

# Interface-utilisateur graphique d'une application Java

DUT Informatique 1<sup>ère</sup> année

Kaï Poutrain & Henri Garreta & Cyril Pain-Barre

## TP 2 - Boutons, étiquettes, icônes, etc.

Pour effectuer les TP de Swing aidez-vous du support de TP intitulé "[Support Swing](#)", qui comporte pour chaque classe un descriptif des méthodes les plus utiles regroupées suivant leur fonction.

Créez un nouveau projet **TP2** (toujours en séparant les sources et les binaires). Nous utiliserons à nouveau des packages afin de garder une trace de la progression dans le TP.

L'une des particularités de Swing est la possibilité de personnaliser les interfaces avec des images. Télécharger l'archive [RESGRAF.zip](#) et l'extraire dans le répertoire racine du projet, ce qui crée le répertoire **RESGRAF** contenant 19 images au format GIF.

### 2.1 - Création d'une fenêtre de base

En Swing, une fenêtre est un *conteneur de premier niveau* (Top-Level container) appelé *cadre*, de la classe **JFrame**.

Nous allons créer une classe publique **FenetreSimple** qui va nous permettre d'avoir une fenêtre de travail pour la suite.

#### Exo 2.1.1 - Création et affichage du cadre

[\[Corrigé\]](#)

Dans le projet **TP2**, créez un package **exo\_02\_01** (il servira pour l'ensemble des exercices de cette section 2.1).

Créez une nouvelle classe publique exécutable **FenetreSimple** qui étend **javax.swing.JFrame**. Le fichier source (**FenetreSimple.java**) doit contenir :

- une instruction `import javax.swing.*;` qui permet de travailler avec les noms courts des classes de Swing
- la classe publique ayant le même nom que le fichier, et qui hérite de **javax.swing.JFrame** par utilisation de `extends` :

```
public class FenetreSimple extends JFrame {  
    ...  
}
```

Notons que ce squelette est automatiquement créé par *eclipse* si on adapte la zone *Superclass* dans la fenêtre de création d'une classe.

Consultez la section **JFrame** du support de TP consacré à Swing ou de l'API et créez le constructeur de votre classe de manière à ce qu'il prenne le titre de la fenêtre en argument :

```
public FenetreSimple(String titre) {  
    ...  
}
```

*Indications* : en Java, un constructeur commence explicitement ou implicitement par une instruction `super(...)` qui fait appel à un constructeur de la super classe (classe parente). Par exemple, `super(12, "hello")` fait appel au constructeur de la classe parente acceptant ces paramètres.

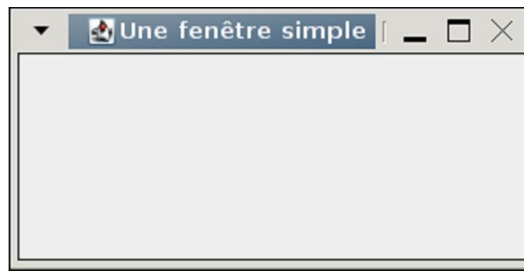
Donnez une taille à votre fenêtre en définissant deux constantes de classes (variables à la fois **static** et **final**) **LARGEUR** et **HAUTEUR**, valant respectivement 300 et 150, qu'on utilisera à bon escient en appelant la méthode `setSize(int width, int height)` héritée indirectement de **java.awt.Window**.

Notez que, pour le moment, si nous créons un objet **FenetreSimple**, celui-ci serait invisible. Complétez le constructeur en ajoutant en dernier lieu l'instruction `setVisible(true)` pour que le cadre s'affiche

automatiquement à sa création.


Ecrivez la méthode `main()` de la classe, qui se contentera de créer une fenêtre simple comportant le titre que vous choisirez.

Compilez et exécutez ce `FenetreSimple.java`, vous obtiendrez une fenêtre rudimentaire de la taille indiquée, ressemblant à :



Elle est cependant totalement opérationnelle, puisqu'elle peut-être agrandie ou réduite, icônifiée, déplacée, placée devant ou derrière une autre fenêtre...

En revanche, si vous cliquez sur le bouton de fermeture vous ne la ferez que disparaître, le programme continuera à tourner...

Terminez l'exécution de cette superbe application en cliquant sur le bouton  de la console d'exécution d'*eclipse* ou par `Ctrl-C` si vous l'exécutez depuis la ligne de commandes.

### Exo 2.1.2 - Traiter la fermeture d'un cadre Swing (version 1)

[\[Corrigé\]](#)

Nous allons modifier le comportement de fermeture du cadre en indiquant ce que l'application doit faire lorsque l'utilisateur clique sur le bouton de fermeture.

