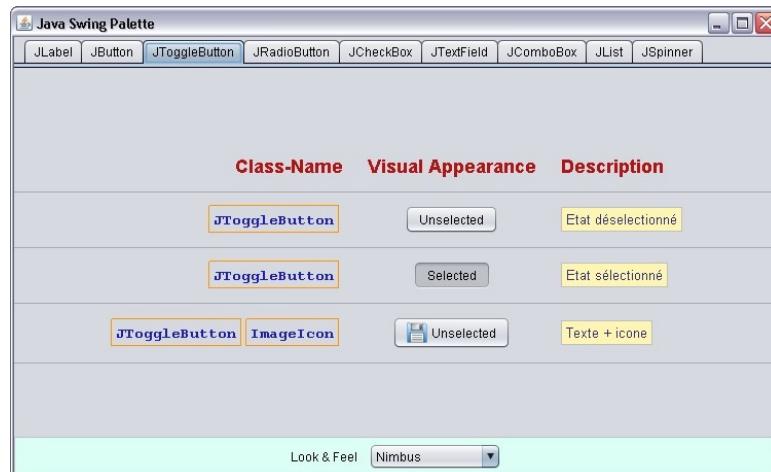
IHM1_02

17



Exemples de composants Swing [3]

- Boutons bistables : JToggleButton



EIA-FR / Jacques Bapst



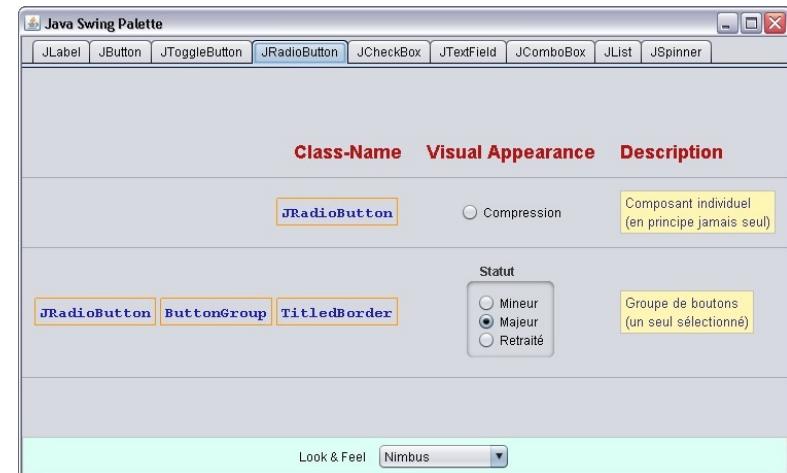
IHM1_02

18



Exemples de composants Swing [4]

- Boutons radio : JRadioButton



EIA-FR / Jacques Bapst



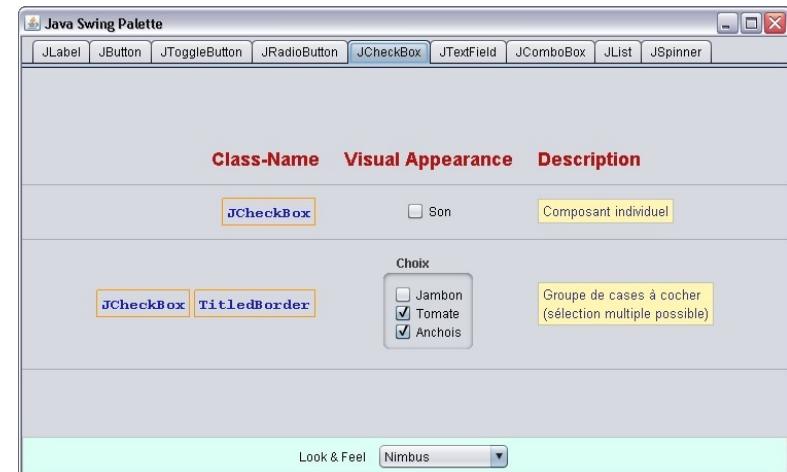
IHM1_02

19



Exemples de composants Swing [5]

- Cases à cocher : JCheckBox



EIA-FR / Jacques Bapst



IHM1_02

20



Exemples de composants Swing [6]

- Champs texte : JTextField / JPasswordField

The Java Swing Palette window displays several examples of JTextField and JPasswordField components:

- JTextField:** Example input field labeled "Exemple". Description: Champ texte.
- JTextField + JLabel:** A composite example where a JTextField is paired with a JLabel. Description: Champ texte avec libellé (2 composants visuels), Alignement à droite.
- JTextField:** Input field containing "3.14159265". Description: Champ text non éditable (affichage de valeurs).
- JPasswordField:** Input field showing masked text "*****". Description: Pour saisie de mots de passe (sous-classe de JTextField).

Look & Feel: Nimbus

EIA-FR / Jacques Bapst



IHM1_02

21



Exemples de composants Swing [7]

- Champs texte formatés : JFormattedTextField

The Java Swing Palette window displays examples of JFormattedTextField:

- JFormattedTextField:** Input field with mask "00 41 (0)7 / ..". Description: Champ texte formaté pour saisir un n° de tél. Formateur : MaskFormatter.
- JFormattedTextField + JLabel:** A composite example where a JFormattedTextField is paired with a JLabel. Description: Champ texte formaté. Formateur : DateFormat (JJ.MM.YYYY).

Look & Feel: Nimbus

EIA-FR / Jacques Bapst



IHM1_02

22



Exemples de composants Swing [8]

- Zones de texte : JTextArea

The Java Swing Palette window displays examples of JTextArea:

- JTextArea:** A large text area component. Description: Le composant JTextArea représente un champ textuel multiligne, sélectionnable, éditable, mais non formaté.
- Zone de texte (mêmes propriétés pour tout le texte):** A smaller text area component. Description: Zone de texte (mêmes propriétés pour tout le texte).

Look & Feel: Nimbus

EIA-FR / Jacques Bapst



IHM1_02

23



Exemples de composants Swing [9]

- Zones de texte formaté : JEditorPane

The Java Swing Palette window displays examples of JEditorPane:

- JEditorPane:** A text area component. Description: Creates a new JEditorPane. The document model is set to null.
- JEditorPane:** A text area component. Description: Affichage d'une page web (interpréteur HTML).

Look & Feel: Nimbus

EIA-FR / Jacques Bapst



IHM1_02

24



Exemples de composants Swing [10]

- Zones de texte structuré + images : **JTextPane**

| Class-Name | Visual Appearance | Description |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
| | <p>History: Distant</p> <p>Lee Fitzpatrick Marjan Bace</p> <p>When we started doing business under the Manning name, about 10 years ago, we were a very different company. What we are now is the end result of an evolutionary process in which accidental events played as big a role, or bigger, as planning and foresight.</p> <p>Visit Manning</p> | <p>Image d'un JTextPane (simulation) Possibilité de créer une application de type "Desktop Publishing"</p> |

Look & Feel Nimbus

EIA-FR / Jacques Bapst



IHM1_02

25



Exemples de composants Swing [11]

- Listes déroulantes : **JComboBox**

| Class-Name | Visual Appearance | Description |
|------------|-------------------------------------------------------------------------------|-------------------------------|
| | <p>Allemande</p> | Liste déroulante non-éditable |
| | <p>Français</p> <p>Deutsch</p> <p>English</p> <p>Français</p> <p>Italiano</p> | Liste déroulante éditable |

Look & Feel Nimbus

EIA-FR / Jacques Bapst



IHM1_02

26



Exemples de composants Swing [12]

- Listes : **JList**

| Class-Name | Visual Appearance | Description |
|------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|
| | <p>A : Alpha B : Bravo C : Charlie D : Delta E : Echo F : Foxtrot G : Golf H : Hotel I : India J : Juliette</p> | Liste (ensemble des éléments affichés) |
| | <p>A : Alpha B : Bravo C : Charlie D : Delta E : Echo</p> | Liste intégrée avec scrolling (conteneur spécialisé) |

Look & Feel Nimbus

EIA-FR / Jacques Bapst



IHM1_02

27



Exemples de composants Swing [11]

Exemples de composants Swing [13]

- Listes à défilement : **JSpinner**

| Class-Name | Visual Appearance | Description |
|------------|-------------------|------------------|
| | <p>Vendredi</p> | Liste déroulante |
| | <p>4.5</p> | Avec nombres |

Look & Feel Nimbus

EIA-FR / Jacques Bapst



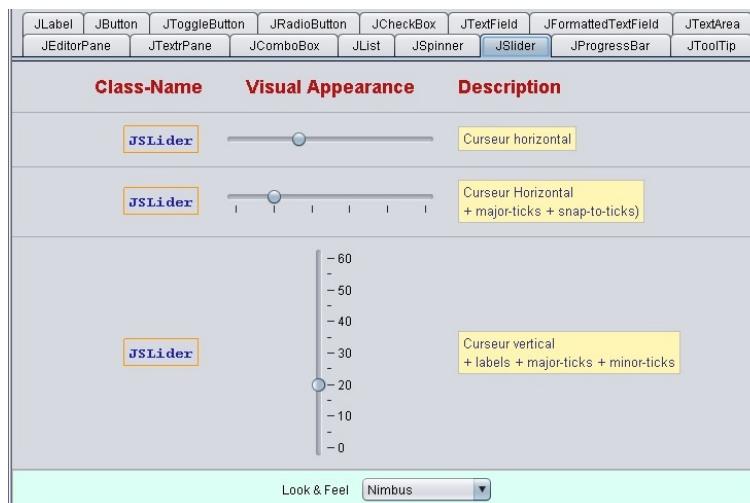
IHM1_02

28



Exemples de composants Swing [14]

▪ Curseurs : JSlider



EIA-FR / Jacques Bapst



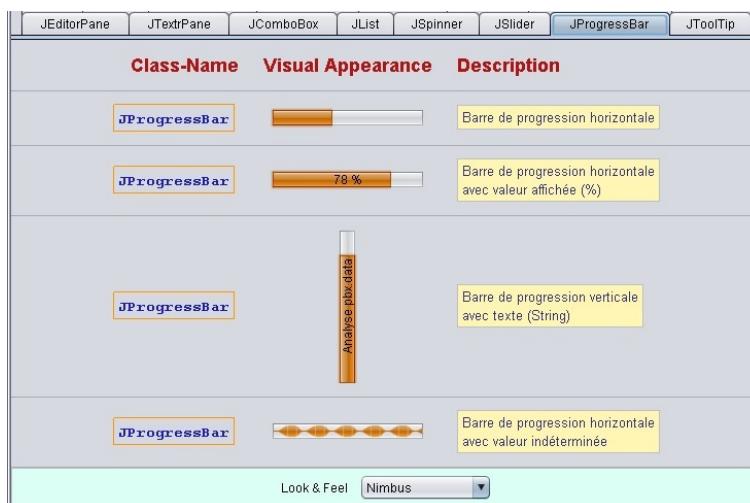
IHM1_02

29



Exemples de composants Swing [15]

▪ Barres de progression : JProgressBar



EIA-FR / Jacques Bapst



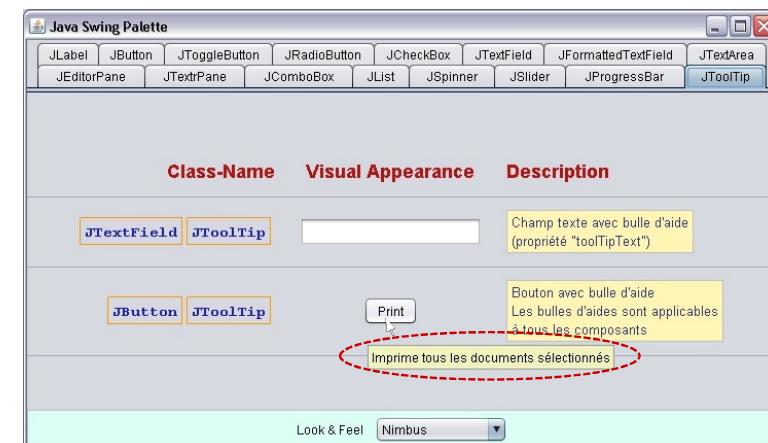
IHM1_02

30



Exemples de composants Swing [16]

▪ Bulles d'aide : JToolTip



EIA-FR / Jacques Bapst



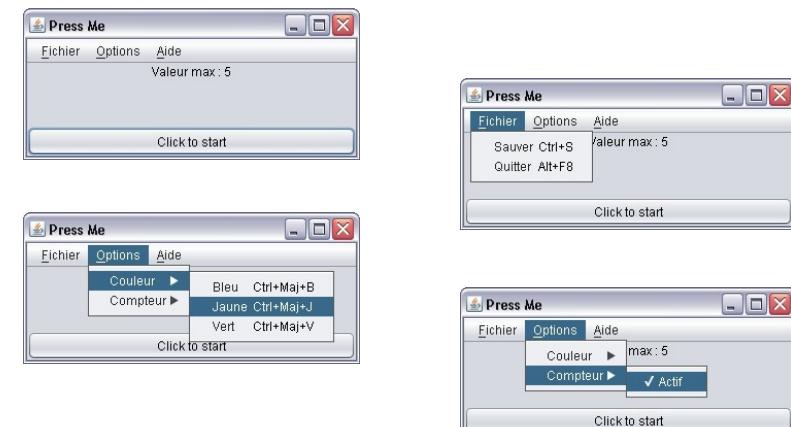
IHM1_02

31



Exemples de composants Swing [17]

▪ Menus : JMenuBar / JMenu / JMenuItem JRadioButtonMenuItem / JCheckBoxMenuItem



EIA-FR / Jacques Bapst



IHM1_02

32



Exemples de composants Swing [18]

Tabelles : JTable

The top screenshot shows a table titled "TableSelectionDemo" with columns: First Name, Last Name, Sport, # of Years, and Vegetarian. It contains data for six people. Below the table is a "Selection Mode" section with radio buttons for "Multiple Interval Selection", "Single Selection", and "Single Interval Selection". Under "Selection Options", there are checkboxes for "Row Selection", "Column Selection", and "Cell Selection". Status messages at the bottom indicate "ROW SELECTION EVENT: Lead: 3, 0. Rows: 3. Columns: 0." and "COLUMN SELECTION EVENT: Lead: 3, 0. Rows: 3. Columns: 0.".

The bottom screenshot shows a "Bit Chart" application with a large grid of colored bars representing data values. A status bar at the bottom shows "Qua... 11 Name Type Size Speed Bitrate".

EIA-FR / Jacques Bapst

IHM1_02

33



Exemples de composants Swing [20]

Conteneurs généraux : JPanel

The palette shows two entries for "JPanel": one with a white background labeled "Conteneur seul Arrière-plan blanc" and another with a yellow border labeled "Conteneur seul Bordure jaune".

EIA-FR / Jacques Bapst



IHM1_02

35



Exemples de composants Swing [19]

Arbres : JTree

The top screenshot shows a tree view titled "TreeDemo" with nodes like "The Java Series" and "Books for Java Programmers". Below it is a "Tree Demo" panel containing text about the help file and a note about selecting leaf nodes.

The bottom screenshot shows a "Dynamic TreeDemo" application with a tree structure and buttons for "Add", "Remove", and "Clear".

EIA-FR / Jacques Bapst

IHM1_02

34



Exemples de composants Swing [21]

Conteneurs à double panneaux : JSplitPane

The palette shows two entries for "JSplitPane": one split horizontally with a yellow left pane and an orange right pane, labeled "HORIZONTAL_SPLIT"; and one split vertically with a green top pane and a red bottom pane, labeled "VERTICAL_SPLIT".

EIA-FR / Jacques Bapst



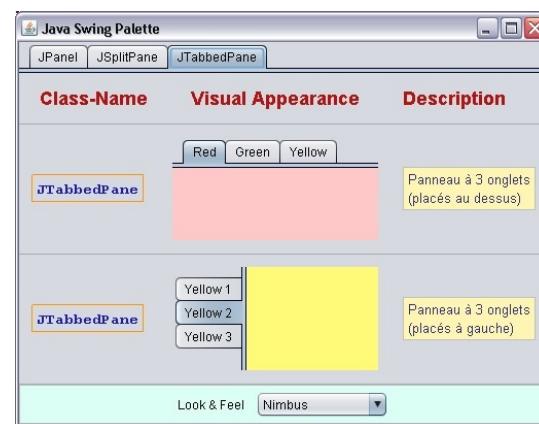
IHM1_02

36



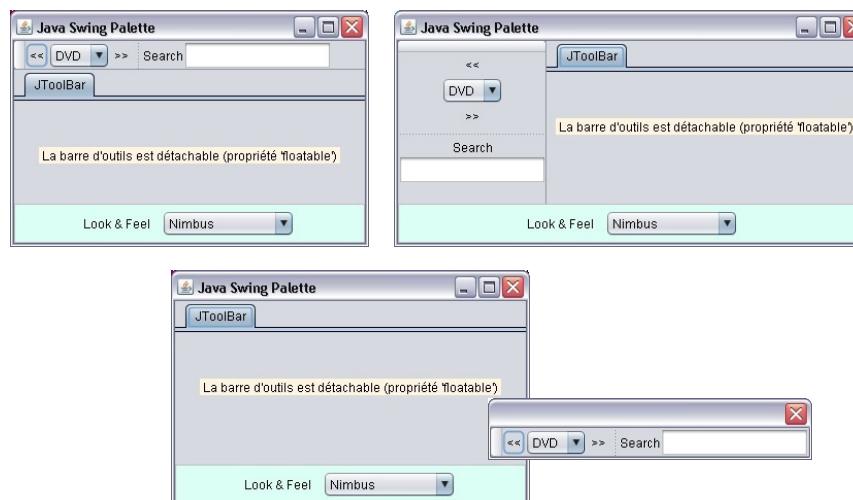
Exemples de composants Swing [22]

- Panneaux à onglets : **JTabbedPane**



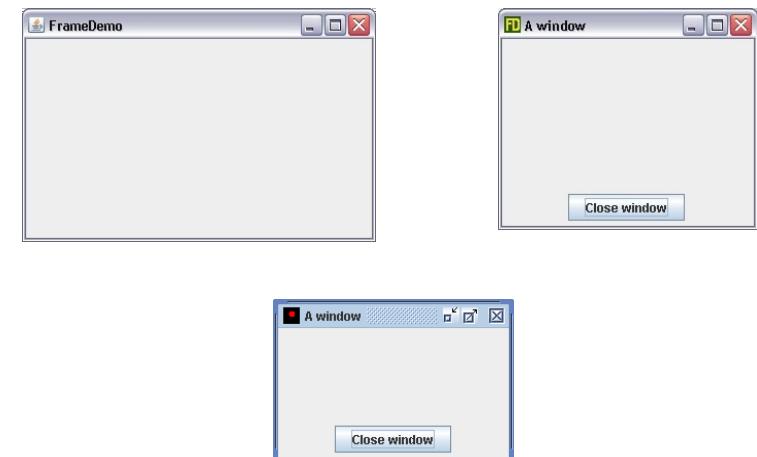
Exemples de composants Swing [23]

- Barres d'outils : **JToolBar**



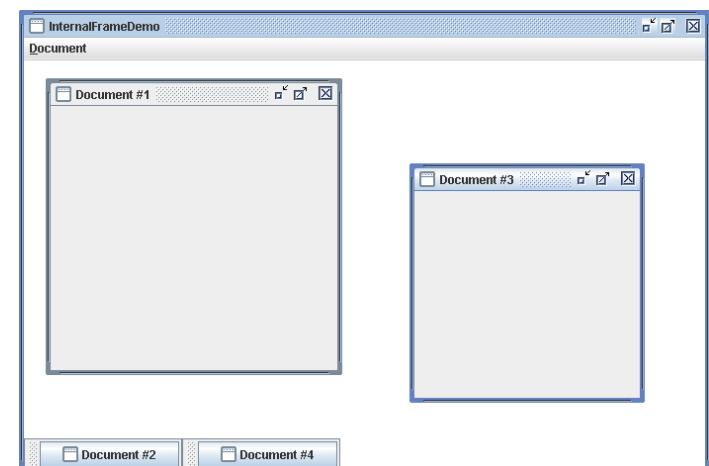
Exemples de composants Swing [24]

- Conteneurs d'applications : **JFrame**



Exemples de composants Swing [25]

- Conteneurs d'applications de type MDI : **JDesktopPane**
JInternalFrame





Exemples de composants Swing [26]

- Fenêtres sans éléments décoratifs : **JWindow**



EIA-FR / Jacques Bapst

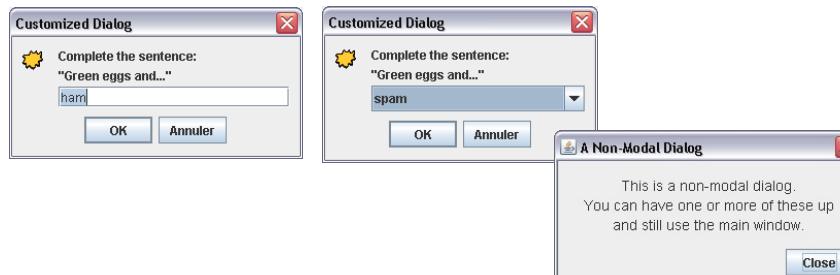


IHM1_02 41



Exemples de composants Swing [27]

- Boîtes de dialogue générales : **JOptionPane**



EIA-FR / Jacques Bapst

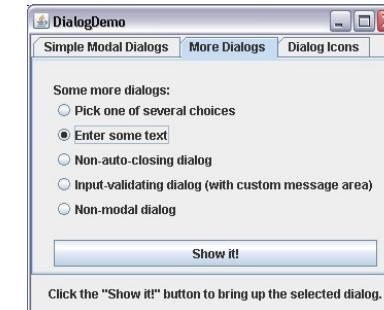


IHM1_02 42



Exemples de composants Swing [28]

- Boîtes de dialogue spécifiques : **JDialog**



EIA-FR / Jacques Bapst

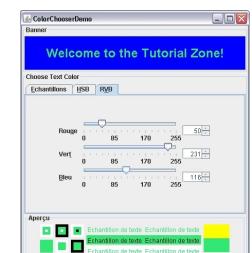
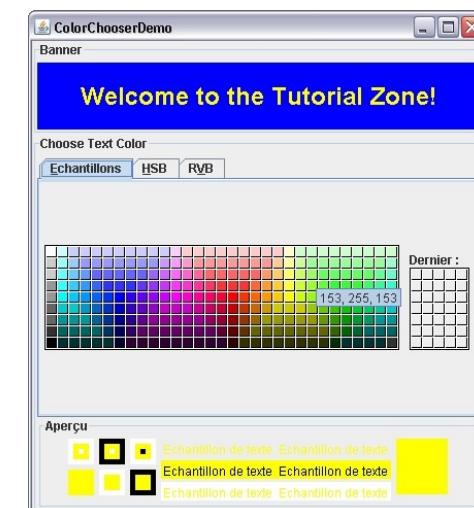


IHM1_02 43



Exemples de composants Swing [29]

- Sélection d'une couleur : **JColorChooser**



EIA-FR / Jacques Bapst

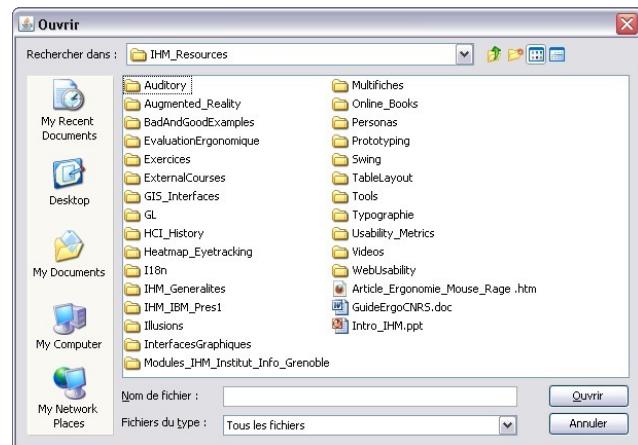


IHM1_02 44



Exemples de composants Swing [30]

- Sélection de fichiers : **JFileChooser**



Propriétés des composants [1]

- Chaque composant Swing peut être personnalisé en apparence et en comportement en spécifiant des valeurs de **propriétés**.
- Les **propriétés** ne sont pas des notions qui font formellement partie des classes Java. Ce sont des conventions de nommage qui ont été définies avec le concept de *JavaBeans* (composants réutilisables).
- Quand un composant définit des méthodes publiques du type :

`getppp()` ou `isppp()`

`setppp()`

Ces méthodes (appelées **accesseurs** en lecture et **mutateurs** en écriture) définissent la **propriété ppp**.

- Par exemple `getFont()` et `setFont()` définissent la propriété "font" (police de caractères) d'un composant.
- Quand la propriété est de type booléenne, l'accesseur `getppp()` est souvent remplacé par `isppp()`.
- `isVisible()` et `setVisible()` définissent la propriété "visible".



Propriétés des composants [2]

- En plus des propriétés définies par un composant, toutes les **propriétés héritées** des classes parentes sont naturellement également disponibles pour ce composant.
- Les propriétés de **JComponent**, **Container** et **Component** sont héritées par pratiquement tous les composants Swing.
- Les possibilités de **personnalisation d'un composant** sont déterminées par les propriétés qu'il définit lui-même et par les propriétés dont il hérite.
- La consultation des propriétés d'un composant donne des indications importantes sur son fonctionnement.
- Dans la description des **API** des composants, les propriétés ne sont pas mises en évidence. Les méthodes associées (`get...`, `is...`, `set...`) sont mêlées aux autres méthodes de la classe.
Les méthodes sont simplement décrites dans l'ordre alphabétique.



Propriétés des composants [3]

- Certaines propriétés ne sont définies qu'en lecture (`get...`, `is...`), d'autres en lecture et en écriture (`get...`, `is...`, `set...`) et certaines, beaucoup plus rarement, en écriture seulement (`set...`).
- Certaines propriétés sont "liées" (**bound properties**) et déclenchent des événements (**propertyChangeEvent**) lorsque leur valeur change.
 - Cela permet, par exemple, de changer automatiquement la taille d'un conteneur lorsque la taille d'un composant change.





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

03

Conteneurs et Gestionnaires de disposition

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Conteneurs et Gestionnaires de disposition

Conteneur et confinement

- Dans une interface graphique, les composants visuels élémentaires sont toujours contenus dans un élément de confinement appelé **conteneur** (ou **container**).
- La librairie Swing définit un certain nombre de conteneurs qui héritent tous de la classe `java.awt.Container`.

Remarque : Comme tous les composants Swing héritent de `Container`, ils peuvent tous être considérés comme des conteneurs. Cependant, seuls certains agissent vraiment comme des conteneurs et sont utilisés comme tels.
- Les **fenêtres principales** (*Frames*) et les **boîtes de dialogue** (*Dialog boxes*) constituent des conteneurs couramment employés. Ils fournissent des fenêtres de premier niveau dans lesquelles les composants graphiques peuvent être disposés pour créer l'interface utilisateur d'une application.
- Certains conteneurs comportent des bordures et des éléments décoratifs (titre, menu, boutons, ...). D'autres ne comportent aucun éléments visibles (éléments de confinement pur).

EIA-FR / Jacques Bapst



IHM1_03

2



Interface Homme-Machine 1 / Conteneurs et Gestionnaires de disposition

Conteneur et confinement

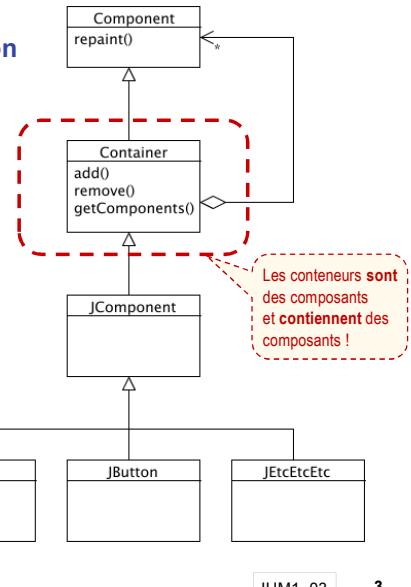
- Les fonctions de base d'un conteneur sont **l'ajout**, le **retrait** et la **consultation de ses composants**.

- Ces fonctions élémentaires sont assurées par les méthodes

- `add()`
- `remove()`
- `getComponent()`
- `getComponents()`

- Une autre méthode importante permet **d'associer un gestionnaire de disposition** au conteneur

- `setLayout()`



EIA-FR / Jacques Bapst

IHM1_03

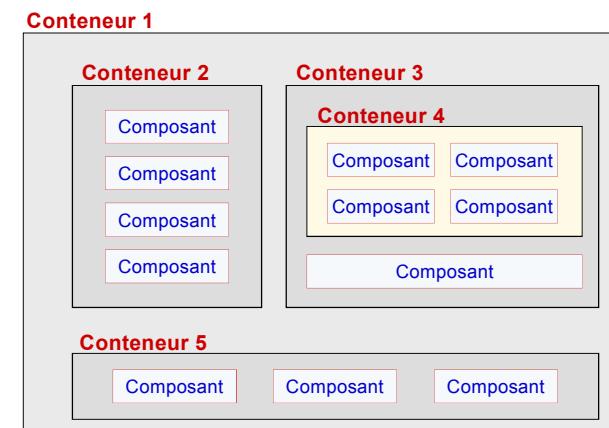
3



Interface Homme-Machine 1 / Conteneurs et Gestionnaires de disposition

Emboîtement

- Il est très fréquent d'**emboîter des conteneurs** afin de mieux gérer la disposition des composants :



EIA-FR / Jacques Bapst



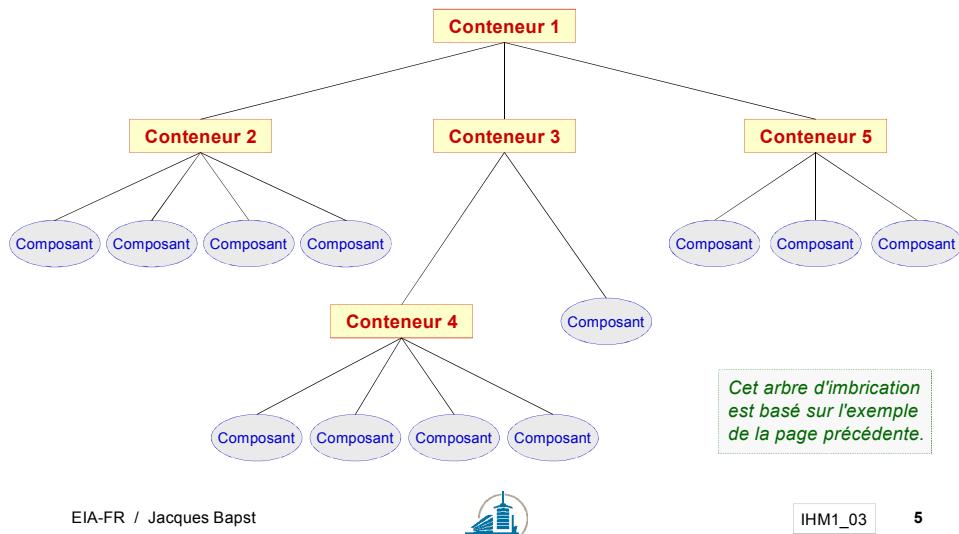
IHM1_03

4



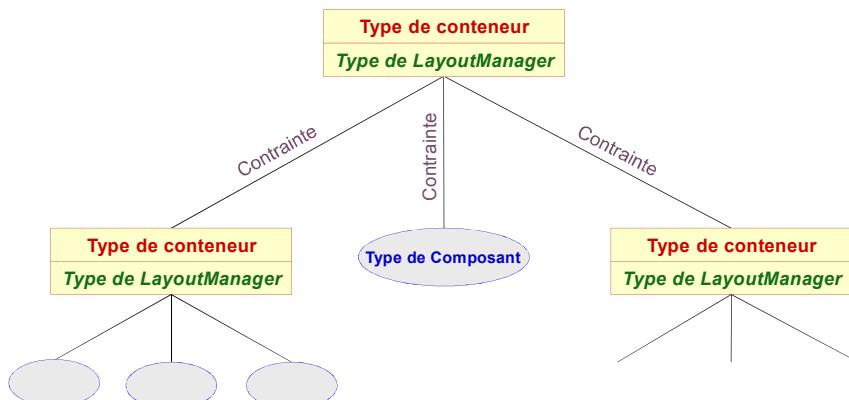
Arbre d'imbrication

- On peut représenter l'emboîtement des conteneurs par un diagramme hiérarchique appelé **arbre d'imbrication** ou **arbre de confinement**.



Arbre d'imbrication - Formalisme

- La **notation** suivante sera utilisée pour représenter les arbres d'imbrication :



Genres de conteneurs

- Dans les librairies Java, on trouve plusieurs genres de conteneurs.
- Il existe des **conteneurs génériques** qui peuvent contenir n'importe quel nombre d'enfants arrangés de manière quelconque (selon le gestionnaire de disposition utilisé). Le composant **JPanel** constitue le conteneur générique par excellence.
- Certains conteneurs sont plus **spécialisés** et disposent les composants enfants d'une manière très spécifique, en imposant des restrictions concernant le nombre ou le type de composants qu'ils peuvent afficher (**JTabbedPane**, **JSplitPane**, **JScrollPane**, **JViewport**, ...).
- Les conteneurs utilisent généralement un **gestionnaire de disposition (Layout Manager)** pour déterminer la manière dont leurs enfants (conteneurs ou composants) seront disposés (taille et position) ainsi que pour définir le comportement à adopter suite à un redimensionnement de la fenêtre englobante (par l'utilisateur ou par l'application).
- Les tables suivantes donnent un bref aperçu des principaux conteneurs disponibles dans la librairie Swing.



Conteneurs Swing [1]

| | |
|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| JRootPane | Un conteneur complexe utilisé en interne par JApplet , JDialog , JFrame et JWindow . Fournit à ces conteneurs de premier-niveau, ainsi qu'à JInternalFrame , un grand nombre de fonctionnalités importantes de Swing. |
| JFrame [1 ^{er} niv.] | Le conteneur utilisé pour les fenêtres de premier niveau (par exemple la fenêtre principale de l'application). |
| JDialog [1 ^{er} niv.] | Le conteneur utilisé pour afficher des boîtes de dialogue. |
| JWindow [1 ^{er} niv.] | Fenêtre de premier-niveau qui n'affiche pas de barre de titre ou d'autre décoration (<i>Splash screen</i>) |
| JApplet [1 ^{er} niv.] | Une sous-classe de java.applet.Applet contenant un JRootPane permettant d'ajouter des fonctionnalités (menus, ...) |
| JDesktopPane | Un conteneur de composants JInternalFrame ; simule le comportement d'un bureau à l'intérieur d'une fenêtre (MDI). |
| JInternalFrame | Un conteneur de fenêtres imbriquées. Se comporte comme un JFrame et affiche une barre de titre et des poignées de modification de taille, mais n'est pas une fenêtre indépendante. |





Conteneurs Swing [2]

| | |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| JPanel | Un conteneur générique qui regroupe des composants Swing. Typiquement utilisé avec un Layout-Manager approprié (utilise FlowLayout par défaut). |
| Box | Un conteneur générique qui arrange les enfants selon le gestionnaire de disposition BoxLayout . |
| JLayeredPane | Un conteneur qui permet à ses enfants de se recouvrir et qui gère leur ordre d'empilement (utilisé dans JRootPane). |
| JTabbedPane | Un conteneur qui affiche un seul dossier-enfant à la fois, permettant à l'utilisateur de sélectionner le dossier à afficher en cliquant sur des onglets. |
| JSplitPane | Un conteneur qui affiche deux enfants en se divisant horizontalement ou verticalement. Permet à l'utilisateur d'ajuster l'espace alloué à chaque enfant. |
| JViewport | Un conteneur de taille fixe qui affiche une partie d'un enfant. Employé par JScrollPane . |
| JScrollPane | Un conteneur qui permet à un composant enfant unique de défiler horizontalement ou verticalement grâce à des ascenseurs. |



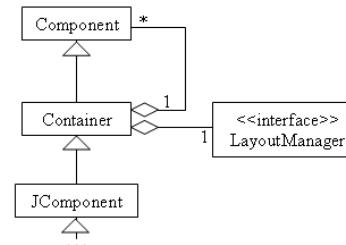
Gestionnaires de disposition [1]

- Certains conteneurs, comme **JTabbedPane** et **JSplitPane**, définissent une disposition particulière pour leurs enfants (le terme "enfant" est utilisé pour désigner les composants contenus dans le conteneur).
- D'autres conteneurs génériques comme **JPanel** peuvent contenir un nombre quelconque de composants qui seront disposés selon le **gestionnaire de disposition** choisi (par défaut le composant **JPanel** utilise le gestionnaire de disposition **FlowLayout**).
- Dans un conteneur, la disposition des composants (position et dimension) est déléguée à un **gestionnaire de disposition** (ou **Layout Manager**) qui se charge d'arranger les enfants à l'intérieur du conteneur.
- Les librairies AWT et Swing offrent un certain nombre de gestionnaires de disposition (Layout Manager) prédéfinis. Il est également possible de créer des gestionnaires de disposition personnalisés en implémentant l'interface **java.awt.LayoutManager** ou **LayoutManager2** (c'est assez rarement nécessaire).



Gestionnaires de disposition [2]

- Les **gestionnaires de disposition (layout manager)** sont donc des composants associés aux conteneurs et qui sont chargés de
 - Positionner** les composants
 - Dimensionner** les composants
- Le programmeur peut généralement paramétriser les gestionnaires afin d'adapter les règles au contexte spécifique (on parle de **contraintes**).



Gestionnaires de disposition [3]

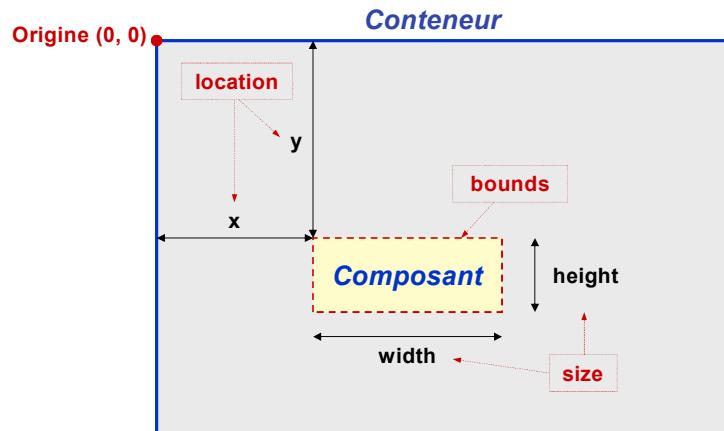
- Il est possible (comme on le verra plus loin) de travailler sans gestionnaire de disposition et de se charger soi-même de la taille et de la position des composants (valeurs absolues en pixels).
- Dans une application professionnelle, l'**utilisation des gestionnaires de disposition est cependant impérative** car ils offrent de grands avantages pour prendre en compte les situations suivantes :
 - Ils peuvent tenir compte de la **taille naturelle** (préférée) des composants qui dépend de plusieurs facteurs :
 - ✓ des libellés de certains composants (label, boutons, champs de texte, ...) qui peuvent varier, notamment dans une application multilingue
 - ✓ de la configuration de la machine cible (police par défaut, taille de la police, style)
 - ✓ du *Look & Feel* choisi par l'utilisateur ou défini par le système d'exploitation
 - Ils peuvent tenir compte du **redimensionnement des fenêtres** qui peut être nécessaire en fonction de la taille de l'écran et de sa résolution sur la machine cible (les utilisateurs n'ont pas le même environnement que vous !)
- Une utilisation judicieuse des gestionnaires de disposition permet au programmeur de **définir les règles de disposition** plutôt que de réagir et prendre en compte tous les événements qui affectent la présentation.





Système de coordonnées

- Chaque conteneur dispose de son propre **système de coordonnées**. Les composants enfants sont placés relativement à l'**origine** (0, 0) du conteneur (et non pas de l'écran).



