www.jeanjoux.fr

# Tutoriel Gtk2Hs

**Jean-Luc JOULIN**

**27 sept. 2014**

# Table des matières

# Tutoriel Gtk2Hs - License

## 1. Informations sur la licence

# Tutoriel Gtk2Hs 1 - Introduction

Gtk2Hs est une librairie graphique GUI pour Haskell basée sur Gtk+. Gtk+ est une boite à outils extensible, évoluée et multi-plateforme pour créer des interfaces graphiques.

Ce tutoriel est destiné à un programmeur Haskell de niveau intermédiaire et couvre ce qui est requis pour écrire les interfaces graphiques les plus classiques. Ceci est résumé dans la table des matières de ce tutoriel. Pour les utilisateurs avancés, la documentation de Gtk2Hs contient plus d'informations.

Gtk+ est une librairie de composants appelés "widgets", qui sont organisés dans un classement orienté objet. Dans Gtk2Hs, cela est implémenté en donnant à chaque widget à la fois un type Haskell et une classe Haskell. De cette façon, le système de type de Haskell est préservé et le programmeur a tous les avantages de la vérification du typage par le compilateur GHC et l'interpréteur. Gtk2Hs possède également des fonctions pour la conversion de types (casting).

Les widgets ont des attributs qui contrôlent leur comportement et leur apparence. Il y a deux fonctions générales pour définir et récupérer les attributs.

Pour définir un ou plusieurs attributs d'un widget, on utilise :

```
set widget [widgetAttr1 := foo, widgetAttr2 := fie, widgetAttr3 := bar]
```

Pour obtenir un attribut d'un widget, on utilise :

```
x <- get widget widgetAttr1
```

Il existe également des fonctions spécifiques pour fixer et obtenir les attributs, mais à l'avenir ces fonctions peuvent devenir obsolètes.

Le fonctionnement d'une application Gtk2Hs est déterminé par les actions de l'utilisateur telles que, cliquer sur la souris, appuyer sur une touche, sélectionner un bouton, etc… Ces actions sont appelées "événements" (events) et peuvent se matérialiser par l'émission d'un "signal". L'information sur les événements, par exemple le clique gauche ou droite à la souris , peut être capturée dans des champs de données qui peuvent être récupérées par le programmeur. Les signaux sont habituellement utilisés par des fonctions spécifiques pour ce type de widget. Elles sont listées dans la section **Signaux** de la documentation pour ce widget. La plupart de ces fonctions ont pour type IO () comme paramètres, mais certaines peuvent retourner des résultats. Le programmeur Gtk2Hs doit donc écrire ces fonctions qui déterminent le comportement du programme en réponse à l'action de l'utilisateur.

Enfin, définir l'organisation des widgets dans une application et déterminer l'apparence visuelle peut être fait graphiquement avec l'éditeur d'interface Glade. Gtk2Hs supporte intégralement Glade. Une introduction à Glade se trouve sur le site web de Gtk2Hs. Il est recommandé d'utiliser Glade lors de la programmation d'interfaces graphiques avec Gtk2Hs.