Pour cela, ajoutons (avant le `setVisible()`) dans le constructeur `FenetreSimple` un appel à la méthode de profil :

```
public void setDefaultCloseOperation(int operation)
```

héritée de la classe `JFrame`. L'opération indique comment l'application doit réagir lorsque l'utilisateur clique sur le bouton de fermeture. Par défaut, l'opération est `JFrame.HIDE_ON_CLOSE` (héritée de `javax.swing.WindowConstants`) : la fenêtre est simplement cachée. Pour terminer l'application, il faudra utiliser `JFrame.EXIT_ON_CLOSE`.

### Exo 2.1.3 - Traiter la fermeture d'un cadre Swing (version 2)

[\[Corrigé\]](#)

La solution précédente semble suffisante. Cependant, elle est un peu brutale car l'application se termine aussitôt le clic effectué sur le bouton de fermeture. Par exemple, il n'est pas possible en l'état de demander à l'utilisateur de confirmer son souhait de terminer l'application, ni même d'opérer le moindre traitement, comme afficher un simple message.

Nous allons changer cela. Plutôt que d'utiliser `setDefaultCloseOperation()`, nous allons capturer l'événement généré lorsque l'utilisateur clique sur le bouton de fermeture. Pour cela, remplacer l'appel de `setDefaultCloseOperation()` par l'ajout d'un **écouteur d'événement de fenêtre** à l'aide de la méthode `addWindowListener(WindowListener l)`.

`WindowListener` est une interface, ce qui signifie que nous devrions normalement redéfinir toutes les méthodes qu'elle contient.... or une seule nous intéresse pour l'instant, `windowClosing(WindowEvent e)`, automatiquement invoquée quand l'utilisateur clique sur le bouton de fermeture. Heureusement, la classe `WindowAdapter` existe pour simplifier la vie du développeur. Elle ne fait que définir les méthodes imposées par `WindowListener` (et d'autres interfaces) mais n'entreprend aucune action suite à un événement. Il est alors beaucoup plus confortable pour le développeur de sous-classer `WindowAdapter` pour éviter de devoir définir les méthodes imposées par `WindowListener`, et de redéfinir (surcharger) les seules méthodes qui l'intéressent. Aussi, allons nous sous-classer `WindowAdapter`, et redéfinir `windowClosing()` :

```
...
addWindowListener(new WindowAdapter() {
```

```

        public void windowClosing(WindowEvent e) {
            ...
            System.exit(0);
        }
    });
    ...

```

L'instruction `System.exit(0)` terminera l'application, mais il est maintenant possible de réaliser d'autres traitements avant cette instruction.

*Remarque 1.* Si nous voulions effectivement demander une confirmation de fermeture (par exemple, en appelant une méthode statique `showConfirmDialog()` de la classe `JOptionPane`), il serait judicieux d'empêcher la fenêtre d'être cachée suite au clic de fermeture en ajoutant, avant l'instruction `addWindowListener()`, l'instruction suivante :

```
setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

*Remarque 2.* La méthode proposée peut paraître complexe mais il n'en est rien. Qu'avons-nous fait et qu'aurions-nous pu faire d'autre ? Nous avons simplement créé un objet d'une **classe anonyme**, qui étend `WindowAdapter`, en redéfinissant `windowClosing()`, c'est à dire juste ce qu'il nous faut. On comprendra à quel point cette facilité offerte par Java de créer ainsi des objets et des classes est appréciable en se demandant quelle serait l'alternative si ce n'était pas possible. Eh bien, il faudrait définir une classe nommée par exemple `MonWindowAdapter` ainsi :

```

class MonWindowAdapter extends WindowAdapter {

    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

```

puis en créer une instance lors de l'appel à `addWindowListener()` :

```

...
addWindowListener(new MonWindowAdapter());
...

```

Bref, on ne peut pas dire que ce soit ni plus simple, ni mieux, car cette classe n'aura servi qu'à un seul endroit...

*Remarque 3.* La classe anonyme (dérivée de `WindowAdapter`) définie "à la volée" dans la solution proposée est **une classe interne** de `FenetreSimple`. Une instance de cette classe est forcément liée à une instance de `FenetreSimple`. Une conséquence est qu'à l'intérieur de sa définition, on a accès à l'ensemble des données membres de `FenetreSimple`. D'autre part, `this` y représente l'instance de cette classe, alors que `FenetreSimple.this` y représente l'instance de `FenetreSimple` à laquelle *appartient* cette instance.

*Remarque 4.* Dans le code, nous utilisons les noms courts des classes `java.awt.event.WindowAdapter` et `java.awt.event.WindowEvent`. Par conséquent, nous devons ajouter en début de fichier les instructions `import` permettant d'utiliser ces noms courts :