Propriétés de positionnement

- Les propriétés suivantes servent à gérer la **position** et la **taille** des composants et des conteneurs :
 - location** : Position du coin supérieur gauche du composant par rapport à l'origine du conteneur (coin supérieur gauche)
 - size** : Dimension (taille) du composant en hauteur et en largeur
 - bounds** : Rectangle qui entoure le composant et qui définit à la fois la position et la dimension du composant
 - minimalSize** : Dimension minimale du composant
 - preferredSize** : Dimension idéale (naturelle, souhaitée) du composant
 - maximalSize** : Dimension maximale du composant
- Les gestionnaires de disposition utilisent ces propriétés (mais pas forcément toutes) pour dimensionner et placer les composants enfants dans le conteneur en respectant certaines règles propres à chaque gestionnaire de disposition.
- Certains gestionnaires respectent la taille préférée des composants, d'autres redimensionnent les composants (ou une partie seulement) pour respecter certaines règles de disposition spécifiques.



Marges

- Les **marges d'un conteneur** (c'est-à-dire l'espace entre les bords du conteneur et les composants qu'il contient) sont définies par un objet de type **Insets** (`java.awt`).
- Pour modifier les marges d'un conteneur, on peut créer une sous-classe de ce conteneur et redéfinir la méthode `getInsets()`.


```
public class MonConteneur extends . . . {
    . . .
    public Insets getInsets() {
        int top=10, bottom=10, left=20, right=20;
        return new Insets(top, left, bottom, right);
    }
    . . .
}
```
- Pour créer une marge à l'intérieur des composants et des conteneurs Swing, il est également possible (et même conseillé) d'utiliser une **bordure invisible** (un objet de type **EmptyBorder**) et de l'appliquer à l'aide de la méthode `setBorder()` (de `JComponent()`).
- Certains *Layout-Manager* permettent également de définir des marges.



Principe général

- La création d'une interface graphique comporte généralement les étapes suivantes :
 - ▶ **Création des composants élémentaires**
 - ▶ **Création des conteneurs**
 - ▶ **Création et association des gestionnaires de disposition**
 - ▶ **Ajout des composants (et sous-conteneurs) dans les conteneurs**
- L'ajout des composants dans le conteneur utilise l'une des méthodes `add()` définie par le conteneur ou héritée de la classe **Container**.
- Exemple :

```
JButton unBouton = new JButton("Stop"); // Composant
JPanel unPanel = new JPanel(); // Conteneur
unPanel.setLayout(new BorderLayout()); // Gestionnaire de disposition
unPanel.add(unBouton); // Ajout du composant dans le conteneur
```





Conteneurs de premier-niveau

- Les composants **JFrame**, **JDialog**, **JWindow** et **JApplet** sont des **conteneurs** dits de **premier-niveau** (ou **haut-niveau**).
- Ils représentent des fenêtres qui seront affichées par le système de fenêtrage de la plate-forme d'exécution (ils cohabitent avec les fenêtres des autres applications en cours d'exécution).
- Contrairement aux autres composants Swing, ces conteneurs de premier-niveau n'héritent pas de **JComponent** mais de leur équivalent dans la librairie AWT (c'est à dire de **Frame**, **Dialog**, **Window** ou **Applet**).
- Lors de leur instantiation, les conteneurs de premier-niveau créent automatiquement un composant-enfant qui est un conteneur de type **JRootPane**. C'est finalement ce conteneur **JRootPane** (qui étend, lui, **JComponent**) qui contiendra tous les composants placés dans le conteneur de premier-niveau.
- A retenir :** Les composants élémentaires ne sont donc pas ajoutés directement dans les conteneurs de premier-niveau, ils sont ajoutés dans des sous-conteneurs de **JRootPane**.

EIA-FR / Jacques Bapst



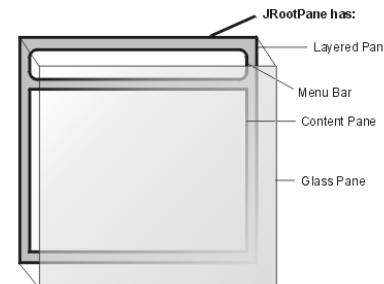
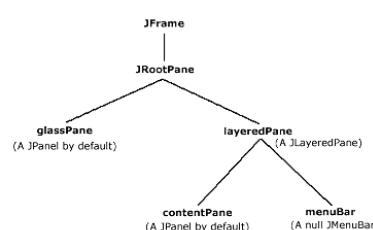
IHM1_03

17



Structure de JRootPane

- L'architecture du conteneur **JRootPane** est assez complexe et fait intervenir un certain nombre de sous-conteneurs.
- Deux de ces sous-conteneurs sont essentiels pour la création de la plupart des applications
 - Content Pane** : Conteneur principal destiné aux composants de l'application
 - MenuBar** : Conteneur prévu pour le menu principal



EIA-FR / Jacques Bapst



IHM1_03

18



Conteneurs de premier-niveau : utilisation [1]

- On ne peut donc pas ajouter de composant directement dans les conteneurs de premier-niveau, on les ajoute dans le panneau de contenu (**Content-Pane**) du **JRootPane**.
- La référence du panneau de contenu (le **Content-Pane** qui est un objet de type **Container**) peut être obtenue à l'aide de la méthode **getContentPane()** définie dans l'interface **RootPaneContainer**.
- Le **schéma général**, pour l'ajout de composants dans un conteneur de premier-niveau est donc le suivant :

```

JButton b1 = new JButton("Ok");           // Composant
JFrame frame = new JFrame("Test");        // Conteneur de premier-niveau
frame.getContentPane().add(b1);            // Ajout du composant dans le
                                         // panneau de contenu de la frame
  
```

- Si nécessaire, le gestionnaire de disposition du **Content-Pane** (par défaut : **BorderLayout**) peut être modifié avant d'y ajouter les composants enfants.

EIA-FR / Jacques Bapst



IHM1_03

19



Conteneurs de premier-niveau : utilisation [2]

- Par défaut, le **Content-Pane** est un conteneur de type **JPanel** (géré par le gestionnaire de disposition **BorderLayout**) mais on peut **définir un conteneur personnalisé** à l'aide de la méthode **setContentPane()**.
- Le conteneur **JRootPane** offre également une autre fonctionnalité importante : il permet d'afficher une **barre de menu** (**JMenuBar**) en utilisant la méthode **setJMenuBar()**.
- La méthode **pack()** permet de forcer le redimensionnement des conteneurs de premier-niveau pour qu'ils prennent la **taille minimale** en tenant compte de la taille préférée de leurs composants enfants.

Remarque : Le conteneur **JInternalFrame** (qui n'est pas un conteneur de premier-niveau) utilise également un sous-conteneur de type **JRootPane** comme unique enfant.

L'ajout d'un menu et/ou de composants dans un tel conteneur s'effectue donc de la même manière que pour un conteneur de premier-niveau, c'est-à-dire en utilisant le **Content-Pane**.

EIA-FR / Jacques Bapst



IHM1_03

20



Conteneurs de premier-niveau : utilisation [3]

- A partir de la version 1.5 de Java (Java 5), les méthodes
 - > **add()**
 - > **remove()**
 - > **setLayout()**

de tous les conteneurs de premier-niveau ont été redéfinies pour qu'elles s'appliquent directement au *Content-Pane* du conteneur.

- **Sur le principe, il n'y a aucun changement** mais, à partir de cette version, on peut donc écrire :

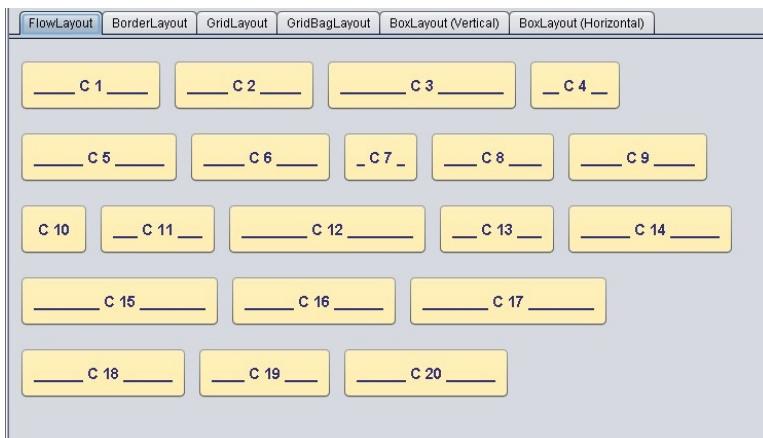
```
JButton b1 = new JButton("Ok"); // Création d'un composant (bouton)
JFrame frame = new JFrame("Test"); // Création du conteneur de premier-
// niveau
frame.setLayout(new FlowLayout()); // Nouveau gestionnaire de disposition
// pour le Content-Pane de la frame
frame.add(b1); // Ajout du composant dans le
// Content-Pane de la frame
```



FlowLayout [1]

▪ **FlowLayout (java.awt)**

- Arrange les composants comme du texte sur une page : de gauche à droite en ligne, puis de haut en bas lorsque chaque ligne est remplie.



FlowLayout [2]

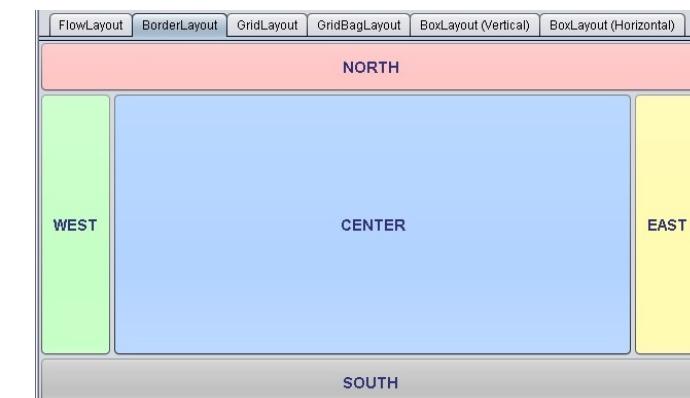
- Le gestionnaire **FlowLayout** respecte la taille préférée des composants (définie par la propriété **preferredSize** des composants).
- Lorsque la taille du conteneur est modifiée (suite à un redimensionnement de la fenêtre par exemple), les composants enfants sont automatiquement réarrangés par le gestionnaire.
- **Propriétés du gestionnaire :**
 - Les rangées peuvent être justifiées à gauche, à droite ou centrées à l'aide de la propriété **alignment** et des constantes de la classe (**LEFT**, **RIGHT**, **CENTER**, ...).
Par défaut **alignment=CENTER**.
 - L'espace (horizontal et vertical) entre les composants peut être défini en gérant les propriétés **hgap** et **vgap** (nombre de pixels entre les composants)
Par défaut : **hgap=5** et **vgap=5**.



BorderLayout [1]

▪ **BorderLayout (java.awt)**

- Dispose un maximum de cinq composants (le long des quatre bords et au centre) selon la disposition suivante :





BorderLayout [2]

- La position des composants est déterminée par une contrainte mentionnée lors de l'ajout du composant (paramètre de la méthode `add()`). Elle est définie en utilisant une constante de la classe `BorderLayout` (`NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`).
- Le gestionnaire `BorderLayout` redimensionne les composants de manière à obtenir le comportement suivant :
 - Les composants "`nord`" et "`sud`" occupent **toute la largeur** du conteneur
 - Les composants "`est`" et "`ouest`" occupent toute **la hauteur qui reste**
 - Le composant "`centre`" occupe **toute la place restante**
- Les cinq emplacement prévus ne doivent pas forcément être tous occupés (mais il ne peut pas y avoir plus d'un composant par emplacement).
- Propriétés du gestionnaire :**
 - L'espace (horizontal et vertical) entre les composants peut être défini en gérant les propriétés `hgap` et `vgap` (nombre de pixels entre les composants)
 Par défaut : `hgap=0` et `vgap=0`.

EIA-FR / Jacques Bapst



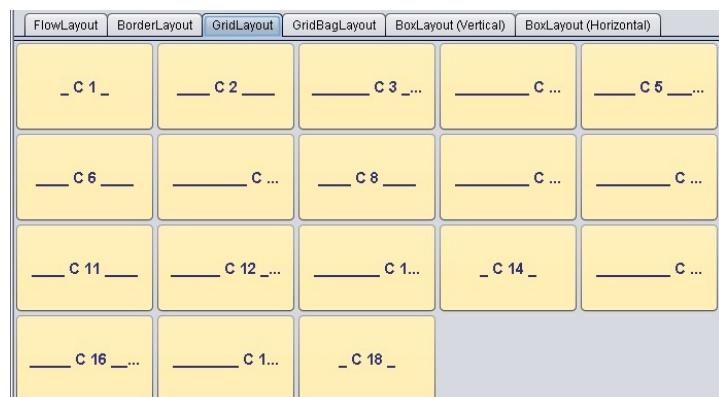
IHM1_03

25



GridLayout [1]

- GridLayout (java.awt)**
 - Arrange les composants de gauche à droite et de haut en bas dans une **grille régulière** (tous les composants ont une taille identique).
 - Toutes les cellules ont la même taille.



EIA-FR / Jacques Bapst



IHM1_03

26



GridLayout [2]

- Dans le constructeur, on fixe le nombre de lignes (`rows`) ou le nombre de colonnes (`columns`). Si `rows>0`, `columns` est ignoré.
- Les composants sont placés dans les cellules **ligne par ligne** (remplies de gauche à droite) dans l'ordre d'ajout. La grille peut être incomplète.
- Les composants sont redimensionnés pour occuper tout l'espace de chacune des cellules. Toutes les cellules ont la même taille.
- Propriétés du gestionnaire :**
 - L'espace (horizontal et vertical) entre les composants (entre les cellules) peut être défini en gérant les propriétés `hgap` et `vgap`.
Par défaut : `hgap=0` et `vgap=0`.
 - Nombre de lignes : `rows`
 - Nombre de colonnes : `columns`

EIA-FR / Jacques Bapst



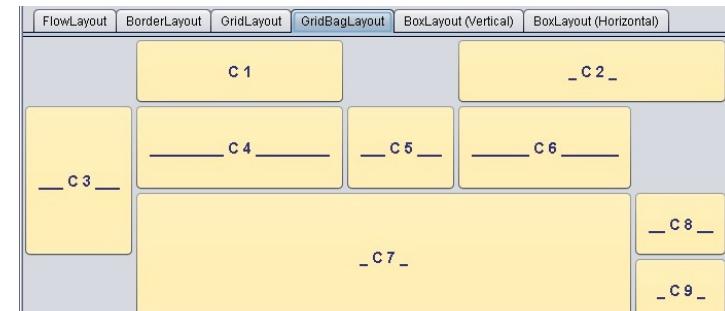
IHM1_03

27



GridBagLayout [1]

- GridBagLayout (java.awt)**
 - Arrange les composants dans une grille dont les cellules sont de tailles variables. Un composant peut s'étendre (`span`) sur plusieurs cellules.
 - A chaque composant est associé un ensemble de **contraintes**, spécifiées par un objet de type `GridBagConstraints`.
 - Permet un contrôle précis de la dimension et du positionnement des composants, notamment quand le conteneur change de taille.



EIA-FR / Jacques Bapst



IHM1_03

28



GridBagLayout [2]

- Lors de l'ajout des composants dans le conteneur, on leur associe des **contraintes** sous la forme d'objets de type **GridBagConstraints**.
- Un objet de type **GridBagConstraints** possède un certain nombre de champs publics qui sont utilisés pour définir les propriétés de la contrainte (il n'y a pas d'accesseurs/mutateurs pour ces champs).
- Il est possible de déclarer un seul objet pour définir les contraintes et de l'adapter avant l'ajout de chacun des composants selon le schéma d'utilisation suivant :

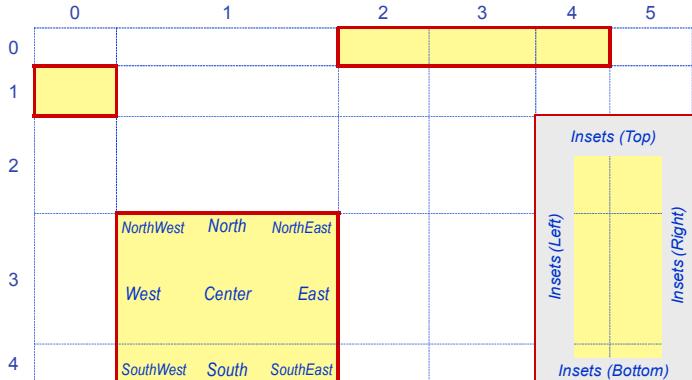
```
GridBagConstraints contrainte = new GridBagConstraints();
contrainte.a = x; // On donne la valeur x à la contrainte a
contrainte.b = y;
conteneur.add(composant1, contrainte);
contrainte.a = w;
contrainte.c = z;
conteneur.add(composant2, contrainte);
...
```

Attention aux risques liés à l'utilisation d'un objet global !



GridBagLayout [3]

- Les composants occupent des **zones** rectangulaires qui peuvent être composées d'une ou plusieurs cellules.
- On peut définir le point d'ancre du composant à l'intérieur de la zone



GridBagConstraints [1]

- Les **contraintes** associées à chaque composant enfant définissent :
 - Les **coordonnées** (numéro de la colonne, numéro de la ligne) de la zone dans laquelle sera placé le composant (**gridx**, **gridy**)
La zone du composant pouvant occuper plusieurs cellules, on indiquera les coordonnées de la première cellule occupée (en haut à gauche)
 - Le **nombre de cellules occupées** (nombre de lignes, nombre de colonnes) par la zone (**gridwidth**, **gridheight**)
 - La **position du composant** (ancre) à l'intérieur de sa zone : *nord*, *sud*, *est*, *ouest*, *centre*, *nord-est*, ... (**anchor**)
 - Un **indicateur de remplissage** qui précise si le composant doit s'agrandir pour occuper tout l'espace de la zone si la zone est plus grande que sa taille préférée. On peut distinguer le remplissage horizontal et vertical (**fill**)
 - Un **indicateur d'agrandissement** qui précise dans quelle proportion les lignes et les colonnes doivent modifier leurs tailles lorsque le conteneur change de dimension (**weightx**, **weighty**). La valeur maximale de chaque ligne, respectivement de chaque colonne, est prise en compte.
 - Une **marge externe**, en pixels, autour du composant (**insets**)
 - Une **marge interne**, en pixels, à ajouter, de chaque côté, à la taille minimale du composant (**ipadx**, **ipady**)

[Très rarement utilisé]



GridBagConstraints [2]

| Contrainte | Type | Description | Valeurs possibles |
|---------------------------------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| gridx gridy | int | Position (n° colonne, ligne) de la première cellule (en haut, à gauche) de la zone associée au composant | Valeur entière ≥ 0 Par défaut : placé à droite (resp. en dessous) du précédent (RELATIVE) |
| gridheight gridwidth | int | Nombre de cellules que comprend la zone (nombre de lignes et nombre de colonnes) | Valeur entière > 0 Par défaut : 1 |
| anchor | int | Position d'ancre du composant dans sa zone | CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST Par défaut : CENTER |
| fill | int | Indicateur de remplissage dans le sens horizontal, vertical, ou les deux (étalement du composant dans la zone) | NONE, HORIZONTAL, VERTICAL, BOTH Par défaut : NONE |
| weightx weighty | double | Poids de la répartition des espaces additionnels dans les lignes et les colonnes lors du redimensionnement du conteneur. La valeur maximale de chaque ligne resp. de chaque colonne est prise en compte dans le calcul | Valeurs réelles ≥ 0.0 Par défaut : 0.0 Les valeurs n'ont de significations que relativement les unes par rapport aux autres. |
| insets | Insets | Marge externe, en pixels, autour du composant (à l'intérieur de sa zone) | Objet de type Insets (valeurs entières pour les marges) |
| ipadx ipady | int | Marge interne (en x et en y) qui augmente la taille minimale du composant (ajoutée de chaque côté) | Valeurs entières ≥ 0 Par défaut : 0 |





GridBagConstraints [3]

▪ Quelques remarques additionnelles :

- Les valeurs associées à **weightx** et **weighty** influenceront le redimensionnement des colonnes, respectivement des lignes de la grille. C'est la **valeur maximale** de chaque colonne respectivement de chaque ligne qui est prise en compte dans le calcul.
- La valeur **weightx** d'une colonne est comparée à la somme des valeurs **weightx** de toutes les colonnes de la grille pour déterminer la proportion d'espace additionnel qui lui sera attribuée.
- La valeur **fill** indique si le composant doit s'adapter à la grandeur de sa zone. Le composant peut s'étaler horizontalement, verticalement, dans les deux directions ou ne pas s'étaler du tout (même si la zone s'agrandit).
- Si **weightx** (**weighty**) est égal à zéro pour tous les composants, l'espace additionnel sera ajouté autour de la grille, à gauche et à droite (resp. en haut et en bas). La grille sera ainsi centrée à l'intérieur du conteneur.
- Certaines constantes particulières de la classe **GridBagConstraints** (**RELATIVE**, **REMAINDER**) peuvent être utilisées avec les propriétés **gridx**, **gridy**, **gridwidth**, **gridheight**.



GridBagConstraints [4]

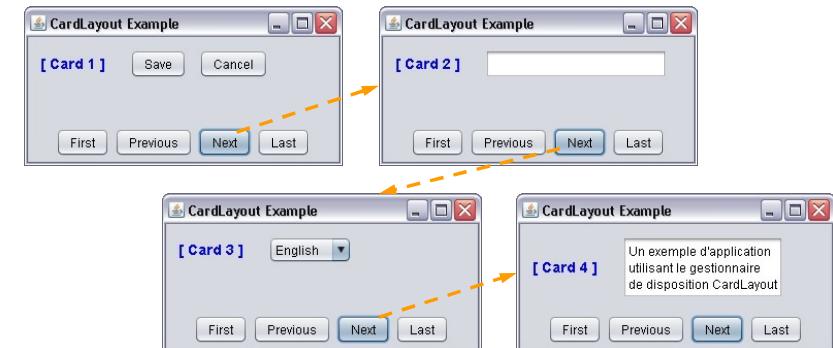
```
JPanel conteneur = new JPanel(); // Création du conteneur
JLabel lbA = new JLabel("Texte");
JButton btA = new JButton("Ok");
GridBagLayout gbLayout = new GridBagLayout(); // Layout Manager
conteneur.setLayout(gbLayout);
GridBagConstraints gbc = new GridBagConstraints(); // Contraintes
gbc.gridx = 0;
gbc.gridy = 0;
gbc.anchor = GridBagConstraints.NORTH;
conteneur.add(lbA, gbc); // Ajout du composant lbA avec ses contraintes
gbc.gridx = 1;
gbc.anchor = GridBagConstraints.CENTER;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.weightx = 1.0;
conteneur.add(btA, gbc); // Ajout du composant btA avec ses contraintes
. . .
```



CardLayout

▪ **CardLayout (java.awt)**

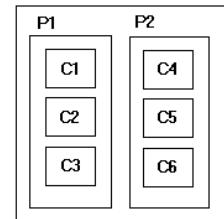
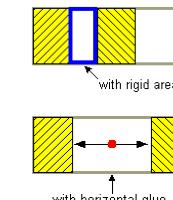
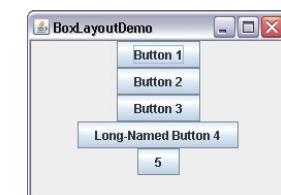
- Élargit chaque composant à la taille du conteneur et n'en affiche qu'un à la fois à la manière d'un jeu de cartes dont on ne voit que la première.
- Plusieurs méthodes permettent de modifier le composant actuellement affiché (**first()**, **last()**, **next()**, **previous()**, **show()**).
- Le conteneur **JTabbedPane** est généralement plus pratique pour l'utilisateur



BoxLayout [1]

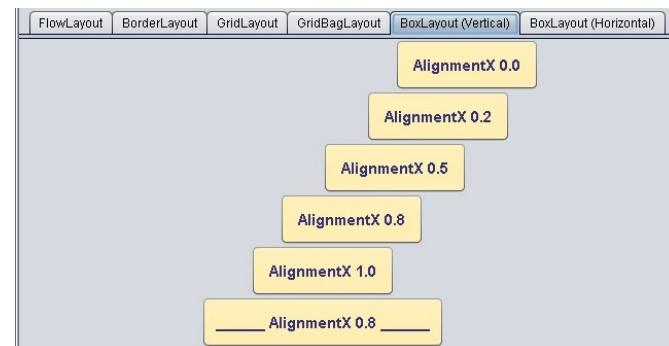
▪ **BoxLayout (javax.swing)**

- C'est le gestionnaire employé par le conteneur **Box**.
- Il arrange ses enfants **horizontalement** (dans une ligne, de gauche à droite) ou **verticalement** (dans une colonne, de haut en bas).
- Des espaces fixes et variables peuvent être définis au moyen d'**espaces** (**Struts** ou **RigidArea**) et de **ressorts** (**Glue**). Ces composants invisibles peuvent être créés à l'aide de méthodes statiques de la classe **Box**.
- Un emboîtement de différents conteneurs **Box** peut parfois remplacer l'utilisation du conteneur **GridBagLayout** et ses contraintes (mais les fonctionnalités ne sont pas totalement équivalentes).





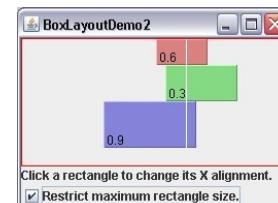
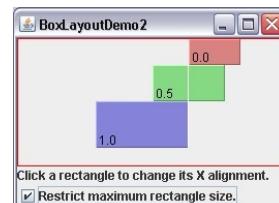
BoxLayout [2]



BoxLayout [3]

- Le gestionnaire de disposition **BoxLayout** prend en compte les propriétés **alignmentX** et **alignmentY** des composants enfants.
- Ces propriétés permettent de contrôler la position relative des composants horizontalement (x) ou verticalement (y) selon le type de gestionnaire utilisé.
- La classe **Box** représente un conteneur géré par **BoxLayout**. Elle contient un certain nombre de méthodes statiques utilitaires permettant de simplifier le travail lors de l'utilisation de ce gestionnaire de disposition.

Exemples :
Différentes valeurs pour la propriété **alignmentX**



BoxLayout [4]

- Le gestionnaire de disposition **BoxLayout** respecte généralement la propriété **preferredSize** mais agrandit - curieusement - certains composants (par exemple la hauteur de **JTextField** et **JComboBox**) jusqu'à la valeur de la propriété **maximumSize** et, ceci, même en présence de composants de type **Glue** sensés occuper tout le reste de l'espace disponible.

- On peut cependant contourner ce problème en limitant la taille maximale des composants concernés :

```
comp.setMaximumSize(comp.getPreferredSize());
```

- Exemple :

```
JLabel lbPrenom = new JLabel("Firstname");
JTextField tfPrenom = new JTextField(10);
JLabel lbNom = new JLabel("Lastname");
JTextField tfNom = new JTextField(8);

Box pnFields = Box.createVerticalBox();
```

[Suite à la page suivante...]



BoxLayout [5]

[Suite ...]

```
pnFields.add(Box.createVerticalGlue());
lbPrenom.setAlignmentX(0.5f);
pnFields.add(lbPrenom);

pnFields.add(Box.createVerticalStrut(3));
tfPrenom.setMaximumSize(tfPrenom.getPreferredSize());
tfPrenom.setAlignmentX(0.5f);
pnFields.add(tfPrenom);

pnFields.add(Box.createVerticalGlue());
lbNom.setAlignmentX(0.5f);
pnFields.add(lbNom);
pnFields.add(Box.createVerticalStrut(3));

tfNom.setMaximumSize(tfNom.getPreferredSize());
tfNom.setAlignmentX(0.5f);
pnFields.add(tfNom);
pnFields.add(Box.createVerticalGlue());

getContentPane().add(pnFields);
...
```

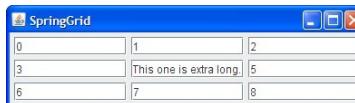




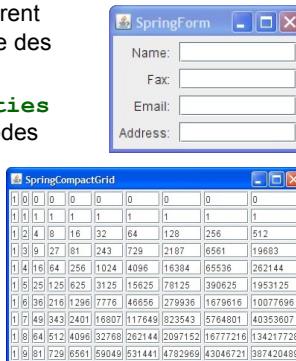
SpringLayout

▪ SpringLayout (javax.swing)

- Introduit avec la version 1.4 du SDK, le gestionnaire **SpringLayout** a été conçu pour être utilisé par les générateurs visuels d'interfaces graphiques (*GUI Builder*). Il prête assez mal à une programmation manuelle.
- Comme pour GridBagLayout, le gestionnaire **SpringLayout** se base sur des contraintes (classe interne **SpringLayout.Constraints**).
- En outre, des objets de type **Spring** enregistrent différentes propriétés et se comportent comme des ressorts qui peuvent être combinés.
- Une classe utilitaire nommée **SpringUtilities** (qui doit être téléchargée) offre différentes méthodes (**makeGrid**, **makeCompactGrid**, ...) permettant de simplifier la création d'interfaces de type grille, par exemple l'association de libellés et de champs textes ou autres tableaux.



EIA-FR / Jacques Bapst



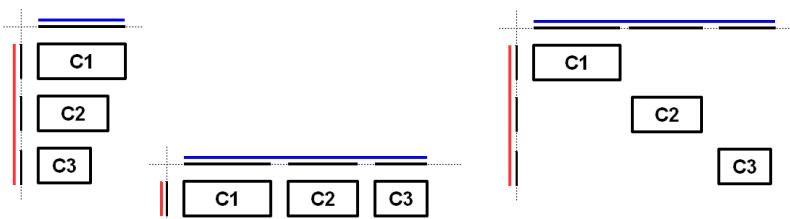
IHM1_03 41



GroupLayout

▪ GroupLayout (javax.swing)

- Le gestionnaire **GroupLayout** a été introduit avec la version 1.6 du SDK. Il a été principalement conçu pour la génération automatique de code (*GUI Builder*) et est notamment utilisé par le générateur *Matisse* (*NetBeans*). Il reste cependant plus ou moins utilisable pour un codage "à la main".
- Le gestionnaire **GroupLayout** traite les deux axes (horizontal et vertical) de manière indépendante. Des espacements par défauts sont gérés (Gaps).
- La disposition est effectuée grâce à la classe interne (abstraite) **Group** et aux deux sous-classes internes (concrètes) **SequentialGroup** et **ParallelGroup**



EIA-FR / Jacques Bapst

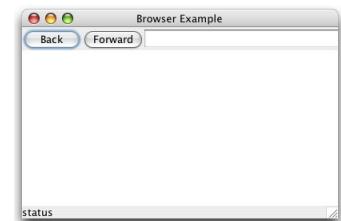
IHM1_03 42



ZoneLayout [1]

- **ZoneLayout** est un gestionnaire de disposition original qui ne fait pas partie de la plate-forme Java.
- Il est cependant intéressant de considérer ce gestionnaire *open-source* que l'on peut télécharger sur le site www.zonelayout.com.
- Son intérêt principal vient du fait que les contraintes de disposition sont données sous une forme très visuelle (un tableau de caractères) ce qui rend le code très lisible tout en offrant des possibilités proches de celles du gestionnaire **GridBagLayout** (sans être véritablement équivalent).
- Principe :

| b | f | a | . | . | a |
|---|---|---|---|---|---|
| w | . | . | . | . | . |
| . | . | . | . | . | w |
| s | . | . | . | . | s |



EIA-FR / Jacques Bapst

IHM1_03 43



ZoneLayout [2]

- Après avoir créé un gestionnaire de type **ZoneLayout** à l'aide de la méthode statique **newZoneLayout()** (de la classe **ZoneLayoutFactory**), il y a ensuite deux étapes principales :
 - Définir la disposition à l'aide de la méthode **addRow()** ;
 - Ajouter les composants dans le conteneur avec la méthode **add()** , en les liant à l'une des zones préalablement définies (*Binding*).

```
ZoneLayout zLayout = ZoneLayoutFactory.newZoneLayout();
zLayout.addRow("bfa...a"); // Layout definition
zLayout.addRow("w.....");
zLayout.addRow(".....w");
zLayout.addRow("s....s");
JPanel zPanel = new JPanel(zLayout);
zPanel.add(new JButton("Back"), "b"); // Binding
zPanel.add(new JButton("Forward"), "f"); //
zPanel.add(new JTextField(30), "a"); //
zPanel.add(. . .);
```

EIA-FR / Jacques Bapst

IHM1_03 44



ZoneLayout [3]

- Dans la définition de la disposition, chaque **zone** est identifiée par une **lettre** (`Character.isLetter()`) qui délimite le début et la fin de la zone.
- Une zone peut être représentée par une seule lettre ou alors par deux fois la même lettre qui délimite le début et la fin de la **zone rectangulaire**.
- Une zone peut commencer sur une ligne et se terminer sur une autre.
- Des points ('.') sont utilisés pour conserver l'alignement (sans aucune autre signification).
- Des **modificateurs** peuvent être ajoutés à l'intérieur de chaque zone pour préciser le comportement de la zone elle-même et du composant qui se trouve à l'intérieur (taille, alignement, ...).
- Les modificateurs sont des caractères particuliers dont la signification est décrite à la page suivante.

| | | |
|---|---|---|
| b | f | a |
| w | . | w |
| s | . | s |

```
"u..up...px.."
"k....."
".....k..."
"g.....gx.."
```

```
"u>.up<..px|."
"k*....."
".....k..."
"~g.....gx.."
```



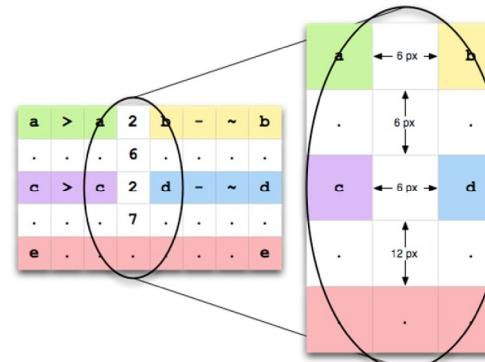
ZoneLayout / Modificateurs

| | | | |
|---|-------------------------------------------------------------------------|---------------------|--|
| < | Aligne le composant à gauche de la zone | (centré par défaut) | |
| > | Aligne le composant à droite de la zone | (centré par défaut) | |
| ^ | Aligne le composant en haut de la zone | (centré par défaut) | |
| - | Aligne le composant en bas de la zone (Underscore) | (centré par défaut) | |
| - | Étend le composant pour occuper horizontalement toute la zone (Dash) | | |
| | Étend le composant pour occuper verticalement toute la zone | | |
| + | Étend le composant pour occuper toute la zone (horizontal + vertical) | | |
| ~ | Étend la zone horizontalement pour qu'elle occupe tout l'espace | | |
| ! | Étend la zone verticalement pour qu'elle occupe tout l'espace | | |
| * | Étend la zone pour qu'elle occupe tout l'espace (horizontal + vertical) | | |



ZoneLayout / Espaceurs

- Des modificateurs particuliers appelé **espaceurs** peuvent être insérés en dehors et entre les zones.
- Ces espaces sont codés par des chiffres selon la table ci-contre.
- Exemple d'utilisation :



| | | |
|---|---------------------|-----------|
| 1 | Espaceur horizontal | 3 pixels |
| 2 | Espaceur horizontal | 6 pixels |
| 3 | Espaceur horizontal | 12 pixels |
| 4 | Espaceur horizontal | 18 pixels |
| 5 | Espaceur vertical | 3 pixels |
| 6 | Espaceur vertical | 6 pixels |
| 7 | Espaceur vertical | 12 pixels |
| 8 | Espaceur vertical | 18 pixels |



ZoneLayout / Templates

- Le gestionnaire de disposition **ZoneLayout** permet de créer des **modèles de zones (Templates)** auquel on donne un nom et qui peuvent être ensuite réutilisés en invoquant la méthode `insertTemplate()`.

```
ZoneLayout zL = ZoneLayoutFactory.newZoneLayout();
zL.addRow("2a.....a2"); // Layout definition
zL.addRow("....7....");
zL.addRow("2k>.k2t~.t2", "inputTemplate"); // Template definition
zL.addRow("....6.....", "inputTemplate"); //

JPanel zPanel = new JPanel(zL);
zPanel.add(new JLabel("Titre"), "a");
for (int i=0; i<4; i++) {
    zL.insertTemplate("inputTemplate");
    zPanel.add(new JLabel("Label "+i), "k");
    zPanel.add(new JTextField(10), "t");
}
```

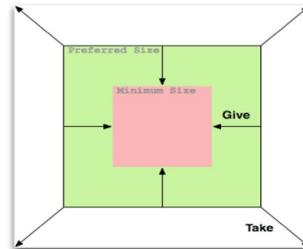




ZoneLayout / Give and Take [1]

- La répartition de l'espace à disposition des zones peut être contrôlée en donnant une pondération appelée **Give** pour la réduction de l'espace (s'il n'y a pas assez de place pour la taille préférée) et **Take** pour la répartition de l'espace supplémentaire (s'il y a trop de place à disposition).
- Ces pondérations peuvent être définies (sur l'axe x et sur l'axe y) par des valeurs entières (comprises entre 0 et 100) transmises aux méthodes suivantes :
 - `setGive(int xWeight, int yWeight); // Par défaut : compact size`
 - `setTake(int xWeight, int yWeight); // Par défaut : (0, 0)`

- Avant d'invoquer ces méthodes, il est nécessaire de *compiler* la disposition (c'est-à-dire de transformer et compacter le tableau de caractères et de l'enregistrer dans une structure interne).



EIA-FR / Jacques Bapst



IHM1_03 49



ZoneLayout / Give and Take [2]

- Exemple :

```

zL.addRow("b.....bc.....c");
zL.addRow("m>~m2c*~.....c2p<~p");
zL.addRow("f.....fg.....gh.....h");

zL.compile();           // Nécessaire avant de modifier Take ou Give

zL.getZone("m").setTake( 3, 1); // Horizontal, Vertical
zL.getZone("c").setTake(10, 2);
zL.getZone("p").setTake( 1, 1);
  
```

Pour plus de détails concernant le gestionnaire de disposition **ZoneLayout**, il faut consulter la documentation (manuel, Javadoc, etc.) disponible sur le site du projet (www.zonelayout.com).

EIA-FR / Jacques Bapst



IHM1_03 50



MigLayout

- MigLayout** est un gestionnaire de disposition *open-source* (créé par Mikael Grev) qui ne fait pas partie de la plate-forme Java mais qui offre des caractéristiques intéressantes et qui peut constituer un très puissant substitut à **GridBagLayout** (avec un côté nettement moins "cryptique" et une paramétrisation plus étendue).
- Ce gestionnaire (+ documentation) est disponible sur www.miglayout.com
- Il est conçu de manière à être indépendant de la librairie de composants utilisée. Des versions existent pour Swing, SWT (Eclipse) et JavaFX.
- Les contraintes (que l'on peut définir à différents niveaux) sont données sous forme de chaînes de caractères (liste de mots-clés) ce qui favorise la lisibilité du code (au détriment des contrôles possibles à la compilation !).
- Ce gestionnaire prend en compte, par défaut, certaines éléments propres à chaque système d'exploitation (espacement standard des composants, ordre de certains boutons, etc.) et simplifie grandement la création d'interfaces classiques tout en permettant des dispositions plus "exotiques".