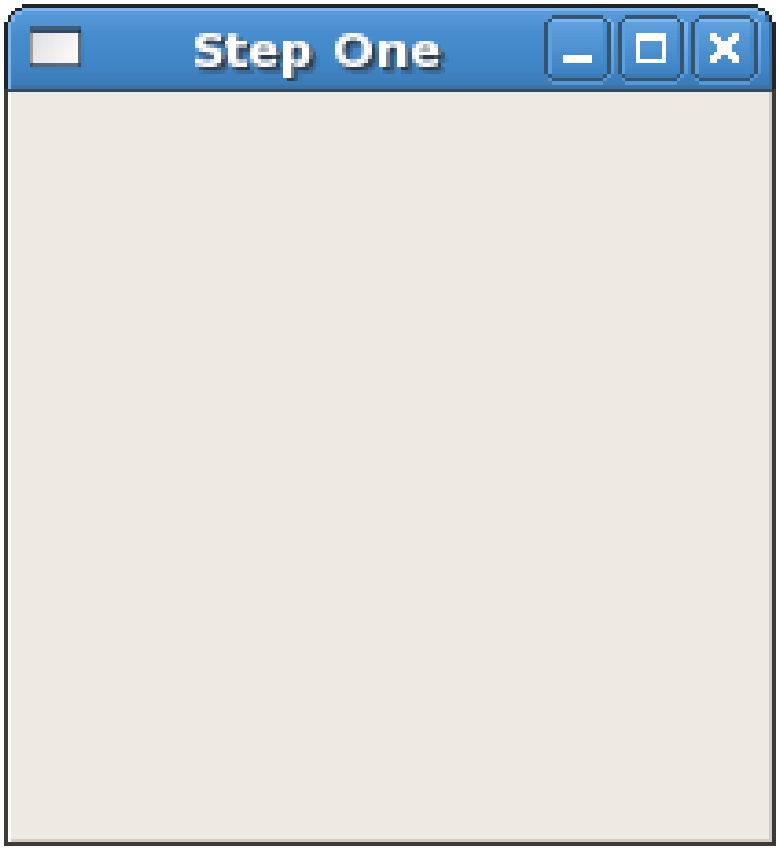
# Tutoriel Gtk2Hs 2 - Démarrage

## 1. Démarrage

La première chose à faire est de télécharger et d'installer Gtk2Hs. Vous pouvez obtenir la dernière version à l'adresse http ://projects.language-haskell.org/gtk2hs La plupart des distributions Linux ont leurs propres paquets de Gtk2Hs. Pour Windows, un installateur est disponible.

La deuxième chose à faire est d'ouvrir la documentation de référence de l'interface de Gtk2Hs pour cette version. Vous aurez besoin de l'utiliser de façon constante pour trouver le nom des widgets, des méthodes, des attributs et des signaux que vous voudrez utiliser. Le contenu liste tous les modules et il y a aussi un index. A l'intérieur de chaque description d'un module, il y a également une hiérarchie des classes. Si une méthode, un attribut ou un signal que vous cherchez manque, il peut appartenir à une des superclasses dont le widget est dérivé.

Pour commencer notre introduction à Gtk2Hs, nous allons faire le programme le plus simple qui existe. Ce programme va créer une fenêtre de 200x200 pixels qui ne possède pas de fonctions pour s'arrêter (à moins d'être "tuée" depuis un Shell).




```haskell
import Graphics.UI.Gtk
main :: IO ()
main = do
  initGUI
  window <- windowNew
  widgetShowAll window
  mainGUI
```

Vous pouvez compiler le programme ci-dessus avec le compilateur GHC (Glasgow Haskell Compiler) avec :

```
ghc --make GtkChap2.hs -o Chap2
```

En supposant que le fichier s'appelle GtkChap2.hs.Il est également possible d'utiliser GHCI en interactif avec les versions récentes de Gtk2Hs.

La première ligne du programme importe la librairie graphique Gtk2Hs.

Tous les programmes Gtk2Hs se lancent dans main. La première ligne du bloc do de cette fonction :

```
initGUI
```

est une fonction appelée dans toutes les applications Gtk2Hs.

Les deux lignes suivantes du bloc créent et affichent une fenêtre et son contenu :

```
window <- windowNew
widgetShowAll window
```

Plutôt que de créer une fenêtre d'une taille de 0x0, une fenêtre sans "enfants" (children) est définie à 200x200 par défaut de façon À pouvoir quand même la manipuler. Les widgets qui peuvent s'afficher (tous les widgets ne le peuvent pas) peuvent être affichés ou cachés en utilisant leurs propres méthodes, mais la seconde ligne s'applique sur un widget (ici une fenêtre) et tous ses enfants.