```

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

```

ce que nous pouvons résumer en :

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

ce qui nous évitera d'ajouter d'autres instructions `import` si nous devons utiliser d'autres classes du package `java.awt.event`. Ceci ne pose en outre aucun problème au niveau optimisation de l'espace mémoire puisque cette instruction ne charge pas les classes en question : en Java, les classes ne sont chargées qu'à la demande lors de l'exécution.

Modifiez `FenetreSimple` pour utiliser cette méthode de fermeture.

## 2.2 - Ajout d'un composant Swing

Les cadres de la classe `JFrame` sont des **conteneurs**, c'est à dire des objets destinés à contenir d'autres objets : les composants Swing, qui héritent tous de la classe `JComponent`.

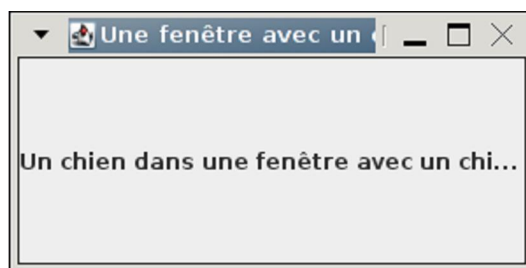
### Exo 2.2.1 - Une étiquette (JLabel)

[\[Corrigé\]](#)

Afin d'agrémenter notre fenêtre simple, nous allons y ajouter une étiquette simple contenant du texte, chose que nous allons effectuer en créant une classe publique (exécutable) `EtiquetteSimple` qui étend `JLabel`, définie dans un fichier séparé.

Définir le constructeur de la classe `EtiquetteSimple`, de manière à pouvoir passer le texte de l'étiquette en argument.

Votre classe devra comporter la très classique méthode `main()` dans laquelle vous créerez un objet `cadre`, de la classe `FenetreSimple`, dont le titre sera par exemple "Une fenêtre avec un chien" et un objet `etiquette` de la classe `EtiquetteSimple`, et dont le texte sera "Un chien dans une fenêtre avec un chien !" :



Remarques :

- en swing, avant Java 5, il n'était pas possible d'ajouter directement un composant à un cadre, il fallait passer par l'intermédiaire d'un panneau de contenu (`ContentPane`). Tous les cadres en comportent un par défaut, et on y accède par la méthode `getContentPane()`. Pour ajouter l'étiquette au panneau de contenu de notre cadre, il fallait utiliser la méthode `add(Component comp)` dans une instruction similaire à :

```
cadre.getContentPane().add(etiquette);
```

Depuis Java 5, cette contrainte a été considérablement assouplie : ce qui est dit au paragraphe précédent reste vrai, mais la classe `JFrame` a été étendue de telle manière que l'expression :

```
unJFrame.add(unComponent)
```

soit désormais légitime, et comprise comme :

```
unJFrame.getContentPane().add(unComponent)
```

- on ne doit modifier une fenêtre visible qu'en réaction à un événement. Ceci inclut l'ajout de composants. Il en découle qu'il nous faut modifier le constructeur de `FenetreSimple` et en supprimer l'appel de `setVisible()`. Après l'ajout de l'étiquette dans le `cadre`, il faudra le rendre visible par l'instruction :

```
cadre.setVisible(true);
```

- notez que nous aurons une méthode `main()` dans chacune des deux classes, mais en Java ceci est tout à fait autorisé. C'est même un moyen pratique pour implémenter des procédures de tests particulières pour chaque classe. En effet, en tapant :

```
$ java FenetreSimple
```

c'est la méthode `main()` de `FenetreSimple.java` qui est invoquée, alors qu'en tapant :