EIA-FR / Jacques Bapst



IHM1_03 51



MigLayout – Principe [1]

- MigLayout** permet de disposer les composants dans une grille selon différentes approches
 - Flowing Layout** : disposition séquentielle (horizontale ou verticale) des composants
 - Grid-Based Layout** : disposition basée sur les coordonnées de la grille
 - Docking Layout** : composants placés sur les bords du conteneur
 - Absolute Positioning** : positionnement absolu des composants (ou positionnement relatif aux autres composants)
- Les **contraintes** peuvent être données sur **trois niveaux**
 - Layout Constraints** : au niveau du conteneur (global)
 - Row/Column Constraints** : au niveau de chaque ligne et colonne
 - Component Constraints** : au niveau de chaque composant

```

// Contraintes données lors de la création du gestionnaire de disposition
MigLayout migL = new MigLayout("fill", // Layout Constraints
                               "[right][left]", // Columns Constraints
                               "40[]20[top]"); // Rows Constraints
  
```

EIA-FR / Jacques Bapst

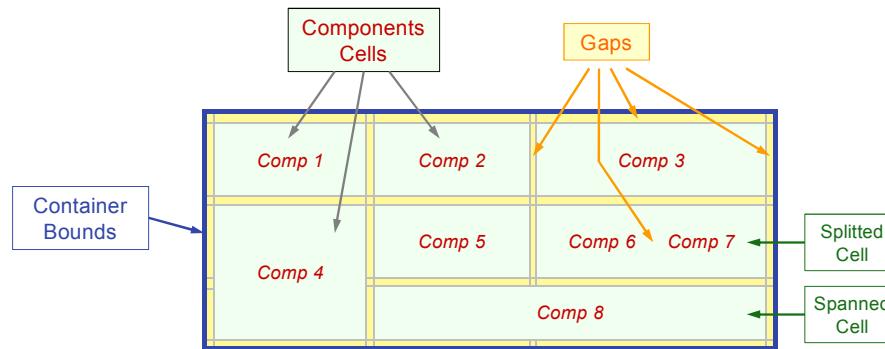


IHM1_03 52



MigLayout – Principe [2]

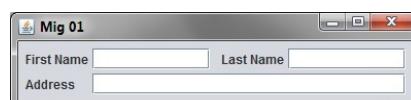
- La grille est créée automatiquement lors de l'insertion des composants, en fonction de la séquence d'insertion et des contraintes données
- Les cellules peuvent couvrir plusieurs lignes et/ou colonnes (*span*)
- Plusieurs composants peuvent occuper la même cellule (*split*)



MigLayout – Exemple [1]

- Un premier exemple
 - Positionnement séquentiel, grille créée automatiquement (*Flowing Layout*)
 - Gestion des espaces entre les composants (*gaps*)
 - Ajout de gauche à droite (retour à la ligne avec *wrap*)

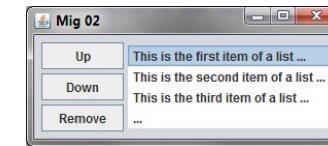
```
JPanel p = new JPanel(new MigLayout());
p.add(lblFName);
p.add(txrFName);
p.add(lblLName, "gap unrelated"); // Component Constraint
p.add(txrLName, "wrap");
p.add(lblAddress);
p.add(txrAddress, "span, growx");
```



MigLayout – Exemple [2]

- Exemple simple
 - Positionnement séquentiel, grille créée automatiquement (*Flowing Layout*)
 - Les composants peuvent occuper toute la cellule (*growx, growy*)
 - Les composants peuvent occuper plusieurs cellules de la grille (*spanx, spany*)

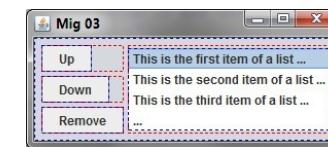
```
JPanel p = new JPanel(new MigLayout());
p.add(btnUp, "growx");
p.add(lstItems, "spany 3, wrap");
p.add(btnDown, "growx, wrap");
p.add(btnRemove, "growx");
```



MigLayout – Exemple [3]

- Exemple simple
 - Positionnement explicite dans la grille (*Grid-Based Layout*)
 - Utilisation des coordonnées des cellules (*cell x y*)
 - Mode *Debug* (affichage des zones des cellules)

```
JPanel p = new JPanel(new MigLayout("debug"));
p.add(btnUp, "cell 0 0"); // x, y
p.add(btnDown, "cell 0 1");
p.add(btnRemove, "cell 0 2");
p.add(lstItems, "cell 1 0 1 3"); // x, y, spanx, spany
```





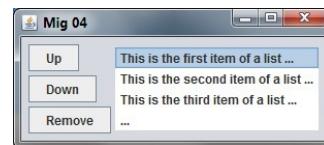
MigLayout – Exemple [4]

- Exemple simple

- Positionnement séquentiel, grille créée automatiquement (*Flowing Layout*)
- Ajout des composants de haut en bas (*flowy*)
- Résultat identique

```
JPanel p = new JPanel(new MigLayout("flowy"));

p.add(btnUp,      "");
p.add(btnDown,    "");
p.add(btnRemove, "wrap");
p.add(lstItems,  "spany 3");
```



MigLayout – Exemple [5]

- Exemple simple

- Les cellules peuvent être partagées (*split*) et occupées par plusieurs composants

```
JPanel p = new JPanel(new MigLayout());

p.add(lblFrom);
p.add(txfFrom,      "wrap");
p.add(lblTo);
p.add(txfTo,        "wrap");
p.add(lblDuration);
p.add(txfDuration, "split 2");
p.add(lblSeconds);
```



MigLayout – Exemple [6]

- Exemple simple

- L'alignement peut être défini dans les contraintes des colonnes (*right*)
- Effet global si plusieurs composants occupent une cellule (*split*)

```
JPanel p = new JPanel(new MigLayout("", "[right][]")); // Default is "left"

p.add(lblFrom);
p.add(txfFrom,      "wrap");
p.add(lblTo);
p.add(txfTo,        "wrap");
p.add(lblDuration);
p.add(txfDuration, "split 2");
p.add(lblSeconds);
```



MigLayout – Unités [1]

- La taille et la position des éléments peuvent être exprimées dans différentes unités. Par défaut l'unité est le pixel (*px*).

- Principales unités

| Unité | Description |
|-----------|-----------------------------------------------------------------------------------------------------------------|
| px | Pixel |
| lp | Logical pixel (tient compte du facteur d'agrandissement des polices de la plate-forme) |
| % | Pourcentage de la taille du conteneur Alignements : 0% = aligné à gauche 50% = centré 100% = aligné à droite |
| sp | Screen percentage : 100sp=bord droit/inférieur de l'écran |
| in | Pouce (25.4 mm) |
| pt | Point typographique (1/72 ^{ème} de pouce ≈ 0.353 mm) |
| mm | Millimètre (avec prise en compte de la résolution de l'écran [DPI]) |
| cm | Centimètre (avec prise en compte de la résolution de l'écran [DPI]) |
| al | Alignement utilisé avec positionnement absolu 0al=aligné à gauche 0.5al=centré 1al=aligné à droite |
| n | Absence de valeur (synonyme: null) |





MigLayout – Unités [2]

- Quelques autres mots-clés peuvent être utilisés comme unités. Ils seront convertis en pixels en prenant en compte les caractéristiques de la plate-forme cible (le composant `PlatformConverter` se charge de cette conversion).
- Mots-clés :

| Unité | Description |
|------------------------------------|-------------------------------------------------------------------------------------------------|
| <code>r / rel / related</code> | Espace standard entre deux composants associés (par exemple un libellé et le composant associé) |
| <code>u / unrel / unrelated</code> | Espace standard entre deux composants indépendants (sans relations particulières) |
| <code>p / para / paragraph</code> | Espace standard entre deux paragraphes (utilisé généralement pour un espace vertical) |
| <code>i / ind / indent</code> | Espace standard pour une indentation (retrait) (utilisé généralement pour un espace horizontal) |



MigLayout – Unités [3]

- Les unités peuvent également être utilisées pour référencer la taille de certains composants.
- Ces unités peuvent être utilisées dans les contraintes de lignes et de colonnes ou comme référence dans des contraintes de composants.
- Mots-clés :

| Unité | Description |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>min / minimum</code> | Une référence à la taille minimale du composant ou à la plus grande taille minimale de la colonne ou de la ligne considérée |
| <code>pref / preferred</code> | Une référence à la taille préférée du composant ou à la plus grande taille préférée de la colonne ou de la ligne considérée |
| <code>max / maximum</code> | Une référence à la taille maximale du composant ou à la plus petite taille maximale de la colonne ou de la ligne considérée |
| <code>button</code> | Taille minimale d'un bouton sur la plateforme cible. <u>Remarque</u> : Cette unité ne peut être utilisée que pour définir la largeur d'un composant |



MigLayout – Alignement

- Pour définir l'alignement des composants dans les cellules, les mots-clés suivants peuvent être utilisés.
- Mots-clés :

| Unité | Description |
|-------------------------------|----------------------------------------------------------------------------------------|
| <code>t / top</code> | Haut |
| <code>l / left</code> | Gauche |
| <code>b / bottom</code> | Bas |
| <code>r / right</code> | Droite |
| <code>c / center</code> | Centre |
| <code>lead / leading</code> | Alignement dépendant de l'orientation (<i>left-to-right</i> ou <i>right-to-left</i>) |
| <code>trail / trailing</code> | Alignement dépendant de l'orientation (<i>left-to-right</i> ou <i>right-to-left</i>) |
| <code>base / baseline</code> | Alignement sur la ligne de base des textes (dès JDK 6) |
| <code>label</code> | Alignement standard des libellés selon la plate-forme |



MigLayout – BoundSize [1]

- La notion de **BoundSize** permet de définir des contraintes sur la taille en indiquant les tailles **minimale / préférée / maximale**
- La syntaxe est la suivante :

$$\text{min} : \text{preferred} : \text{max}$$
- Si une des valeurs est manquante ou `null` elle sera remplacée par une valeur appropriée (la valeur par défaut du composant)
- Une valeur seule indique la taille préférée
 - "12" est donc équivalent à "`null:12:null`" ou "`n:12:n`" ou "`:12:`"
- Deux valeurs indiquent la taille minimale et préférée
 - "6:12" est donc équivalent à "`6:12:null`" ou "`6:12:`"
- Une valeur suivie de "!" indique trois valeurs identiques
 - "12!" est donc équivalent à "`12:12:12`"





MigLayout – BoundSize [2]

- La notion de **BoundSize** peut notamment être utilisée pour définir les **espacements (gaps)** entre les lignes, les colonnes, les composants, ...
- Dans ce cas, le mot-clé "**push**" peut être ajouté à la fin du *BoundSize* (ou utilisé seul) pour rendre l'espacement "glouton" (*greedy*) qui occupera ainsi tout l'espace restant. Il agira comme un ressort qui repousse les composants voisins.
 - Exemples : `"1cm:push"`
`"rel:push"`
`"[] [] 6:12:push[] []"`

EIA-FR / Jacques Bapst



IHM1_03 65



MigLayout – Contraintes Colonnes/Lignes

- La syntaxe pour les contraintes sur les colonnes et les lignes est identique. Exemple pour les colonnes :


```
gap [col0-constr] gap [col1-constr] gap [col2-constr] gap...
```
- Chaque contrainte peut être composée de plusieurs éléments séparés par une virgule


```
[coln-constr1, coln-constr2, coln-constr3, ...]
```
- Exemple

```
new MigLayout(
    "",
    "[80]unrel[120!]10px[10:20:30]",
    "6[]25mm[grow,fill]paragraph[]");
    // Layout constraints
    // Columns constraints
    // Rows constraints
```

EIA-FR / Jacques Bapst



IHM1_03 66



MigLayout – Espacements (Gaps)

- L'espacement autour des composants peut être défini dans les contraintes des composants par le mot clé "**gap**" suivi des valeurs (avec les unités) selon la syntaxe suivante : `gap left right`
`gap top bottom`
- Les valeurs sont spécifiées sous la forme de *BoundSize*
- Alternativement, les mots-clés suivants (avec une seule valeur) peuvent également être utilisés : `"gapleft"`, `"gapright"`, `"gaptop"`, `"gapbottom"`, `"gapbefore"`, `"gapafter"`.
- Exemple

```
p.add(lblName, "gap 12 6");
p.add(lblName, "gap 6:push");
p.add(txfName, "gap 10:20:40 1cm!");
p.add(lstMail, "gapleft unrelated");
```

EIA-FR / Jacques Bapst



IHM1_03 67



MigLayout – Espacements (Insets)

- L'espacement autour du conteneur parent (marges) peut être défini dans les contraintes du conteneur par le mot clé "**insets**" suivi des valeurs (avec les unités) selon la syntaxe suivante :


```
insets top left bottom right
```
- Si une seule valeur est mentionnée, elle s'appliquera à toutes les marges
- Les valeurs `"panel"` et `"dialog"` correspondent à des marges par défaut de la plate-forme cible pour les conteneurs standards et les boîtes de dialogue
- Exemple

```
new MigLayout("insets 10 20 10 20");
new MigLayout("insets 0.8cm");
new MigLayout("insets dialog");
```

EIA-FR / Jacques Bapst

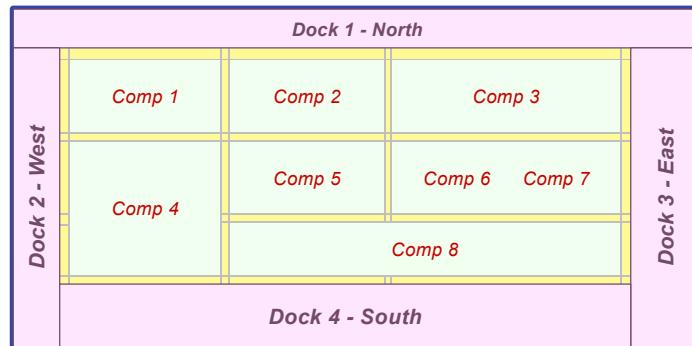


IHM1_03 68



MigLayout – Docking [1]

- La notion de **docking** permet d'arrimer certains composants sur les bords de la grille et au centre (un peu à la manière de *BorderLayout* mais avec plus de flexibilité et sans être limité par le nombre de composants)
- L'ordre dans lequel les composants sont insérés influence la disposition (les premiers occupent la place disponible)



EIA-FR / Jacques Bapst



IHM1_03

69



MigLayout – Docking [2]

- Les contraintes suivantes (associées aux composants) permettent de les placer en bordure ainsi qu'au centre de la grille

[dock] north
 [dock] west dock center [dock] east
 [dock] south

- Le mot-clé "dock" est optionnel, sauf pour le centre
- Plusieurs composants peuvent occuper les bords et même le centre
- Le redimensionnement des composants ainsi arrimés suit les mêmes règles que celles qui s'appliquent à *BorderLayout* (redimensionnement dans une des directions pour les composants en bordure et redimensionnement dans les deux directions pour ceux placés au centre)
- L'espace par rapport aux composants voisins (et par rapport aux limites du conteneur) peut être géré avec la contrainte **gap** (par défaut 0 px)



EIA-FR / Jacques Bapst

IHM1_03

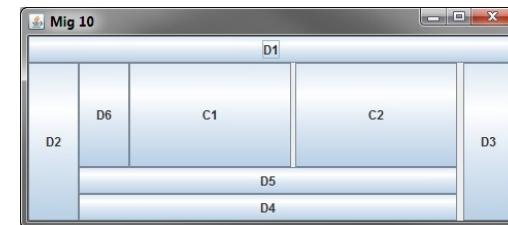
70



MigLayout – Docking [3]

- Exemple

```
 JPanel p = new JPanel(new MigLayout());
 p.add(btnD1, "north");
 p.add(btnD2, "west");
 p.add(btnD3, "east,gap 6 0");
 p.add(btnD4, "south");
 p.add(btnD5, "south");
 p.add(btnD6, "west");
 p.add(btnC1, "dock center");
 p.add(btnC2, "dock center");
```



EIA-FR / Jacques Bapst



IHM1_03

71



MigLayout – Grow/Shrink [1]

- Si le conteneur ne possède pas la dimension optimale (qui respecte la taille préférée de l'ensemble des composants qu'il contient), la taille des lignes/colonnes est gérée par les contraintes **grow** et **shrink** que l'on peut associer aux lignes/colonnes ou aux composants individuels
- On peut associer un poids (valeur numérique) aux contraintes pour indiquer la proportion relative de l'espace qui doit être alloué ou restitué (la valeur absolue du poids n'a pas de signification, c'est seulement relativement aux autres valeurs qu'elle est interprétée)

grow 50 grow shrink 2.5 shrink

- Le poids par défaut est 100
- Si la contrainte n'est pas mentionnée, cela correspond à **grow 0** resp. **shrink 0** (la ligne/colonne ne changera pas de taille)
- Le redimensionnement s'effectue dans les limites définies par la taille minimale (voir **min**) et la taille maximale (voir **max**) des composants



EIA-FR / Jacques Bapst

IHM1_03

72



MigLayout – Grow/Shrink [2]

- Les contraintes **growx**, **growy**, **shrinkx** et **shrinky** peuvent être associées à un composant individuel pour indiquer le comportement dans chacun des axes (en donnant un poids [par défaut 100] permettant par exemple de modifier la valeur par défaut de la ligne/colonne)

```
growx 100  growy 50  shrinkx 0  shrinky 0
```

- Les contraintes **growprio p** et **shrinkprio p** permettent de définir des groupes de priorités dans le redimensionnement
- Lors de l'agrandissement, toutes les lignes/colonnes avec la plus haute priorité s'agrandissent jusqu'à leur taille maximale puis les lignes / colonnes avec une valeur inférieure sont prises en considération et ainsi de suite à chaque niveau

```
growprio 150  growprio 80  shrinkprio 50
```

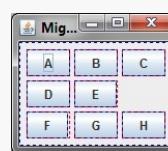
- La valeur par défaut de la priorité **p** est 100



MigLayout – Grow/Shrink [3]

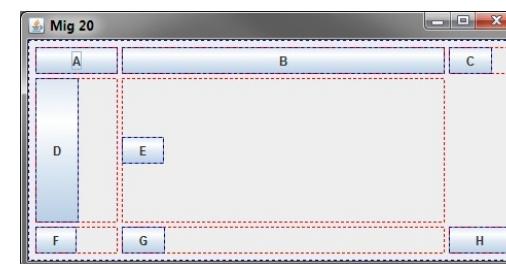
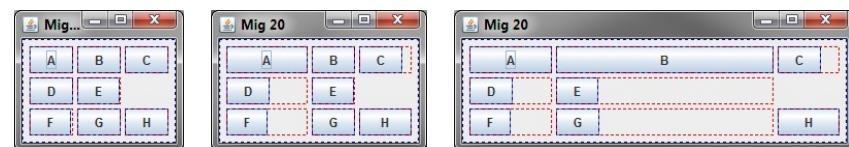
- Exemple

```
MigLayout migL = new MigLayout(
    "debug",
    "[6::80,grow 80] [growprio 1,grow] [6::60,grow 20]",
    "[] [grow] []"
);
migPanel = new JPanel(migL);
//--- First Row
migPanel.add(btnA, "grow");
migPanel.add(btnB, "grow");
migPanel.add(btnC, "wrap");
//--- Second Row
migPanel.add(btnD, "growy");
migPanel.add(btnE, "wrap");
//--- Third Row
migPanel.add(btnF);
migPanel.add(btnG);
migPanel.add(btnH, "grow");
```




MigLayout – Grow/Shrink [4]

- Comportement lors du redimensionnement



MigLayout – Autres fonctionnalités

- Le gestionnaire de disposition **MigLayout** offre encore de nombreuses possibilités qui ne sont pas décrites ici (consulter la documentation)

- Positionnement absolu et relatif (lien avec la position d'autres composants) (**pos**)
- Groupement des colonnes, lignes, composants pour s'assurer qu'ils possèdent la même hauteur ou la même largeur (**sizemode**)
- Spécification du comportement des composants non-visible (**hidemode**)
- Taille et ordre des boutons standard selon la plate-forme (**tag**)
- Mode **Flow-only**, sans grille (**nogrid**)
- Composants non gérés par *MigLayout* (**external**)
- Utilisation d'expressions (**parent.width*0.2+1cm**)
- Appel de méthodes pour les contraintes (**API constraints**) en lieu et place des chaînes de caractères
- Et la liste n'est pas exhaustive...





Autres gestionnaires de disposition

- Il existe d'autres gestionnaires spécialisés comme **OverlayLayout**, **JViewportLayout** ou **ScrollPaneLayout** qui sont spécifiquement utilisés par certains conteneurs spécialisés.
On ne les emploie généralement pas directement.
- Il est également possible de travailler **sans gestionnaire de disposition** (sans *Layout Manager*) en enregistrant la référence **null** à la place d'un objet de type **LayoutManager**.

Dans ce cas, la position absolue des composants doit être définie explicitement (*x*, *y*) par exemple en utilisant une des méthodes **setLocation()** ou **setBounds()** de **java.awt.Component**.

Sans gestionnaire de disposition, la taille des composants doit être définie à l'aide de la méthode **setSize()**.

Sauf cas particuliers, **ce mode de fonctionnement n'est pas recommandé** (il n'est d'ailleurs pas autorisé et lève une exception dans certains conteneurs).



Enregistrement d'un Layout Manager [1]

- Si, pour un conteneur, on ne souhaite pas utiliser le *Layout Manager* défini par défaut, il faut le modifier en utilisant la méthode **setLayout()** héritée de la classe **Container**.
- Pour une partie des conteneurs on peut également définir le *Layout Manager* en utilisant certaines variantes des constructeurs.

```
// Création d'un conteneur JPanel
JPanel p1 = new JPanel(); // FlowLayout par défaut
// Enregistrement d'un gestionnaire BorderLayout
p1.setLayout(new BorderLayout());

// Création d'un conteneur JPanel en définissant le LayoutManager
// dans le constructeur
JPanel p2 = new JPanel(new BorderLayout());
```



Enregistrement d'un Layout Manager [2]

- Pour les **conteneurs de premier-niveau** l'enregistrement du Layout Manager s'effectue dans le panneau de contenu (**Content-Pane**) du **JRootPane** car c'est lui qui contient effectivement les composants.
- Par défaut, le *Content-Pane* est un conteneur de type **JPanel** qui utilise le gestionnaire **BorderLayout**.

```
// Création d'une Frame
JFrame f1 = new JFrame();

// Enregistrement d'un gestionnaire GridLayout (2 lignes et 3 colonnes)
f1.getContentPane().setLayout(new GridLayout(2, 3));

f1.setLayout(new GridLayout(2, 3)); // Autorisé depuis version 1.5
```

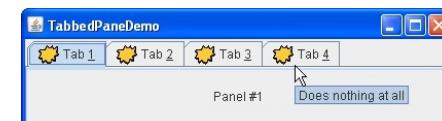
Remarque : Le *Layout Manager* doit être déterminé avant l'ajout des composants (enfants) dans le conteneur.

Remarque : Si un conteneur est déjà affiché, il faut invoquer la méthode **revalidate()** (de la classe **JComponent**) sur le conteneur après l'ajout ou la suppression dynamique d'un composant.



JTabbedPane [1]

- Le conteneur **JTabbedPane** est très utile pour enregistrer plusieurs sous-conteneurs (généralement de type **JPanel**) dont un seul est affiché à la fois (même idée de base que celle du gestionnaire **CardLayout**).
- Tous les sous-conteneurs partagent donc le même espace d'affichage.
- L'utilisateur peut choisir le sous-conteneur à afficher en cliquant sur un des onglets (**Tab**) affichés à la périphérie du conteneur.
Les onglets sont identifiés par un libellé (titre) et/ou une icône.
- L'emplacement des onglets est configurable dans le constructeur et par la propriété **tabPlacement** (**TOP**, **BOTTOM**, **LEFT**, **RIGHT**).





JTabbedPane [2]

- Les différents composants enfants (sous-conteneurs) sont identifiés par un **index** qui va de 0 à **tabCount -1**.
- L'ajout des sous-conteneurs s'effectue à l'aide des différentes méthodes **addTab()** et **insertTab()** qui permettent également de définir le texte (titre) et/ou l'icône associés à l'onglet correspondant.
- On ne peut pas changer le *Layout-Manager* d'un **JTabbedPane**.
- Quelques propriétés de **JTabbedPane** :
 - tabCount** : Nombre d'onglets (en lecture uniquement)
 - tabPlacement** : Emplacement des onglets
 - selectedIndex** : Index de l'onglet sélectionné
 - titleAt** : Titre (libellé) de l'onglet (pour un index donné)
 - iconAt** : Icône de l'onglet (pour un index donné)
 - toolTipTextAt** : Texte de la bulle d'aide (pour un index donné)
 - componentAt** : Sous-conteneur (pour un index donné)



JTabbedPane [3]

```
===== Exemple d'utilisation d'un JTabbedPane =====

JTabbedPane tabbedPane = new JTabbedPane(JTabbedPane.LEFT);

ImageIcon icon = new ImageIcon("D:\\images\\bluedot.gif");

 JPanel panel1 = makePanel1(); // La méthode makePanel1 est déclarée ailleurs
 tabbedPane.addTab("One", icon, panel1); // Ajout du premier sous-conteneur

 JPanel panel2 = makePanel2();
 tabbedPane.addTab("Two", panel2); // Ajout du deuxième sous-conteneur

 JSplitPane panel3 = makePanel3("Source", "Destination");
 tabbedPane.addTab("Three", panel3); // Ajout du troisième sous-conteneur

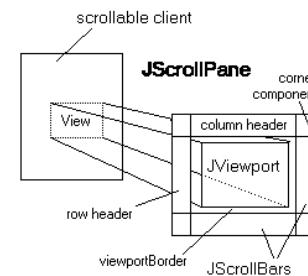
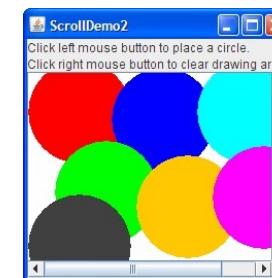
 tabbedPane.setSelectedIndex(1); // Sélection du deuxième onglet
 . . .

 // Add the tabbed pane to a JFrame
 JFrame frame = new JFrame("Test JTabbedPane");
 . . .
 frame.setContentPane(tabbedPane);
 . . .
```



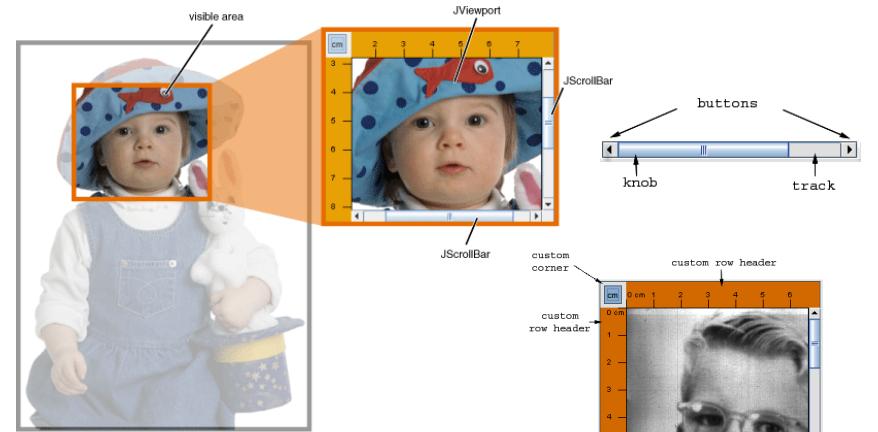
JScrollPane [1]

- Le conteneur **JScrollPane** permet d'offrir les fonctionnalités de **défilement (scrolling)** à son unique composant enfant.
- Le conteneur **JScrollPane** utilise en interne un composant de type **JViewport** qui permet d'observer une certaine zone (*View*) d'un composant qui ne peut (généralement) pas être affiché en entier.



JScrollPane [2]

- En plus de la zone d'observation (*Viewport*) et des barres de défilement (*Scrollbar*), le conteneur **JScrollPane** permet d'afficher également d'autres éléments décoratifs (*custom row and column headers, corners*).





JScrollPane [3]

- La manière la plus simple d'insérer un composant dans un conteneur **JScrollPane** est de passer ce composant en paramètre dans un des constructeurs qu'offre la classe **JScrollPane**.

```
Icon icon = new ImageIcon("D:\\images\\largePicture.jpg");
JLabel imageLabel = new JLabel(icon);
JScrollPane scrollImage = new JScrollPane(imageLabel);
//-----
JTextArea textZone = new JTextArea();
textZone.append(...);
...
JScrollPane scrollText = new JScrollPane(textZone);
```

- Sans indications particulières, les barres de défilement (ascenseurs) apparaissent seulement lorsque c'est nécessaire (mais cette propriété peut être modifiée dans le constructeur ou après coup).
- Le gestionnaire de disposition (*Layout-Manager*) d'un **JScrollPane** doit obligatoirement être de type **JScrollPaneLayout**.



JScrollPane [4]

- Quelques propriétés de **JScrollPane** :
 - horizontalScrollBarPolicy** : Règle d'apparition de la barre de défilement horizontale. Les valeurs correspondent à des constantes importées par la classe **JScrollPane** :

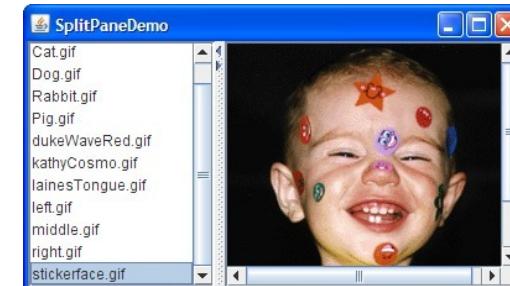

```
HORIZONTAL_SCROLLBAR_NEVER,
HORIZONTAL_SCROLLBAR_AS_NEEDED,
HORIZONTAL_SCROLLBAR_ALWAYS
```
 - verticalScrollBarPolicy** : Règle d'apparition de la barre de défilement verticale. Les valeurs correspondent à des constantes importées par la classe **JScrollPane** :


```
VERTICAL_SCROLLBAR_NEVER,
VERTICAL_SCROLLBAR_AS_NEEDED,
VERTICAL_SCROLLBAR_ALWAYS
```
 - viewportBorder** : Bordure autour de la zone visible (*Viewport*)
 - wheelScrollingEnabled** : Défilement possible avec la roulette de la souris (cette fonctionnalité n'est disponible que depuis la version 1.4 du JDK)



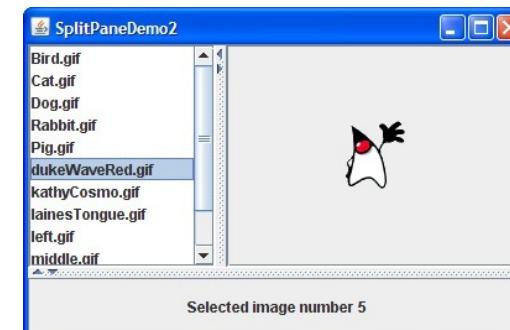
JSplitPane [1]

- Le conteneur **JSplitPane** permet d'afficher deux composants (et pas plus) en divisant horizontalement ou verticalement sa zone d'affichage.
- La barre de séparation entre les deux zones (*Divider*) peut être déplacée par l'utilisateur (modification de la taille des sous-conteneurs)
- Les composants insérés sont fréquemment des conteneurs de type **JScrollPane** de manière à ce que l'utilisateur puisse faire défiler le contenu si nécessaire.



JSplitPane [2]

- Les composants enfants peuvent également être de type **JScrollPane**.
- Il est ainsi possible d'avoir simultanément des divisions horizontales et verticales.
- L'utilisateur ne peut pas modifier la barre de séparation au delà de la taille minimale (**minimumSize**) des composants enfants.





JSplitPane [3]

- L'ajout des composants dans un conteneur **JSplitPane** s'effectue généralement en utilisant un des constructeurs qu'offre la classe. Les méthodes `setLeftComponent()`, `setRightComponent()`, `setTopComponent()` et `setBottomComponent()` permettent également de définir dynamiquement les composants placés dans les sous-conteneurs (`Left↔Top=1er`, `Right↔Bottom=2ème` sous-conteneur).

```
JScrollPane scrollPaneLeft = new JScrollPane(panel1);
JScrollPane scrollPaneRight = new JScrollPane(panel2);
JSplitPane sPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
                                   scrollPaneLeft,
                                   scrollPaneRight );
```

- Les constantes `HORIZONTAL_SPLIT` et `VERTICAL_SPLIT` (classe `JSplitPane`) servent à définir l'orientation de la division du conteneur.
- L'orientation de la division du conteneur peut être modifiée après avoir ajouté les sous-composants (`setOrientation()`).
- On ne modifie pas le gestionnaire de disposition d'un **JSplitPane**.



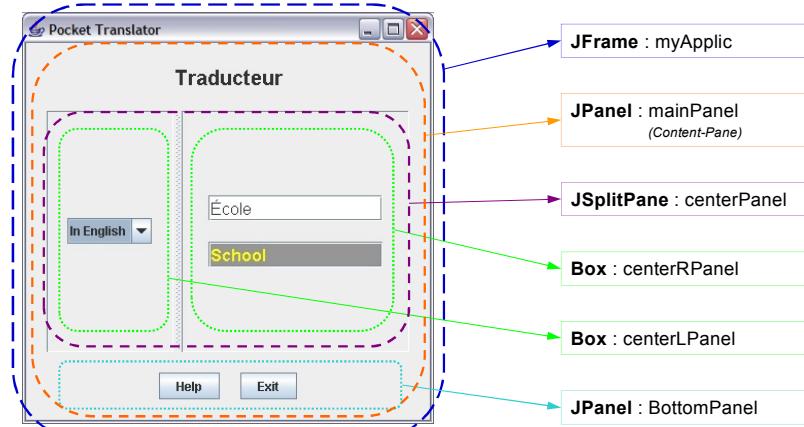
JSplitPane [4]

- Quelques propriétés de **JSplitPane** :
 - `dividerLocation` : Position de la barre de séparation (en pixels ou en % de la taille du conteneur)
 - `dividerSize` : Taille de la barre de division (en pixel)
 - `orientation` : Orientation de la division du conteneur
 - `continuousLayout` : Indique si la taille des sous-conteneurs est redessinée en continu lors du déplacement de la barre de séparation (true/false)
 - `topComponent` : Composant placé en haut (ou à gauche)
 - `bottomComponent` : Composant placé en bas (ou à droite)
 - `leftComponent` : Composant placé à gauche (ou en haut)
 - `rightComponent` : Composant placé à droite (ou en bas)
 - `resizeWeight` : Répartition de l'espace additionnel lors du redimensionnement du **JSplitPane** (valeurs entre 0.0 et 1.0; la valeur indique la proportion de l'espace attribuée au sous-conteneur gauche/haut)



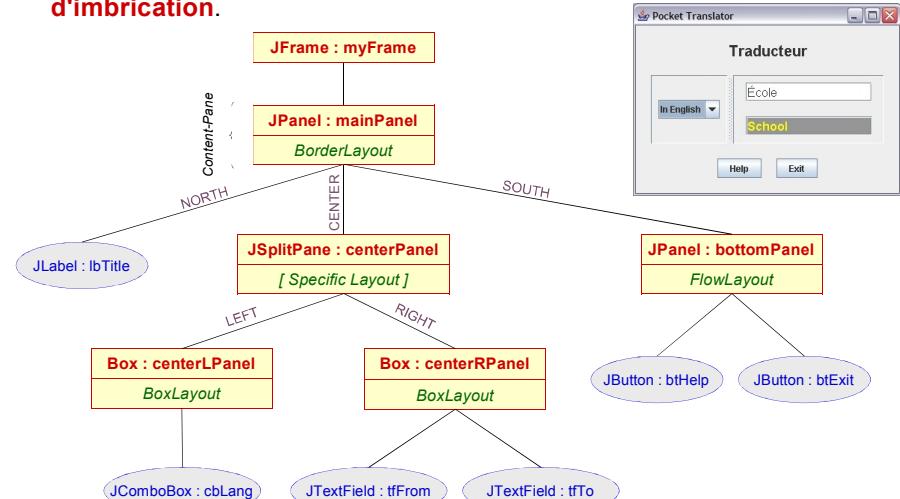
Exemple imbrication

- La conception (et la réalisation) d'une interface graphique nécessite généralement l'imbrication de plusieurs conteneurs qui sont associés à différents gestionnaires de disposition.



Exemple d'arbre d'imbrication

- En plus des **types**, le nom des **objets** peut figurer dans l'**arbre d'imbrication**.





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

04

Architecture MVC / Événements Structure d'une application

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Architecture MVC

Interactions avec le système

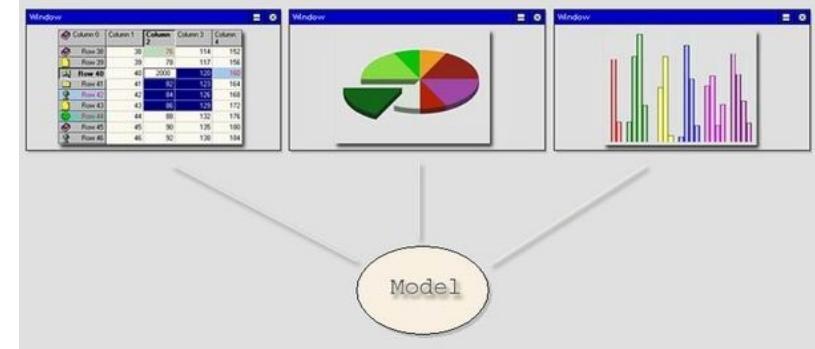
- Dans les applications informatiques classiques, l'utilisateur interagit avec le système en utilisant essentiellement
 - Le **clavier** et la **souris** (touchpad, joystick, ...)
 - ✓ Modalité d'**entrée** dans le système
- L'**écran de visualisation**
- ✓ Modalité de **sortie** du système
- L'affichage des informations sur l'écran (fenêtres, boîtes de dialogue) constitue ce que l'on nomme la **vue (view)** du système.
- La **vue** représente – d'une manière ou d'une autre – l'état du système interactif (logiciel, site web)
 - ✓ Mécanisme de visualisation de l'état interne (image du système)



Interface Homme-Machine 1 / Architecture MVC

Interactions avec le système : modèle et vue

- L'interface utilisateur est chargée de représenter, sous une forme interprétable par un humain, les informations internes de l'application.
- Ces informations constituent le **modèle (model)** de l'application (les données représentant son état actuel).
- On peut créer plusieurs **vues** distinctes d'un même modèle.



Interface Homme-Machine 1 / Architecture MVC

Interactions avec le système : contrôleur



- Lorsque l'utilisateur interagit avec le système
 - ✓ Déplacement de la souris, clic, double-clic, ...
 - ✓ Frappe sur les touches du clavier (caractères, touches de fonctions, ...)
- le composant logiciel qui gère ces actions est nommé le **contrôleur (controller)** du système.
- Le rôle du contrôleur est donc de **réagir aux actions de l'utilisateur**
- Si l'utilisateur agit au moyen de la souris (ou *touchpad, trackpad, joystick, tablette graphique, etc.*), l'interaction se fait au travers de la vue
- Lors de la saisie, au clavier, d'un champ de texte (ou d'une zone de texte, ou lors de la navigation à l'aide du clavier), la vue est également impliquée dans l'interaction
- Il y a donc souvent une **relation assez forte entre le contrôleur et la vue**





Architecture MVC

- L'**architecture MVC** (*Model-View-Controller*) est un **modèle de conception** (*Design Pattern*) très classique qui a été introduit avec le langage *Smalltalk-80*.
- Le principe de base de l'architecture MVC est relativement simple, on divise le système interactif en **trois parties distinctes** :
 - le **modèle (Model)** qui offre l'accès et permet la gestion des données (état du système)
 - la **vue (View)** qui a pour tâche de présenter les informations (visualisation) et qui participe à la détection de certaines actions de l'utilisateur
 - le **contrôleur (Controller)** qui est chargé de réagir aux actions de l'utilisateur (clavier, souris) et à d'autres événements internes et externes
- Ce modèle de conception simplifie le développement et la maintenance des applications en répartissant et en découplant les activités dans différents sous-systèmes (plus ou moins) indépendants.

EIA-FR / Jacques Bapst



IHM1_04

5



MVC / La vue (View)

- La **vue (View)** se charge de la **représentation visuelle** des informations.
- La vue utilise les données provenant du modèle pour afficher les informations. La vue doit être informée des modifications intervenues dans certaines données du modèle (celles qui influencent l'affichage).
- Plusieurs vues différentes peuvent utiliser le même modèle (plusieurs représentations possibles d'un même jeu de données).
- La vue intercepte certaines actions de l'utilisateur et les transmet au contrôleur pour qu'il les traite (souris, événements clavier, ...).
- Le contrôleur peut également modifier la vue en réaction à certaines actions de l'utilisateur (par exemple afficher une nouvelle fenêtre).
- La représentation visuelle des informations affichées peut dépendre du *Look-and-Feel* adopté (ou imposé) et peut varier d'un système d'exploitation à l'autre. L'utilisateur peut parfois modifier lui-même la présentation des informations en choisissant par exemple un thème.