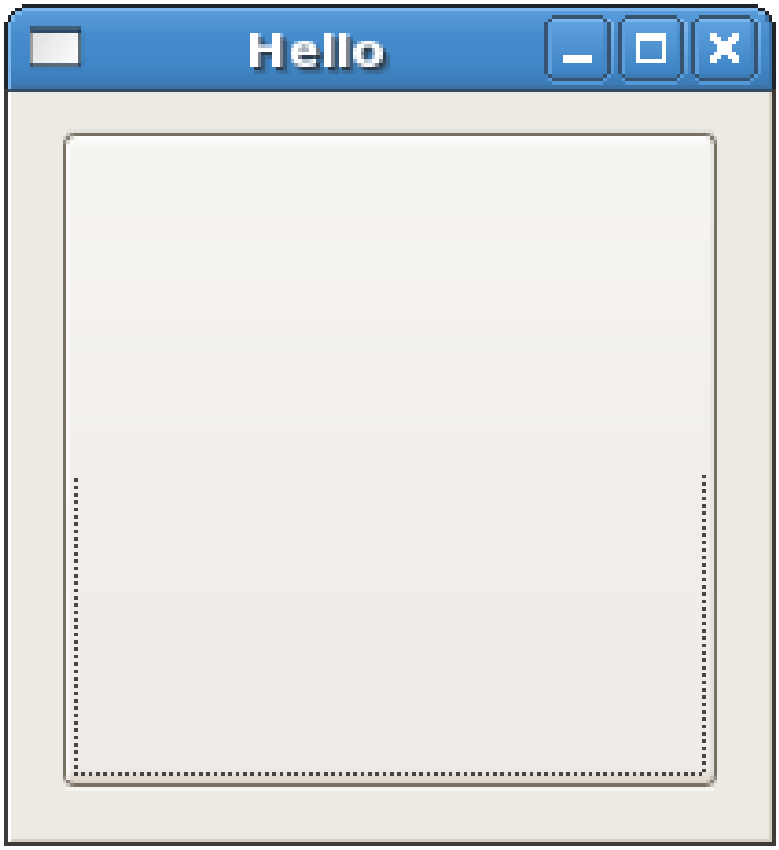
La dernière ligne de main lance la boucle de fonctionnement principale de Gtk2Hs :

```
mainGUI
```

Il s'agit d'un autre appel que l'on retrouve dans toutes les applications Gtk2Hs. Quand le programme atteint ce point, Gtk2Hs se met en attente d'événements X (clic souris, appui sur une touche, ...), délais ou notifications d'entrées-sorties. Dans cet exemple, tous les événements sont ignorés.

# 2. "Hello World" avec Gtk2Hs

Maintenant, faisons un programme avec un widget (un bouton) : Le traditionnel "hello world" façon Gtk2Hs.




Si on clique sur le bouton, cela affichera le texte "Hello World". Cela est implémenté dans Haskell avec la fonction hello avec un bouton b comme argument. Le type de la variable o est une instance de la class

ButtonClass.Gtk2Hs utilise énormément les classes de Haskell pour reproduire la hiérarchie des widgets de Gtk+. Bien sur, chaque widget de Gtk2Hs possède un type Haskell.

Les widgets et les classes auxquelles ils appartiennent ont généralement des attributs. Ils peuvent être définis soit par des méthodes nommées ou par la fonction générale set, qui utilise une notation ayant l'apparence d'une liste telle que montrée ci-dessous. Il est intéressant de noter l'attribut containerChild de la fenêtre qui définit la relation avec le bouton. Grâce à cette relation widgetShowAll window rendra également le bouton visible.