```
$ java EtiquetteSimple
```

c'est celle de `EtiquetteSimple.java` qui l'est.

Dans Eclipse, il faut se placer dans le fichier ciblé et, dans le menu *Run*, choisir *Run As* → *Java Application* (ou utiliser le raccourci Shift + Alt + XJ).

Il faudra alors penser à rendre visible le cadre dans le `main()` de `FenetreSimple.java` pour qu'elle garde son intérêt.

Exécutez les deux fichiers.

## Exo 2.2.2 - Dimensionnement automatique de la fenêtre

[\[Corrigé\]](#)

Le label de l'exercice précédent est probablement trop long pour entrer en totalité dans la fenêtre. Nous allons modifier le programme pour que la taille du cadre s'ajuste en fonction de celle de l'étiquette.

Pour cela, déplacez l'appel de la méthode `setSize()` depuis le constructeur de `FenetreSimple` dans le `main()` de `FenetreSimple.java` (on ne change pas l'exécution de cette classe), **avant** l'appel de `setVisible()`.

Pour un dimensionnement automatique, il nous faut appeler la méthode `pack()` de `JFrame`, héritée de `java.awt.Window` (consulter la documentation). Elle ne doit être appelée que lorsque tous les composants ont été ajoutés dans la fenêtre. Nous placerons donc son appel dans la méthode `main()` de `EtiquetteSimple.java`.

Exécutez `EtiquetteSimple`. Constatez que l'étiquette est à présent totalement contenue dans la fenêtre :



Exécutez aussi `FenetreSimple` pour s'assurer qu'elle n'a pas changé en fonctionnalité.

## Exo 2.2.3 - Personnalisation de l'étiquette

[\[Corrigé\]](#)

Les composants Swing possèdent la particularité de pouvoir être personnalisés à l'aide d'icônes. Nous allons remplacer le texte de l'étiquette par une image.

Ajoutez un constructeur de `EtiquetteSimple` qui prend en argument un `ImageIcon` et qui intègre l'icône dans le label, sans texte.

Dans la méthode `main()` de `EtiquetteSimple.java` créez un objet `chien` de la classe `ImageIcon`, et chargez l'image du fichier `Chien.gif` contenu dans le répertoire `RESGRAF` (attention au chemin utilisé : pour Eclipse, il doit commencer à la racine du projet. Si vous avez placé `RESGRAF` à la racine du projet, le chemin doit être `RESGRAF/Chien.gif`). Remplacez l'argument de `EtiquetteSimple()` par l'icône.

Exécutez `EtiquetteSimple.java`. Remarquez que la fenêtre se redimensionne automatiquement à la taille de l'image. Vous connaissez à présent notre féroce chien de garde : Rex !



## 2.3 - Ajuster l'apparence des labels

### Exo 2.3.1 - Mettons un peu d'espace

[\[Corrigé\]](#)

Ajoutez un constructeur de `EtiquetteSimple` qui prend deux arguments :

```
EtiquetteSimple(String texte, ImageIcon icone) {
```

```
    ...
}
```

On peut constater qu'il n'existe pas de constructeur de la super-classe (`JLabel`) qui ne prenne en argument qu'un texte et une icône. C'est pourquoi nous utiliserons le constructeur de `JLabel` (en utilisant `super()` !) suivant :

```
JLabel(String text, Icon icon, int horizontalAlignment)
```

en utilisant un alignement horizontal centré (`JLabel.CENTER`).

Toujours dans le corps du constructeur ajoutez un espace de 20 pixels entre l'image et le texte à l'aide de l'instruction :

```
setIconTextGap(20);
```

puis créez une bordure vide de 10 pixels autour de l'étiquette à l'aide de l'instruction :