EIA-FR / Jacques Bapst



IHM1_04

7



MVC / Le modèle (Model)

- Le **Modèle (Model)** est responsable de la gestion de l'**état du système** (son contenu actuel, la valeur de ses données).
- Il offre également les **méthodes** et **fonctions** permettant de gérer, transformer et manipuler ces données (logique "métier" de l'application).
- Le modèle peut informer les vues des changements intervenus dans ses données. La communication de ces changements intervient en général sous la forme d'événements qui seront gérés par des contrôleurs (les vues s'enregistrent auprès du modèle pour être notifiées lors des changements dans les données)
- Les informations gérées par le modèle sont indépendantes de la manière dont elles seront affichées. En fait, le modèle doit pouvoir exister indépendamment de la représentation visuelle des données.
- Dans certaines situations (simples) le modèle peut contenir lui-même les données, mais la plupart du temps, il agit comme un intermédiaire (*proxy*) vers les données qui sont stockées dans une base de données ou un serveur d'informations (en Java, le modèle est souvent défini par une **interface**).

EIA-FR / Jacques Bapst



IHM1_04

6



MVC / Le contrôleur (Controller)

- Le **contrôleur (Controller)** est chargé de réagir aux différents événements qui peuvent survenir.
- Les **événements** sont constitués soit par des **actions de l'utilisateur** (presser sur une touche, cliquer sur un bouton, fermer une fenêtre, ...) ou par des **directives venant du programme** lui-même (un changement intervenu dans un autre composant, l'écoulement d'un certain temps, etc...).
- Le contrôleur **définit le comportement** de l'application (comment elle réagit aux sollicitations).
- Dans les applications simples, le contrôleur gère la synchronisation entre la vue et le modèle (rôle de chef d'orchestre).
- Le contrôleur est informé des événements qui doivent être traités et sait d'où ils proviennent.
- Le contrôleur peut agir sur la vue en modifiant les éléments affichés.
- Le contrôleur peut également, si nécessaire, modifier le modèle en réaction à certains événements.

EIA-FR / Jacques Bapst

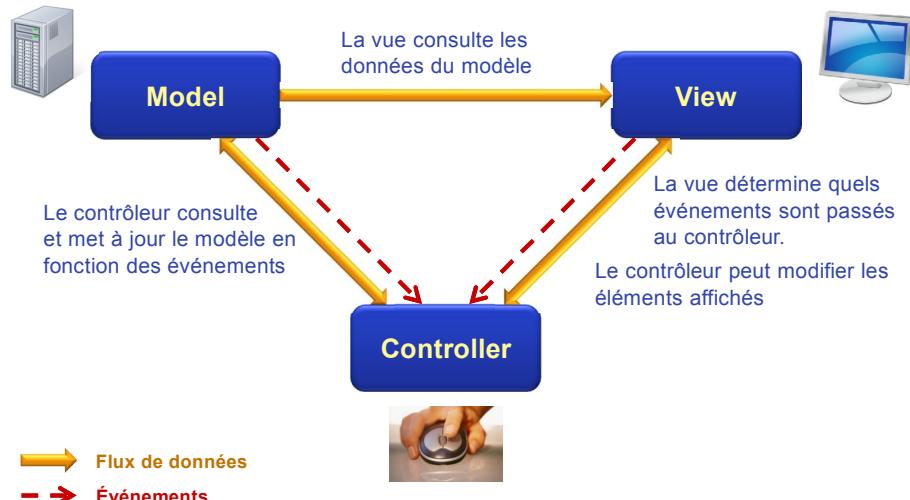


IHM1_04

8

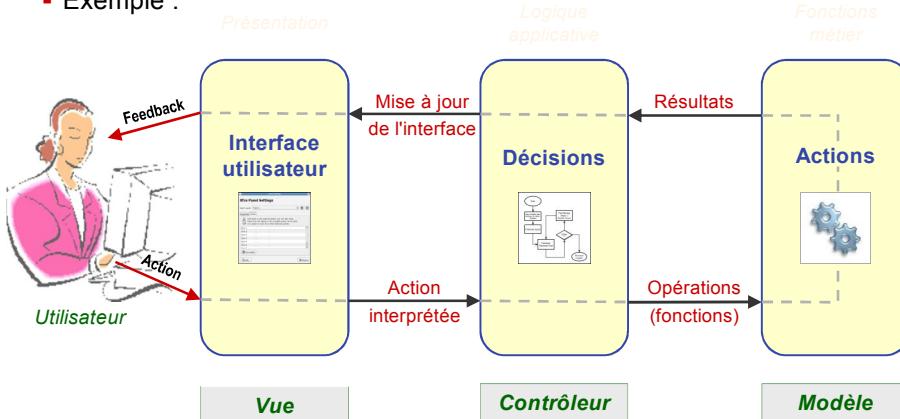


Interactions MVC



Rôles des éléments de l'architecture MVC

- Lorsqu'un utilisateur interagit avec une interface, les différents éléments de l'architecture MVC jouent chacun un rôle bien défini.
- Exemple :



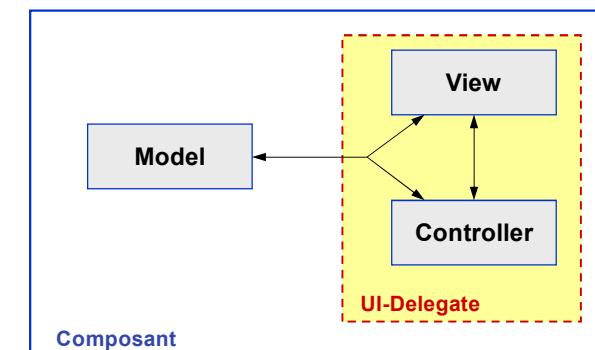
Importance de l'architecture MVC

- Pour le développement d'applications Java, l'**architecture MVC** est intéressante à plusieurs titres.
- D'une part, c'est un **modèle de conception** (ou, plus exactement, un ensemble de modèles de conception) qu'il est recommandé d'utiliser pour le développement des applications interactives car il offre des principes de découpage du code (modularisation) qui visent à **minimiser les couplages** entre les différents éléments qui constituent le programme.
- Ce découplage entre les modules minimise les dépendances et favorise ainsi la maintenance des applications (on peut modifier un élément du système sans que tous les autres en soient affectés).
- D'autre part, la **librairie Swing** est elle-même basée sur une variante de cette architecture (voir plus loin).
- Il est donc important de connaître l'architecture MVC pour comprendre et exploiter au mieux les différents éléments de la librairie Swing (paramétrier et personnaliser les composants, gérer les événements, ...).



MVC et Swing [1]

- La librairie Swing utilise une **variante** légèrement simplifiée de l'architecture MVC appelée **Model-Delegate** (Modèle-Délégué).
- Cette variante **combine** dans un seul élément **la vue et le contrôleur**.
- Cet élément combiné est appelé **UI-Delegate**.





MVC et Swing [2]

- En résumé, chaque composant Swing contient **un modèle** et **un UI-Delegate**. Certains modèles (par exemple `BoundedRangeModel` ou `Document`) sont utilisés par plusieurs composants différents.
- Le modèle** est responsable de maintenir les informations associées au composant (son état, ses données).
- Le UI-Delegate** est responsable de la représentation visuelle du composant ainsi que de la réaction du composant aux différents événements qui peuvent l'affecter (actions de l'utilisateur ou instructions provenant du programme).
- Cette séparation permet :
 - D'avoir **différentes représentations d'un même modèle**. Les composants `JSlider`, `JProgressBar` et `JScrollBar` partagent, par exemple, le même modèle : `BoundedRangeModel`.
 - De **modifier la représentation (Look-and-Feel)** d'un composant sans affecter ses données (*Pluggable Look-and-Feel*). La représentation peut être modifiée dynamiquement (durant l'exécution de l'application).

EIA-FR / Jacques Bapst



IHM1_04

13



Programmation événementielle

- La programmation des interfaces graphiques (GUI) se base généralement sur la notion d'**événement**.
- On parle alors de **programmation événementielle (Event-Driven Programming)**.

```
LinearProg {
  DataDeclaration;
  Init();
  Loop {
    cmd = GetCommand();
    Switch (cmd) {
      Case: command1
        code1;
      Case: command2
        code2;
      Case: command3
        code3;
      Case: ...
        ...
    }
  }
}
```

```
EventProg {
  DataDeclaration;
  Init();
  CreateGUI();
  RegisterCallback();
  StartEventLoop(); // Thread
  Callback1() // Button clicked
  code1;
  Callback2() // Key pressed
  code2;
  Callback3() // Window resized
  code3;
  ...
}
```

EIA-FR / Jacques Bapst



IHM1_04

14



Événements

- Les **événements** sont constitués par :
 - Des **actions de la part de l'utilisateur** (un clic de souris, la pression sur une touche du clavier, le déplacement d'une fenêtre, etc.)
 - Des **changements provoqués par le programme** lui-même ou par un sous-système externe (modification des propriétés d'un composant, informations provenant du réseau, échéance d'une temporisation, etc.)
- Les **événements** (groupés par genre) sont représentés par différentes classes qui ont toutes comme classe parente la classe `EventObject` (du package `java.util`).
- Les événements AWT sont représentés par des objets des classes qui se trouvent dans le paquetage `java.awt.event`.
- La librairie Swing utilise également ces classes et en définit d'autres dans le paquetage `javax.swing.event`.
- La classe `javax.beans.PropertyChangeEvent`, définie dans le modèle de composant JavaBeans, est également utilisée par la librairie Swing pour le changement des propriétés "liées" (*bound properties*)

EIA-FR / Jacques Bapst



IHM1_04

15

EIA-FR / Jacques Bapst



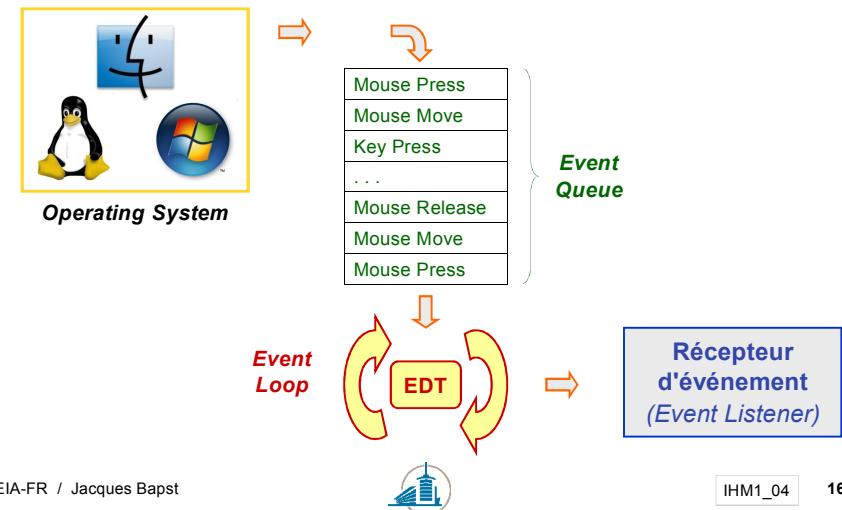
IHM1_04

16



Gestion des événements

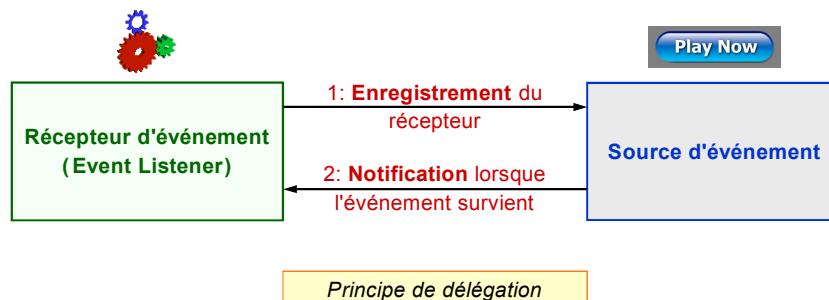
- Les **événements élémentaires** sont transmis à l'application active par le système d'exploitation. Ils sont ensuite enregistrés dans un buffer avant d'être traités par un processus indépendant (*Event Dispatch Thread*).





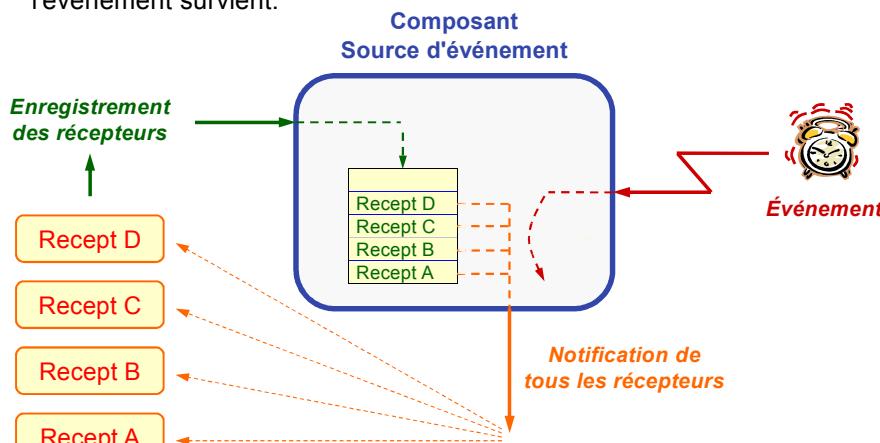
Sources et récepteurs d'événements

- Certains composants génèrent des événements; ce sont des **sources d'événements**.
- Un objet qui **désire être notifié** (informé) lorsqu'un événement survient est un **récepteur d'événement (Event Listener)**.
On pourrait également parler d'**intercepteur d'événement**, d'**auditeur d'événement**, de **traite événement**, etc.



Source d'événement

- Les composants qui sont des sources d'événements maintiennent (pour chacun des événements) une liste des récepteurs à notifier lorsque l'événement survient.



Gestion des événements – Déroulement [1]

- Préparation** (lors de l'initialisation du programme)
 - Création d'un récepteur d'événement (contrôleur) ...Controller
 - Instanciation d'une classe implémentant l'interface `evTypListener`
 - Enregistrement du récepteur d'événement auprès du composant qui est la source de l'événement (un bouton par exemple)
 - Invocation d'une méthode `addEvTypListener()`

Lorsque l'événement survient

- (clic sur un bouton par exemple)
- Un objet "événement" est automatiquement créé par le composant source d'événement
 - L'objet créé est une instance de la classe `evTypEvent`
 - La méthode du contrôleur (`evController`) associée à l'événement est automatiquement appelée par le composant source d'événement
 - L'objet "événement" est toujours passé en paramètre à cette méthode
 - Cette méthode se charge du traitement de l'événement



Gestion des événements – Déroulement [2]

- Préparation** (lors de l'initialisation du programme)
 - Création d'un récepteur d'événement (contrôleur) ...Controller
 - `public class ClickController implements ActionListener {`
 - `... public void actionPerformed(ActionEvent event) {`
 - `... // Traitement de l'événement`
 - `}`
 - `}`
 - `btController = new ClickController(...);`
 - Enregistrement du récepteur d'événement auprès du composant qui est la source de l'événement (un bouton par exemple)
 - `myButton.addActionListener(btController);`

Lorsque l'événement survient

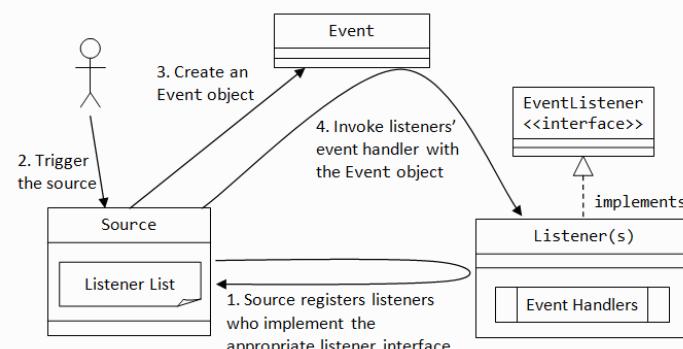
- (clic sur le bouton)
- Le bouton créera une instance `event` de la classe `ActionEvent`
 - Le bouton invoquera la méthode `actionPerformed(event)`





Gestion des événements – Déroulement

- Représentation de la gestion des événements sous forme d'un diagramme de collaboration

Source : <http://www.ntu.edu.sg>

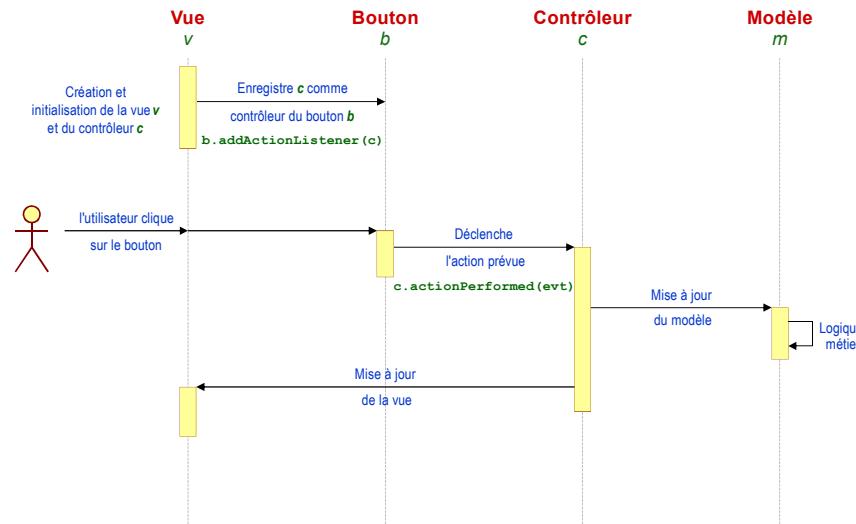
EIA-FR / Jacques Bapst

IHM1_04

21



Séquence des opérations



EIA-FR / Jacques Bapst

IHM1_04

22



Objets représentant les événements

- Les objets de type événement (**evTypEvent**) disposent tous de deux méthodes générales :

• **getSource()** : Renvoie l'objet qui a généré ou déclenché l'événement

• **getID()** : Renvoie une valeur permettant de distinguer les différents types d'événements qui sont représentés par la même classe d'événement (pas souvent nécessaire). Par exemple **FocusEvent** a deux types possibles : `FocusEvent.FOCUS_GAINED` et `FocusEvent.FOCUS_LOST`.

- Les différentes sous-classes d'événements définissent également des méthodes spécialisées qui dépendent de l'événement considéré.

- Par exemple **MouseEvent** possède les méthodes `getX()`, `getY()`, `getButton()`, `getClickCount()`, ... et hérite des méthodes `getModifiers()` et `getWhen()` de sa classe parente **InputEvent**.

Quand on reçoit un **MouseEvent**, on peut donc connaître l'emplacement où l'utilisateur a cliqué, avec quel bouton, combien de fois, quels modificateurs clavier étaient simultanément activés et quand (à quel moment) il a cliqué.

EIA-FR / Jacques Bapst

IHM1_04

23



Gestion des événements [1]

- La gestion des événements est identique pour tous les composants, et des conventions de nommage régissent les éléments qui interviennent.
- Chaque type d'événement **evTyp** se nomme **evTyp Event** et possède un **listener** (un récepteur) qui est défini par une interface appelée **evTyp Listener** (une sous-classe de `java.util.EventListener`).
- Les interfaces **listener** définissent des méthodes que la source d'événement invoquera lorsqu'un type donné d'événement se produira (principe de *Callback*). Ces méthodes listener prennent toujours un **objet événement** comme unique paramètre.
- Les sources d'événements doivent mettre à disposition des méthodes permettant aux récepteurs (**listeners**) de s'inscrire auprès de la source et annoncer ainsi leur intérêt concernant certains événements (principe de *Publication/Souscription*).
- Par convention, ces méthodes d'inscription (enregistrement du contrôleur) se nomment `addevTypListener()`. Les méthodes de retrait (résiliation, désinscription) se nomment `removeevTypListener()`.

EIA-FR / Jacques Bapst

IHM1_04

24





Gestion des événements [2]

- Par exemple, pour la classe **JButton** (source d'événement), le type d'événement est **Action**, le composant génère donc un événement appelé **ActionEvent** lorsque l'utilisateur clique sur le bouton.
- La classe fournit donc deux méthodes **addActionListener()** et **removeActionListener()** qui permettent à un objet (passé en paramètre) de s'inscrire (resp. de résilier l'inscription) comme récepteur (listener) pour l'événement de type **ActionEvent**.
- Si un objet s'intéresse à l'événement **ActionEvent** du bouton, il doit implémenter l'interface **ActionListener**. Cette interface contient une unique méthode **actionPerformed()** qui sera invoquée lorsque le bouton sera pressé.
- Plusieurs récepteurs différents peuvent s'intéresser à un même événement en s'inscrivant chacun à l'aide de la méthode **addMouseListener()**. La source d'événement maintient une **liste des récepteurs** (listeners) inscrits. Si l'événement survient, chacun des récepteurs sera notifié (par invocation de la méthode correspondante).



Schéma de fonctionnement [1]

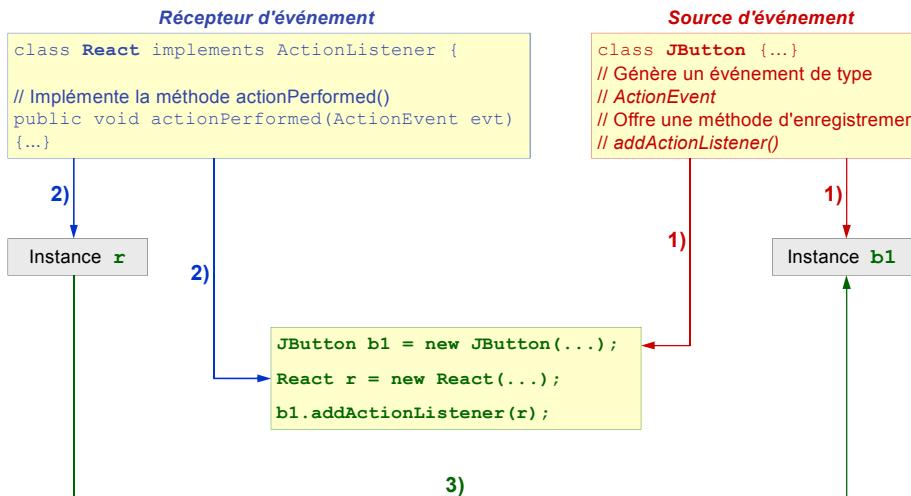


Schéma de fonctionnement [2]

- Création d'un composant **JButton** qui génère un événement de type **ActionEvent** (**JButton** est une source d'événement).
- Création d'un récepteur d'événement qui implémente les méthodes définies dans l'interface **ActionListener** (dans ce cas, il y a une unique méthode appelée **actionPerformed()**). Le traitement de l'événement est réalisé dans le corps de la méthode **actionPerformed()**.
- Enregistrement du récepteur d'événement (listener) auprès de la source d'événement, le bouton **b1**. Ce mécanisme s'appelle **délégation** : le bouton **b1** délègue la gestion de l'événement **ActionEvent** à l'objet **r**.
- Finalement : Lorsqu'un utilisateur pressera le bouton **b1**, la méthode **actionPerformed()** sera invoquée avec, comme unique argument, l'objet événement **e** (de type **ActionEvent**). Si nécessaire, il sera possible dans le corps de **actionPerformed()** de déterminer l'objet qui a généré l'événement (**b1** dans l'exemple présenté) en invoquant la méthode **getSource()** de l'objet événement **evt**.



Synthèse du fonctionnement

```
JButton btOk = new JButton("OK");
```



```
public class ButtonController implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        ...
    }
}
```

```
ButtonController btCtrl = new ButtonController(...);
```

```
btOk.addActionListener(btCtrl);
```





Adaptateur d'événement

- Quand une interface de récepteur d'événement (*Event Listener*) définit plus d'une méthode, elle est souvent accompagnée d'une classe abstraite appelée **Adaptateur d'événement (Event Adapter)** qui fournit des implémentations vides pour chacune des méthodes de l'interface.
- Les adaptateurs d'événement (classes abstraites) portent le suffixe **Adapter** (au lieu de **Listener**).
- Exemples : **MouseListener** **KeyListener** (*Interfaces*)
MouseAdapter **KeyAdapter** (*Classes*)
- Si l'on ne s'intéresse qu'à certaines méthodes de l'interface, il est **plus facile de sous-classer l'adaptateur** et de ne redéfinir que les méthodes souhaitées.
- Par exemple, l'interface **MouseListener** définit cinq méthodes différentes. Si l'on ne s'intéresse qu'à la méthode **mouseClicked()** il est plus facile de sous-classer **MouseAdapter** et de redéfinir uniquement **mouseClicked()** au lieu d'implémenter les cinq méthodes de l'interface **MouseListener**.



Listeners / Méthodes / Composants

- Pour connaître la **liste des événements** générés par un composant, il faut consulter son API et **observer la liste des méthodes d'enregistrement `addTyplistener()`** que la classe offre et celles dont elle hérite.
- Vous pouvez également consulter la documentation en ligne et notamment la page "*Listeners Supported by Swing Components*" sur le site d'Oracle (*Swing Tutorial*), à l'adresse :
docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html
- Les **tabelles qui suivent** donnent un bref aperçu des gestionnaires d'événements et événements les plus courants des librairies AWT et Swing (la liste n'est de loin pas exhaustive).
- Les classes qui héritent des événements ne sont pas mentionnées, seules les classes qui déclarent les méthodes sont indiquées (par exemple **AbstractButton** et non pas **JButton**).



Listeners courants [1]

| Listeners / Event | Méthodes de listener | Générés par |
|-------------------------------|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| ActionListener ActionEvent | actionPerformed() | AbstractButton, Button, ButtonModel, ComboBoxEditor, JComboBox, JFileChooser, JTextField, List, MenuItem, TextField, Timer |
| FocusListener FocusEvent | focusGained() focusLost() | Component |
| ItemListener ItemEvent | itemStateChanged() | AbstractButton, ButtonModel, Checkbox, CheckboxMenuItem, Choice, ItemSelectable, JComboBox, List |
| KeyListener KeyEvent | keyPressed() keyReleased() keyTyped() | Component |



Listeners courants [2]

| Listeners / Event | Méthodes de listener | Générés par |
|-----------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| MouseListener MouseEvent | mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased() | Component |
| MouseMotionListener MouseEvent | mouseDragged() mouseMoved() | Component |
| TextListener TextEvent | textValueChanged() | TextComponent |
| WindowListener WindowEvent | windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened() | Window |





Listeners courants [3]

| Listeners / Event | Méthodes de listener | Générés par |
|---------------------------------------------|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ComponentListener ComponentEvent | componentHidden() componentShown() componentMoved() componentResized() | Component |
| ChangeListener ChangeEvent | stateChanged() | AbstractButton, BoundedRangeModel, ButtonModel, JProgressBar, JSlider, JTabbedPane, JViewport, MenuSelectionManager, SingleSelectionModel |
| ListDataListener ListDataEvent | contentsChanged() intervalAdded() intervalRemoved() | AbstractListModel, ListModel |
| ListSelectionListener ListSelectionEvent | valueChanged() | JList, ListSelectionModel |
| CaretListener CaretEvent | caretUpdate() | JTextComponent |



Structure d'une application [2]

```
public class MyApplic {
    public static void main(String[] args) {
        ...
        MyModel m = new MyModel(...);
        View1 v1 = new View1(...);
        ...
        v1.setVisible(true);
    }
}
```

Classe principale

```
public class MyModel {
    private float tempCueve;
    private int nbPompes;
    ...
    public float getTemp(int noCueve) {
        ...
    }
    ...
}
```

Modèle

```
public class View1 extends JFrame {
    ...
    private JLabel lbNom = new JLabel("Nom");
    ...
    public View1(...) {
        ...
    }
    ...
}
```

Vue(s)

```
public class Controller1 implements ... {
    ...
    public void action(...) {
        ...
    }
    ...
}
```

Contrôleur(s)



Structure d'une application [1]

- Les applications utilisant les librairies AWT et Swing peuvent être structurées de différentes manières.
 - D'une manière générale, il est conseillé de séparer dans différentes classes **les fonctions suivantes** :
 - L'initialisation et lancement de l'application (classe principale)
 - La gestion de l'interface graphique (GUI) *Vue*
 - La logique fonctionnelle de l'application (gestion des événements) *Contrôleur*
 - Les données et les méthodes d'accès et de traitement associées *Modèle*
- Chacun de ces modules peut naturellement être composé de plusieurs classes.
- La **fenêtre principale** de l'application est généralement constituée par un objet de type **JFrame** (dans la pratique, on crée, la plupart du temps, une sous-classe de **JFrame**) .
 - La **classe principale** de l'application comprendra une méthode **main()** (dont la signature est imposée) qui se chargera d'effectuer les initialisations nécessaires, de créer le modèle ainsi que de créer et d'afficher la fenêtre principale de l'application.



Structure d'une application [3]

- Un certain nombre d'**éléments de base** sont nécessaires pour réaliser une application comportant une interface graphique.
- Les pages qui suivent proposent un squelette général pour une application utilisant les librairies AWT/Swing.
- Il s'agit d'un exemple qui peut être utilisé comme modèle pour les exercices, en tenant compte du fait que :
 - Il ne s'agit que d'un cadre général comportant un petit nombre de classes et certains détails sont omis
 - Le modèle, la vue et les contrôleurs de l'application sont volontairement très simples
 - La création d'une classe séparée pour chaque gestion d'événement est préférable à la création de classes qui gèrent tous les événements ou de classes-internes anonymes (notamment durant la phase d'apprentissage)
 - Il y a souvent plusieurs manières d'effectuer une même opération
 - Toutes les classes sont placées dans le même *package*
 - Suivant les classes utilisées, les instructions **import** doivent être complétées





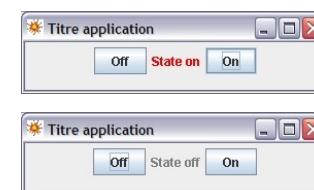
Classe principale : TestApplic

```
package cours_ihm1.template;

public class TestApplic {
    public static void main(String[] params) {
        boolean initialValue = false;

        MyModel model      = new MyModel(initialValue); // Création du modèle
        MyView view       = new MyView(model);           // Création de la vue

        view.setVisible(true); // Affichage de la fenêtre principale
    }
}
```



Modèle : MyModel

```
package cours_ihm1.template;

import java.awt.Color;

public class MyModel {
    public static final String ON_TEXT   = "State on";
    public static final String OFF_TEXT  = "State off";
    public static final Color  ON_COLOR   = Color.RED;    // Ou dans la vue !
    public static final Color  OFF_COLOR  = Color.GRAY;   // Ou dans la vue !

    private boolean on;

    public MyModel(boolean initState) {
        on = initState;
    }

    public boolean isOn() {
        return on;
    }

    public void setOn(boolean on) {
        this.on = on;
    }
}
```



Interface graphique : MyView [1]

```
package cours_ihm1.template;

import java.awt.*;
import javax.swing.*;

public class MyView extends JFrame {

    private MyModel model;

    //-----
    // Déclaration et création des conteneurs et des composants de la frame
    // comme champs de la classe TestFrame (pour pouvoir y accéder)
    //-----
    private JPanel panelA = new JPanel(); // FlowLayout par défaut

    private JLabel lbText = new JLabel("<== Click ==>");
    private JButton btOn  = new JButton("On");
    private JButton btOff = new JButton("Off");

    public MyView() {
        panelA.add(lbText);
        panelA.add(btOn);
        panelA.add(btOff);

        add(panelA, "Center");
    }
}
```

(suite...)



Interface graphique : MyView [2]

```
(...suite)

public MyView(MyModel model) {
    super("Titre application"); // Invocation du constructeur de JFrame

    this.model = model;

    //--- Termine l'application lorsqu'on ferme la fenêtre principale
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //--- Dimensionnement (largeur, hauteur)
    setSize(300, 80);

    //--- Positionnement (x, y)
    setLocation(400, 200);

    //--- Icône de l'application
    setIconImage(new ImageIcon("ApplIcon.gif").getImage());

    //--- Création de l'interface
    buildUI();
}
```

(suite...)





Interface graphique : MyView [3]

```
//-----
private void buildUI() {
    //--- Configuration et ajout des composants dans les conteneurs
    lbText.setForeground(Color.BLUE);

    panelA.add(btOff);
    panelA.add(lbText);
    panelA.add(btOn);

    //--- Création et enregistrement des gestionnaires d'événement des boutons
    BtOnController onController = new BtOnController(model, this);
    BtOffController offController = new BtOffController(model, this);
    btOn.addActionListener(onController);
    btOff.addActionListener(offController);

    //--- Ajout du conteneur de premier niveau dans le ContentPane de la Frame
    getContentPane().add(panelA);
}
```

(...suite...)



Interface graphique : MyView [4]

```
//-----
// Update state display (on/off)
//-----
public void setOnState(boolean onState) {
    if (onState) {
        lbText.setText(MyModel.ON_TEXT);
        lbText.setForeground(MyModel.ON_COLOR);
    }
    else {
        lbText.setText(MyModel.OFF_TEXT);
        lbText.setForeground(MyModel.OFF_COLOR);
    }
}

} // Fin de la classe MyView
```

(...suite...)



Gestion des événements : BtOnController

```
package cours_ihm1.template;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BtOnController implements ActionListener {
    private MyModel model; // Référence vers le modèle
    private MyView view; // Référence vers la vue
    //-----
    public BtOnController(MyModel model, MyView view) {
        this.model = model;
        this.view = view;
    }
    //-----
    // Méthode qui sera invoquée lorsque l'utilisateur clique sur le bouton "On"
    //-----
    public void actionPerformed(ActionEvent evt) {
        model.setOn(true);
        view.setOnState(true);
    }
}
```



Gestion des événements : BtOffController

```
package cours_ihm1.template;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BtOffController implements ActionListener {
    private MyModel model; // Référence vers le modèle
    private MyView view; // Référence vers la vue
    //-----
    public BtOffController(MyModel model, MyView view) {
        this.model = model;
        this.view = view;
    }
    //-----
    // Méthode qui sera invoquée lorsque l'utilisateur clique sur le bouton "Off"
    //-----
    public void actionPerformed(ActionEvent evt) {
        model.setOn(false);
        view.setOnState(false);
    }
}
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

05

**Labels / Boutons / Champs de texte
Validation / Cases à cocher / Boutons radio**

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Composants graphiques Swing

Composants graphiques

- Les différents chapitres qui suivent décrivent **très succinctement** les principaux composants graphiques de la librairie Swing.
- La consultation des API des différentes classes qui composent la librairie Swing (ainsi que les classes associées) est indispensable pour utiliser les composants.
- Le site de Sun constitue la **référence** pour la consultation des API de la plate-forme Java :

<http://java.sun.com/javase/7/docs/api/>
- Lors de l'étude d'une classe, il faut notamment consulter
 - La **description** générale de la classe
 - Les éventuelles **constantes** de la classe (et celles qui sont héritées)
 - Les **constructeurs** disponibles et leurs paramètres
 - Les **propriétés** du composant (méthodes `get...()`, `is...()` et `set...()`)
 - Les **méthodes** disponibles dans la classe du composant
 - Les **méthodes et propriétés héritées** des classes parentes
 - Les **événements** générés par le composant (méthodes `add...Listener()`)



Interface Homme-Machine 1 / Labels et boutons

JLabel [1]

- Le composant **JLabel** permet d'afficher un **texte** (libellé), une **icône** (image) ou **les deux** à la fois.
- Le texte du composant **JLabel** n'est pas éditable par l'utilisateur.

```
JLabel lNom = new JLabel("Nom");
Icon image1 = new ImageIcon("H:/pictures/image1.gif");
JLabel lImage = new JLabel(image1);
JLabel lCombi = new JLabel("Texte", image1, JLabel.CENTER);
```

- Les libellés (ou labels) peuvent être associés à un autre composant (un champ texte par exemple) avec la méthode `setLabelFor()`.
- Une déclaration de mnémonique pour le composant associé est ensuite possible en utilisant la méthode `setDisplayMnemonic()`.
- L'activation du mnémonique passera le focus au composant associé (et non pas au label qui n'est, de toutes manières, pas éditable).



Interface Homme-Machine 1 / Labels et boutons

JLabel [2]

- La création d'icône fait appel à la classe **ImageIcon** qui possède un constructeur permettant de lire l'icône à partir d'un fichier au format **GIF** (y compris les images animées GIF89A) ou **JPEG** ou **PNG**.
- La disposition générale du label dans le conteneur est déterminée par les propriétés **horizontalAlignment** et **verticalAlignment**.
- La position relative de l'image et du texte est déterminée par les propriétés **horizontalTextPosition** et **verticalTextPosition**.
- Pour définir ces différentes positions, on utilise des constantes (héritées de l'interface **SwingConstants**) : **LEFT**, **RIGHT**, **CENTER**, **TOP**, **BOTTOM**
- L'espace entre l'image et le texte est déterminé par la propriété **iconTextGap**.
- **Remarque** : Les textes peuvent être formatés en utilisant des balises HTML. Dans ce cas, le texte doit commencer par "`<HTML>`" (des informations plus détaillées se trouvent au chapitre "Éléments de programmation Swing").





AbstractButton

- La classe abstraite **AbstractButton** est la classe parente des différents types de **boutons**, des **menus** et des **options de menu**.
- Les propriétés de base de tous les boutons sont définies dans la classe **AbstractButton**.

```
java.awt.Component
  |
  +--java.awt.Container
    |
    +--javax.swing.JComponent
      |
      +--AbstractButton
        |
        +--JButton
        |
        +--JToggleButton
          |
          +--JCheckBox
          +--JRadioButton
        |
        +-- JMenuItem
          |
          +--JMenu
          +--JCheckBoxMenuItem
          +--JRadioButtonMenuItem
```

EIA-FR / Jacques Bapst



IHM1_05

5



JButton [1]

- Le composant **JButton** permet d'afficher un bouton appelé également bouton poussoir (*Push Button*) qui peut comprendre un **texte** (libellé), une **icône** (image) ou **les deux à la fois**.

```
JButton ok = new JButton("Ok");
JButton bImage = new JButton(new ImageIcon("image1.gif"));
```

- Une des caractéristiques essentielles d'un bouton est qu'il permet d'enregistrer des **récepteurs d'événements** (*Event-Listener*) de type **ActionListener** qui seront notifiés (informés) lorsque le bouton sera sélectionné par un utilisateur.
- Lors de l'activation du bouton, la méthode **actionPerformed()** du récepteur d'événement sera alors exécutée.
- Un bouton permet donc de **déclencher des actions** (activités).

EIA-FR / Jacques Bapst



IHM1_05

6



JButton [2]

- La plupart des propriétés de **JButton** sont héritées de la classe parente **AbstractButton**.
- La propriété **enabled** définit si le bouton est activé ou désactivé. Le texte (ou l'image) d'un bouton désactivé est grisé et ce bouton ne génère plus d'événement de type **ActionEvent**.
- La propriété **margin** (de type **Insets**) définit l'espace entre le contenu du bouton (texte, icône ou les deux) et le bord du bouton.
- Les propriétés **d'alignement** et de **position relative** des textes et des icônes sont identiques à celles qui sont définies pour le composant **JLabel**.
- Les conteneurs de haut-niveau peuvent définir un **bouton par défaut** (qui sera activé lorsque l'on presse la touche **Return**) en utilisant la méthode **setDefaultButton (JButton b)** de **JRootPane** (la méthode **getRootPane ()** permet d'obtenir la référence de ce conteneur).

EIA-FR / Jacques Bapst



IHM1_05

7



Exemple JButton [1]

```
public class TestBouton {
    public static void main(String[] params) {
        //--- Crédit de la fenêtre principale
        //--- (la classe ButtonFrame est définie dans les pages suivantes)
        ButtonFrame frame = new ButtonFrame();

        frame.pack();
        frame.setVisible(true);
    }
}
```

EIA-FR / Jacques Bapst



IHM1_05

8



Exemple JButton [2]

```
import java.awt.*;
import javax.swing.*;

public class ButtonFrame extends JFrame {

    JButton b1 = new JButton("Ceci est un bouton de type JButton");

    //--- Constructeur
    public ButtonFrame() {
        super("Test Bouton");           // Titre de la frame
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().add(b1);       // Ajout du bouton dans le ContentPane
        //--- Crédit de récepteur d'événement
        ActionBouton recepteur = new ActionBouton();

        //--- Enregistrement du récepteur d'événement
        b1.addActionListener(recepteur);
    }
}
```



JToggleButton [1]

- Le composant **JToggleButton** (qui étend **AbstractButton**) représente un **bouton qui peut prendre deux états différents** (enclenché/déclenché, on/off).



- L'état du bouton change à chaque fois que l'on clique sur lui.
- Visuellement, le composant **JToggleButton** se présente sous la même forme que le composant **JButton**. La représentation de son état (enclenché/déclenché) dépend de la plate-forme d'exécution (sous Windows, l'état enclenché est représenté par un effet 3D [bouton enfoncé]).
- La classe **JToggleButton** comprend deux sous-classes qui représentent des composants fréquemment utilisés (et qui sont décrits dans les pages suivantes) :

- **JCheckBox**
- **JRadioButton**



Exemple JButton [3]

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActionBouton implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        //--- Recherche l'objet qui a généré l'événement
        JButton b = (JButton)e.getSource();

        //--- Modifie le texte et la couleur de fond du bouton
        b.setText("Le bouton a été presse");
        b.setBackground(Color.RED);
    }
}
```



JToggleButton [2]

- Comme pour les composants **JButton**, on peut enregistrer un récepteur d'événement de type **ActionListener** sur les composants de type **JToggleButton**.
- Mais il est cependant parfois plus pratique d'utiliser un récepteur de type **ItemListener** qui permet de consulter plus facilement l'état du bouton (enclenché/déclenché).
- L'implémentation de l'interface **ItemListener** nécessite de créer une seule méthode **itemStateChanged()** qui prend pour unique argument un objet événement de type **ItemEvent**.
- La méthode **getStateChange()** de **ItemEvent** permet de consulter l'état actuel du bouton (après l'événement).
- Les constantes **SELECTED** et **DESELECTED** de **ItemEvent** représentent les deux états possibles rentrés par la méthode **getStateChange()**.