Les widgets sont connectés aux autres widgets dans un arbre de dépendance graphique (à ne pas confondre avec la hiérarchie des classes).Gtk2Hs fonctionne également avec le modeleur d'interface Glade. Si vous l'utilisez vous pourrez voir ces relations entre les widgets avec le panneau **Inspector**. Un tutoriel d'initiation est disponible pour utiliser Glade avec Gtk2Hs.

```
import Graphics.UI.Gtk

hello :: (ButtonClass o) => o -> IO ()
hello b = set b [buttonLabel := "Hello World"]

main :: IO ()
main = do
  initGUI
  window <- windowNew
  button <- buttonNew
  set window [windowDefaultWidth := 200, windowDefaultHeight := 200,
              containerChild := button, containerBorderWidth := 10]
  onClicked button (hello button)
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI
```

Gtk2Hs est piloté par événements. La fonction `mainGUI` se met en sommeil jusqu'à ce que quelque chose se passe, comme un clic souris, une destruction de fenêtre ou quelque chose d'autre spécifique à chaque widget. Ces événements déclenchent des signaux qui à leur tour déclenchent l'évaluation des fonctions définies par le programmeur. Dans le cas présent le signal onClicked, émis lorsque l'utilisateur clique sur le bouton est lié à l'affichage du test sur ce même bouton. Quand l'utilisateur détruit la fenêtre (contrairement au premier programme), main se termine proprement.

# Tutoriel Gtk2Hs 3.1 - Empaquetage des widgets

Lorsque vous voudrez créer une application, vous voudrez mettre plus qu'un seul widget dans une fenêtre. Notre premier exemple "hello world" utilise seulement un widget et on pouvait simplement utiliser set pour spécifier le containerChild widget pour window, ou utiliser `containerAdd` pour "empaqueter" le widget dans la fenêtre. Mais lorsque vous voudrez mettre plus qu'un seul widget dans une fenêtre, comment contrôler où ces widgets seront positionnés ? C'est là qu'intervient : l'empaquetage.

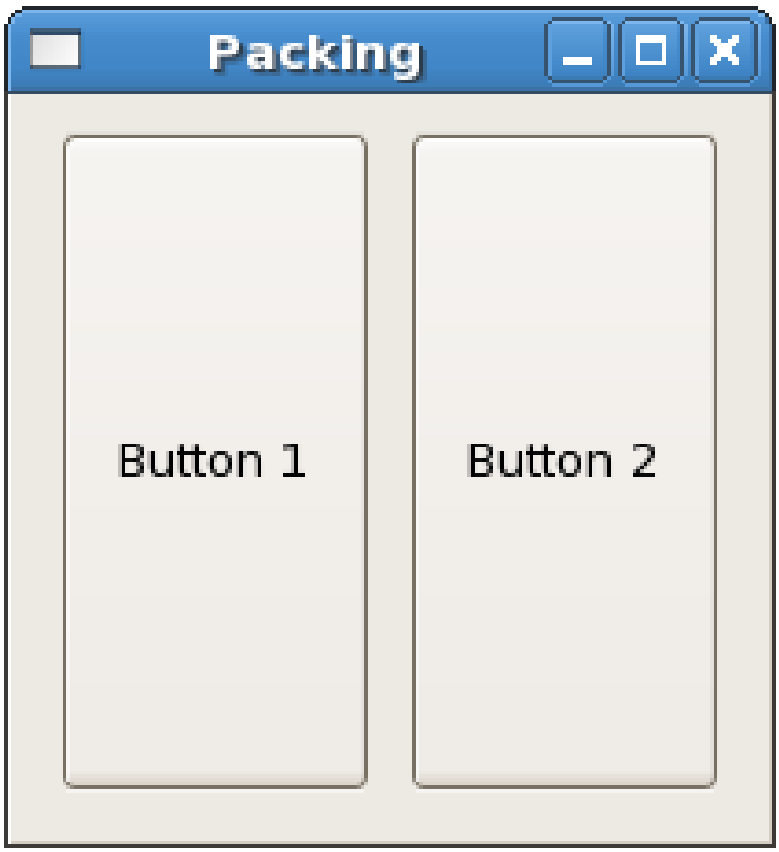
## 1. Théorie de l'empaquetage des boites