```
setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
```

Dans `main()`, modifiez la création de l'objet `etiquette` qui doit à présent comporter l'image de Rex et un texte (par exemple "Un chien").

Exécutez `EtiquetteSimple.java`, vous devriez obtenir quelque chose comme :



### Exo 2.3.2 - Séance de mise en forme

[\[Corrigé\]](#)

Dans la méthode `main()` expérimentez le positionnement du texte par rapport à l'icône, horizontalement et verticalement à l'aide des méthodes de `JLabel` suivantes :

- `setHorizontalTextPosition(int textPosition)` qui définit la position horizontale du texte par rapport à l'image ;
- `setVerticalTextPosition(int textPosition)` qui définit la position verticale du texte par rapport à l'image.

Par exemple, modifiez `main()` pour placer le texte sur l'icône :



Un chien en forme !

## 2.4 - Laissons l'utilisateur choisir la position

Maintenant que nous savons placer le texte et l'icône, proposons à l'utilisateur de choisir lui même :

- la position horizontale du texte par rapport à l'icône (à gauche, au centre ou à droite)
- la position verticale du texte par rapport à l'icône (en haut, au centre ou en bas)

Pour cela, nous pourrions ajouter un menu (**JMenuBar**) mais ce sera étudié dans un autre TP. Nous allons plutôt étudier les mécanismes de création de palettes d'outils (**JToolBar**) et créer une palette qui va permettre de modifier cette position.

Au final, nous obtiendrons une fenêtre qui ressemble à :



Comme on le voit, la fenêtre est composée d'une palette d'outils (en haut) et d'une étiquette. La palette est constituée de deux groupes de 3 boutons :

- les trois boutons de gauche vont contrôler la position horizontale du texte (représenté par la petite barre à gauche, au centre et à droite) ;
- ceux de droite contrôleront sa position verticale (en haut, au centre et en bas)..

Ici, la position choisie, matérialisée par les boutons rouges, est à gauche et centrée verticalement, comme on peut le voir sur l'image. La position change quand l'utilisateur clique sur l'un des boutons.

Pour réaliser l'application, nous ne modifierons ni **FenetreSimple.java** ni **EtiquetteSimple.java**. Nous allons créer une nouvelle classe qui utilisera nos travaux précédents.

### Exo 2.4.1 - Création d'une barre d'outils

[\[Corrigé\]](#)

Créez une classe publique **ControleEtiquette**, qui étend **JToolBar**.

Déclarez les données membres privées suivantes :

- un tableau de 18 icônes (classe **ImageIcon**) nommé **iconesBoutons** ;
- un tableau de 6 boutons radios (classe **JRadioButton**) nommé **boutons**.

N'oubliez pas de créer les tableaux (avec **new**) directement dans leur déclaration. Nous les remplirons plus tard.

La classe **ControleEtiquette** doit aussi comporter deux méthodes privées :

- **private void chargerIcones()**
- **private void creerBoutons()**

ainsi qu'un constructeur public qui ne prend aucun argument, et la très classique méthode **main()**.

Mettez tous ces éléments en place, les corps des méthodes seront pour le moment vides, et vont être

progressivement remplis par la suite.

## Exo 2.4.2 - Charger les icônes

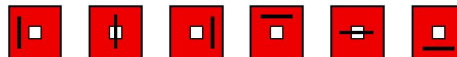
[\[Corrigé\]](#)

Pour personnaliser les boutons radios de la palette d'outils, nous allons avoir besoin de 18 icônes qui vont servir à animer les boutons radio. Il y a en fait 3 groupes de 6 icônes :

- le groupe principal composé des icônes représentant les versions non sélectionnées des boutons : `bhgauche.gif`, `bhcentre.gif`, `bhdroite.gif`, `bvhaut.gif`, `bvcentre.gif`, `bvbas.gif`. Ces images sont dans l'ordre :



- le groupe d'icônes rouges représentant un bouton sélectionné : les noms de fichiers sont similaires, mais comportent un `R` avant le point qui sépare le nom du fichier de l'extension (`bhgauche.gif` devient `bhgaucheR.gif`). Ces images sont, dans l'ordre :



- le groupe d'icônes bleues qui permettront de donner un effet de rollover, c'est à dire lorsque le pointeur de souris passe sur un bouton : la convention utilisée dans les noms est semblable, le `R` est remplacé par un `B`. Ces images sont, dans l'ordre :



Ecrivez le corps de la méthode `chargerIcones()` qui doit charger dans `iconesBoutons` les trois groupes d'icônes. Ils seront stockés à la suite dans le tableau, on aura donc dans l'ordre : les 6 icônes de base, les 6 icônes rouges, et les 6 icônes bleues.

Pour faciliter l'écriture de la méthode, déclarez auparavant la donnée membre privée suivante :