Exemple ItemListener

- Exemple de récepteur d'événement **ItemListener** que l'on peut enregistrer sur un composant de type **JToggleButton** (ou sur une des sous-classes **JCheckBox** et **JRadioButton**).

```
import java.awt.event.*;
public class ToggleAction implements ItemListener {
    public void itemStateChanged(ItemEvent ie) {
        //--- Lecture de l'état du bouton
        int state = ie.getStateChange();
        if (state == ItemEvent.SELECTED) {
            //--- Actions effectuées quand le bouton a été sélectionné
            . . .
        } else {
            //--- Actions effectuées quand le bouton a été désélectionné
            . . .
        }
    }
}
```

EIA-FR / Jacques Bapst



IHM1_05 13



JCheckBox

- Le composant **JCheckBox** est une sous-classe de **JToggleButton** et représente une case à cocher qui peut prendre deux états (sélectionné/non-sélectionné).



- Le composant prend donc également deux états qui peuvent être traités de manière identique à ceux des composants **JToggleButton** à l'aide d'un récepteur d'événement de type **ItemListener**.
- L'utilisation des composants **JCheckBox** permet de saisir un certain nombre d'**options** parmi une liste de choix non-exclusifs.
- Plusieurs variantes de constructeurs existent et permettent notamment de définir l'état initial de la case à cocher :

```
JCheckBox cb1 = new JCheckBox("Olives", true);
```

EIA-FR / Jacques Bapst



IHM1_05 14



JRadioButton

- Le composant **JRadioButton** est également une sous-classe de **JToggleButton** et représente une option que l'utilisateur peut sélectionner (généralement parmi un groupe d'options).



- Le composant prend donc également deux états qui peuvent être traités de manière identique à ceux des composants **JToggleButton** à l'aide d'un récepteur d'événement de type **ItemListener**.
- L'utilisation des composants **JRadioButton** doit être réservée à la saisie de choix mutuellement exclusifs (le composant lui-même n'impose pas cette sémantique).
- Pour obtenir ce comportement, il suffit d'insérer les composants **JRadioButton** (mutuellement exclusifs) dans un **groupe de boutons** représenté par un objet de la classe **ButtonGroup**.

EIA-FR / Jacques Bapst



IHM1_05 15



ButtonGroup

- La classe **ButtonGroup** permet de grouper (logiquement) des boutons qui sont enregistrés dans le groupe à l'aide de la méthode **add()**.
- En théorie, n'importe quel type de bouton (qui étend **AbstractButton**) peut être inséré dans un groupe de boutons.
- Cependant l'insertion dans un groupe de boutons n'a **d'effet que si** le composant est de type **JToggleButton**.
- Si un bouton est inséré dans un groupe de boutons, la **sélection de l'un d'eux** provoquera automatiquement la **désélection de tous les autres** membres du groupe (permet de garantir une seule sélection parmi un choix d'options mutuellement exclusives).
- L'insertion de boutons dans un groupe de boutons n'a **aucun effet visuel**. Pour visualiser les groupes de boutons, il est recommandé de créer une **bordure** qui délimite l'ensemble des boutons du groupe.
- En principe, l'insertion dans un groupe de boutons devrait être **réservée exclusivement aux boutons de type JRadioButton**.

EIA-FR / Jacques Bapst



IHM1_05 16



Exemple JRadioButton dans ButtonGroup

- L'extrait de code ci-dessous montre comment placer les composants **JRadioButton** dans un groupe de boutons (**ButtonGroup**).

```
//--- Création des composants et conteneurs
JRadioButton rb1 = new JRadioButton("Toutes les pages");
JRadioButton rb2 = new JRadioButton("Page courante");
JRadioButton rb3 = new JRadioButton("Selection");
ButtonGroup bGroup = new ButtonGroup();

JPanel p1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 0));

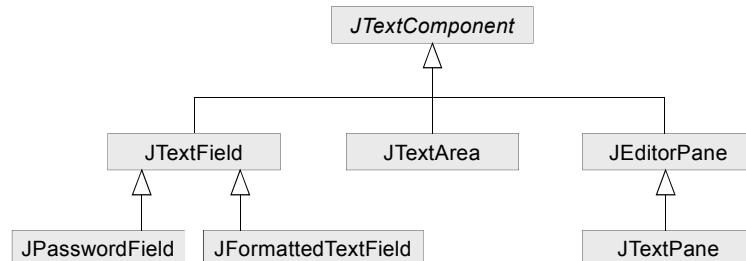
//--- Ajout des boutons radio dans le groupe
bGroup.add(rb1);
bGroup.add(rb2);
bGroup.add(rb3);

//--- Ajout des boutons radio dans le conteneur p1 et création d'une bordure titre
p1.add(rb1);
p1.add(rb2);
p1.add(rb3);
p1.setBorder(BorderFactory.createTitledBorder("Options d'impression"));
getContentPane().add(p1); // Ajout de p1 dans le ContentPane de la Frame
```



JTextComponent

- La classe abstraite **JTextComponent** (de `javax.swing.text`) est la classe de base de tous les composants qui permettent d'éditer des textes.

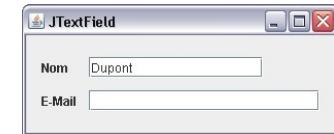


- Tous ces composants utilisent en interne un modèle basé sur l'interface **Document** pour enregistrer les textes qu'ils gèrent.
- Le **curseur** indiquant le point d'insertion dans le texte est appelé "**Caret**". Il est généralement représenté par une barre verticale qui clignote.



JTextField [1]

- Le composant **JTextField** représente un **champ de texte** (sur une seule ligne) qui permettra à l'utilisateur d'entrer une ligne de texte.



- Si l'utilisateur presse sur la touche **Return** lorsqu'il se trouve dans un composant **JTextField**, l'événement **ActionEvent** est généré.
- Il suffit donc d'enregistrer un récepteur d'événement (qui implémente **ActionListener**) sur un composant **JTextField** pour lire et traiter le texte introduit (saisi) par l'utilisateur.



JTextField [2]

- Différents **constructeurs** permettent de créer des champs de texte en mentionnant un **texte initial** à afficher et/ou le **nombre de colonnes** à prévoir pour l'affichage.
- La largeur d'une colonne est basée sur la grandeur du caractère 'm' de la police utilisée.

```
//--- Grandeur basée sur le contenu initial
JTextField t1 = new JTextField("Texte initial");
```

```
//--- Grandeur fixe de 15 colonnes
JTextField t2 = new JTextField(15);
```

- Pour éviter d'avoir la grandeur du champ qui évolue avec son contenu, il est recommandé de fixer la largeur lors de la création en mentionnant le nombre de colonnes souhaité (selon le gestionnaire de disposition).
- A l'affichage, la grandeur du champ de texte est également influencée par le gestionnaire de disposition du conteneur englobant.





JTextField [3]

- Plusieurs propriétés sont définies pour les composants **JTextField** (certaines sont héritées de **JTextComponent**).
- Les principales sont :

| | |
|----------------------------|---------------------------------------------------------------------------------------------|
| text | Le texte contenu dans le champ |
| editable | Indique si le champ est éditable ou non |
| selectedText | La portion de texte sélectionnée (consultation) |
| font | La police de caractères du texte |
| columns | Nombre de colonnes du champ |
| horizontalAlignment | Alignement horizontal du texte (LEFT , CENTER , RIGHT) dans le champ |
| margin | Marges entre le texte et le bord du composant (de type Insets) |



Validation du contenu d'un champ texte [1]

- Il est fréquent de devoir **valider le contenu** d'un champ de texte (avec différents critères de validation) avant de passer au champ suivant.
- Cette validation peut être effectuée dans un récepteur d'événement **FocusListener** (implémentation de la méthode **focusLost()**) au moment où l'utilisateur veut quitter le champ.
- Une autre technique, légèrement différente, a été introduite (à partir de la version JDK 1.3) sous la forme d'une classe abstraite **InputVerifier** (**javax.swing**) qui comporte deux méthodes :


```
boolean verify(JComponent input)
boolean shouldYieldFocus(JComponent input)
```
- La méthode **verify()** est chargée de vérifier si le contenu du champ est valide.
- La méthode **shouldYieldFocus()** détermine si le focus est transféré au champ de destination ou s'il doit rester dans le champ courant. Cette méthode est invoquée lorsque l'utilisateur veut quitter le champ (en cliquant ailleurs par exemple).



Validation du contenu d'un champ texte [2]

- La méthode (abstraite) **verify()** retournera **true** si le contenu du composant est valide et **false** sinon. Elle ne doit pas comporter d'effet de bord (le système doit pouvoir l'invoquer plusieurs fois).
- La méthode (concrète) **shouldYieldFocus()** retournera **true** si le focus doit être transféré au champ de destination et **false** si le focus doit rester dans le champ courant. Généralement la valeur renournée correspond à celle renournée par la méthode **verify()**.
- L'implémentation par défaut de la méthode **shouldYieldFocus()** invoque simplement la méthode **verify()** et retourne son résultat.
- La méthode **shouldYieldFocus()** est fréquemment redéfinie pour **effectuer des actions**, généralement dans le cas où le contenu du champ n'est pas correct (on invoque d'abord la méthode **verify()** pour le vérifier et si ce n'est pas le cas, on affiche par exemple une message d'erreur).
- Une fois la classe **InputVerifier** implémentée, il suffit d'enregistrer ce vérificateur au moyen de la méthode **setInputVerifier()** du composant à vérifier (la méthode est héritée de **JComponent**).



Exemple de validation d'un champ texte

```
public class Verificateur extends InputVerifier {
    ///-- Vérifie que le champ texte contienne un entier valide
    public boolean verify(JComponent comp) {
        boolean returnValue = true;
        JTextField textField = (JTextField)comp;
        String content = textField.getText();
        if (content.length() != 0) {
            try {
                Integer.parseInt(content);
            } catch (NumberFormatException e) {
                returnValue = false;
            }
        }
        return returnValue;
    }
    ///-- Bip sonore si le contenu est non-valide
    public boolean shouldYieldFocus(JComponent input) {
        if (verify(input)) {
            //--- Vérifie le contenu du champ
            //--- Transfère le focus à la destination
            Toolkit.getDefaultToolkit().beep();
            return false;
        }
    }
}
```





Validation avec affichage d'un message

```
public class MsgVerifier extends InputVerifier {
    --- Verify field contents (valid integer) -----
    public boolean verify(JComponent comp) {
        ...
        -- Same code as previous page
    }

    --- Error message (pop-up) if not ok -----
    public boolean shouldYieldFocus(JComponent input) {
        if (verify(input)) {
            return true;
        }

        --- Pop up the message dialog
        String message = "Warning : Input format should be 'PP.NNN.Z'\n"
            + "Press F1 (Help) to see the correct syntax";
        JOptionPane.showMessageDialog(null, message, "Invalid value",
            JOptionPane.WARNING_MESSAGE);

        return false;
        -- We don't want to yield focus
    }
}
```



Désactivation de la validation

- Par défaut, si un vérificateur est enregistré sur un composant, la validation est activée dès que le focus du composant à vérifier est transféré à un autre composant (à l'aide du clavier ou de la souris).
- Dans certaines situations, on souhaite que la validation **ne soit pas effectuée** si l'utilisateur active certains composants particuliers (bouton d'annulation "Cancel", barre de défilement, déroulement de menus, etc.).
- Si le focus est demandé pour un de ces composants particuliers, il est possible de **désactiver la validation** du composant courant en utilisant la méthode (de **JComponent**) :

setVerifyInputWhenFocusTarget(false)

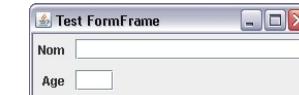
sur le ou les composants de destination (bouton *Cancel*, ...).

On évite ainsi des messages intempestifs et d'être contraint de devoir remplir correctement (et inutilement) des champs alors qu'on souhaite quitter la fenêtre courante (boîte de dialogue par exemple).



Exemple TestForm [1]

- Exemple d'application avec des labels et des champs de texte et un contrôle de validation sur un des champs de texte (*Age*)
- Résultat affiché :



```
===== Classe principale de l'application =====
class TestForm {
    public static void main(String[] params) {
        --- Création de l'écran principal
        FormFrame frame = new FormFrame();

        frame.pack();
        frame.setVisible(true);
    }
}
```



Exemple TestForm [2]

```
import java.awt.*;
import javax.swing.*;

class FormFrame extends JFrame {
    --- Déclaration de 2 labels et de 2 champs de texte
    JLabel lNom      = new JLabel("Nom");
    JLabel lAge      = new JLabel("Age");
    JTextField tfNom = new JTextField(20);
    JTextField tfAge = new JTextField(3);

    public FormFrame() {
        super("Test FormFrame");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel pChamps = new JPanel(new GridBagLayout());

        --- Contraintes pour le gestionnaire GridBagLayout
        GridBagConstraints gc = new GridBagConstraints();
        gc.insets=new Insets(5, 5, 5, 5);

        --- Fixe la taille minimale à la taille préférée
        tfNom.setMinimumSize(tfNom.getPreferredSize());
        tfAge.setMinimumSize(tfAge.getPreferredSize());
    }
}
```





Exemple TestForm [3]

...Suite

```

    //--- Validation du contenu du champ 'Age'
    Verificateur valideur = new Verificateur(); // Voir code dans les pages précédentes
    tfAge.setInputVerifier(valideur);

    gc.gridx=0; gc.gridy=0; // Position x, y dans la grille
    gc.anchor=gc.EAST; // Ancrage du composant
    pChamps.add(lNom, gc); // Ajout du composant

    gc.gridx=1; gc.gridy=0;
    gc.anchor=gc.WEST;
    pChamps.add(tfNom, gc);

    gc.gridx=0; gc.gridy=1;
    gc.anchor=gc.EAST;
    pChamps.add(lAge, gc);

    gc.gridx=1; gc.gridy=1;
    gc.anchor=gc.WEST;
    pChamps.add(tfAge, gc);

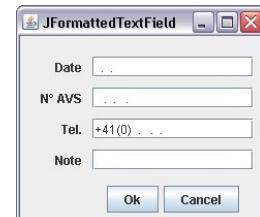
    setContentPane(pChamps);
}
}

```



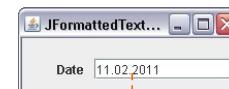
JFormattedTextField [1]

- Le composant **JFormattedTextField** est une sous-classe du composant **JTextField** qui spécialise son comportement en lui intégrant un **formateur (Formatter)** chargé de mettre en forme le texte saisi et faire en sorte qu'il respecte un format prédéfini.
- Plusieurs types de formateurs peuvent être définis :
 - Généraux : **MaskFormatter**
 - Dates : **DateFormatter**
 - Nombres : **NumberFormatter**
 - ... Instance de **JFormattedTextField.AbstractFormatter**
- Visuellement, le composant **JFormattedTextField** se présente de la même manière que le composant **JTextField** :
- Certaines informations, liées au formateur associé, peuvent apparaître dans les champs (parties fixes, séparateurs, etc.)



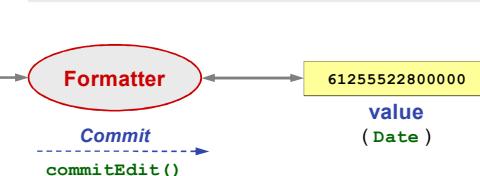
JFormattedTextField [2]

- Pour le composant **JFormattedTextField**, il faut distinguer :
 - la **valeur affichée** (propriété **text**) de type **String** qui est associée à la classe parente **JTextField**
 - la **valeur interne** (propriété **value**) qui peut être de n'importe quel type (**Object**) et qui représente la dernière valeur validée par le formateur
- Le formateur se charge de la conversion entre le texte affiché et la valeur interne (conversion bidirectionnelle).



"11.02.2011"
text
(String)

Avec les composants **JFormattedTextField**
on travaille généralement avec la propriété **value**
plutôt que **text** (méthodes **getValue()** et **setValue()**)



JFormattedTextField [3]

- Le formatage du texte est effectué par une instance de la classe interne (abstraite) **AbstractFormatter** (conversions **Object/Value** \Rightarrow **String** et **String** \Rightarrow **Object/Value**).
- Une autre classe interne **AbstractFormatterFactory** est utilisée par le composant **JFormattedTextField** pour rechercher l'instance du formateur utilisée pour convertir les données.
- La **mise en place d'un formateur** peut s'effectuer de différentes manières, la plus simple consiste à passer au constructeur du composant **JFormattedTextField** une instance d'une des deux familles suivantes (classes abstraites) :

• Format

: **DateFormat**, **SimpleDateFormat**,
NumberFormat, **DecimalFormat**, **MessageFormat**

• AbstractFormatter

: **DefaultFormatter**, **MaskFormatter**,
InternationalFormatter, **DateFormatter**,
NumberFormatter





Champs de texte formatés et validation

- Pour connaître la syntaxe permettant de définir les formats valides, il faut consulter la documentation des classes correspondantes (`MaskFormatter`, `NumberFormatter`, `DecimalFormat`, ...).
- Certaines classes peuvent tenir compte de la configuration locale de l'application (les paramètres régionaux). Typiquement, les dates sont formatées différemment selon les différentes régions du globe.
- Les sous-classes de `Format` et de `InternationalFormatter` sont prévues pour tenir compte de la localisation de l'application (`Locale`) et permettent de simplifier leur internationalisation.
- L'utilisation d'un champ `JFormattedTextField` avec un formateur n'empêche pas l'utilisateur de quitter le champ avec une valeur invalide (la propriété `value` ne sera simplement pas mise à jour).
- Pour éviter que l'on puisse quitter le champ (perdre le focus) avec une valeur invalide, il faut enregistrer un vérificateur (`InputVerifier`) qui empêchera le changement de focus tant que le format n'est pas valide (voir pages précédentes).



Exemple `JFormattedTextField`

- L'extrait de code ci-dessous montre comment créer des composants `JFormattedTextField` et de leur associer des formateurs.

```
///-- Utilisation de MaskFormatter pour formater un numéro de téléphone
MaskFormatter mFmt = null;
try {
    mFmt = new MaskFormatter("+41(0)##.###.##.##");
}
catch (ParseException e) {
    System.out.println("MaskFormatter Parser Error");
    e.printStackTrace();
}
JFormattedTextField ftf1 = new JFormattedTextField(mFmt);

///-- Utilisation de DecimalFormat pour formater un nombre avec
///-- au moins un chiffre après la virgule et au plus trois
DecimalFormat dFmt = new DecimalFormat("##.0##");
JFormattedTextField ftf2 = new JFormattedTextField(dFmt);
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI) 06 Éléments de programmation Swing

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Gestion du focus

Gestion du focus [1]

- Quand il y a plus d'une fenêtre ouverte sur un écran, une des fenêtres est déclarée la **fenêtre active** ou **fenêtre courante**. Cette fenêtre est mise en évidence par le système d'exploitation.
- Quand on utilise le clavier, les touches frappées (événements clavier) sont redirigées vers la fenêtre active.
- La fenêtre active contient généralement plusieurs composants visuels susceptibles de recevoir les événements clavier.
- Les événements du clavier sont redirigés vers un seul des composants de la fenêtre appelé **composant actif** ou **composant possédant le focus**. Ce composant est généralement mis en évidence.
- En règle générale, un composant Swing ne peut accepter des caractères ou être contrôlé depuis le clavier que quand il possède le focus.
- Il est cependant possible de programmer des événements clavier indépendants du focus (accélérateurs de menus par exemple).



Interface Homme-Machine 1 / Gestion du focus

Gestion du focus [2]

- Un composant peut **obtenir le focus** de différentes manières :
 - L'utilisateur clique sur le composant
 - L'utilisateur utilise les touches *Tab* et *Shift-Tab* qui font passer le focus d'un composant à l'autre selon un cycle par défaut (de gauche à droite et de haut en bas) ou selon un cycle défini par le programmeur (en créant une sous classe de *FocusTraversalPolicy*)
 - Par le programme lui-même en invoquant - sans garantie - la méthode *requestFocusInWindow()*. Si la méthode retourne **false**, on est certain que le focus n'a pas été transféré, si elle retourne **true**, ce n'est pas une garantie de transfert (pour en être certain il faut écrire un *FocusListener*)
- Quand on donne le focus à un **conteneur**, il le passe au premier composant (enfant) activable. Quand le focus atteint le dernier composant du conteneur il peut, soit retourner au premier enfant du conteneur, soit quitter le conteneur. Le comportement dépend de la valeur renournée par la méthode *isFocusCycleRoot()* du conteneur.
- Les touches *Ctrl-Tab* et *Ctrl-Shift-Tab* permettent à l'utilisateur de sortir d'un conteneur dont le focus a un comportement cyclique.



Interface Homme-Machine 1 / Gestion du focus

Gestion du focus [3]

- Il est possible de gérer les événements liés au focus en enregistrant un gestionnaire d'événement de type **FocusListener** sur le composant dont on souhaite surveiller la prise ou la perte du focus.
- Le gestionnaire **FocusListener** possède deux méthodes permettant de traiter les deux événements suivants :
 - **focusGained()** : Invoquée lorsque le composant **reçoit le focus**
 - **focusLost()** : Invoquée lorsque le composant **perd le focus**
- Tous les composants qui héritent de **Component** disposent de la paire de méthodes *addFocusListener()* et *removeFocusListener()* permettant d'enregistrer (respectivement de résilier) un gestionnaire de focus sur le composant considéré.
- **Attention** : Il existe des **changements de focus temporaires** (générés par le système). Ils peuvent être détectés en utilisant la méthode **isTemporary()** sur l'objet événement (l'instance de **FocusEvent** passée au gestionnaire d'événement).
On ignore généralement ces changements temporaires.





Gestion du focus [4]

- Pour transférer le focus à un composant particulier lorsque une fenêtre devient active, on peut utiliser le code suivant (le contrôleur est créé ici sous forme de classe interne anonyme) :

```
-----
// Contrôleur qui donne le focus au composant btOk lorsque la fenêtre viewA
// devient la fenêtre active
-----
viewA.addWindowListener(new WindowAdapter() {
    public void windowActivated(WindowEvent event) {
        btOk.requestFocusInWindow();
    }
});
```

- Un tel contrôleur peut être enregistré sur tous conteneurs de premier niveau, notamment les fenêtres principales (**JFrame**) et les boîtes de dialogue (**JDialog**).



Gestion du focus [5]

- Pour modifier ou compléter les **touches associées au transfert du focus**, il faut modifier l'ensemble (Set) nommé *FocusTraversalKeys* qui existe pour chaque **Container** (rappel : tous les composants le sont).
- Par exemple pour faire en sorte que la touche *Enter* joue le même rôle que la touche *Tab*, on peut utiliser le code suivant :

```
import static java.awt.KeyboardFocusManager.*; // Pour les constantes
. .
. .
// Ajoute la touche Enter dans la liste des touches associées au transfert du focus
// (FORWARD_TRAVERSAL_KEYS)
. .
Set<AWTKeyStroke> fwKeys =
    getFocusTraversalKeys(FORWARD_TRAVERSAL_KEYS);

Set<AWTKeyStroke> newFwKeys = new HashSet<AWTKeyStroke>(fwKeys);
newFwKeys.add(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0));
setFocusTraversalKeys(FORWARD_TRAVERSAL_KEYS, newFwKeys);
```



Gestion du focus [6]

- Pour modifier l'ordre de circulation du focus, il faut écrire une sous-classe de **FocusTraversalPolicy** et redéfinir les méthodes suivantes :

- `getComponentAfter()`
- `getComponentBefore()`
- `getDefaultComponent()`
- `getFirstComponent()`
- `getLastComponent()`

- Il suffit ensuite de créer une instance de cette sous-classe et de l'enregistrer sur le conteneur dont on veut modifier le comportement

```
view.setFocusTraversalPolicy(new NewFocusTraversalPolicy());
```

- La page suivante donne un **exemple** de classe qui modifie le parcours du focus (compte tenu de la place disponible, les importations, les champs et le constructeur ont été omis).



Gestion du focus [7]

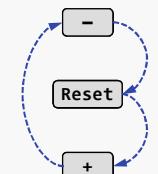
```
public class NewFocusTraversalPolicy extends FocusTraversalPolicy {
    . .
    public Component getComponentAfter(Container cont, Component comp) {
        if (comp == view.btMinus) return view.btReset;
        else if (comp == view.btReset) return view.btPlus;
        else if (comp == view.btPlus) return view.btMinus;
        else return btReset;
    }

    public Component getComponentBefore(Container cont, Component comp) {
        if (comp == view.btPlus) return view.btReset;
        else if (comp == view.btReset) return view.btMinus;
        else if (comp == view.btMinus) return view.btPlus;
        else return btReset;
    }

    public Component getDefaultComponent(Container cont) {
        return view.btReset;
    }

    public Component getFirstComponent(Container cont) {
        return view.btMinus;
    }

    public Component getLastComponent(Container cont) {
        return view.btPlus;
    }
}
```





Bulles d'aide

- Pratiquement tous les composants Swing (ceux qui héritent de la classe **JComponent**) peuvent afficher une **bulle d'aide (ToolTip)** lorsque l'utilisateur laisse le curseur au-dessus.
 - Les bulles d'aide sont très utiles (et très conviviales) pour expliquer le but ou la fonction d'un composant de l'interface graphique.
 - Le texte à afficher est spécifié par la méthode **setToolTipText()**.
- ```
 JButton b1 = new JButton("Suppression");
 b1.setToolTipText("Effacement de l'enregistrement courant");
```
- Il est possible de désactiver l'affichage des bulles d'aide en modifiant la propriété "**enable**" de l'objet **ToolTipManager**. La méthode statique **sharedInstance()** retourne l'instance courante du gestionnaire.
- ```
ToolTipManager.sharedInstance().setEnabled(false);
```
- La désactivation des bulles d'aide peut faire partie des options de configuration de l'application qui sont mises à disposition de l'utilisateur.



Libellés HTML [1]

- Les libellés (textes) de certains composants peuvent être formatés en utilisant des **balises HTML** dans le texte.
- Il suffit de spécifier une chaîne de texte qui commence par la balise "**<HTML>**" (en majuscules ou minuscules).
Remarque : La balise "**</HTML>**" à la fin du texte n'est pas indispensable.
- Des balises de formatage de texte (niveau HTML 3.2) peuvent ensuite être insérées dans le texte.
- Cela permet, entre autres, de :
 - Créer des **libellés multilignes**
 - Utiliser **plusieurs polices** de caractères
 - Utiliser **plusieurs couleurs**
 - Placer du texte en **indice** ou **exposant**
 - ...
- Cette fonctionnalité s'applique aux composants suivants : **JLabel**, **JButton**, **JMenuItem**, **JMenu**, **JCheckBoxMenuItem**, **JTabbedPane**, **JRadioButtonMenuItem** et **JToolTip**.

Attention :

La police de caractères par défaut change (interpréteur HTML)



Libellés HTML [2]

- Affichage d'une bulle d'aide sur plusieurs lignes :

```
comp.setToolTipText("<HTML>Première ligne<BR>deuxième ligne");
```

- En mode HTML, le texte d'un bouton sera automatiquement placé sur plusieurs lignes si la taille du bouton ne permet pas de l'afficher en entier (en mode standard, le texte est tronqué avec "..." à la fin) :

```
 JButton b1 = new JButton("<HTML>Texte éventuellement réparti"
 + "sur plusieurs lignes</HTML>");
```

- Texte en exposant :

```
 JLabel t1 = new JLabel("<HTML>3<sup>ème</sup></HTML>");
```

- Utilisation de CSS (Cascading Style Sheet) :

```
 String cssHead = "<HEAD><LINK rel=STYLESHEET TYPE='text/css' "
 + "HREF=''" + MyClass.class.getResource("fmt.css")
 + "'></HEAD>";
```

```
 JButton b2 = new JButton("<HTML>" + cssHead + "... + "</HTML>");
```



Curseur

- Le **curseur (pointeur de souris)** peut être géré à l'aide de la classe **java.awt.Cursor** et de la méthode **setCursor()** qui existe pour tous les objets de type **Component**.
- La classe **Cursor** permet de créer un certain nombre de curseurs prédéfinis qui suffisent dans la plupart des applications (flèche, sablier, croix, redimensionnement, etc).
- La méthode **setCursor()** peut appliquer un curseur à tous les composants qui héritent de **Component**.
- Si l'on définit le curseur pour un **conteneur**, tous les **composants** qu'il contient (enfants) héritent de ce curseur à moins qu'il ne définissent eux-mêmes un curseur personnalisé.
- Exemple :

```
 JFrame frame = new JFrame("Test sablier");
 Cursor sablier = new Cursor(Cursor.WAIT_CURSOR);
 frame.getRootPane().setCursor(sablier);
```
- Curseur personnalisé : **Cursor createCustomCursor(Image i, ...)** (dans **java.awt.Toolkit**)





Cadres / Bordures [1]

- Tous les composants qui héritent de **JComponent** possèdent une propriété de **bordure** (**border**) qui peut être accédée et manipulée à l'aide des méthodes **getBorder()** et **setBorder()**.
- Le paquetage **javax.swing.border** contient différentes classes (**EmptyBorder**, **LineBorder**, **BevelBorder**, **SoftBevelBorder**, **EtchedBorder**, **MatteBorder**, **TitledBorder**, **CompoundBorder**) qui permettent de créer des bordures prédéfinies que l'on peut appliquer aux composants à l'aide de la méthode **setBorder()**.
- La classe **CompoundBorder** permet d'imbriquer deux bordures pour former une bordure composite qui peut être utilisée comme une bordure simple (on peut donc avoir plusieurs niveaux d'emboîtements).
- Exemple :

```
 JPanel panel1 = new JPanel();
 Border bordureTitre = new TitledBorder("Options");
 panel1.setBorder(bordureTitre);
```



Cadres / Bordures [2]

- Il est également possible de créer des bordures en utilisant les différentes méthodes statiques (**createTitledBorder()**) contenues dans la classe utilitaire **javax.swing.BorderFactory**.

- Exemple :

```
 JPanel panel2 = new JPanel();
 Border bordureRelief =
     BorderFactory.createBevelBorder(BevelBorder.RAISED);
 panel2.setBorder(bordureRelief);
 //-----
 Border bordureEspace = new EmptyBorder(20, 30, 20, 30);
 Border bordureComplexe =
     BorderFactory.createCompoundBorder(bordureRelief,
                                         bordureEspace);
```

- Un même objet **Border** peut être appliqué à plusieurs composants différents (c'est un **composant partageable**).



Cadres / Bordures [3]

- Les bordures permettent de **grouper visuellement** des composants qui ont des caractéristiques communes.
- C'est un élément important permettant de satisfaire le critère ergonomique de *Guidage* et du sous-critère *Groupement/Distinction*.
- Une utilisation fréquente des bordures est le **groupement de boutons radio** de manière à indiquer clairement quels sont les boutons qui font partie du groupe mutuellement exclusif (ne pas oublier que le composant **ButtonGroup** n'a aucun effet visuel). Les bordures de type **TitledBorder** sont fréquemment utilisées dans ce but car elles permettent d'ajouter un libellé à la bordure et de décrire ainsi le groupement.
- Les bordures de type **EmptyBorder** sont très utiles pour placer des **marges autour d'un conteneur** (par exemple un **JPanel**) :

```
/// Define top, left, bottom and right margin (in pixels)
panel.setBorder(BorderFactory.createEmptyBorder(10, 20, 10, 20));
```



Dimensions de l'écran

- Recherche des **dimensions de l'écran** de l'utilisateur (en pixels).
- Cette information est utile par exemple pour centrer (ou positionner de manière judicieuse) la fenêtre de l'application.

```
/// Recherche de la dimension de l'écran et de la fenêtre de l'application
Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
Dimension frameSize = frame.getSize();
//---
// Evite que la fenêtre ne déborde de l'écran
if (frameSize.height > screenSize.height) {
    frameSize.height = screenSize.height;
}
if (frameSize.width > screenSize.width) {
    frameSize.width = screenSize.width;
}
//---
// Centrage de la fenêtre de l'application
frame.setLocation((screenSize.width - frameSize.width) / 2,
                  (screenSize.height - frameSize.height) / 2);
```





Centrage d'une fenêtre de premier niveau

- A partir de la version 1.4 du JDK, il est possible de centrer une fenêtre de premier niveau (par exemple une fenêtre principale [JFrame] ou une boîte de dialogue [JDialog]) par rapport à l'écran ou par rapport à une autre fenêtre.
- On utilise la méthode `setLocationRelativeTo(Parent)`
- `Parent` doit être un composant affiché (typiquement une fenêtre), sinon la fenêtre sera centrée par rapport à l'écran (vrai également si `Parent == null`)
- Exemples :

```
//--- Centrage par rapport à l'écran
mainFrame.setLocationRelativeTo(null);
```

```
//--- Centrage par rapport à une autre fenêtre
dialogBox.setLocationRelativeTo(mainFrame);
```



Fermeture de la fenêtre principale [1]

- On souhaite généralement que la **fermeture de la fenêtre principale** entraîne la fermeture de l'application.
- Dans les cas simples, on peut le faire en utilisant la méthode :


```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```
- Si l'on souhaite demander confirmation à l'utilisateur ou si des opérations de clôture doivent être effectuées lors de la fermeture de l'application (fermeture de fichiers, DB, connexions réseau, etc), il faut créer un récepteur d'événement (`WindowListener` ou `WindowAdapter`) et l'enregistrer sur la frame principale.
- Il faut d'abord créer une classe (récepteur d'événement) qui effectue les opérations nécessaires lors de la fermeture de la fenêtre (voir la classe `CloseWindowAction` à la page suivante) puis enregistrer ce récepteur sur la fenêtre principale :


```
frame.addWindowListener(new CloseWindowAction(frame));
```



Fermeture de la fenêtre principale [2]

```
import java.awt.event.*;
import javax.swing.*;

public class CloseWindowAction extends WindowAdapter {
    JFrame frame;
    public CloseWindowAction(JFrame f) {
        frame = f;
    }
    public void windowClosing(WindowEvent e) {
        //--- Finalisation de l'application (fermeture de fichiers, DB, réseau, etc.)
        ...
        frame.dispose(); // Libération de toutes les ressources liées à l'écran principal
        System.exit(0); // Arrêt de l'application
    }
}
```



Bip sonore

- Dans certaines circonstances, il peut être utile d'émettre un **bip sonore** pour accompagner un message d'erreur, une frappe erronée au clavier, la fin d'une longue opération, etc.
- La méthode `beep()` de la classe `java.awt.Toolkit` permet d'émettre un bip qui dépend du système d'exploitation (et de sa configuration).


```
Toolkit.getDefaultToolkit().beep();
```
- Attention à **ne pas abuser** de ce genre de retour sonore qui peut vite devenir dérangeant pour l'utilisateur et, surtout, pour son entourage.
- Les applications qui offrent un retour sonore devraient toujours offrir à l'utilisateur la **possibilité de le désactiver** (dans les options de configuration de l'application par exemple).





Position et dimension des composants [1]

- La position et la dimension des composants dans un conteneur sont généralement déterminées par le type de conteneur et le gestionnaire de disposition qui est associé (*Layout-Manager*).
- Les méthodes `setSize()`, `setLocation()` et `setBounds()` (de `Component`) peuvent être utilisées pour définir la taille, la position ou les deux à la fois (ces méthodes sont à utiliser **avec les conteneurs de haut-niveau** [qui n'ont pas de gestionnaire de disposition] ainsi qu'avec les conteneurs sans gestionnaire de disposition, c'est-à-dire où la référence `null` est mentionnée comme gestionnaire de disposition).
- Selon le gestionnaire de disposition, ces méthodes n'auront pas d'effet.
- Les gestionnaires de disposition utilisent de préférence les **propriétés suivantes** pour prendre en compte et gérer la taille des composants :
 - `minimumSize` : Taille minimale qu'un composant peut avoir
 - `preferredSize` : Taille préférée d'affichage (si possible)
 - `maximumSize` : Taille maximale d'affichage



Position et dimension des composants [2]

- Ces trois propriétés peuvent être définies à l'aide des méthodes `setMinimumSize()`, `setPreferredSize()` et `setMaximumSize()` de `JComponent`.
- On peut également consulter ces propriétés à l'aides des méthodes `getMinimumSize()`, `getPreferredSize()` et `getMaximumSize()`.
- Si l'on ne définit pas ces dimensions, elles sont automatiquement initialisées par le *UI-Delegate* ou par le gestionnaire de disposition en fonction du contenu du composant (**taille naturelle**).
- Le gestionnaire `FlowLayout` (qui est le gestionnaire par défaut du conteneur `JPanel`) et le gestionnaire `BoxLayout` respectent la taille préférée des composants.
- Les autres gestionnaires tentent de redimensionner les composants en fonction des contraintes mentionnées (`GridBagLayout`) ou de leurs comportements intrinsèques (`GridLayout`, `BorderLayout`, ...).



Police de caractères [1]

- Les polices de caractères (fontes) sont représentées par des instances de la classe `java.awt.Font`.
- Pour créer une fonte, on passe au constructeur un **nom de police**, un **style** et une **taille** exprimée en points (1/72^e de pouce).
- Pour les noms de fonte, on peut utiliser soit des **noms logiques** (disponibles sur toutes les plates-formes) soit des **noms de fontes installées** sur la machine cible (la méthode `getAvailableFontFamilyNames()` de la classe `GraphicsEnvironment` retourne la liste des polices installées). Pour les noms logiques, des constantes sont définies dans la classe `Font` :

| Noms logiques | Équivalents courants |
|--------------------------------|-----------------------------|
| <code>Font.SERIF</code> | <i>Times Roman</i> |
| <code>Font.SANS_SERIF</code> | <i>Helvetica, Arial</i> |
| <code>Font.MONOSPACED</code> | <i>Courier</i> |
| <code>Font.DIALOG</code> | (selon OS et configuration) |
| <code>Font.DIALOG_INPUT</code> | (selon OS et configuration) |



Police de caractères [2]

- Les **styles** des polices de caractères sont définis par des constantes de la classe `Font` (`PLAIN`, `ITALIC` et `BOLD`).
 - L'expression `BOLD+ITALIC` combine les deux styles.
 - Exemple de création d'une police de caractères :
- ```
Font ftTitle = new Font(Font.SANS_SERIF, Font.BOLD, 14);
```
- La méthode `setFont()` de `JComponent` peut être utilisée pour **définir la police** d'un composant qui affiche du texte.
- ```
JLabel lbTitle = new JLabel("Statistiques");
lbTitle.setFont(ftTitle);
```

Attention :

A l'écran, la taille effective de la police de caractères dépend de la résolution de l'écran (configuration de la carte graphique et paramètres de l'OS)





Couleurs

- La classe `java.awt.Color` permet de définir des **couleurs** sur la base de différents espaces de couleur.
- Différentes **constantes** sont définies dans la classe `Color` pour représenter des couleurs prédéfinies (`BLUE`, `RED`, `WHITE`, `PINK`, ...).
- Plusieurs **constructeurs** permettent de créer une couleur en spécifiant les valeurs RGB (canaux rouge, vert et bleu).

```
Color mauve = new Color(200, 100, 150);
Color noir = Color.BLACK;
```

- Les méthodes `setForeground()` et `setBackground()` (héritées de `JComponent`) permettent de **définir la couleur du premier plan** respectivement la couleur de l'**arrière-plan** des composants.

```
JLabel lbName = new JLabel("Pierre");
lbName.setForeground(new Color(50, 50, 200));
```

- La couleur de l'arrière-plan ne peut être définie que si le composant est **opaque** (voir méthodes `isOpaque()` et `setOpaque()` de `JComponent`).