La plupart des empaquetages sont réalisés en créant des boites. Ce sont des widgets "conteneurs" invisibles que l'on peut remplir avec nos widgets et qui sont disponibles sous deux formes : une boite horizontale ou verticale. Lorsque l'on empaquette les widgets dans une boite horizontale, les objets sont insérés de gauche à droite ou de droite à gauche suivant l'appel utilisé. Dans une boite verticale, les widgets sont empaquetés du haut vers le bas ou inversement. On peut utiliser n'importe quelle combinaison de boites à l'intérieur ou à l'extérieur d'autres boites pour créer l'effet désiré.

Pour créer une boite horizontale, on utilise `hBoxNew`, et pour une boite verticale `vBoxNew`. Chaque fonction prend comme argument un `Bool` et un `Int`. Le premier paramètre donne à tous les enfants un espacement égal s'il est défini à True et le deuxième donne l'espace en pixels par défaut entre les enfants.

Les fonctions `boxPackStart` et boxPackEnd sont utilisées pour placer des objets à l'intérieur de ces conteneurs. La fonction `boxPackStart` commence en haut et va vers le bas dans une `VBox`, commence à gauche et va vers la droite dans une `HBox`. L'utilisation de ces fonctions permet de justifier à droite ou à gauche et peuvent être combinées de n'importe quelle manière pour obtenir l'effet escompté. Dans la plupart des exemples, nous utiliserons boxPackStart.

Un objet peut être un autre conteneur ou un widget. Dans la réalité, beaucoup de widgets sont des conteneurs eux-mêmes, comme button, mais on utilise généralement seulement un label à l'intérieur d'un button.




```haskell
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window  <- windowNew
  hbox    <- hBoxNew True 10
  button1 <- buttonNewWithLabel "Button 1"
  button2 <- buttonNewWithLabel "Button 2"
  set window [windowDefaultWidth := 200, windowDefaultHeight := 200,
              containerBorderWidth := 10, containerChild := hbox]
  boxPackStart hbox button1 PackGrow 0
  boxPackStart hbox button2 PackGrow 0
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI
```

En utilisant `boxPackStart` ou `boxPackEnd`, Gtk sait où vous voulez placer votre widget, il peut alors modifier leur taille automatiquement et d'autres choses sympathiques.

```haskell
boxPackStart :: (WidgetClass child, BoxClass self) => self -> child -> Packing -> Int -> IO ()
```

```haskell
boxPackEnd :: (WidgetClass child, BoxClass self) => self -> child -> Packing -> Int -> IO ()
```

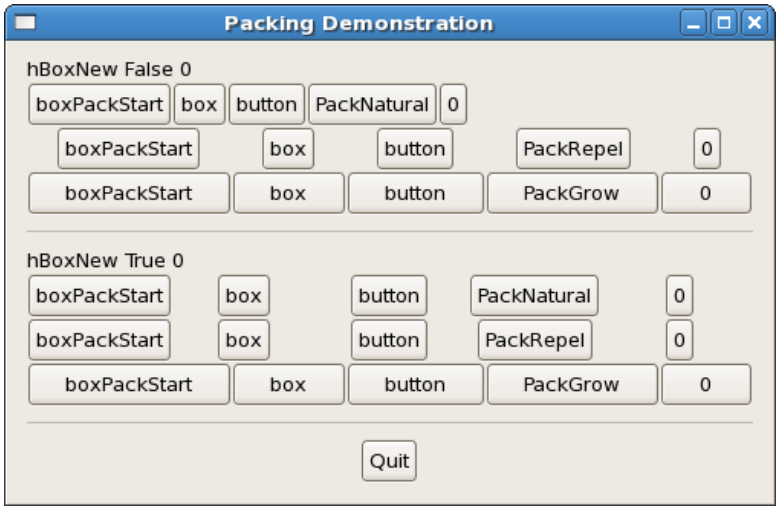
Le paramètre `Packing` spécifie la façon dont les widgets dans le conteneur se comportent quand la fenêtre est redimensionnée. `PackNatural` signifie que les widgets garderont leur taille et resteront ou ils se trouvent, `PackGrow` signifie qu'ils seront redimensionnés, et `PackRepel` signifie que les widgets seront également répartis des deux cotés. Le dernier paramètre est un Int qui spécifie l'espace additionnel à mettre entre cet enfant et ses voisins.