```
private final String nomsIcones[] = { "bhgauche", "bhcentre", "bhdroite",
                                     "bvhaut",    "bvcentre", "bvbas" };
```

## Exo 2.4.3 - Création des boutons radio

[\[Corrigé\]](#)

Ecrivez le corps de la méthode `creerBoutons()` qui doit remplir la table des 6 boutons radios (`JRadioButton`), chacun étant composé d'une icône de base (représentant la version non sélectionnée du bouton), sans texte. Fixez également, pour chaque bouton radio, l'icône qui doit s'afficher lorsque le bouton est sélectionné : vous utiliserez les icônes rouges pour cela.

## Exo 2.4.4 - Un premier constructeur...

[\[Corrigé\]](#)

*Rappel.* En tant qu'extension de la classe `JToolBar`, le constructeur de `ContrôleEtiquette` fera automatiquement appel au constructeur par défaut de `JToolBar`, à moins que ce ne soit fait explicitement en utilisant la méthode `super(...)` en première instruction du constructeur. Dit autrement, si `super(...)` n'est pas la première instruction du constructeur, le compilateur ajoute en début l'instruction `super()` (sans argument). Un `ContrôleEtiquette` est un `JToolBar`.

Ecrivez le corps du constructeur de la classe qui se contente de charger les icônes, de construire les boutons, et d'ajouter à la barre d'outils, dans l'ordre, les trois premiers boutons, un séparateur, et les trois derniers boutons du tableau des boutons. Consultez la documentation de `JToolBar` pour réaliser ces opérations.

## Exo 2.4.5 - Le principal !

[\[Corrigé\]](#)



Le corps de la méthode `main()` se contente de créer une fenêtre simple, à laquelle on ajoutera un objet `ContrôleEtiquette`. Pensez à faire ajuster la taille du cadre et à le rendre visible. Compilez et exécutez `ContrôleEtiquette`. Vous devriez obtenir une fenêtre ressemblant à :



J'ai des boutons !!

Remarquez que si vous cliquez sur les boutons, ils changent de couleur : ils passent du blanc (non sélectionné) au rouge (sélectionné).

## Exo 2.4.6 - Une seule position à la fois, c'est mieux

[\[Corrigé\]](#)

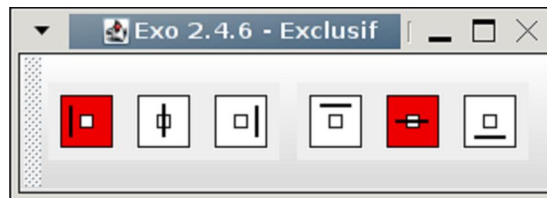
Vous avez sûrement remarqué qu'il est possible de sélectionner en même temps plusieurs boutons d'un même groupe. Or ce n'est pas ce que nous voulons puisque le texte ne peut avoir qu'une seule position horizontale et qu'une seule position verticale.

Nous allons donc mettre en place un pratique courante avec les boutons radios : **les rendre mutuellement exclusifs**. Ainsi, à tout moment, un seul bouton d'un même groupe peut être sélectionné.

Pour rendre exclusifs des boutons, il faut les ajouter à un objet `ButtonGroup`. Ainsi, la sélection d'un bouton d'un `ButtonGroup` entraîne la désélection des autres.

Modifiez le corps de la méthode `creerBoutons()` de manière à créer des objets `ButtonGroup` : le premier contiendra les trois premiers boutons, et le second les trois suivants. Notez que ces objets peuvent être déclarés localement dans la méthode. Pour autant, ils ne cesseront pas d'exister après son appel... Profitez-en pour fixer une position initiale telle que à gauche et centré verticalement, en sélectionnant les boutons correspondants.

Compilez et testez `ContrôleEtiquette`. Vous devriez obtenir initialement quelque chose comme :



et les boutons doivent être exclusifs.

## Exo 2.4.7 - Roll Over...

[\[Corrigé\]](#)

Swing permet de mettre en place un effet de **rollover**, que vous avez peut-être déjà rencontré si vous êtes un ou une adepte du javascript. Pour le moment nous avons spécifié l'apparence qu'un bouton doit prendre lorsqu'il est sélectionné : rouge. Indispensable pour rendre compte de l'état du groupe de boutons radios.

Le rollover est un effet similaire, dont le but est de permettre à l'utilisateur de reconnaître un composant actif, i.e. avec lequel il peut interagir. C'est le principe de *retour d'information*, votre interface doit dialoguer avec l'utilisateur, de manière implicite ou explicite. Dans le cas de boutons radios d'apparence courante (ronds) cette information est implicite, c'est une sorte de convention que l'on retrouve d'une plateforme à une autre.

Dans notre cas, les boutons ont une apparence personnalisée, le dessin porte en lui une information qui traduit sa fonction (si nous n'avons pas trop mal dessiné...). Nous allons ajouter un message visuel supplémentaire en provoquant une modification de la couleur du bouton lorsque le pointeur de souris passe dessus : c'est le **rollover**. Le bouton informe alors explicitement l'utilisateur qu'il est actif, qu'il peut interagir avec lui.

Les six dernières icônes bleues sont destinées à l'effet de rollover, modifiez votre méthode `creerBoutons()` de façon à ce qu'elle affecte cette icône à chaque bouton comme icône de rollover. Pour cela, consultez la

documentation de `JRadioButton` : il faut indiquer l'icône à utiliser pour le rollover sur un bouton normal, et sur un bouton sélectionné (on prendra dans les deux cas, l'icône bleue du bouton), et activer le rollover pour le bouton.

Activez également le rollover pour la barre d'outils (methode `setRollover(boolean b)` de la classe `JToolBar`) dans le constructeur.

Compilez et exécutez `ControleEtiquette`. Lorsque la souris passe sur un bouton, vous devriez obtenir quelque chose comme :



## Exo 2.4.8 - Ajout de l'étiquette

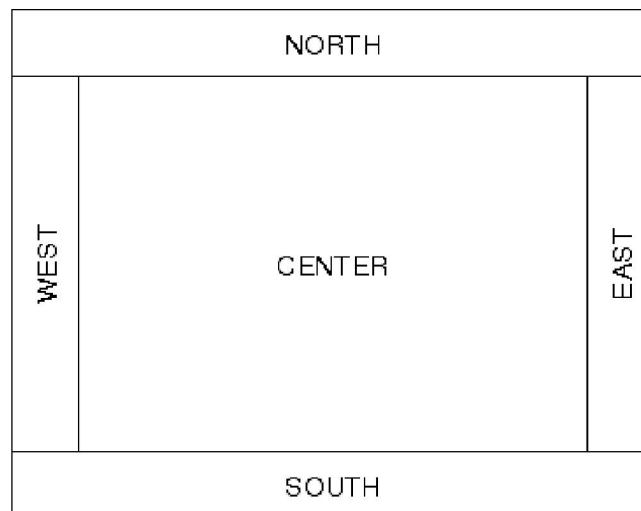
[\[Corrigé\]](#)

Il est temps d'ajouter l'étiquette à notre cadre. Celui-ci contiendra donc deux objets :

- un objet `ControleEtiquette` qui est la palette d'outils et qui doit se situer en haut ;
- un objet `EtiquetteSimple` qui est l'étiquette et qui doit se trouver en dessous.

Il faut savoir que la position et la taille d'un objet (un `Component`) ajouté dans un conteneur (`Container`) est régie par un **gestionnaire de disposition** (`LayoutManager`). Il est possible de fixer un gestionnaire de disposition particulier pour un conteneur. On peut même combiner plusieurs gestionnaires de disposition afin de fabriquer des interfaces complexes. Nous étudierons en détail ces gestionnaires au cours du prochain TP.

Le `LayoutManager` par défaut d'un `JFrame` est un `java.awt.BorderLayout`. Celui-ci décompose l'intérieur de la fenêtre en 5 zones (`NORTH`, `WEST`, `CENTER`, `EAST` et `SOUTH`) :



et redimensionne les objets pour qu'ils occupent la place disponible. Dans notre cas :

- la palette d'outils **doit** se trouver au nord : c'est l'emplacement privilégié d'une palette d'outils dans un `JFrame` ;
- l'étiquette **doit** se trouver au centre.

Pour fixer la position d'un objet, on utilisera la méthode :