JOptionPane [1]

- Le composant complexe `JOptionPane` permet de créer une **fenêtre de dialogue** (fenêtre séparée ou *pop-up*) qui servira à **afficher un message** ou à **obtenir des informations** de la part de l'utilisateur.
- La classe `JOptionPane` crée automatiquement un conteneur de haut-niveau `JDialog` pour afficher les informations et recevoir les réponses de l'utilisateur.
- Une fenêtre `JOptionPane` comprend 4 zones (optionnelles)
 - Message**
 - Icône**
 - Entrée**
 - Boutons**



JOptionPane [2]

- Les composants `JOptionPane` peuvent être créés de deux manières différentes :
 - Au moyen d'un des **constructeurs** disponibles (offre une flexibilité maximale mais nécessite plus de travail)
 - Au moyen d'une des nombreuses **méthodes statiques** nommées `showTypDialog()` (permet de couvrir la plupart des situations courantes)
- La classe `JOptionPane` permet également de créer des **fenêtres internes de dialogue** (`JInternalFrame`) qui ne s'afficheront pas comme fenêtre de haut-niveau mais comme sous-fenêtre d'un conteneur `JDesktopPane` (réservé aux applications utilisant le mode MDI : *Multiple Document Interface*).
- Les pages qui suivent ne présentent que des exemples qui s'affichent dans des fenêtres de haut-niveau (l'utilisation la plus fréquente).



JOptionPane [3]

- Les fenêtres de dialogue peuvent être divisées en quatre catégories différentes :
 - Message** : Affichage d'un message d'information
 - Input** : Saisie d'une information entrée par l'utilisateur
 - Confirm** : Demande de confirmation (*Ok*, *Oui*, *Non*, *Annulation*)
 - Option** : Choix d'une option spécifique (parmi une liste de boutons)
- A chacune de ces catégories correspond un **ensemble de méthodes statiques** surchargées pour différents profils de paramètres.
- Les méthodes statiques `showTypDialog()` créent des fenêtres de dialogue de haut-niveau. Ces fenêtres sont **modales** (ce qui signifie que le focus ne peut pas être transféré à un composant d'une autre fenêtre de l'application sans fermer la fenêtre de dialogue).
- Les méthodes statiques `showInternalTypDialog()` créent des sous-fenêtres affichées dans des conteneurs de type `JDesktopPane`. (dans ce cas, la fenêtre parente doit être de type `JDesktopPane`).





JOptionPane : Message

- Les méthodes `showMessageDialog()` affichent un message et attendent la quittance de l'utilisateur.
- Il n'y a pas de valeur de retour.
- Le premier paramètre (de toutes les méthodes `showTypDialog()`) indique à quel composant (parent) la fenêtre de dialogue est liée ce qui détermine sa dépendance et son positionnement (si `null`, la fenêtre est centrée à l'écran).
- Exemple :

```
JOptionPane.showMessageDialog(parent,
                           "Aucun virus détecté");
```



JOptionPane : Input

- Les méthodes `showInputDialog()` affichent des fenêtres qui comportent une zone d'entrée (input) qui permet à l'utilisateur d'introduire un texte ou de sélectionner un élément dans une liste à choix.
- Ces méthodes retournent une valeur de type `String` ou `Object` selon la version utilisée (ou `null` si l'utilisateur annule ou ferme la fenêtre).
- Exemple :

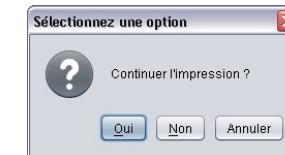
```
String s = JOptionPane.showInputDialog(parent,
                                       "Entrer votre nom :");
```



JOptionPane : Confirm

- Les méthodes `showConfirmDialog()` affichent des fenêtres qui comportent des boutons de type "Ok", "Cancel", "Yes", "No" (différentes combinaisons sont possibles sur la base des constantes (*option type*) définies dans la classe `JOptionPane`).
- Ces méthodes retournent une valeur entière correspondant à une des constantes de `JOptionPane` (`CANCEL_OPTION`, `CLOSED_OPTION`, `NO_OPTION`, `YES_OPTION`, `OK_OPTION`).
- Exemple :

```
int i = JOptionPane.showConfirmDialog(null,
                                      "Continuer l'impression ?",
                                      if (i==JOptionPane.YES_OPTION) {...}
```



JOptionPane : Input

- Les méthodes `showInputDialog()` affichent des fenêtres qui comportent une zone d'entrée (input) qui permet à l'utilisateur d'introduire un texte ou de sélectionner un élément dans une liste à choix.
- Ces méthodes retournent une valeur de type `String` ou `Object` selon la version utilisée (ou `null` si l'utilisateur annule ou ferme la fenêtre).
- Exemple :

```
String s = JOptionPane.showInputDialog(parent,
                                       "Entrer votre nom :");
```



JOptionPane : Option

- Les méthodes `showOptionDialog()` complètent les possibilités offertes par les méthodes `showConfirmDialog()` et permettent de transmettre un tableau d'objets qui seront affichés sous forme de boutons (ils remplacent les boutons standard de `showConfirmDialog()`).
- Ces méthodes retournent une valeur entière correspondant à l'indice du tableau identifiant l'objet qui a été sélectionné (ou `CLOSED_OPTION`).
- Exemple :

```
String[] sOpt = {"Froid", "Tiède", "Chaud"};
int i = JOptionPane.showOptionDialog(null,
                                    "Vos préférences ?",
                                    JOptionPane.DEFAULT_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null,
                                    sOpt, sOpt[1]);
```



Type d'option
(pas utilisé si options définies)

Valeur initiale

Type de message
(détermine l'icône par défaut)

Icône





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

07

Mnémoniques / Accélérateurs / Actions Menus déroulants / Menus contextuels

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Menus déroulants et contextuels

Gestion des menus

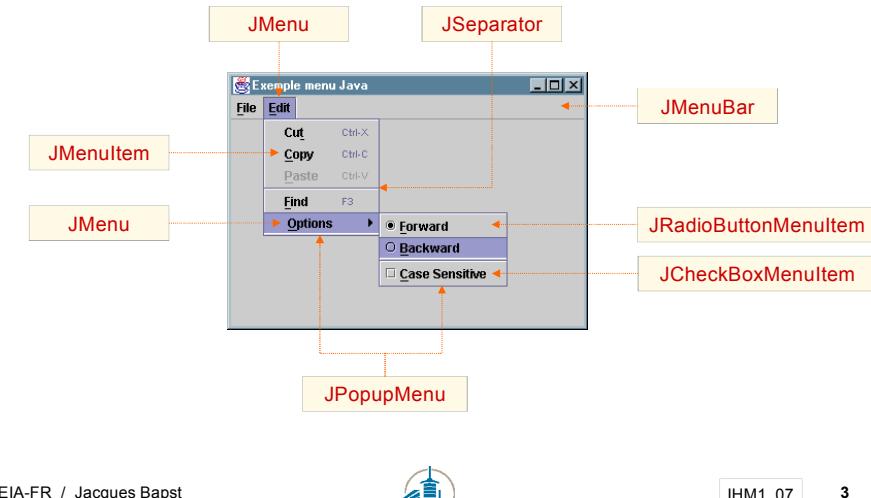
- Les **menus** (déroulants et contextuels) sont gérés par un ensemble de composants dont les principaux sont :
 - JMenuBar** : Barre de menus contenant des composants **JMenu**
 - JMenu** : Menu déroulant ou sous-menu représenté par un libellé qui constitue le point d'entrée (titre)
 - JMenuItem** : Élément (option) d'un menu ou d'un sous-menu
 - JPopupMenu** : Conteneur des éléments du menu
- Les composants suivants peuvent également être utilisés comme éléments d'un menu :
 - JCheckBoxMenuItem** : Identique au composant **JCheckBox**
 - JRadioButtonMenuItem** : Identique au composant **JRadioButton**
 - JSeparator** : Ligne séparatrice utilisée pour grouper les options



Interface Homme-Machine 1 / Menus déroulants et contextuels

Composants utilisés dans les menus

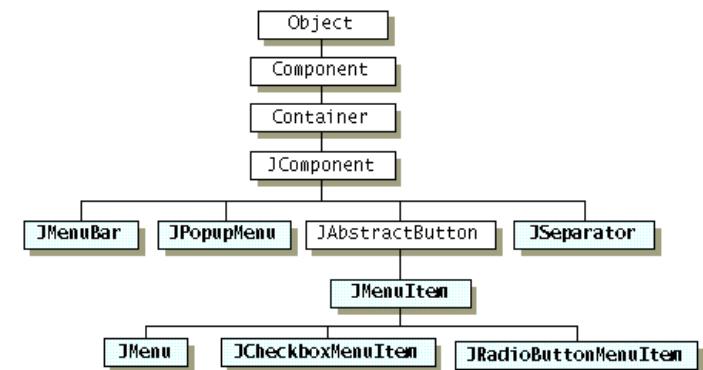
- Les différents **composants** qui interviennent dans la gestion des **menus déroulants** sont illustrés dans l'exemple suivant :



Interface Homme-Machine 1 / Menus déroulants et contextuels

Menus : hiérarchie des composants

- La **hiérarchie des classes** utilisées pour la gestion des menus est la suivante :





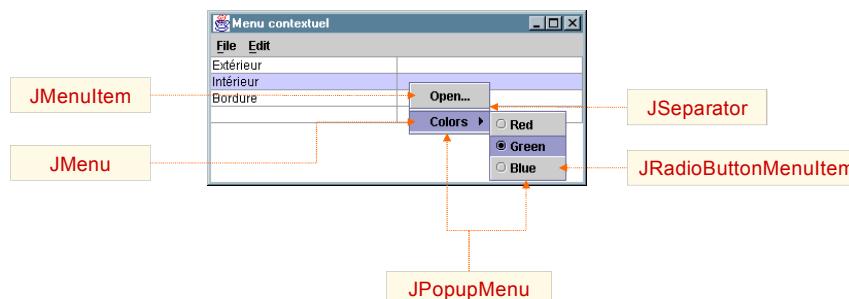
Création des menus

- Les menus peuvent être utilisés de deux manières différentes :
 - Menus déroulants** (*Drop-down Menu*)
 - Menus contextuels** (*Pop-up Menu*)
- Les éléments des **menus déroulants**, généralement rassemblés dans une barre de menus (**JMenuBar**), sont affichés (déroulés) lorsque l'utilisateur clique sur le titre du menu (**JMenu**).
- Les éléments des **menus contextuels** sont affichés en réaction à un événement (généralement lorsqu'on clique sur le bouton droit de la souris) et sont présentés dans une fenêtre *Pop-up* indépendante (**JPopupMenu**) qui est habituellement affichée à l'emplacement où l'on a déclenché le menu contextuel.
- L'affichage et la sélection des éléments du menu sont traités de manière identique pour les menus déroulants et les menus contextuels. Le composant **JPopupMenu** est utilisé dans les deux cas.



Menus contextuels

- Les différents composants qui interviennent dans la gestion des **menus contextuels** sont illustrés dans l'exemple suivant :



Définition de mnémoniques

- La définition d'un **mnémonique** permet d'associer une **touche d'abréviation** (caractère alphanumérique) à un **menu** (placé dans une barre de menus), à une **option de menu** ou à un **bouton**.
- A l'exécution, le mode d'**activation** de l'abréviation dépend du système d'exploitation. Sur *Windows*, c'est l'association de la touche **Alt** avec le **caractère** défini qui provoquera l'activation du menu ou du bouton considéré.
- A l'affichage, le mnémonique est généralement indiqué par le **soulignement** du caractère d'abréviation.
- Le mnémonique est enregistré à l'aide de la méthode **setMnemonic()** héritée de la classe **AbstractButton**.



```
JMenuItem miQuit = new JMenuItem("Quitter");
miQuit.setMnemonic('Q');
```



Accélérateurs de menu

- Un **accélérateur de menu** est une commande clavier unique qui peut être utilisée pour invoquer une option de menu même quand celui-ci n'est pas affiché.
- Un accélérateur est représenté par un objet de type **KeyStroke** (dans **javax.swing**) et comprend généralement un **modificateur clavier** comme **Alt** ou **Ctrl** (ou les deux).
- L'enregistrement de l'accélérateur s'effectue à l'aide de la méthode **setAccelerator()** de **JMenuItem**.

```
KeyStroke ctrlS = KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                             Event.CTRL_MASK);
miSave.setAccelerator(ctrlS);
```

- Différentes surcharges de la méthode statique **getKeyStroke()** permettent de créer une commande clavier.

- Autre variante :

```
KeyStroke ctrlS = KeyStroke.getKeyStroke("control S");
```





Mnémoniques et accélérateurs

- Il est fréquent de créer des **mnémoniques** et/ou des **accélérateurs** sur certains éléments des menus :

```
JMenuItem mItem3 = new JMenuItem("Sauver");
mItem3.setMnemonic('S');

KeyStroke ctrlS = KeyStroke.getKeyStroke("control S");
mItem3.setAccelerator(ctrlS);
```

- La création de mnémoniques et d'accélérateurs doit être réservée aux **actions les plus fréquemment utilisées** de l'application de manière à simplifier le travail des utilisateurs expérimentés.
- La possibilité de pouvoir travailler quitter le clavier peut être un critère ergonomique important dans certains types d'applications (saisie de grande quantité de données par exemple).
- Selon les plates-formes, des conventions (ou habitudes) existent quant au choix des accélérateurs. Dans la mesure du possible, il est important de s'y conformer.



Action

- Une interface graphique permet souvent à un utilisateur d'invoquer une **même opération de plusieurs manières** différentes.
- Par exemple, l'utilisateur peut sauvegarder un fichier soit en utilisant le **menu** (*Fichier → Sauver*) soit en cliquant sur un bouton disponible dans une **barre d'outils** (*Toolbar*) ou alors en invoquant un **menu contextuel** (*en cliquant sur le bouton droit de la souris*).
- L'interface **Action** (`javax.swing`) a été définie pour faciliter la gestion de ce genre d'opérations.
- Si l'on passe (dans la méthode `add()`) un objet de type **Action** à un composant de type **JMenu** ou **JToolBar**, le composant créera automatiquement une entrée dans le menu (**JMenuItem**) respectivement un bouton (**JButton**) dans la barre d'outils pour représenter l'action.
- Lorsque l'on implémente l'interface **Action**, on doit définir plusieurs méthodes dont `actionPerformed()` qui sera activée lorsque l'utilisateur sélectionnera l'option du menu ou pressera sur le bouton correspondant.

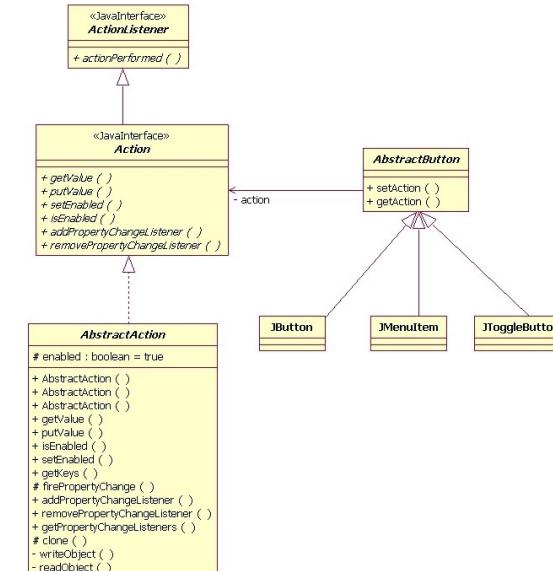


AbstractAction

- La classe **AbstractAction** (`javax.swing`) est une implémentation par défaut de l'interface **Action**.
- La classe **AbstractAction** fournit une implémentation pour toutes les méthodes de l'interface **Action** sauf pour la méthode essentielle `actionPerformed()`.
- Si l'on sous-classe **AbstractAction** il suffit donc de d'implémenter le constructeur (dans lequel on définira le **texte** et/ou l'**icône** à afficher) ainsi que la méthode `actionPerformed()` dont le code sera exécuté lorsque l'utilisateur sélectionnera l'option du menu ou pressera sur le bouton de la barre d'outils.
- La sous-classe ainsi définie permet de créer un objet de type **Action** qui peut ensuite être utilisé dans les méthodes `add()` des composants **JMenu** et **JToolbar**.
- Lorsque que l'on ajoute un objet de type **Action**, la méthode `add()` retourne une référence vers le composant qui a été créé (bouton ou option de menu).



Action – AbstractAction





Gestion des événements des menus

- L'activation d'un élément de menu (option) a pour but de déclencher l'exécution de certaines opérations.
- Pour associer des instructions à un élément de menu, il faut enregistrer un récepteur d'événement sur cet élément de menu.
- Plusieurs types de récepteurs d'événement peuvent être enregistrés sur les éléments du menu (**JMenuItem**).
- Dans la plupart des cas (éléments de type **JMenuItem**), on enregistre un récepteur d'événement de type **ActionListener** et la gestion de l'événement s'effectue **de la même manière que pour un bouton**.
- Pour les éléments de type **JCheckBoxMenuItem** et **JRadioButtonMenuItem** on utilisera de préférence un récepteur d'événement **ItemListener** qui permet de consulter plus facilement l'état de l'élément.
- Rappel : les composants **JMenuItem**, **JCheckBoxMenuItem** et **JRadioButtonMenuItem** sont des sous-classes de **AbstractButton** et sont donc des boutons.



JMenuItem

- Le composant **JMenuItem** constitue l'élément de base (option) d'un menu déroulant ou contextuel.
- Les éléments des menus sont considérés comme des boutons spécialisés (**JMenuItem** est une sous-classe de **AbstractButton**).
- Les composants **JMenuItem** peuvent comprendre un **texte**, une **icône** ou **les deux** (comme les boutons).
- Différents constructeurs sont à disposition pour créer les éléments d'un menu :

```
JMenuItem mItem1 = new JMenuItem("Copier");

Icon printIcon = new ImageIcon("printer.gif");
JMenuItem mItem2 = new JMenuItem("Imprimer", printIcon);
```



JMenu

- Le composant **JMenu** représente un conteneur d'éléments de menu (**JMenuItem**) qui peut être placé dans une barre de menus (**JMenuBar**) ou être inséré comme sous-menu dans un autre **JMenu**.
- Initialement le composant **JMenu** affiche uniquement son libellé (titre) dans la barre de menus ou comme élément d'un menu déroulant (il identifie dans ce cas un sous-menu).
- Lorsque le composant **JMenu** est sélectionné, les éléments qu'il contient sont affichés dans une fenêtre de type **JPopupMenu**.
- La position de la fenêtre **JPopupMenu** est différente s'il s'agit du déroulement d'un menu de premier niveau ou s'il s'agit de l'ouverture d'un sous-menu.
- Lorsque le composant **JMenu** identifie un sous-menu, il est accompagné d'un indicateur visuel (généralement une flèche dirigée vers la droite) qui le distingue d'un élément terminal (élément de dernier niveau).



Création de JMenu

- La création d'un composant **JMenu** s'effectue à l'aide d'un des constructeurs disponibles.
- Dans la très grande majorité des cas, on utilise la variante dans laquelle on transmet en paramètre le libellé (titre) du menu sous forme d'un **String**.

```
// --- Création de trois menus avec les titres "File", "Edit" et "Options"
JMenu fileMenu      = new JMenu("File");
JMenu editMenu      = new JMenu("Edit");
JMenu optionsMenu = new JMenu("Options");
```

Remarque : Selon les plates-formes, il existe des conventions (ou des habitudes) concernant le libellé et la disposition des menus et de leurs options. Si l'on a pas de raisons particulières de faire autrement, il est recommandé de s'y conformer de manière à ce que l'utilisateur se retrouve en terrain connu et faciliter ainsi la phase d'apprentissage.





Ajout des éléments dans un menu [1]

- Différentes méthodes `add()` et `insert()` permettent d'ajouter des éléments dans un menu (`JMenu`).
- Les éléments insérés dans les menus sont généralement de type `JMenuItem`, `JCheckBoxMenuItem`, `JRadioButtonMenuItem`, `JMenu` ou `Action` (bien qu'en théorie, n'importe quelle sous-classe de `Component` puisse être insérée)
- Les méthodes `add()` ajoutent les éléments de haut en bas, dans l'ordre d'invocation de la méthode (ajout à la fin de la liste).
- Les méthodes `insert()` permettent d'insérer des éléments à des positions spécifiques en mentionnant un nombre positif qui détermine l'ordre relatif des éléments.
- Les éléments du menu peuvent être séparés (groupés) en utilisant les méthodes `addSeparator()` ou `insertSeparator()`. Ces méthodes insèrent un composant de type `JSeparator` comme élément du menu (ligne séparatrice).



Ajout des éléments dans un menu [2]

```
===== Menu "File" =====
JMenu fileMenu = new JMenu("File");
JMenuItem mOpen = new JMenuItem("Open");
JMenuItem mClose = new JMenuItem("Close");
JMenuItem mExit = new JMenuItem("Exit");

fileMenu.add(mOpen);
fileMenu.add(mClose);
fileMenu.add(mExit);

===== Menu "Options" =====
JMenu optionsMenu = new JMenu("Options");

---Utilisation d'un objet de type Action comme élément du menu
---La classe ProgSettings doit implémenter l'interface Action
Action setConfig = new ProgSettings();
optionsMenu.add(setConfig);
```

Remarque : L'utilisation des objets de type `Action` a été décrite dans les pages précédentes.



Création de sous-menus

- La création de **sous-menus** s'effectue simplement en insérant dans un menu (`JMenu`) un autre objet `JMenu` contenant les éléments (options) du sous-menu (le niveau d'imbrication n'est pas limité).

```
--- Création du menu et des éléments de premier niveau
JMenu premierNiveau = new JMenu("Premier niveau");
JMenuItem elem1 = new JMenuItem("Option 1 niveau 1");
JMenuItem elem3 = new JMenuItem("Option 3 niveau 1");

--- Création du sous-menu avec ses éléments
JMenu sousMenu = new JMenu("Deuxième niveau");
JMenuItem sousElem1 = new JMenuItem("Option 1 niveau 2");
JMenuItem sousElem2 = new JMenuItem("Option 2 niveau 2");

sousMenu.add(sousElem1);
sousMenu.add(sousElem2);

--- Ajout des éléments au premier niveau
premierNiveau.add(elem1);
premierNiveau.add(sousMenu);
premierNiveau.add(elem3);
```



JMenuBar

- Le composant `JMenuBar` est utilisé pour créer une barre de menus.
 - La classe `JMenuBar` comporte un unique constructeur sans argument.
- ```
JMenuBar menuBar = new JMenuBar();
```
- Les barres de menus sont généralement insérées dans des conteneurs de premier niveau : `JFrame`, `JDialog`, `JApplet`, `JInternalFrame` ou dans `JRootPane` au moyen de la méthode `setJMenuBar()`.
  - Elles peuvent également être insérées dans n'importe quel conteneur au moyen de la méthode `add()` de `Container`.
  - Exemple d'insertion d'une barre de menus dans un conteneur de premier niveau de type `JFrame` :
- ```
JFrame frame = new JFrame("Test JMenuBar");
frame.setJMenuBar(menuBar);
```





Ajout de menus dans une barre de menus

- L'ajout de menus (**JMenu**) dans une barre de menus (**JMenuBar**) s'effectue à l'aide de la méthode **add()** de **JMenuBar**.
- Les menus sont ajoutés de gauche à droite dans la barre de menus selon l'ordre d'exécution des méthodes **add()**.

```
JMenuBar menuBar = new JMenuBar();

JMenu fileMenu = new JMenu("File");
menuBar.add(fileMenu);

JMenu editMenu = new JMenu("Edit");
menuBar.add(editMenu);
```



JCheckBoxMenuItem

- Le composant **JCheckBoxMenuItem** se comporte comme le composant **JCheckBox** mais il est spécialisé pour être inséré dans un menu déroulant ou contextuel (**JMenu** ou **JPopupMenu**).
- La classe **JCheckBoxMenuItem** est une sous-classe de **JMenuItem**.
- La gestion des événements associés à un élément de menu **JCheckBoxMenuItem** s'effectue de la même manière que pour un composant **JCheckBox**. La technique la plus simple consiste à créer et enregistrer un récepteur d'événement de type **ItemListener** qui permet de consulter l'état du composant.

```
JMenu optionsMenu = new JMenu("Options");
JCheckBoxMenuItem choix1 = new JCheckBoxMenuItem("Muet");
optionsMenu.add(choix1);
```



JRadioButtonMenuItem

- Le composant **JRadioButtonMenuItem** se comporte comme le composant **JRadioButton** mais il est spécialisé pour être inséré dans un menu déroulant ou contextuel (**JMenu** ou **JPopupMenu**).
- La classe **JRadioButtonMenuItem** est une sous-classe de **JMenuItem**.
- La gestion des événements associés à un élément de menu **JRadioButtonMenuItem** s'effectue de la même manière que pour un composant **JRadioButton**. La technique la plus simple consiste à créer et enregistrer un récepteur d'événement de type **ItemListener** qui permet de consulter l'état du composant.
- Pour obtenir des choix mutuellement exclusifs, les composants **JRadioButtonMenuItem** doivent être insérés dans un **ButtonGroup** (comme pour les composants **JRadioButton**).

```
JMenu optionsMenu = new JMenu("Options");
JRadioButtonMenuItem cYes = new JRadioButtonMenuItem("Yes");
optionsMenu.add(cYes);
```



Menus contextuels (Pop-up)

- Les **menus contextuels (Pop-up)** sont des menus qui ne sont pas liés à une barre de menus. Ce sont des **menus flottants** qui sont **associés à un composant** visuel et dont l'affichage est déclenché par une action de l'utilisateur.
- Le mode d'activation du menu dépend de la plate-forme d'exécution (généralement, c'est l'utilisation de la touche droite de la souris qui provoque l'affichage des menus contextuels).
- Une fois invoqué (le menu s'affiche généralement à l'endroit où l'utilisateur a cliqué), le menu s'utilise de la même manière que les menus déroulants.
- Le composant **JPopupMenu** est utilisé pour créer un menu contextuel.
- Les éléments du menu (options) sont ajoutées à l'aide des méthodes **add()**, **addSeparator()** et **insert()** comme pour l'insertion des éléments dans un **JMenu**.
- Les éléments insérés dans les menus sont généralement de type **JMenuItem**, **JCheckBoxMenuItem**, **JRadioButtonMenuItem**, **JMenu** OU **Action**.





Création des menus contextuels

- La création des menus contextuels est très similaire à la création des menus déroulants (dans les menus contextuels, le composant **JPopupMenu** remplace le composant **JMenu** qui est utilisé dans les menus déroulants).

```
JPopupMenu popUp = new JPopupMenu();

JMenuItem itemDelete = new JMenuItem("Delete");
JMenuItem itemFind   = new JMenuItem("Find");
JMenuItem itemCheck  = new JMenuItem("Check");

popUp.add(itemDelete);
popUp.addSeparator();
popUp.add(itemFind);
popUp.add(itemCheck);
```



Affichage des menus contextuels [1]

- Contrairement aux composants **JMenu**, les composants **JPopupMenu** nécessitent de définir un **gestionnaire d'événement** pour déclencher l'affichage du menu contextuel.
- L'affichage du menu contextuel est effectué par invocation de la méthode **show()** (de **JPopupMenu**) qui a la signature suivante :

```
public void show(Component invoker, int x, int y)
```

- Cette méthode affiche le menu contextuel à la position *x*, *y* du composant parent (*invoker*).
- La méthode **show()** doit être appelée dans le cadre d'un récepteur d'événement de type **MouseListener** (ou de la classe associée **MouseAdapter**).
- Avant d'appeler la méthode **show()**, le récepteur d'événement doit s'assurer que l'événement correspond bien à l'invocation d'un menu contextuel.



Affichage des menus contextuels [2]

- Pour être indépendant de la plate-forme, il faut utiliser la méthode **isPopupTrigger()** (de **MouseEvent** ou de **JPopupMenu**) pour déterminer si l'événement correspond à l'invocation du menu contextuel.
- La méthode **isPopupTrigger()** doit être invoquée aussi bien dans la méthode **mousePressed()** que dans la méthode **mouseReleased()** si l'on souhaite être indépendant de la plate-forme.
- Le code pour activer les menus contextuels peut être intégré dans une classe réutilisable pour tous les menus contextuels.
- La classe **PopupMenuDisplay** (voir page suivante) est un exemple de classe qui peut être réutilisée pour afficher les menus contextuels.

Remarque : Par manque de place sur la page suivante, les instructions d'importation (`java.awt.*`, `java.awt.event.*`, `javax.swing.*`) n'ont pas été mentionnées dans le code.



Affichage des menus contextuels [3]

```
=====
// Récepteur d'événement pour l'affichage des menus contextuels
=====
public class PopupMenuDisplay extends MouseAdapter {

    private JPopupMenu popup;

    public PopupMenuDisplay(JPopupMenu popup) {
        this.popup = popup;
    }

    private void showIfPopupTrigger(MouseEvent mouseEvent) {
        //--- Test si l'événement correspond à l'activation du menu contextuel
        if (mouseEvent.isPopupTrigger())
            popup.show(mouseEvent.getComponent(), mouseEvent.getX(),
                      mouseEvent.getY());
    }

    public void mousePressed(MouseEvent mouseEvent) {
        showIfPopupTrigger(mouseEvent);
    }

    public void mouseReleased(MouseEvent mouseEvent) {
        showIfPopupTrigger(mouseEvent);
    }
}
```



Affichage des menus contextuels [4]

- A partir de la version 1.5 du JDK, une nouvelle méthode a été ajoutée à la classe **JComponent** pour enregistrer un menu contextuel sur un composant :

```
setComponentPopupMenu(JPopupMenu popup)
```

- Dans la même classe, une autre méthode permet de dire si un composant doit *hériter* du menu contextuel du conteneur parent :

setInheritsPopupMenu(boolean value)

- Par défaut, il n'y a pas d'héritage du menu contextuel.

- Remarque : Un bug dans ces méthodes (corrige en version 1.6) empêche le fonctionnement correct des menus contextuels sur les conteneurs de type **JPanel**. L'enregistrement d'une bulle d'aide (même vide) sur le conteneur permet cependant de contourner cette erreur (voir exemple à la page suivante).



Menus contextuels : gestion des événements

- La **gestion des événements** associés à l'activation des éléments (options) d'un menu contextuel est strictement identique à la gestion des événements d'un menu déroulant.
 - Il faut créer des récepteurs d'événements **ActionListener** ou **ItemListener** (selon le type de l'élément) et les enregistrer sur les éléments du menu (qui sont traités comme des boutons).
 - Pour plus d'informations voir les remarques et exemples mentionnés dans la section consacrée à la gestion des événements des menus déroulants. Ces indications s'appliquent sans restrictions aux menus contextuels.



Affichage des menus contextuels [5]

```
// Enregistrement des menus contextuels (possible à partir de la version 1.5 du JDE
=====
private JButton btOk;
private JTextField tfName;
private JPanel pnTop;
private JPopupMenu popupA;
private JPopupMenu popupB;
.
.
popupA.add(...); // Création des menus contextuels
popupA.add(...); // " " " "
.
.
popupB.add(...); // " " " "
popupB.add(...); // " " " "
.
.
// Enregistrement du menu contextuel popupA sur le bouton btOk
btOk.setComponentPopupMenu(popupA);

// Enregistrement du menu contextuel popupB sur le conteneur pnTop
pnTop.setToolTipText(""); // Pour éviter un bug en version 1.5
pnTop.setComponentPopupMenu(popupB);

// Le champ texte tfName héritera du menu contextuel du conteneur parent
tfName.setInheritsPopupMenu(true);
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI)

08

Barres d'outils, Listes, Listes déroulantes, JSpinner

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Barres d'outils

JToolBar [1]

- Le composant **JToolBar** permet de créer une **barre d'outils**.
- Les barres d'outils sont fréquemment utilisées pour permettre à l'utilisateur d'accéder de manière directe à des fonctions fréquentes sans avoir à dérouler des menus.
- Le composant **JToolBar** est un conteneur spécialisé dans lequel on peut insérer différents composants (généralement des boutons et des listes déroulantes).
- Les barres d'outils **JToolBar** sont généralement placées (ancrées) sur l'un des bords d'un conteneur dont la disposition est gérée par **BorderLayout**.
- Si la propriété **floatable** de **JToolBar** est à **true**, l'utilisateur pourra détacher la barre d'outils de son emplacement et elle deviendra alors une barre d'outils indépendante (flottante).
- Une **barre d'outils flottante** peut être placée n'importe où sur l'écran et peut être à nouveau ancrée sur l'un des bords du conteneur.

EIA-FR / Jacques Bapst



IHM1_08

2



Interface Homme-Machine 1 / Barres d'outils

JToolBar [2]

- Pour que le mécanisme de barre d'outils flottante fonctionne correctement, il faut que le conteneur destiné à ancrer la barre d'outils dispose d'un **gestionnaire de disposition** de type **BorderLayout** et que les emplacements à la périphérie du conteneur (*North*, *South*, *East* et *West*) ne contiennent aucun autre composant.
- La méthode **addSeparator()** peut être utilisée pour insérer un espace entre les composants de la barre d'outils.
- Dans une barre d'outils horizontale, les composants insérés devraient avoir une hauteur plus ou moins identique (si la barre est verticale, c'est la largeur des composants qui devrait être plus ou moins identique).
- La méthode **add()** de **JToolBar** permet aussi d'ajouter des objets de type **Action** qui sont automatiquement transformés en boutons et qui sont associés à un récepteur d'événement (permet de simplifier le travail si la même action peut être activée par un menu et par une barre d'outils).

Remarque : L'utilisation des objets de type **Action** a été décrite dans le chapitre consacré aux menus déroulants et contextuels.

EIA-FR / Jacques Bapst



IHM1_08

3



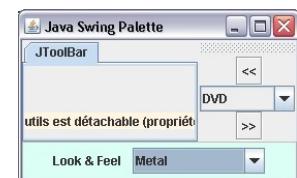
Interface Homme-Machine 1 / Barres d'outils

Exemple JToolBar

```
//--- Création de la barre d'outils avec deux boutons
JToolBar toolBar = new JToolBar("Outils");
//--- Création de deux boutons et ajout dans la barre d'outils
 JButton b1 = new JButton("Envoyer");
 JButton b2 = new JButton("Imprimer");
 toolBar.add(b1);
 toolBar.addSeparator(); // Espace entre les deux boutons
 toolBar.add(b2);
//--- Création d'un conteneur JPanel avec le gestionnaire BorderLayout
//--- Insertion de la barre d'outils dans la zone Nord du conteneur
 JPanel p1 = new JPanel(new BorderLayout());
 p1.add(toolBar, BorderLayout.NORTH);
```



Barre d'outils ancrée au Nord



Barre d'outils ancrée à l'Est



Barre d'outils flottante

EIA-FR / Jacques Bapst



IHM1_08

4



JList

- Le composant **JList** permet de présenter à l'utilisateur une **liste d'éléments** parmi lesquels il peut en **sélectionner un ou plusieurs**.
- Le composant **JList** affiche en permanence le contenu de la liste. Si tous les éléments ne peuvent être affichés, il faut offrir une barre de défilement (**JScrollPane**) pour parcourir les éléments de la liste (ce n'est pas automatique, le programmeur doit s'en charger).



Conseil : Insérer systématiquement les composants **JList** dans un composant **JScrollPane** pour qu'une barre de défilement s'affiche automatiquement lorsque c'est nécessaire.

```
JList maListe = new JList(...);
JScrollPane scListe = new JScrollPane(maListe);
```

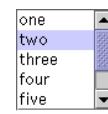


JList : Mode de sélection

- Différents **modes de sélection** peuvent être configurés en définissant la propriété **selectionMode** de **JList**.
- Les valeurs possibles sont définies par des constantes de la classe **ListSelectionModel** :

SINGLE_SELECTION

Un seul élément peut être sélectionné

**SINGLE_INTERVAL_SELECTION**

Une zone d'**éléments contigus** peut être sélectionnée

**MULTIPLE_INTERVAL_SELECTION**

Un **ensemble d'éléments quelconques** peut être sélectionné (comportement par défaut)



Sélection dans une JList

- Pour l'utilisateur, la manière de **sélectionner les éléments** dans ces différents modes est dépendante du *Look-and-Feel* utilisé sur la plate-forme d'exécution.
- Dans la plupart des cas, le comportement est le suivant :
 - Le **bouton gauche de la souris** est utilisé pour sélectionner un élément individuel.
 - La touche **Shift** est utilisée en combinaison avec le bouton gauche de la souris pour sélectionner une zone d'éléments contigus (de...à).
 - La touche **Ctrl** utilisée en combinaison avec le bouton gauche de la souris permet de sélectionner (ou de désélectionner) des éléments individuels sans modifier l'état de la sélection des autres éléments.



Architecture de JList

- Le composant **JList** est composé de trois éléments principaux :
 - Un **modèle de données** qui contient les éléments de la liste
 - Un **modèle de sélection** qui enregistre l'état de la sélection des éléments
 - Un **afficheur d'élément** qui est chargé de la représentation visuelle de chaque élément de la liste (l'élément est également appelé **cellule**)
- Le **modèle de données (data model)** doit être une implémentation de l'interface **ListModel**. Par défaut, le composant **JList** utilise la classe **DefaultListModel**, une implémentation de **ListModel** qui enregistre les éléments dans une structure de type **Vector**.
- Le **modèle de sélection (selection model)** doit implémenter l'interface **ListSelectionModel**. Par défaut, le composant **JList** utilise la classe **DefaultListSelectionModel**.
- L'**afficheur d'élément (cell renderer)** doit implémenter l'interface **ListCellRenderer**. Par défaut, le composant **JList** utilise la classe **DefaultListCellRenderer** qui est une sous-classe de **JLabel**.





Création d'une JList [1]

- Différents constructeurs permettent de créer un composant **JList** et de l'alimenter avec un ensemble d'éléments.
- Les éléments peuvent être transmis au constructeur sous forme de tableau d'objets (**Object[]**), d'une collection de type **Vector** ou d'un objet qui implémente l'interface **ListModel**.