Notez que Packing s'applique uniquement à la dimension de la boite. Si, par exemple, vous spécifiez `PackNatural` au lieu de `PackGrow` dans l'exemple ci-dessus, le redimensionnement horizontal gardera les boutons à leur place originale, mais le redimensionnement vertical va changer la taille des boutons. Ceci est dû au fait que les boutons sont placés de façon homogène dans la boite horizontale (Le premier paramètre est True) et la boite elle-même sera redimensionnée avec la fenêtre. L'exemple qui va suivre montrera cela de façon plus claire.

# Tutoriel Gtk2Hs 3.2 - Programme de démonstration de l'empaquetage

La base de tous les widgets montrés ici est une boite verticale, qui est elle-même un enfant de la fenêtre. Les widgets enfants ne sont pas affichés de façon homogène et il n'y a pas d'espace additionnel (autre que l'espace standard). Il y a six boites horizontales dans la boite verticale, définie par la fonction `makeBox`. En outre, il y a deux labels dans la boite verticale ainsi que deux séparateurs. Le dernier widget est le bouton **Quit** dont le signal `onClicked` est connecté à la fonction `mainQuit`.

Les séparateurs sont créés avec `hSeparatorNew` et ils sont espacés avec `boxPackStart` avec un espacement de dix pixels. Les labels sont créés avec `labelNew` qui prend comme argument `Maybe String` et leur positionnement est défini avec `miscSetAlignment` pour être justifié en haut à gauche.