```
public Component add(Component comp, int index);
```

de `JFrame` (héritée de `java.awt.Container`). L'`index` est la position de l'objet. Pour un `BorderLayout`, `index` doit être l'une des constantes suivantes de la classe `BorderLayout` :

- `BorderLayout.NORTH`
- `BorderLayout.WEST`
- `BorderLayout.CENTER`
- `BorderLayout.EAST`
- `BorderLayout.SOUTH`

Au début de `ControleEtiquette.java`, ajoutez l'instruction :

```
import java.awt.*;
```

afin d'utiliser les noms courts des classes de `AWT` (et en particulier de `BorderLayout`).

Puis, modifiez la méthode `main()` pour :

- fixer la position de la palette d'outils au nord du cadre ;
- créer une `EtiquetteSimple` composée du texte "Un chien" et l'image du chien (comme dans la méthode `main()` de `EtiquetteSimple.java`), en précisant que la position du texte doit être à gauche et centrée verticalement ;
- ajouter l'étiquette au centre du cadre.

Compilez et exécutez `ControleEtiquette.java`. Au final, vous devriez obtenir une fenêtre ressemblant à :



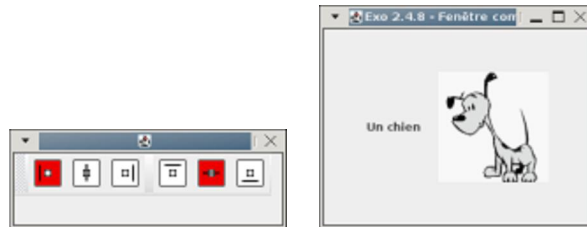
ça se précise...

Essayez de saisir la barre d'outils par la poignée située sur sa gauche et déplacez-la en suivant les bords intérieurs de la fenêtre. Vous pouvez la placer dans n'importe quelle solution en relâchant le bouton de la souris (si besoin redimensionnez la fenêtre pour que tous les boutons soient visibles) :



Plusieurs emplacements de la barre d'outils

Si vous sortez la palette de la fenêtre et relâchez le bouton de souris, vous obtenez une palette flottante (qu'on peut remettre dans la fenêtre principale) :



## Exo 2.4.9 - Pour finir, écouter les boutons

[\[Corrigé\]](#)

Nous avons presque fini. Il ne reste "plus qu'à" détecter la sélection d'un bouton et agir en conséquences.

Nous étudierons en détail au cours du prochain TP différentes techniques permettant de réagir à un événement produit sur une fenêtre, tel que le clic sur un bouton, le passage de la souris sur la fenêtre (*focus*) ou un objet graphique, etc.

Pour le moment, il vous suffit de savoir que la sélection d'un `JRadioButton` génère un événement pris en compte par le processus (en réalité le *thread*) qui gère la fenêtre. Celui-ci demande alors aux objets s'étant déclarés "écouteurs d'action" du bouton de réagir.

Plus précisément :

- un bouton peut avoir plusieurs écouteurs d'action qui devront réagir quand le bouton est actionné. On ajoute un écouteur d'actions à un bouton (notamment un `JRadioButton`) en appelant sa méthode :