```
String[] nomsJours = {"Lundi", "Mardi", "Mercredi",
                      "Jeudi", "Vendredi", "Samedi",
                      "Dimanche"};
JList listeJours = new JList(nomsJours);
```



Création d'une JList [2]

- Par défaut (utilisation de **DefaultListCellRenderer**), les objets transmis aux constructeurs sont convertis sous forme textuelle (par invocation de la méthode **toString()**) avant d'être représentés (au niveau de chaque élément) par des composants de type **JLabel**.
- Les éléments** enregistrés dans une **JList** par transmission d'un tableau d'objets ou d'une collection de type **Vector** **ne peuvent plus être modifiés après coup** (ce n'est possible qu'en remplaçant le modèle de données).
- Le **remplacement du modèle** de données d'une **JList** s'effectue soit en invoquant la méthode **setModel()** (qui prend en paramètre un objet de type **ListModel**), soit en utilisant la méthode **setListData()** à laquelle on peut passer un tableau d'**Object** ou un **Vector** et qui créera un nouveau modèle qui remplacera le modèle actuel.



JList : Propriétés

- Le composant **JList** dispose d'un grand nombre de propriétés.
- Les principales sont les suivantes :

| Propriété | Type | R/W | Description |
|---------------------------|--------------------|-----|-------------------------------------------------------|
| model | ListModel | RW | Modèle de données |
| cellRenderer | ListCellRenderer | RW | Afficheur d'élément |
| selectionModel | ListSelectionModel | RW | Modèle de sélection |
| fixedCellHeight | int | RW | Hauteur de chaque élément (cellule) |
| fixedCellWidth | int | RW | Largeur de chaque élément (cellule) |
| prototypeCellValue | Object | RW | Cellule modèle pour déterminer les dimensions |
| selectedIndex | int | RW | Indice du 1 ^{er} élément sélectionné (ou -1) |
| selectedIndices | int[] | RW | Indices des éléments sélectionnés |
| selectedValue | Object | R | Elément sélectionné |
| selectedValues | Object[] | R | Eléments sélectionnés (tableau) |
| selectionMode | int | RW | Mode de sélection |
| visibleRowCount | int | RW | Nombre d'éléments affichés (si ds ScrollPane) |



JList : Méthodes

- En plus des méthodes permettant de gérer les propriétés du composant **JList**, il existe d'autres méthodes, par exemple :

| Méthodes | Description |
|-----------------------------|-----------------------------------------------------------------------------------------------------------|
| ensureIndexIsVisible | Assure que l'élément dont l'indice est passé en paramètre est visible à l'écran (scrolling si nécessaire) |
| clearSelection | Efface la sélection courante |

- Pour voir l'ensemble des propriétés et des méthodes disponibles, il faut consulter l'API de la classe **JList**.

Remarque : Dans les API de la plate-forme Java, les méthodes permettant de manipuler les propriétés de l'objet ne sont pas décrites à part. Elles font partie de l'ensemble des méthodes (qui sont listées par ordre alphabétique).





JList : Événements

- Si l'application doit être informée dès qu'une sélection a été effectuée dans une **JList**, il faut enregistrer un récepteur d'événement de type **ListSelectionListener**.
- L'interface **ListSelectionListener** définit une seule méthode :

```
public void valueChanged(ListSelectionEvent e);
```
- Les objets **ListSelectionEvent** disposent de trois méthodes spécifiques :

| | |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| getFirstIndex | Indice du premier élément dont la sélection a été modifiée |
| getLastIndex | Indice du dernier élément dont la sélection a été modifiée |
| getValueIsAdjusting | Retourne true si la sélection est en cours de changement (modifications multiples dans un certain intervalle de temps) |



JComboBox [1]

- Le composant **JComboBox** représente une **liste déroulante** dans laquelle l'utilisateur peut **sélectionner un seul élément**.
- Visuellement, le composant **JComboBox** est composé de trois sous-composants :
 - Un **champ de texte (JLabel ou JTextField)**
 - Un **bouton** permettant de dérouler la liste
 - Une **liste d'éléments (JList)** affichée dans une fenêtre pop-up



État initial



Liste déroulée



Liste déroulée + scrollbar



JComboBox [2]

- Le composant **JComboBox** peut être déclaré **éditable** (en invoquant la méthode **setEditable()**).
- Dans ce cas, le champ de texte n'est plus un composant **JLabel** mais un composant **JTextField** dans lequel l'utilisateur peut introduire une valeur qui ne figure pas dans la liste des éléments (c'est-à-dire une valeur qui n'est pas contenue dans le modèle de données).
- Les **éléments affichés** dans la liste déroulante (composant **JList**) sont **automatiquement insérés dans un panneau JScrollPane** qui affichera, si nécessaire, une barre de défilement pour parcourir la liste des éléments (rappel : pour les composants **JList** ce mécanisme n'est pas automatique).



Architecture de JComboBox

- Le composant **JComboBox** est composé de quatre éléments principaux :
 - Un **modèle de données (data model)** qui enregistre les éléments et qui doit implémenter l'interface **ComboBoxModel** qui est une sous-interface de **ListModel**. Par défaut, le composant **JComboBox** utilise la classe **DefaultComboBoxModel**.
 - Un **afficheur d'élément (cell renderer)** qui est chargé de la représentation visuelle de chaque élément de la liste (l'élément est également appelé **cellule**). Ce composant doit implémenter l'interface **ListCellRenderer** (comme pour le composant **JList**).
 - Un **éditeur** pour les valeurs introduites par l'utilisateur (si le composant est éditable). Par défaut **JComboBox** utilise un champ de type **JTextField** qui possède un éditeur par défaut pour la ligne de texte.
 - Un **gestionnaire clavier (keystroke manager)** pour réagir aux touches pressées par l'utilisateur (si le composant n'est pas éditable). Ce composant doit implémenter l'interface **KeySelectionManager** (interface interne de **JComboBox**).





Création d'un JComboBox [1]

- Différents constructeurs permettent de créer un composant **JComboBox** et de l'alimenter avec un ensemble d'éléments.
- Les éléments peuvent être transmis au constructeur sous forme de tableau d'objets (**Object[]**), d'une collection de type **Vector** ou d'un objet qui implémente l'interface **ComboBoxModel**.

```
String[] nomsJours = {"Lundi", "Mardi", "Mercredi",
                      "Jeudi", "Vendredi", "Samedi",
                      "Dimanche"};
JComboBox listeJours = new JComboBox(nomsJours);
```



Création d'un JComboBox [2]

- Par défaut la représentation des éléments d'un **JComboBox** est identique à celle utilisée par le composant **JList**. La classe **DefaultListCellRenderer** affiche la représentation textuelle des éléments (par invocation de la méthode **toString()**) en utilisant des composants de type **JLabel**.
- Contrairement aux éléments des composants **JList**, la liste des éléments d'un **JComboBox** peut être modifiée après l'alimentation initiale effectuée par le constructeur.

On utilisera pour cela les méthodes **addItem()**, **insertItemAt()**, **removeItem()**, **removeItemAt()** et **removeAllItems()** qui modifient le contenu du modèle de données (pour autant que ce modèle implémente la sous-interface **MutableComboBoxModel**).



JComboBox : Propriétés

- Le composant **JComboBox** dispose d'un grand nombre de propriétés.
- Les principales sont les suivantes :

| Propriété | Type | R/W | Description |
|----------------------------|---------------------|-----|-------------------------------------------------------------------|
| model | ComboBoxModel | RW | Modèle de données |
| renderer | ListCellRenderer | RW | Afficheur d'élément |
| editor | ComboBoxEditor | RW | Editeur, si composant éditable |
| keySelectionManager | KeySelectionManager | RW | Gestionnaire clavier |
| editable | boolean | RW | Champ de texte éditable |
| action | Action | RW | Action exécutée lors de la sélection |
| maximumRowCount | int | RW | Nombre maximal d'éléments affichés dans la liste (par défaut : 8) |
| popupVisible | boolean | RW | Affichage de la liste déroulante |
| enabled | boolean | W | Actif / Inactif |
| itemCount | int | R | Nombre d'éléments dans la liste |
| selectedIndex | int | RW | Indice de l'élément sélectionné (ou -1) |
| selectedItem | Object | RW | Elément sélectionné |



JComboBox : Méthodes

- En plus des méthodes permettant de gérer les propriétés du composant **JComboBox** et celles permettant de manipuler le modèle de données, il existe d'autres méthodes, par exemple :

| Méthodes | Description |
|------------------|-------------------------------------------------|
| showPopup | Affiche (par programmation) la liste déroulante |
| hidePopup | Ferme (par programmation) la liste déroulante |

- Pour voir l'ensemble des propriétés et des méthodes disponibles, il faut consulter l'API de la classe **JComboBox**.

Remarque : Dans les API de la plate-forme Java, les méthodes permettant de manipuler les propriétés de l'objet ne sont pas décrites à part. Elles font partie de l'ensemble des méthodes (qui sont listées par ordre alphabétique).





JComboBox : Événements

- Si l'application doit être informée dès qu'une sélection a été effectuée dans un **JComboBox**, le plus simple est d'enregistrer un récepteur d'événement de type **ItemListener** (un récepteur d'événement de type **ActionListener** est également envisageable mais l'élément sélectionné doit alors être recherché au travers des méthodes `getSource()` de **ActionEvent** et `getSelectedItem()` de **JComboBox**).
- L'interface **ItemListener** définit une seule méthode :

```
public void itemStateChanged(ItemEvent e);
```
- Les objets **ItemEvent** disposent des méthodes suivantes :

| | |
|-----------------------------|-------------------------------------------------------------------------------------------------------|
| <code>getStateChange</code> | Retourne l'état de l'élément (<code>SELECTED</code> , <code>DESELECTED</code>) après l'événement |
| <code>getItem</code> | Retourne l'élément affecté par l'événement (l'élément sélectionné ou désélectionné) |



KeySelectionManager [1]

- La classe **JComboBox** comprend un **gestionnaire clavier** (*Keystroke Manager*) dont la fonction est de **sélectionner** un élément de la liste en fonction des touches qui sont pressées (événements clavier).
- Ce mécanisme ne fonctionne que si le **JComboBox** est **non-éditable** (s'il est éditable les caractères pressés au clavier s'affichent dans le champ de texte).
- Le **comportement par défaut** de gestionnaire clavier de **JComboBox** est de sélectionner l'élément dont le premier caractère correspond à celui de la touche qui a été pressée (par exemple, une pression sur la touche 'B' sélectionnera le premier élément de la liste dont le texte commence par la lettre 'B').
- Ce comportement par défaut peut être modifié en définissant un nouveau gestionnaire clavier (voir détails à la page suivante) et en l'associant au composant **JComboBox** à l'aide de la méthode `setKeySelectionManager()`



KeySelectionManager [2]

- Pour définir un nouveau **gestionnaire clavier**, il faut créer une classe qui implémente l'interface **KeySelectionManager** (qui est une interface interne de **JComboBox**).
- L'interface **KeySelectionManager** comprend une seule méthode :

```
public int selectionForKey(char aKey, ComboBoxModel m);
```

La méthode **reçoit en paramètre** le caractère qui a été pressé (`aKey`) ainsi que le modèle de données du **JComboBox** (`m`).

La **valeur rentrée** représente l'**indice de l'élément à sélectionner**, une valeur -1 indique qu'aucun élément ne doit être sélectionné.
- Une instance de cette classe peut ensuite être associée au composant **JComboBox** à l'aide de la méthode `setKeySelectionManager()`.



JSpinner [1]

- Le composant **JSpinner** permet à l'utilisateur de **sélectionner une information** parmi une **séquence ordonnée** de valeurs, à l'aide de deux boutons permettant de faire défiler les valeurs possibles.
- Fonctionnellement, le composant **JSpinner** est très proche du composant **JComboBox** mais sans liste déroulante (seule la valeur courante est visible: Ne masque donc pas d'autres composants de l'interface)
- Le composant **JSpinner** est un conteneur composé de trois sous-composants :
 - Un **champ** appelé **éditeur** dans lequel seront affichées ou éventuellement éditées les valeurs
 - Deux **boutons** permettant de faire défiler les valeurs possibles
- Selon le **Look & Feel** adopté, le composant **JSpinner** peut se présenter sous différentes formes :





JSpinner [2]

- Par défaut le champ d'édition est un **JFormattedTextField** (mais n'importe quel **JComponent** peut être utilisé).
- Le composant **JSpinner** ne devrait être utilisé que lorsque les valeurs possibles et leur séquence sont évidentes pour l'utilisateur.
- Le modèle du composant **JSpinner** doit implémenter la classe abstraite **AbstractSpinnerModel**.
- La librairie Swing offre trois modèles par défaut :
 - **SpinnerListModel**
 - **SpinnerNumberModel**
 - **SpinnerDateModel**



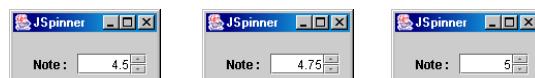
Exemple JSpinner

```
/*--- Créeation du modèle (numérique) pour le composant JSpinner
float init = 4.5f; // Valeur initiale
float min = 1.0f; // Valeur minimale
float max = 6.0f; // Valeur maximale
float step = 0.25f; // Incrément

SpinnerNumberModel snm = new SpinnerNumberModel(init,
                                                min,
                                                max,
                                                step);

/*--- Créeation du composant JSpinner
JSpinner note = new JSpinner(snm);

mainPanel.add(new JLabel("Note : ", JLabel.RIGHT));
mainPanel.add(note);
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI) 09 Visualisation et édition de textes

Jacques Bapst

jacques.bapst@hefr.ch



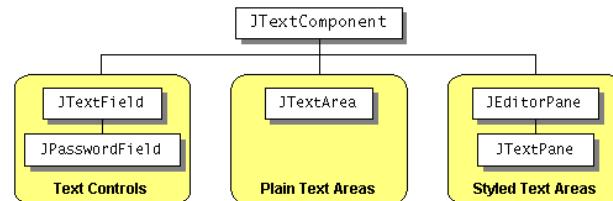
Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Visualisation et édition de textes

Visualisation et édition de textes [1]

- Plusieurs composants de la librairie *Swing* permettent d'afficher et d'éditer des textes.
- Tous ces composants sont dérivés d'une classe parente commune **JTextComponent** (`javax.swing.text`) et sont structurés selon l'arborescence suivante :



- La classe **JTextComponent** regroupe toutes les caractéristiques et comportements communs aux composants qui manipulent des textes (insertion, sélection, navigation, édition, gestions des marges, etc.).



Interface Homme-Machine 1 / Visualisation et édition de textes

Visualisation et édition de textes [2]

- Un grand nombre de classes et d'interfaces utilitaires sont associées aux composants qui visualisent des textes ([Document](#), [Caret](#), [EditorKit](#), [Keymap](#), [TextAction](#), [View](#), [Style](#), [StyleSheet](#), [AttributeSet](#), ...).
- Le composant **JTextField** (qui a déjà été étudié) permet à l'utilisateur de saisir une ligne de texte.
- Le composant **JPasswordField** est une spécialisation du composant **JTextField** et permet d'introduire des mots de passe ou autres textes secrets (les caractères saisis ne sont pas affichés dans le champ, mais ils peuvent être remplacés par un caractère fixe, configurable).

En plus des méthodes héritées, le composant **JPasswordField** comporte quelques méthodes spécifiques, notamment la méthode **setEchoChar()** qui permet de définir le caractère affiché à chaque frappe clavier ainsi que la méthode **getPassword()** qui permet de lire le mot de passe qui a été introduit.



Interface Homme-Machine 1 / Visualisation et édition de textes

JTextComponent : Propriétés

- Le composant **JTextComponent** rassemble un certain nombre de **propriétés communes** aux composants qui manipulent des textes :

| Propriété | Type | R/W | Description |
|--------------------------------|----------|-----|-----------------------------------------|
| <code>caretColor</code> | Color | RW | Couleur du curseur de texte (Caret) |
| <code>caretPosition</code> | int | RW | Position du curseur dans le texte |
| <code>disabledTextColor</code> | Color | RW | Couleur du texte inactif (disabled) |
| <code>document</code> | Document | RW | Modèle de données |
| <code>editable</code> | boolean | RW | Texte éditable (true/false) |
| <code>focusAccelerator</code> | char | RW | Accélérateur clavier (Alt+char ⇌ focus) |
| <code>margin</code> | Insets | RW | Marge autour du texte |
| <code>selectedText</code> | String | R | Texte sélectionné |
| <code>selectedTextColor</code> | Color | RW | Couleur du texte sélectionné |
| <code>selectionColor</code> | Color | RW | Couleur d'arrière-plan de la sélection |
| <code>selectionStart</code> | int | RW | Position du début de la sélection |
| <code>selectionEnd</code> | int | RW | Position de la fin de la sélection |
| <code>text</code> | String | RW | Modèle de données sous forme String |





JTextComponent : Méthodes

- Le composant **JTextComponent** rassemble un certain nombre de **méthodes communes** aux composants qui manipulent des textes :

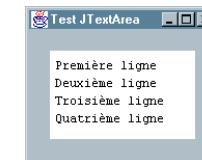
| Méthodes | Description |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <code>copy</code> | Copie la sélection dans le <i>Clipboard</i> (presse-papier) |
| <code>cut</code> | Supprime et enregistre la sélection dans le <i>Clipboard</i> |
| <code>paste</code> | Copie le contenu du <i>Clipboard</i> dans le document (remplace la sélection s'il y en a une) |
| <code>select</code> | Sélectionne une partie du texte |
| <code>selectAll</code> | Sélectionne tout le texte |
| <code>moveCaretPosition</code> | Sélectionne le texte à partir de la position précédente du curseur (<i>Caret</i>) jusqu'à sa nouvelle position (après déplacement) |
| <code>replaceSelection</code> | Remplace le texte sélectionné par un autre texte |
| <code>read</code> | Lit le texte à partir d'un objet de type <i>Reader</i> (flux de caractères) |
| <code>write</code> | Ecrit le contenu du document dans un objet de type <i>Writer</i> (flux de caractères) |



Création de JTextArea [1]

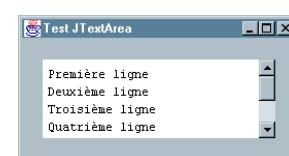
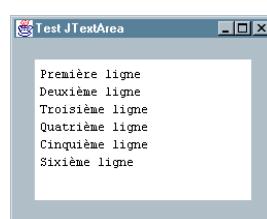
- Différents constructeurs sont disponibles pour créer des composants **JTextArea**.
- Selon les variantes, on peut définir le **nombre de colonnes**, le **nombre de lignes** ainsi que le **texte initial** (sous forme d'un **String** ou d'un objet de type **Document**).

```
JTextArea zoneTexte = new JTextArea("Première ligne\n" +
"Deuxième ligne\n" +
"Troisième ligne\n" +
"Quatrième ligne", 5, 20);
```



JTextArea

- Le composant **JTextArea** représente un champ de texte multiligne, éditable, mais non formaté (les attributs du texte sont identiques pour tous les caractères).
- Par défaut, le texte affiché par un composant **JTextArea** n'est pas scrollable. Il suffit cependant d'insérer le composant **JTextArea** dans un conteneur **JScrollPane** pour pouvoir disposer de barres de défilement (qui s'affichent lorsque c'est nécessaire).



Création de JTextArea [2]

- Si le nombre de lignes et de colonnes n'est pas spécifié dans le constructeur de **JTextArea**, la dimension de la zone de texte sera déterminée par le contenu et le type de gestionnaire de disposition utilisé.

Conseil : Il est prudent d'insérer systématiquement les composants **JTextArea** dans des conteneurs **JScrollPane** de telle sorte que des barres de défilement apparaissent lorsque cela s'avère nécessaire (suite au redimensionnement de la fenêtre englobante ou lorsque le texte déborde de la zone prévue).

```
JTextArea zoneTexte = new JTextArea("Première ligne\n" +
"Deuxième ligne\n" +
"Troisième ligne\n" +
"Quatrième ligne", 5, 20);

JScrollPane zoneTexteScrl = new JScrollPane(zoneTexte);

unConteneur.add(zoneTexteScrl);
```





JTextArea : Propriétés

- En plus des propriétés héritées, le composant **JTextArea** possède un certain nombre de **propriétés spécifiques** dont les principales sont :

| Propriété | Type | R/W | Description |
|---------------|---------|-----|----------------------------------------------------------------------------------------------------------------------------------------------|
| rows | int | RW | Nombre de lignes à afficher (taille préférée) |
| columns | int | RW | Nombre de colonnes à afficher (taille préférée) |
| lineCount | int | R | Nombre de lignes contenues dans le document |
| lineWrap | boolean | RW | Insère un retour à la ligne si le contenu déborde de la zone d'affichage |
| wrapStyleWord | boolean | RW | Si true, insère un retour à la ligne entre les mots (et non pas entre les caractères) A un effet seulement si <code>lineWrap==true</code> |
| font | Font | W | Police de caractères utilisée pour l'affichage du texte |
| tabSize | int | RW | Position des tabulateurs (nombre de colonnes). Par défaut : 8 |
| text | String | RW | Texte affiché (Cette propriété importante est héritée de la classe <code>JTextComponent</code>) |

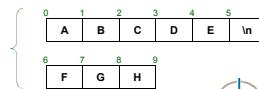


JTextArea : Méthodes

- En plus des méthodes héritées, le composant **JTextArea** possède un certain nombre de **méthodes spécifiques**, notamment :

| Méthodes | Description |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| append | Ajoute du texte à la fin du document |
| insert | Insère du texte à une position donnée dans le document (la position est donnée par le nombre de caractères depuis le début du document; la position 0 est située avant le premier caractère) |
| replaceRange | Remplace une zone de texte par un nouveau texte |
| getLineOffset | Retourne le numéro de ligne correspondant à une position donnée (la position est donnée par le nombre de caractères depuis le début du document ; la position 0 est située avant le premier caractère) |
| getLineStartOffset | Retourne la position avant le premier caractère d'une ligne donnée par son numéro |
| getLineEndOffset | Retourne la position après le dernier caractère d'une ligne donnée par son numéro (correspond à la position du premier caractère de la ligne suivante car le retour à la ligne compte) |

Position des caractères



Si `lineWrap == true`, les lignes affichées ne correspondent pas forcément aux lignes internes



JEditorPane

- Le composant **JEditorPane** est une spécialisation de la classe **JTextComponent** permettant d'**afficher et d'éditer du texte formaté** grâce à un élément interne appelé **EditorKit**.
- Par défaut, le composant peut interpréter trois **types de contenus** :
 - du **texte non formaté** (`contentType : "text/plain"`)
 - du **texte au format HTML** (`contentType : "text/html"`)
 - du **texte au format RTF** (`contentType : "text/rtf"`)
- Ces différents types de contenus sont respectivement traités par les classes **DefaultEditorKit**, **HTMLEditorKit** et **RTFEditorKit** qui représentent différentes implémentations de la classe abstraite **EditorKit**.
- Si l'on souhaite que le composant **JEditorPane** traite d'autres formats de textes il est possible d'écrire une implémentation spécifique de la classe abstraite **EditorKit** (ou de l'une de ses sous-classes par exemple **StyledEditorKit**) et de l'enregistrer avec la méthode `setEditorKit()`.



Utilisation de JEditorPane

- Le composant **JEditorPane** est fréquemment utilisé pour afficher un texte qui a été préalablement créé avec un utilitaire spécifique (un traitement de texte par exemple) et enregistré dans un fichier au format HTML (ou éventuellement RTF).
- Cette fonctionnalité peut être très utile dans une application pour l'affichage de l'aide (*Help*), d'un mode d'emploi ou pour l'affichage de toutes autres informations structurées (images et textes formatés) enregistrées sous l'un de ces deux formats (HTML et RTF).

Attention : Dans la version actuelle de la machine virtuelle, le support HTML pour le composant **JEditorPane** est limité à la version HTML 3.2 avec quelques extensions HTML 4.0.
Les CSS (Cascading Style Sheet) sont partiellement supportés.
Des améliorations sont attendues avec les prochaines versions de la plate-forme Java (JDK 1.6 et suivants).





Création de JEditorPane [1]

- Le composant **JEditorPane** dispose de différents constructeurs qui permettent de créer la zone d'édition avec éventuellement le chargement d'un contenu initial.

```
JEditorPane edPanel = new JEditorPane();
//-----
String type    = "text/html";
String content = "<H1>Ceci</H1>" +
                 "<P>est un <B>Texte</B> HTML</P>";
JEditorPane edPanel2 = new JEditorPane(type, content);
//-----
String fileURL = "File:/D:/Doc/Help.html";
JEditorPane edPanel3 = new JEditorPane(fileURL);
//-----
URL pageWeb = new URL("http://www.jugs.ch");
JEditorPane edPanel4 = new JEditorPane(pageWeb);
```

Remarque : Dans le code ci-dessus, certaines instructions peuvent générer des exceptions.



Création de JEditorPane [2]

- Si le chargement du document à afficher dans le composant **JEditorPane** est effectué en indiquant une adresse URL (que ce soit dans le constructeur ou par invocation de la méthode `setPage()`), le type du contenu (`contentType`) pourra généralement être déterminé automatiquement et l'instance adéquate du composant **EditorKit** sera utilisée.



JEditorPane : Propriétés et Méthodes

- En plus des propriétés héritées, le composant **JEditorPane** possède un certain nombre de **propriétés spécifiques** dont les principales sont :

| Propriété | Type | R/W | Description |
|--------------------------|-----------|-----|--------------------------------------------------------------------------|
| <code>contentType</code> | String | RW | Type de contenu du document (format MIME-Type , ex: "text/html") |
| <code>editorKit</code> | EditorKit | RW | Interpréteur de contenu du document (formatage selon le type de contenu) |
| <code>page</code> | URL | RW | URL du contenu chargé (ou à charger) dans le document |
| <code>page</code> | String | W | URL du contenu à charger dans le document (chaîne de caractères) |
| <code>text</code> | String | RW | Contenu du document sous forme textuelle (chaîne de caractères) |

- Le composant **JEditorPane** ne dispose que de quelques méthodes spécifiques dont la plupart redéfinissent ou complètent des méthodes héritées.



JEditorPane : Événements [1]

- En plus des événements liés à la gestion de textes, le composant **JEditorPane** peut générer un nouveau type d'événement appelé **HyperlinkEvent** qui permet de gérer les liens de type hypertexte dans les documents.
- Cet événement est généré lorsque l'utilisateur survole ou clique sur un hyper-lien dans le document affiché.
- L'événement **HyperlinkEvent** n'est généré que si le composant **JEditorPane** est en mode **non-éditable** (dans ce mode, **JEditorPane** peut fonctionner comme un navigateur).
- L'interface **HyperlinkListener** ne définit qu'une seule méthode :


```
public void hyperlinkUpdate(HyperlinkEvent ev);
```
- Cette méthode est appelée :
 - Lorsque l'utilisateur **entre dans une zone** de lien hypertexte
 - Lorsque l'utilisateur **sourt d'une zone** de lien hypertexte
 - Lorsque l'utilisateur **clique sur un lien** hypertexte





JEditorPane : Événements [2]

- Les objets de type **HyperlinkEvent** possèdent trois méthodes spécifiques qui permettent de consulter certaines propriétés associées à l'événement :

| | |
|-------------------------|----------------------------------------------------------------------------------------------------------------|
| getDescription() | Retourne une chaîne de caractères (<i>String</i>) correspondant à la description du lien (le texte de l'URL) |
| getEventType() | Retourne une constante (définie dans la classe interne EventType) qui précise le type d'événement |
| ENTERED | <i>Le curseur de la souris est entré dans la zone d'un lien hyper-texte</i> |
| EXITED | <i>Le curseur de la souris est sorti de la zone d'un lien hyper-texte</i> |
| ACTIVATED | <i>L'utilisateur a cliqué sur le lien hyper-texte</i> |
| getURL() | Retourne un objet (de type URL) correspondant à l'URL du lien hypertexte |



JTextPane

- Le composant **JTextPane** est une spécialisation du composant **JEditorPane**.
- Il est dédié à l'affichage et à l'édition de **documents complexes** (comprenant des textes formatés et des images) en offrant tous les mécanismes permettant de créer un véritable traitement de texte.
- Un grand nombre de classes et d'interfaces sont associées au composant **JTextPane**.
- Le modèle de données est basé sur l'interface **StyledDocument** qui est une spécialisation de l'interface **Document** permettant de définir des "feuilles de styles" (représentées par des instances de l'interface **Style**). Ces styles déterminent des ensembles d'attributs qui sont représentés par des instances de l'interface **AttributeSet**.
- L'étude détaillée du composant **JTextPane** dépasse le cadre de ce cours.



Exemple HyperlinkListener

```
----- (Instructions package et import non mentionnées) -----
public class FollowHyperlink implements HyperlinkListener {
    JEditorPane edPane;
    public FollowHyperlink(JEditorPane edPane) {
        this.edPane = edPane;
    }
    public void hyperlinkUpdate (HyperlinkEvent hev) {
        if (hev.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            Document oldPage = edPane.getDocument(); //--- Mémorisation ancienne page
            try {
                edPane.setPage(hev.getURL()); //--- Affiche la nouvelle page
            }
            catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Lien incorrect");
                edPane.setDocument(oldPage); //--- Retour à la page précédente
            }
        }
    }
}
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI) 10 Tables et Arbres

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Tables et Arbres

Notion de table

- Les **tables** constituent un des formats les plus courants pour représenter (afficher) des données à caractère bidimensionnel.
- Dans une table, les informations sont présentées dans une **grille** organisée en **lignes** et en **colonnes**. Chaque donnée est affichée dans une **cellule**.
- Les **colonnes** contiennent généralement un ensemble de données de **même type** et peuvent posséder un **en-tête** (*header*) qui décrit le genre d'informations que la colonne représente.

| TableDemo | | | | |
|------------|-----------|------------------|------------|-------------------------------------|
| First Name | Last Name | Sport | # of Years | Vegetarian |
| Mary | Campione | Snowboarding | 5 | <input type="checkbox"/> |
| Alison | Huml | Rowing | 3 | <input checked="" type="checkbox"/> |
| Kathy | Walrath | Chasing toddl... | 2 | <input type="checkbox"/> |
| Mark | Andrews | Speed reading | 20 | <input checked="" type="checkbox"/> |



Interface Homme-Machine 1 / Tables et Arbres

JTable [1]

- Le composant **JTable** est un composant complexe permettant de représenter des données sous forme de table.
- Pas loin de vingt classes et interfaces sont liées au composant **JTable**. La plupart de ces classes se trouvent dans le package `javax.swing.table`
- L'interface **TableModel** définit les spécifications minimales qu'un modèle doit impérativement implémenter pour que les données puissent être représentées dans un composant **JTable**.
- La classe abstraite **AbstractTableModel** implémente la plupart des méthodes de l'interface **TableModel** mais exige de définir les méthodes d'accès aux données à représenter.
- La classe concrète **DefaultTableModel** étend la classe abstraite **AbstractTableModel** en fournissant des méthodes d'accès à des données enregistrées sous forme de vecteur de vecteurs.



Interface Homme-Machine 1 / Tables et Arbres

JTable [2]

- Le composant **JTable** possède plusieurs constructeurs dans lesquels les données peuvent être transmises sous différentes formes (`Object[][]`, `Vector` ou `TableModel`).
- L'en-tête des colonnes (qui définit leur nom) peut également être spécifiée sous forme de tableau ou d'un vecteur (`Object[]`, `Vector`).
- Il est ainsi assez facile de créer une table utilisant le modèle par défaut (`DefaultTableModel`) à partir de données enregistrées dans un tableau ou un vecteur (voir page suivante).
- Dans une table ainsi créée, les cellules sont éditable (sans mise à jour du modèle) et il est possible de sélectionner des lignes (multi-sélection).
- Il est judicieux d'insérer systématiquement le composant **JTable** dans un conteneur **JScrollPane** pour offrir à l'utilisateur, si nécessaire, la possibilité de faire défiler les données.

Remarques : Les en-têtes de colonnes ne seront affichés que si la table est insérée dans un **JScrollPane**.

La barre de défilement horizontale ne sera affichée que si la propriété `autoResizeMode=AUTO_RESIZE_OFF`.





Exemple de JTable

```
public class SimpleTable extends JFrame {
    String[][] tData = {{"Bach", "Jean-Sébastien", "34"}, // Données
                       {"Mozart", "Amadeus", "26"},
                       {"Chopin", "Frédéric", "37"},
                       {"Robert", "Ted", "58"},
                       {"Purcell", "Henry", "19"}};

    String[] cHeader = {"Nom", "Prénom", "Age"}; // En-têtes

    public SimpleTable() {
        super("SimpleTable");
        setSize(300, 200);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTable table = new JTable(tData, cHeader); // Création de la table
        JScrollPane sTable = new JScrollPane(table);
        getContentPane().add(sTable, BorderLayout.CENTER);
    }
}
```



JTable : Propriétés [1]

- Quelques propriétés du composant **JTable** :

| Propriété | Type | R/W | Description |
|------------------------|-----------------|-----|----------------------------------------------------------------|
| showHorizontalLines | boolean | RW | Affichage des lignes horizontales |
| showVerticalLines | boolean | RW | Affichage des lignes verticales |
| showGrid | boolean | W | Affichage de la grille (lignes Horiz+Vert) |
| gridColor | Color | RW | Couleur des lignes de la grille |
| intercellSpacing | Dimension | RW | Espace supplém. autour des cellules |
| selectionForeground | Color | RW | Couleur de la sélection (1er plan) |
| selectionBackground | Color | RW | Couleur de la sélection (arrière-plan) |
| rowSelectionAllowed | boolean | RW | Sélection des lignes activée |
| columnSelectionAllowed | boolean | RW | Sélection des colonnes activée |
| cellSelectionEnabled | boolean | RW | Sélection des cellules activée |
| autoResizeMode | int | RW | Comportement au redimensionnement des colonnes ou du conteneur |
| cellEditor | TableCellEditor | RW | Éditeur de cellule |
| columnCount | int | R | Nombre de colonnes |



JTable : Propriétés [2]

| Propriété | Type | R/W | Description |
|---------------------|--------------------|-----|----------------------------------|
| columnModel | TableColumnModel | RW | Modèle associé aux colonnes |
| model | TableModel | RW | Modèle associé à la table |
| rowCount | int | R | Nombre de lignes |
| rowHeight | int | RW | Hauteur des lignes |
| rowMargin | int | RW | Marge entre les lignes |
| SelectionMode | int | W | Mode de sélection |
| selectionModel | ListSelectionModel | RW | Modèle associé à la sélection |
| selectedRow | int | R | Ligne sélectionnée |
| selectedRows | int[] | R | Lignes sélectionnées |
| selectedRowCount | int | R | Nombre de lignes sélectionnées |
| selectedColumn | int | R | Colonne sélectionnée |
| selectedColumns | int[] | R | Colonnes sélectionnées |
| selectedColumnCount | int | R | Nombre de colonnes sélectionnées |
| tableHeader | JTableHeader | RW | En-tête des colonnes de la table |



Création d'un JTableModel

- Le composant **JTable** peut être alimenté en créant un modèle de données personnalité qui respecte l'interface **TableModel**.
- On peut le faire en étendant la classe **AbstractTableModel** qui nécessite de redéfinir (au minimum) les trois méthodes suivantes :
 - `int getRowCount()`
 - `int getColumnCount()`
 - `Object getValueAt(int row, int col)`
- Il peut être intéressant, dans certains cas, de redéfinir également d'autres méthodes, par exemple :
 - Pour indiquer si la cellule est éditable
 - `boolean isCellEditable(int rowIndex, int colIndex)`
 - Pour modifier une valeur dans le modèle
 - `void setValueAt(Object val, int rowIndex, int colIndex)`
 - Pour définir le nom des colonnes
 - `String getColumnName(int colIndex)`
 - Pour définir la classe (le type) des colonnes
 - `Class getColumnClass(int colIndex)`





Exemple de modèle personnalisé

```
public class TableModelDemo extends JFrame {
    TableModel dataModel = new AbstractTableModel() { // Modèle de données
        public int getColumnCount() { return 10; }
        public int getRowCount() { return 50; }
        public Object getValueAt(int row, int col) {
            return new Integer(row*col);
        }
    };
    public TableModelDemo() {
        super("TableModelDemo");
        setSize(400, 150);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JTable table = new JTable(dataModel); // Création de la table
        JScrollPane sTable = new JScrollPane(table);
        getContentPane().add(sTable, BorderLayout.CENTER);
    }
}
```



TableColumn

- Dans le composant **JTable**, le sous-composant de base est constitué par la **colonne** (et non pas la cellule pour laquelle il n'existe pas de structure de données spécifique).
- Chaque colonne d'une table est représentée (à l'intérieur de la table) par une instance de la classe **TableColumn**.
- Comme toutes les données d'une colonne sont du même type, la classe **TableColumn** est chargée de fournir tous les éléments pour représenter les informations (**Renderer**) et pour les éditer (**Editor**), si nécessaire.
- L'interface **TableColumnModel** permet de définir les caractéristiques d'un **ensemble de colonnes** (y compris un ensemble d'une seule colonne).
- Par défaut, les informations de chacune des cellules sont représentées au moyen de composants de type **JLabel** mais n'importe quel autre composant peut être utilisé à la place en définissant un afficheur de cellule (qui implémente l'interface **TableCellRenderer**).



TableColumn : Propriétés

- Quelques propriétés de la classe **TableColumn** :

| Propriété | Type | R/W | Description |
|----------------|-------------------|-----|----------------------------------------------------------------------|
| cellEditor | TableCellEditor | RW | Editeur de cellule |
| cellRenderer | TableCellRenderer | RW | Afficheur de cellule |
| headerRenderer | TableCellRenderer | RW | Afficheur d'en-tête |
| headerValue | Object | RW | Valeur de l'en-tête |
| identifier | Object | RW | Identificateur de la colonne (par défaut valeur de l'en-tête) |
| minWidth | int | RW | Largeur minimale de la colonne |
| maxWidth | int | RW | Largeur maximale de la colonne |
| width | int | R | Largeur actuelle de la colonne |
| preferredWidth | int | RW | Largeur actuelle de la colonne |
| resizable | boolean | RW | Indique si la largeur de la colonne est modifiable par l'utilisateur |



TableColumnModel

- En plus d'un modèle pour la table elle-même, le composant **JTable** utilise un modèle pour les colonnes qui contient des informations spécifiques aux colonnes de la table.
 - L'interface **TableColumnModel**, définit les caractéristiques (propriétés et méthodes) d'un **ensemble de colonnes**.
 - Par défaut, le composant **JTable** utilise une instance de la classe **DefaultTableColumnModel** comme modèle pour les colonnes.
 - Cette classe est utilisée si l'on ne spécifie pas de modèle différent dans le constructeur du composant **JTable**.
 - La classe **DefaultTableColumnModel** constitue également un bon point de départ si l'on souhaite créer son propre modèle pour les colonnes.
- Il suffit, dans ce cas, de ne redéfinir que les méthodes que l'on souhaite modifier et garder le comportement par défaut des autres.





TableColumnModel : Propriétés

- Quelques propriétés de la classe **TableColumnModel** :

| Propriété | Type | R/W | Description |
|--------------------------------------|--------------------|-----|-------------------------------------------------------------|
| <code>column(index)</code> | TableColumn | R | Colonne selon index |
| <code>columns</code> | Enumeration | R | Liste de toutes les colonnes |
| <code>columnMargin</code> | int | RW | Marge entre les colonnes |
| <code>columnCount</code> | int | R | Nombre de colonnes du modèle |
| <code>columnsSelectionAllowed</code> | boolean | RW | Sélection des colonnes activée |
| <code>selectedColumns</code> | int[] | R | Colonnes sélectionnées |
| <code>selectedColumnCount</code> | int | R | Nombre de colonnes sélectionnées |
| <code>totalColumnWidth</code> | int | R | Largeur totale (en pixels) de toutes les colonnes du modèle |
| <code>selectionModel</code> | ListSelectionModel | RW | Modèle de sélection |

EIA-FR / Jacques Bapst



IHM1_10 13



Utilisation du TableColumnModel

- Pour modifier les propriétés du modèle associé aux colonnes d'une table, il faut préalablement obtenir une référence à ce modèle en utilisant la méthode :

`getColumnModel()`

- On peut ensuite également accéder aux colonnes individuelles (instances de **TableColumn**) et consulter ou modifier leurs propriétés.
- Exemple :

```

    ...
    JTable table = new JTable(dataModel);           // Création de la table
    table.getColumnModel().setColumnMargin(5);       // 5 pixels entre les colonnes
    TableColumn col2= table.getColumnModel().column(2);
    col2.setResizable(false);                      // Col 2 non redimensionnable
    col2.setPreferredWidth(120);                   // Col 2 de taille fixe (120 pixels)

    JScrollPane sTable = new JScrollPane(table);
    getContentPane().add(sTable, BorderLayout.CENTER);
    ...
  
```

EIA-FR / Jacques Bapst



IHM1_10 14



Afficheur de cellules [1]

- Le contenu de chaque cellule d'une table est affiché par un sous-composant appelé **afficheur de cellule (Cell Renderer)**.
- Par défaut le composant **JLabel** est utilisé pour afficher le contenu des cellules (par invocation de la méthode `toString()` sur l'objet à représenter).
- La classe **JTable** dispose également d'autres afficheurs de cellules qui sont utilisés si la méthode `getColumnClass` de **TableModel** retourne un des types suivants¹⁾ :

| Classe | Composant (afficheur) | Description |
|------------------|-----------------------|----------------------------------------|
| Boolean | JCheckBox | Centré |
| Date | JLabel | Aligné à droite (utilise DateFormat) |
| ImageIcon | JLabel | Centré |
| Number | JLabel | Aligné à droite (utilise NumberFormat) |
| Object | JLabel | Aligné à gauche |

¹⁾ Par défaut, la méthode `getColumnClass()` retourne `Object.class` pour toutes les colonnes.

EIA-FR / Jacques Bapst



IHM1_10 15



Afficheur de cellules [2]

- Si l'on souhaite (par exemple pour certaines colonnes), écrire son propre afficheur de cellules, il faut créer un objet qui implémente l'interface **TableCellRenderer**.
- Cette interface ne comporte qu'une seule méthode qui initialise et retourne un composant (de la classe **Component**) :