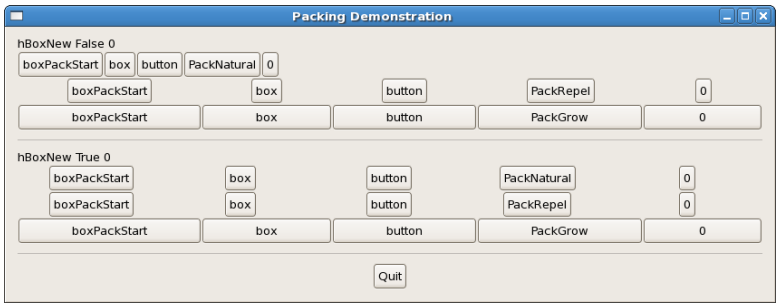



La fonction `makeBox :: Bool -> Int -> Packing -> Int -> IO HBox` montre comment les widgets Gtk2Hs s'adaptent au système de types de Haskell. `Packing` est juste un type comme `Int` et `Bool` et `IO HBox` est juste comme `IO String` dans La fonction crée cinq boutons, les labellise avec le texte approprié et les empilent dans une boite horizontale. La fonction est alors utilisée dans le programme principal pour créer l'empaquetage de la manière souhaitée.

```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window     <- windowNew
  vbox       <- vBoxNew False 0
  set window [containerBorderWidth := 10,
              windowTitle := "Packing Demonstration",
              containerChild := vbox]
  label1     <- labelNew (Just "hBoxNew False 0")
  miscSetAlignment label1 0 0
  boxPackStart vbox label1 PackNatural 0
  box1       <- makeBox False 0 PackNatural 0
  boxPackStart vbox box1 PackNatural 0
  box2       <- makeBox False 0 PackRepel 0
  boxPackStart vbox box2 PackNatural 0
  box3       <- makeBox False 0 PackGrow 0
  boxPackStart vbox box3 PackNatural 0
  sep1       <- hSeparatorNew
  boxPackStart vbox sep1 PackNatural 10
```

```
    label2     <- labelNew (Just "hBoxNew True 0")
    miscSetAlignment label2 0 0
    boxPackStart vbox label2 PackNatural 0
    box4       <- makeBox True 0 PackNatural 0
    boxPackStart vbox box4 PackNatural 0
    box5       <- makeBox True 0 PackRepel 0
    boxPackStart vbox box5 PackNatural 0
    box6       <- makeBox False 0 PackGrow 0
    boxPackStart vbox box6 PackNatural 0
    sep        <- hSeparatorNew
    boxPackStart vbox sep PackNatural 10
    quitbox    <- hBoxNew False 0
    boxPackStart vbox quitbox PackNatural 0
    quitbutton <- buttonNewWithLabel "Quit"
    boxPackStart quitbox quitbutton PackRepel 0
    onClicked quitbutton mainQuit
    onDestroy window mainQuit
    widgetShowAll window
    mainGUI


makeBox :: Bool -> Int -> Packing -> Int -> IO HBox
makeBox homogeneous spacing packing padding = do
    box     <- hBoxNew homogeneous spacing
    button1 <- buttonNewWithLabel "boxPackStart"
    boxPackStart box button1 packing padding
    button2 <- buttonNewWithLabel "box"
    boxPackStart box button2 packing padding
    button3 <- buttonNewWithLabel "button"
    boxPackStart box button3 packing padding
    button4 <- case packing of
                 PackNatural -> buttonNewWithLabel "PackNatural"
                 PackRepel   -> buttonNewWithLabel "PackRepel"
                 PackGrow    -> buttonNewWithLabel "PackGrow"
    boxPackStart box button4 packing padding
    button5 <- buttonNewWithLabel (show padding)
    boxPackStart box button5 packing padding
    return box
```

L'image suivante montre les effets du redimensionnement horizontal de la fenêtre. Dans le premier groupe avec homogeneous définis à False, le redimensionnement horizontal laisse la première ligne de boutons telle qu'elle est, les espaces de la seconde uniformes et élargit les boutons de la troisième ligne. Dans le deuxième groupe, les boutons sont réglés pour être empaquetés de façon homogène et les deux premières lignes se ressemblent. Redimensionner la fenêtre verticalement ajoute seulement un espace additionnel à la fin, car la boite verticale a été initialisée à False.



# Tutoriel Gtk2Hs 3.3 - Empaquetage en utilisant des grilles

Voyons maintenant une autre façon d'empaqueter : **les grilles**. Elles peuvent être extrêmement utiles dans certaines situations. En utilisant les grilles, on crée une grille sur laquelle on peut placer les widgets. Les widgets prennent la place qu'on leur alloue.

La première chose à voir, est la fonction tableNew :

```
tableNew :: Int -> Int -> Bool -> IO Table
```

Le premier argument est le nombre de lignes à créer dans la grille, alors que le second est le nombre de colonnes.

L'argument booléen homogeneous traite de la façon dont les boites de la grille sont dimensionnées. Si homogeneous est mis à `True`, les boites de la grille sont redimensionnées à la taille du widget le plus large de la grille. Si homogeneous et mis à False, la taille d'une boite de la grille est imposée par le widget le plus grand dans la même ligne et le plus large dans la même colonne.