```
public void addActionListener(ActionListener l)
```

héritée de la classe `javax.swing.AbstractButton` ;

- comme l'indique le profil de la méthode `addActionListener()`, un écouteur d'action est un objet (d'une classe) qui implémente l'interface `java.awt.event.ActionListener`. Ce faisant, il doit définir la méthode :

```
public void actionPerformed(ActionEvent evt)
```

qui sera invoquée (par le thread gérant la fenêtre) lorsque l'utilisateur sélectionne le bouton écouté. L'argument `evt` de cette méthode est un `java.awt.event.ActionEvent` et permet notamment de déterminer quel bouton a été sélectionné ; un écouteur d'action pouvant s'occuper de plusieurs boutons à la fois.

Nous ajouterons donc l'instruction :

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

en début de `ContrôleEtiquette.java` afin d'utiliser les noms courts `ActionListener` et `ActionEvent`.

Encore une fois, plusieurs solutions sont envisageables. Par, exemple, utiliser un seul écouteur pour tous les boutons de notre application. Cela nécessite de déterminer quel bouton a été sélectionné pour modifier correctement la position du texte. Nous étudierons cette possibilité dans un autre contexte au prochain TP.

La solution que nous allons adopter est de créer 6 écouteurs : un pour chaque bouton. L'avantage est que l'on peut ainsi associer une fonctionnalité précise à chaque écouteur : fixer une position horizontale donnée pour le texte, ou une position verticale donnée. Les écouteurs des boutons régissant la position horizontale devront modifier cette position en faisant appel à la méthode `setHorizontalTextPosition()` de l'étiquette, alors que ceux des boutons régissant la position verticale utiliseront `setVerticalTextPosition()`. Les écouteurs seront alors de deux nature : les écouteurs horizontaux et les écouteurs verticaux, ce qui donne lieu aux deux **classes internes** de `ContrôleEtiquette` suivantes :

```
...
```

```
public class ContrôleEtiquette extends JToolBar {
```

```
...
```

```
class EcouteurHorizontal implements ... {
    private int position;
```

```
    EcouteurHorizontal(int position) {
```

```

        ...
    }

    public void actionPerformed(ActionEvent evt) {
        ...
    }
}

class EcouteurVertical implements ... {
    private int position;

    EcouteurVertical(int position) {
        ...
    }

    public void actionPerformed(ActionEvent evt) {
        ...
    }
}

...
}

```

Dans ces classes, la donnée membre `position` est la position (horizontale ou verticale) que doit prendre le texte de l'étiquette. Elle sera passée en paramètre du constructeur lors de la création de l'écouteur.

On créera les écouteurs dans la méthode privée de `ContrôleEtiquette` :

```

private void creerEcouteurs() {
    ...
}

```

qui crée les trois écouteurs horizontaux et les trois écouteurs verticaux en leur indiquant la position à utiliser. Il est aussi judicieux de déclarer dans `ContrôleEtiquette` un tableau privé des positions représentées par les boutons :

```

private final int positions[] = { JLabel.LEFT, JLabel.CENTER, JLabel.RIGHT,    // horizontales
                                  JLabel.TOP,  JLabel.CENTER, JLabel.BOTTOM }; // verticales

```

Finalement, pour que les écouteurs puissent modifier la position du texte de l'étiquette, cette dernière doit leur être accessible. Puisque les classes internes ont accès à toutes les données et méthodes membres de la classe englobante, il suffit de créer la donnée membre privée suivante dans `ContrôleEtiquette` :

```

private EtiquetteSimple etiquette;

```

L'étiquette en question sera passée en paramètre d'un **nouveau constructeur** de `ContrôleEtiquette` :

```

public ContrôleEtiquette(EtiquetteSimple etiquette) {
    ...
}

```

qui fera appel à l'ancien constructeur (instruction `this()` ;), initialisera la donnée membre `etiquette` et créera les écouteurs. L'ancien constructeur n'ayant plus de raison de rester public peut maintenant être déclaré privé.

Modifiez `ContrôleEtiquette.java` pour mettre en place cette solution.

---

[La suite des TP...](#)