```

Component getTableCellRendererComponent(JTable table,
                                       Object value,
                                       boolean isSelected,
                                       boolean hasFocus,
                                       int row,
                                       int column );
  
```

- Cette méthode sera invoquée par la table pour afficher le contenu des cellules qui se trouvent dans des colonnes pour lesquelles cet afficheur a été enregistré (voir page suivante).
- Par défaut, un objet de la classe **DefaultTableCellRenderer** est utilisé pour afficher les cellules de toutes les colonnes.

EIA-FR / Jacques Bapst



IHM1_10 16



Afficheur de cellules [3]

- Un afficheur de cellule peut être enregistré pour **une ou plusieurs colonnes individuelles** en utilisant la méthode `setCellRenderer()` de chacune des colonnes concernées :

```
table.getColumnModel().getColumn(3).setCellRenderer(monAfficheur);
table.getColumnModel().getColumn(7).setCellRenderer(monAfficheur);
```

- Il est également possible d'enregistrer un afficheur de cellules pour **toutes les colonnes d'un certain type** en invoquant la méthode `setDefaultRenderer()` de la table.

- Pour remplacer l'afficheur par défaut :

```
table.setDefaultRenderer(Object.class, monAfficheurGeneral);
```

- Pour enregistrer un afficheur sur toutes les colonnes de type *Boolean* :

```
table.setDefaultRenderer(Boolean.class, monAfficheurBool);
```

- Pour enregistrer un afficheur sur toutes les colonnes de type *Volume* :

```
table.setDefaultRenderer(Volume.class, monAfficheurVol);
```



Éditeur de cellules [1]

- Si une cellule est éditable (propriété du modèle de la table), un **éditeur de cellule** est invoqué lorsque l'utilisateur clique sur la cellule.
- L'éditeur de cellule par défaut utilise le composant `JTextField` pour permettre à l'utilisateur d'édition le contenu de la cellule.
- Si l'on souhaite écrire son propre éditeur de cellule, il faut créer un objet qui implémente l'interface `TableCellEditor` (qui étend la super-interface `CellEditor`).
- Cette interface ne comporte qu'une seule méthode qui initialise et retourne un composant (de la classe `Component`) qui sera utilisé comme éditeur de cellule :

```
Component getTableCellEditorComponent(JTable table,
                                     Object value,
                                     boolean isSelected,
                                     int row,
                                     int column );
```



Éditeur de cellules [2]

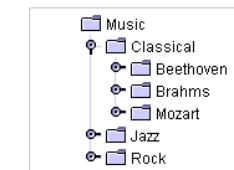
- La classe `DefaultCellEditor` offre une implémentation par défaut de l'interface `TableCellEditor`.
- Les trois constructeurs de cette classe permettent de créer des éditeurs de cellule utilisant les trois composants suivants :
 - `JTextField` pour l'édition de champs de texte
 - `JComboBox` pour offrir des listes à choix
 - `JCheckBox` pour des valeurs booléennes
- L'enregistrement d'un éditeur de cellule s'effectue en passant l'objet éditeur à la méthode `setCellEditor()` de la colonne dont les cellules doivent être gérées par cet éditeur :

```
JTable table = new JTable(...);
...
String[] elements = {"un", "deux", "trois"};
JComboBox listeChoix = new JComboBox(elements);
TableCellEditor editor = new DefaultCellEditor(listeChoix);
...
table.getColumnModel().getColumn(2).setCellEditor(editor);
```



Notion d'arbre [1]

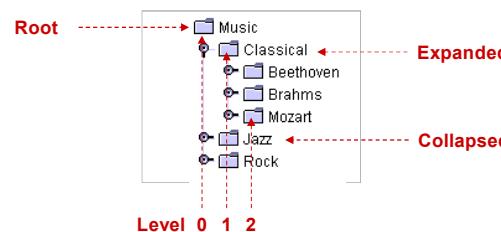
- Le composant `JTree` est un composant complexe permettant de représenter sous forme d'**arbre** des données avec une **organisation hiérarchique** (structure arborescente).
- Les éléments de l'arbre sont appelés des **nœuds (nodes)**.
- Le nœud au sommet de l'arbre est appelé la **racine (root)**
Rappel : les arbres informatiques poussent à l'envers !
- Chaque nœud peut posséder plusieurs **branches** qui lient ce nœud à ses **enfants (children)** appelés également les descendants.
- Les nœuds qui ne possèdent pas de branches (pas d'enfants) sont appelés les **feuilles** de l'arbre (**leaf nodes**).





Notion d'arbre [2]

- Le niveau (**level**) d'un nœud caractérise sa distance (exprimée en nombre d'ancêtres) par rapport à la racine de l'arbre. La racine est au niveau zéro.
- La **hauteur (height)** ou la **profondeur (depth)** d'un arbre est définie par le niveau maximal de tous ses nœuds.
- Visuellement chaque nœud de l'arbre peut être représenté **ouvert (expanded)** ou **fermé (collapsed)** selon que ses enfants sont visibles.



EIA-FR / Jacques Bapst



IHM1_10 21



JTree [1]

- Un grand nombre de classes et d'interfaces sont liées au composant **JTree**. La plupart de ces classes se trouvent dans le package `javax.swing.tree`
- L'interface **TreeModel** définit les spécifications minimales qu'un modèle doit impérativement implémenter pour que les données puissent être représentées dans un composant de type **JTree**.
- La classe **DefaultTreeModel** offre une implémentation par défaut de l'interface **TreeModel** dans laquelle les nœuds doivent être représentés par des objets qui implémentent l'interface **TreeNode** (ou la sous-interface **MutableTreeNode**).
- L'interface **TreeNode** définit une des spécifications possibles qu'un objet doit implémenter pour pouvoir être un nœud dans un arbre **JTree**. Cette interface est utilisée notamment en relation avec la classe **DefaultTreeModel** qui permet de créer un modèle de données pour le composant **JTree**.

EIA-FR / Jacques Bapst



IHM1_10 22



JTree [2]

- L'interface **MutableTreeNode** (une sous-interface de **TreeNode**) définit les spécifications qu'un objet doit implémenter pour pouvoir être un nœud modifiable (auquel on peut ajouter et supprimer des enfants, modifier sa valeur, etc.) dans un arbre **JTree**.
- L'interface **MutableTreeNode** est utilisée notamment par la classe **DefaultTreeModel**.
- La classe **DefaultMutableTreeNode** implémente l'interface **MutableTreeNode** et offre les fonctionnalités de base pour créer des nœuds qui serviront à construire un arbre **JTree**.

EIA-FR / Jacques Bapst



IHM1_10 23



JTree [3]

- Le composant **JTree** possède plusieurs constructeurs dans lesquels les données (les nœuds) peuvent être transmises sous différentes formes (**Object[]**, **Vector**, **Hashtable** OU **TreeModel**).
- Si l'on utilise un tableau d'objet, un vecteur ou une table, il est possible de créer un arbre à plusieurs niveaux si les éléments de ces structures de données sont eux-mêmes des structures composites.
- Avec ces structures de données, le nom des nœuds de la table correspondra à la valeur renournée par la méthode **toString()** invoquée sur chacun des nœuds (pour les tableaux d'objets, la méthode **toString()** ne peut pas être redéfinie et la chaîne de caractères renournée par défaut, par ex : "`Ljava.lang.Object;@2d7a10`", n'a pas grand intérêt).
- Il est donc préférable de créer une sous-classe de **Vector** ou de **Hashtable** et de redéfinir la méthode **toString()** (voir page suivante).
- Il est judicieux d'insérer systématiquement le composant **JTree** dans un conteneur **JScrollPane** pour offrir à l'utilisateur, si nécessaire, la possibilité de faire défiler les nœuds de l'arbre.

EIA-FR / Jacques Bapst



IHM1_10 24



JTree à partir de Vector [1]

```
import java.util.Vector;

public class NamedVector extends Vector {
    private String name;
    public NamedVector(String name) {
        this.name = name;
    }
    public NamedVector(String name, Object elements[]) {
        this.name = name;
        for (int i=0; i<elements.length; i++) {
            add(elements[i]);
        }
    }
    public String toString() {
        return name;
    }
}
```



JTree à partir de Vector [2]

```
public class VectorTree extends JFrame {
    //--- Création des nœuds de l'arbre
    Vector v11 = new NamedVector("VW", new String[]{"Passat", "Golf"});
    Vector v12 = new NamedVector("Ford",
        new String[]{"Galaxy", "Mondeo", "Focus"});
    Vector v13 = new NamedVector("Propriétés");

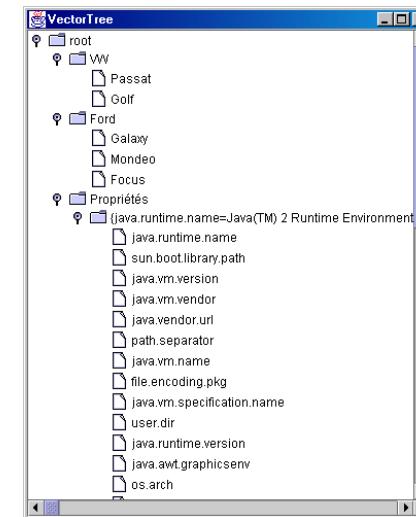
    public VectorTree() {
        super("VectorTree");
        setSize(400, 500);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        v13.add(System.getProperties()); // Ajout des propriétés du système
        Object[] tRoot = {v11, v12, v13};
        Vector vRoot = new NamedVector("Root", tRoot);

        JTree tree = new JTree(vRoot); // Création de l'arbre
        tree.setRootVisible(true); // Rend visible le nœud racine
        JScrollPane sTree = new JScrollPane(tree);
        getContentPane().add(sTree, BorderLayout.CENTER);
    }
}
```



JTree à partir de Vector [3]



JTree : Propriétés [1]

- Quelques **propriétés** du composant **JTree** :

| Propriété | Type | R/W | Description |
|-----------------|--------------------|-----|-------------------------------------------------------------------------------|
| cellEditor | TreeCellEditor | RW | Éditeur de cellules (nœuds) |
| cellRenderer | TreeCellRenderer | RW | Afficheur de cellules (nœuds) |
| editable | boolean | RW | Indique si l'arbre est éditable |
| model | TreeModel | RW | Modèle de donnée associé à l'arbre |
| rootVisible | boolean | RW | Affichage du nœud racine |
| scrollOnExpand | boolean | RW | Scrolling lors de l'ouverture d'un nœud |
| selectionPath | TreePath | RW | Chemin du premier nœud sélectionné |
| selectionPaths | TreePath[] | RW | Chemins des nœuds sélectionnés |
| selectionRow | int | W | Sélection d'un nœud |
| selectionRows | int[] | RW | Nœuds sélectionnés |
| selectionCount | int | R | Nombre de nœuds sélectionnés |
| selectionModel | TreeSelectionModel | RW | Modèle de sélection |
| visibleRowCount | int | RW | Nombre de nœuds visibles si l'arbre est dans un JScrollPane (par défaut : 20) |



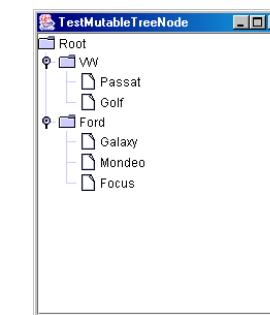
DefaultMutableTreeNode [1]

- Une autre manière de construire un arbre **JTree** est de créer une série de nœuds de type **DefaultMutableTreeNode** et de les assembler (les connecter) pour former un arbre.
- Les différents constructeurs de la classe **DefaultMutableTreeNode** permettent de construire des nœuds en passant un objet (*User-Object*) et optionnellement un indicateur qui précise si le nœud peut avoir des enfants.
- La classe **DefaultMutableTreeNode** possède différentes méthodes permettant d'**assembler** des nœuds (**add()**, **insert()**, **remove()**, etc.).
- On y trouve aussi des méthodes pour **consulter les relations** entre les nœuds d'un arbre (**isNodeAncestor()**, **isNodeChild()**, **isNodeSibling()**, **isNodeDescendant()**, etc.)
- Elle possède également des méthodes pour **naviguer** dans l'arbre (**getNextNode()**, **getPreviousNode()**, **preOrderEnumeration()**, **depthFirstEnumeration()**, etc.).



DefaultMutableTreeNode [3]

- L'exécution du code de la page précédente donne le résultat suivant :



DefaultMutableTreeNode [2]

```

public class TestMutableTreeNode extends JFrame {
    //--- Création des nœuds de l'arbre
    DefaultMutableTreeNode n0 = new DefaultMutableTreeNode("Root");
    DefaultMutableTreeNode n10 = new DefaultMutableTreeNode("VW");
    DefaultMutableTreeNode n110 = new DefaultMutableTreeNode("Passat");
    DefaultMutableTreeNode n111 = new DefaultMutableTreeNode("Golf");
    DefaultMutableTreeNode n20 = new DefaultMutableTreeNode("Ford");
    DefaultMutableTreeNode n210 = new DefaultMutableTreeNode("Galaxy");
    DefaultMutableTreeNode n211 = new DefaultMutableTreeNode("Mondeo");
    DefaultMutableTreeNode n212 = new DefaultMutableTreeNode("Focus");

    public TestMutableTreeNode() {
        super("TestMutableTreeNode");
        setSize(250, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //--- Assemblage des nœuds (création de la topologie)
        n0.insert(n10, 0);      n0.insert(n20, 1);
        n10.insert(n110, 0);    n10.insert(n111, 1);
        n20.insert(n210, 0);    n20.insert(n211, 1);    n20.insert(n212, 2);

        JTree tree = new JTree(n0);          // Création de l'arbre
        JScrollPane sTree = new JScrollPane(tree);
        getContentPane().add(sTree, BorderLayout.CENTER);
    }
}
  
```





Interface Homme-Machine 1

Interface utilisateur graphique (GUI) 11 Swing : Timers & Threads

Jacques Bapst

jacques.bapst@hefr.ch



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg



Interface Homme-Machine 1 / Swing : Timers & Threads

Utilisation de Threads avec Swing

- La conception de la librairie Swing impose que les instructions qui accèdent et manipulent des composants de l'interface graphique soient effectuées dans un seul *Thread*.
- Autrement dit, la librairie **Swing n'est pas Thread-Safe** (à l'exception de quelques rares méthodes mentionnées plus loin).
- C'est par souci d'efficacité et de simplicité que ce modèle de conception a été utilisé pour l'élaboration de la librairie Swing.
Le coût de la gestion des verrous (*locks*) a été jugé prohibitif en terme de performance et de complexité.
- Le *Thread* dans lequel s'exécute le code qui gère l'interface graphique est appelé **Event Dispatch Thread** ou **EDT** (ce n'est pas le *Thread* principal de l'application).
- Ce *Event Dispatch Thread* est démarré automatiquement lorsque l'on utilise des composants de la librairie Swing.



Interface Homme-Machine 1 / Swing : Timers & Threads

Swing Event-Dispatch Model



Operating System



AWT Event Queue

| |
|---------------|
| Mouse Press |
| Mouse Move |
| Key Press |
| ... |
| Mouse Release |
| Mouse Move |
| Mouse Press |

Critical thread
Time-consuming
code freeze the GUI !



EIA-FR / Jacques Bapst

3



IHM1_11



Interface Homme-Machine 1 / Swing : Timers & Threads

Règles à suivre

- La librairie Swing impose la règle suivante :
Dès qu'un composant a été rendu visible (*realized*), tout le code qui dépend ou qui pourrait affecter l'état du composant doit être exécuté dans le Event Dispatch Thread (EDT).
- Un conteneur de premier niveau (*JFrame*, *JDialog*, *JApplet*, *JWindow*) est rendu visible (*realized*) dès qu'une des méthodes suivantes a été invoquée : `setVisible(true)`, `pack()` ou `show()` [deprecated].
- Cette règle permet cependant, dans la plupart des cas, de construire l'interface graphique dans le *Thread* principal avant de la rendre visible.
- Il est donc possible (même si Sun le déconseille maintenant) de créer les composants visuels, de les configurer et de les placer dans des conteneurs dans le cadre du *Thread* principal de l'application mais ensuite (après l'invocation de l'une des méthodes `setVisible(true)` ou `pack()`), toutes les instructions qui accèdent ou manipulent les composants Swing doivent être exécutées dans le *EDT* (sous peine d'effets imprévisibles).





Event Dispatch Thread (EDT)

- Le **Event Dispatch Thread (EDT)** est le *Thread* qui exécute les instructions de dessin (*Drawing*) des composants et qui se charge de la gestion des événements (*Event Handling*) liés à ces composants.
- Toutes les instructions de **dessin** et de **réaffichage** ainsi que celles qui se trouvent dans les **récepteurs d'événements** (*Event Listener*) sont implicitement exécutées dans le *EDT*.
- L'utilitaire représenté par la classe **javax.swing.Timer** permet également d'exécuter des instructions (après un certain délai ou à intervalles périodiques) dans le *EDT*.
- Deux méthodes statiques de la classe **EventQueue** permettent également d'insérer des instructions dans le *EDT* :
 - invokeLater()**
 - invokeAndWait()**
- Ces deux méthodes prennent pour argument un objet qui implémente l'interface **Runnable**.



Timer [2]

- Le principe de la classe **Timer** est de générer à intervalles réguliers des événements de type **ActionEvent** qui seront traités dans un gestionnaire de type **ActionListener**.
- L'objet qui implémente **ActionListener** ainsi que l'intervalle entre deux exécutions (en [ms]) sont passés en paramètre du constructeur de la classe **Timer** .


```
// La classe MyController doit implémenter ActionListener et donc déclarer
// une méthode actionPerformed()
MyController c = new MyController();
```

```
// Création d'un timer qui exécutera la méthode actionPerformed()
// toutes les deux secondes
```

```
Timer horloge = new Timer(2000, c); // Création du Timer (2 s)
horloge.setInitialDelay(500); // 0.5 s avant 1re exécution
horloge.start(); // Démarrage du Timer
```



Timer [1]

- La classe **javax.swing.Timer** permet d'exécuter automatiquement certaines instructions sur la base d'une horloge (temporisateur).
- Un timer peut être utilisé pour :
 - Exécuter une tâche après un certain délai**
 - Répéter à intervalle périodique** une certaine tâche
- Attention : Ne pas confondre la classe **javax.swing.Timer** avec la classe plus générale **java.util.Timer**. La première doit être utilisée de préférence pour des activités liées à une interface utilisateur (GUI).
- La classe **Timer** permet d'effectuer des animations ou des mises à jour d'informations permettant de refléter la progression d'une activité.
- Les instructions exécutées par la classe **Timer** s'exécutent dans le *EDT* et peuvent donc accéder et modifier sans risque les propriétés des composants affichés.
- Pour ne pas perturber la réactivité de l'application, les instructions effectuées par la classe **Timer** doivent s'exécuter rapidement (on risque sinon de figer l'interface utilisateur).



Timer [3]

- La classe **Timer** possède les **propriétés** principales suivantes :

| | |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| initialDelay | Durée d'attente (en [ms]) entre le démarrage du timer et la première exécution Par défaut : identique à delay |
| delay | Intervalle entre deux exécutions (en [ms]) |
| repeats | true si l'exécution doit se répéter périodiquement false si l'exécution est unique Par défaut : true |
| coalesce | Indication (true / false) si le système doit grouper des exécutions multiples en attente (et n'exécuter qu'une seule fois l'action pour tous les événements en attente) Cela peut se produire si des événements n'ont pas pu être traités à temps en raison de la charge du système Par défaut : true |
| actionCommand | Définit la chaîne de caractères associée à la propriété actionCommand de l'événement du timer |





Timer [4]

- La classe **Timer** possède les **méthodes** principales suivantes :

| | |
|--------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>start()</code> | Démarre le timer après le délai initial (<i>initialDelay</i>) |
| <code>restart()</code> | Annule toutes les actions en attente et redémarre le timer après le délai initial (<i>initialDelay</i>) |
| <code>stop()</code> | Annule toutes les actions en attente et stoppe le timer |
| <code>isRunning()</code> | Retourne <code>true</code> si le timer est démarré |

- Il est possible de **modifier la fréquence d'exécution** même si le *timer* est en train de tourner (en utilisant la méthode `setDelay()`).
- La classe **javax.swing.Timer** peut également être utilisée pour temporiser des activités qui n'ont rien à voir avec l'interface utilisateur.
- La classe **java.util.Timer** est un peu plus générale (elle contient plus de fonctionnalités) mais les activités ne se déroulent pas automatiquement dans le *EDT*. Elle est donc un peu plus complexe à utiliser correctement en relation avec une interface utilisateur.



Swing et les Threads [1]

- On l'a vu, les méthodes de la librairie Swing ne sont pas *Thread-Safe* et ne doivent pas être invoquées en dehors du *EDT*.
- Il y a cependant quelques exceptions et les méthodes suivantes peuvent être utilisées depuis n'importe quel *Thread* de l'application :

addEventListerner()
removeEventListerner()
repaint()
invalidate() (rarement nécessaire, invoqué automatiquement)
revalidate() (rarement nécessaire, invoqué automatiquement)

(Il y en a encore quelques autres qui sont mentionnées dans la documentation)

- Lorsqu'une interface graphique a été construite et affichée, la plupart des applications réagissent aux événements par invocation du code inséré dans les récepteurs d'événements (*Event Listener*).
- Ces instructions ne posent pas de problèmes particuliers car elles sont implicitement exécutées dans le *EDT*.



Swing et les Threads [2]

- Il y a cependant des situations où l'exécution d'instructions liées à l'interface graphique ne peut pas s'effectuer dans les *Event Listener*.
- Par exemple :
 - Si certaines instructions durent longtemps, elles ne devraient pas être insérées dans des *Event Listener* car elles bloqueront toute la gestion de l'interface graphique qui ne réagira plus aux sollicitations de l'utilisateur.
 - Si la mise à jour de l'interface graphique résulte d'un événement qui n'est pas lié à un composant visuel aucun gestionnaire d'événement ne pourra être enregistré pour le traiter (par exemple si l'événement est provoqué par un signal provenant d'une autre machine, par la synchronisation avec un autre processus ou par un signal provenant d'un périphérique particulier).
- Dans certaines situations, il peut donc être indispensable de créer un ou plusieurs *Threads* (pour effectuer les opérations qui prennent beaucoup de temps) et de mettre à jour l'interface graphique en invoquant l'une des deux méthodes utilitaires **invokeLater()** ou éventuellement **invokeAndWait()** qui permettront d'insérer les instructions nécessaires dans le *EDT*.



Swing et les Threads [3]

- En résumé, il y a **deux contraintes importantes à respecter** pour garantir la réactivité de l'interface graphique :
 - Toutes les tâches qui prennent du temps doivent être exécutées en dehors du EDT.**
 - Les composants de l'interface graphique ne doivent être accédés que depuis le EDT.**
- Il y a principalement trois types de *threads* à considérer dans une application comportant une interface-utilisateur graphique.
 - Le **Thread principal** (*Main Thread*) qui exécute les instructions au lancement de l'application
 - Le **Event Dispatch Thread** qui exécute le code de tous les gestionnaires d'événement (*Event Listeners*)
 - Les **Worker Threads** (*Background Threads*) qui sont les *threads* dans lesquels sont exécutées les tâches qui prennent du temps (ou qui sont en attente d'un signal asynchrone quelconque)





Utilisation de invokeLater()

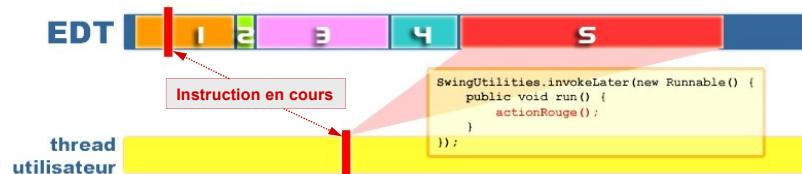
- La méthode **invokeLater()** peut être invoquée depuis n'importe quel *Thread* pour exécuter certaines instructions dans le *EDT*.
- Le code à exécuter doit être inséré dans la méthode **run()** d'un objet qui implémente l'interface **Runnable**.
- Lors de son invocation, la méthode **invokeLater()** retourne immédiatement, sans attendre que les instructions de la méthode **run()** aient été exécutées dans le *EDT*.
- Exemple :

```
Runnable doWorkRunnable = new Runnable() {
    public void run() {
        doWork(); // Code à exécuter dans le Event Dispatch Thread
    }
};
EventQueue.invokeLater(doWorkRunnable);
```



Séquencement des Threads

- L'objet **Runnable** s'exécutera de manière asynchrone par rapport au *Thread* utilisateur (*Main Thread* par exemple).
- Séquencement avec la méthode **invokeLater()** :



- Si l'on veut impérativement attendre la terminaison de l'exécution de l'objet **Runnable** avant de continuer, il faut utiliser la méthode **invokeAndWait()** qui est bloquante.



Utilisation de invokeAndWait()

- La méthode **invokeAndWait()** s'utilise de la même manière que **invokeLater()** en lui passant un objet qui implémente l'interface **Runnable**.
- La seule différence est que lors de son invocation, la méthode **invokeAndWait()** attend que les instructions de la méthode **run()** aient été exécutées dans le *EDT* avant de retourner le contrôle au code qui l'a invoquée.
- D'une manière générale, la méthode **invokeLater()** doit être utilisée de préférence car la méthode **invokeAndWait()** peut poser certains problèmes d'interbloquage (*deadlock*) si le *Thread* dans lequel elle est invoquée maintient des verrous (*locks*) sur certains objets qui doivent être accédés par les instructions exécutées dans le *EDT*.
- Remarque : Si nécessaire, la méthode statique **isDispatchThread()** (de la classe **EventQueue**) permet de tester si un code s'exécute dans le *Event Dispatch Thread*.



SwingWorker Framework [1]

- Pour simplifier la gestion des applications qui comportent des activités dont la durée est indéterminée et qui affectent l'état de l'interface utilisateur, une classe utilitaire appelée **SwingWorker** est à disposition.
- Cette classe (qui se trouve dans le package **javax.swing.swingworker**) est un *framework* permettant de simplifier la séparation des tâches qui doivent être effectuées hors du *EDT* de celles qui touchent à l'interface et qui doivent donc obligatoirement être exécutées dans le *EDT*.
- Avant la version 1.6 du JDK, cette classe ne faisait pas partie de la plate-forme Java standard mais pouvait être téléchargée (site de Sun).
- A partir de la version 1.6, la classe **SwingWorker** a été sensiblement modifiée (avec généricité) et a été intégrée à la plate-forme Java.
- Pour utiliser cette classe abstraite, il faut créer une sous-classe et redéfinir la seule méthode abstraite **doInBackground()** dans laquelle seront placées toutes les instructions qui prennent du temps (sans interactions directes avec le GUI). La méthode **doInBackground()** retourne un objet de type **T** (paramètre générique de la classe).





SwingWorker Framework [2]

- La valeur renvoyée par la méthode `doInBackground()` peut être récupérée par la méthode `get()`.
Attention : la méthode `get()` est bloquante et attendra que la méthode `doInBackground()` se termine. Un *timeout* fixant l'attente maximale peut cependant être passé en paramètre.
- La méthode `done()` peut également être redéfinie si nécessaire, elle sera invoquée lorsque la méthode `doInBackground()` se terminera (ou si `cancel()` est invoqué [voir plus loin]).
- Les instructions de la méthodes `done()` s'exécutent dans le *EDT* c'est donc là qu'il faut placer les instructions qui mettent à jour l'interface utilisateur (attention à ne pas y placer des instructions qui prennent beaucoup de temps !).
- Des mises à jour intermédiaires de l'interface graphique peuvent être effectuées en invoquant la méthode `publish()` dans la méthode `doInBackground()` et en récupérant les résultats dans la méthode `process()` (qui s'exécute dans le *EDT*).



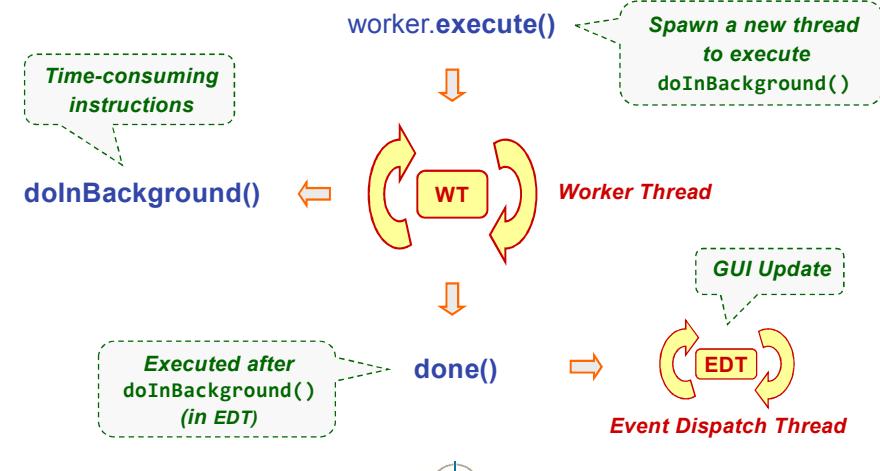
SwingWorker Framework [3]

- Les instructions de la méthode `doInBackground()` sont exécutées (dans un *Thread* indépendant) dès que la méthode `execute()` a été invoquée sur l'instance de `SwingWorker`.
- Attention : L'instance de `SwingWorker` ne peut pas être réutilisée. Un deuxième appel à `execute()` ne ré-exécutera pas les instructions de la méthode `doInBackground()`.
- Attention : Les éventuelles exceptions générées par les instructions de la méthode `doInBackground()` seront perdues si la méthode `get()` n'est pas invoquée pour récupérer un résultat.
Une amélioration a été proposée par Baptiste Wicht (ancien étudiant EIA-FR !). Voir l'article "*A better SwingWorker without exception swallowing*" : www.baptiste-wicht.com/2010/09/a-better-swingworker



SwingWorker Model [1]

```
public class MyWorker extends SwingWorker <T, V> { ... }
MyWorker worker = new MyWorker();
```

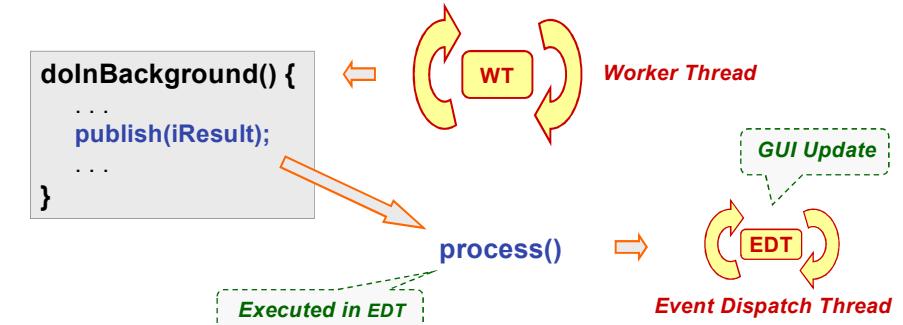


SwingWorker Model [2]

- La méthode `doInBackground()` peut transmettre des résultats intermédiaires (de type *V*) en invoquant, dans ses instructions, la méthode `publish(v... results)`.
- Le traitement de ces résultats partiels est effectué dans la méthode `process()` qui reçoit un liste d'objets de type *V*. La méthode `process()` s'exécute dans le *EDT*.

```
doInBackground() {
    ...
    publish(iResult);
    ...
}
```

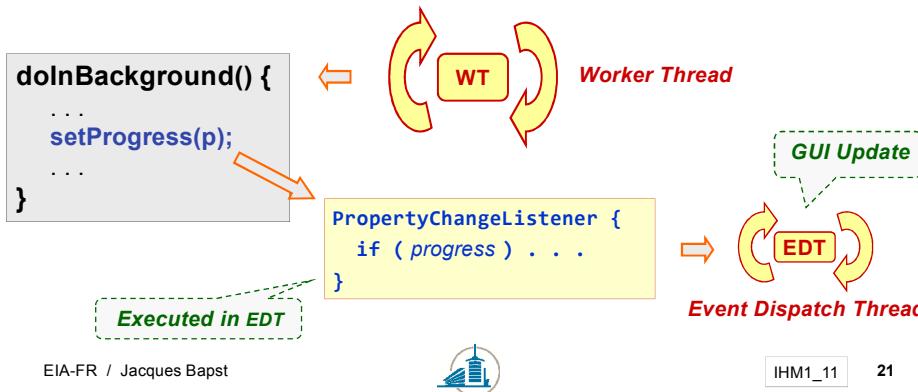
Executed in EDT





SwingWorker Model [3]

- La méthode **doInBackground()** peut également transmettre des informations de progression (entier 0...100) en invoquant, dans ses instructions, la méthode **setProgress()**.
- Le traitement de la progression s'effectue en créant et enregistrant un récepteur d'événement de type **PropertyChangeListener** et en surveillant la propriété "progress".



EIA-FR / Jacques Bapst

IHM1_11

21



SwingWorker : Principe d'utilisation [1]

```
public class WorkerTask extends SwingWorker<Double, String> {
    //-
    // doInBackground() - Code executed in a separated thread
    //-
    protected Double doInBackground() throws Exception {
        ...
        ... publish(result);           // Send intermediate results
        ... setProgress(percentDone);  // Progression [0..100]
        ...
    }
    //-
    // done() - Code executed in EDT, after doInBackground() has finished
    //-
    protected void done() {
        ...
    }
    //-
    // process() - Code executed in EDT to treat intermediate results
    //-
    protected void process(List<String> list) {
        ...
    }
}
```

EIA-FR / Jacques Bapst

IHM1_11

22



SwingWorker : Principe d'utilisation [2]

- Exemple de tâche à exécuter dans un *Thread* séparé

```
//-
// doInBackground() - Code executed in a separated thread
//-
protected Double doInBackground() throws Exception {
    double result = 0;
    //-
    // Long task
    for (long i=1; i<nbSteps; i=i+2) {
        result += 1.0/i;
        if ((i-1)%100000 == 0) publish("V="+result); // Send intermediate results
        setProgress((int)((100*i)/(nbSteps-1))); // Progression [0..100]
    }
    //-
    // Check if task cancelled
    if (isCancelled()) return null;
    //-
    // Give other threads (of same priority) a chance to run
    Thread.yield();
}
return new Double(result); // Result (can be recovered by get())
}
```

EIA-FR / Jacques Bapst

IHM1_11

23



SwingWorker : Principe d'utilisation [3]

- Récupération d'informations intermédiaires et mise à jour de l'interface graphique (les instructions de la méthode **process()** sont exécutées dans le *EDT*).

```
//-
// process() - Code executed in EDT to treat intermediate results
//-
protected void process(List<String> list) {
    //-
    // Get intermediate results sent by publish() and transmitted
    // in a List (multiple results could be grouped)
    String s = list.get(list.size()-1); // Last element
    //-
    // Update view
    view.lbResult.setText(s);
}
```

EIA-FR / Jacques Bapst

IHM1_11

24



SwingWorker : Principe d'utilisation [4]

- Récupération des informations de progression émises par la méthode `setProgress()`.
- Un gestionnaire d'événements de type `PropertyChangeListener` doit être créé et enregistré sur l'instance de `SwingWorker`.
- La propriété porte le nom "`progress`" (valeurs comprises entre [0..100]).
- Exemple sous forme de classe interne anonyme :

```
-----  
// Controller to manage the progression (activated by setProgress())  
// Update the state of the progress-bar  
-----  
worker.addPropertyChangeListener(new PropertyChangeListener() {  
    public void propertyChange(PropertyChangeEvent evt) {  
        if ("progress".equals(evt.getPropertyName())) {  
            view.pbCurState.setValue((Integer)evt.getNewValue());  
        }  
    }  
});
```



Interruption de SwingWorker [1]

- La méthode `cancel()` peut être utilisée pour demander l'interruption de l'activité du *Thread SwingWorker* (instructions contenues dans la méthode `doInBackground()`).
- La méthode `cancel()` prend un paramètre de type booléen :
 - `true` : Le flag `cancel` et le flag `interrupt` sont mis à `true`
 - `false` : Seul le flag `cancel` est mis à `true`
- La prise en compte de l'interruption par le flag `cancel` est à la charge du programmeur (aucun effet sinon). Elle s'effectuera en invoquant la méthode `isCancelled()` dans la méthode `doInBackground()`.
- La prise en compte de l'interruption par le flag `interrupt` s'effectuera
 - Si la méthode `doInBackground()` invoque les méthodes `sleep()` ou `wait()` qui généreront l'exception `InterruptedException` en cas d'interruption
 - Si la méthode `doInBackground()` interroge explicitement l'état du flag d'interruption en invoquant `Thread.interrupted()`
- L'interruption est donc **basée sur la coopération !**



SwingWorker : Principe d'utilisation [5]

- Le code de la méthode `done()` est exécuté après la fin de la méthode `doInBackground()` ou après invocation de la méthode `cancel()`. La méthode `get()` permet de récupérer la valeur renournée par la méthode `doInBackground()`.
- Attention : Compte tenu de son fonctionnement asynchrone, le code de la méthode `process()` peut être exécuté après le code de `done()`.

```
-----  
// done() - Code executed in EDT, after doInBackground() has finished  
-----  
protected void done() {  
    view.btStart.setEnabled(true);  
    view.btStop.setEnabled(false);  
    if (isCancelled()) System.out.println("== Cancelled ==");  
    else {  
        try {  
            view.lbResult.setText(get().toString());  
        } catch (Exception e) {  
            System.out.println("== Interrupted ==");  
        }  
    }  
}
```



Interruption de SwingWorker [2]

- Après exécution de `cancel()`, la méthode `done()` sera invoquée (même si la méthode `doInBackground()` ne s'est pas encore arrêtée).
- La technique utilisant la `isCancelled()` a été illustrée dans l'exemple donné pour la méthode `doInBackground()` (voir pages précédentes).
- Exemple utilisant le flag `interrupt` :

```
protected String doInBackground() throws Exception {  
    try {  
        for(int i=0; i<NUMLOOPS; i++) {  
            updateCounter(i);  
            if (Thread.interrupted()) throw new InterruptedException();  
            doWork(i);  
            Thread.sleep(500); // Can throw InterruptedException  
        }  
    } catch (InterruptedException e) {  
        updateCounter(0);  
        return "Interrupted";  
    }  
    return "All done";  
}
```





SwingWorker et activités GUI

- Si la méthode `doInBackground()` doit effectuer des actions liées à l'interface utilisateur (une mise à jour par exemple) il faut impérativement que ces instructions s'exécutent dans le *EDT*.
- On utilisera de préférence la méthode `setProgress()` vue précédemment qui permet de transférer des résultats intermédiaires à un processus qui s'exécute dans le *EDT* (un `PropertyChangeListener`).
- Dans certaines situations particulières on peut placer ces instructions dans un `Runnable` et les injecter dans le *EDT* avec `invokeLater()`.

```
void updateCounter(final int i) {  
    model.setCounter(i);  
  
    Runnable updateProgressBarValue = new Runnable() {  
        public void run() {  
            progressBar.setValue(i);  
        }  
    };  
  
    EventQueue.invokeLater(updateProgressBarValue);  
}
```



Autres Framework

- Le modèle asynchrone de *SwingWorker* peut présenter certaines faiblesses dans des applications complexes.
- D'autres solutions (*Framework*) ont été proposées pour remédier à certains des inconvénients de *SwingWorker*.
- Parmi ces *Framework* on peut citer : **Foxtrot** (foxtrot.sourceforge.net) qui se base sur un modèle synchrone.
- Parmi les avantages avancés par les concepteurs on peut mentionner :
 - une API simple
 - une meilleure symétrie et lisibilité du code
 - un traitement facilité des exceptions
 - une maintenance plus simple
- **Foxtrot** est composé de trois classes principales (`Worker`, `Task`, `Job`)
- Autre *Framework* avec le même objectif : **Spin** (spin.sourceforge.net)
- Voir API, documentation et exemples sur les sites correspondants.