Les lignes et les colonnes sont numérotées de 0 à n ou n est le numéro spécifié dans l'appel à tableNew. Donc si vous spécifiez rows = 2 et columns = 2, l'agencement ressemblera à quelque chose comme cela :

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

Notez que le système de coordonnées commence dans le coin en haut à gauche. Pour placer un widget dans une boite, utilisez la fonction suivante :

```
tableAttach :: (TableClass self, WidgetClass child)
  => self             -- self         - The table.
  -> child            -- child        - The widget to add.
  -> Int              -- leftAttach   - The column number to attach the left
                      --                side of a child widget to.
  -> Int              -- rightAttach  - The column number to attach the right
                      --                side of a child widget to.
  -> Int              -- topAttach    - The row number to attach the top of a
                      --                child widget to.
  -> Int              -- bottomAttach - The row number to attach the bottom
                      --                of a child widget to.
  -> [AttachOptions]  -- xoptions     - Used to specify the properties of the
                      --                child widget when the table is
                      --                resized.
  -> [AttachOptions]  -- yoptions     - The same as xoptions, except this
                      --                field determines behaviour of
                      --                vertical resizing.
  -> Int              -- xpadding     - An integer value specifying the
                      --                padding on the left and right of the
                      --                widget being added to the table.
  -> Int              -- ypadding     - The amount of padding above and below
                      --                the child widget.
  -> IO ()
```

Le premier argument self est la grille que vous avez créée et le second child est le widget que vous souhaitez placer dans la grille.

Les arguments `leftAttach` et `rightAttach` spécifient où placer le widget et combien de boites à utiliser. Si vous voulez un bouton dans le logement en bas à droite de notre grille 2x2 et qu'il remplisse ce logement **seulement**, leftAttach devra être = 1, rightAttach =2, topAttach = 1, et bottomAttach= 2.

Maintenant, si vous vouliez qu'un widget prenne toute la ligne du haut de notre grille 2x2, vous devrez utiliser leftAttach = 0, rightAttach = 2, topAttach = 0, et bottomAttach = 1.

Les xoptions et yoptions sont utilisés pour spécifier les options d'empaquetage et la liste peut contenir plusieurs options à utiliser.

Ces options sont :

`Fill` : Si la boite de la grille est plus large que le widget et que Fill est spécifié, le widget va s'agrandir pour utiliser tout l'espace disponible.

`Shrink` : Si le widget de la grille prend moins d'espace qu'il n'en demande (typiquement après un redimensionnement de fenêtre), alors le widget devrait normalement être poussé hors de la fenêtre et disparaître. Si Shrink est spécifié, les widgets vont rétrécir avec la grille.

`Expand` : Ceci poussera la grille à s'agrandir pour utiliser tout l'espace restant dans la fenêtre.

L'espacement est exactement comme dans les boites, il créé un espace vide (en pixels) autour du widget tableAttach a plusieurs options, voici un extrait :

```
tableAttachDefaults :: (TableClass self, WidgetClass widget)
  => self    -- self          - The table.
  -> widget  -- widget        - The child widget to add.
  -> Int     -- leftAttach    - The column number to attach the left side of
             --                 the child widget to.
  -> Int     -- rightAttach   - The column number to attach the right side of
             --                 the child widget to.
  -> Int     -- topAttach     - The row number to attach the top of the child
             --                 widget to.
  -> Int     -- bottomAttach  - The row number to attach the bottom of the
             --                 child widget to.
  -> IO ()
```

Les valeurs utilisées pour les paramètres [AttachOptions] sont [Expand, Fill], et l'espacement est mis à 0. Le reste des arguments est identique à la fonction précédente.

Il y a aussi `tableSetRowSpacing` et `tableSetColSpacing`. Il place de l'espace dans les lignes et les colonnes désirées.

```
tableSetRowSpacing :: TableClass self=> self-> Int -> Int -> IO ()
```

```
tableSetColSpacing :: TableClass self => self -> Int -> Int -> IO ()
```

Le premier argument Int est le numéro de ligne (ou de colonne) et le second est l'espace en pixels. Notez que pour les colonnes, l'espace va à droite de la colonne et pour les lignes l'espace va en dessous de la ligne.

On peut également définir un espacement constant pour toutes les lignes (ou colonnes) avec :

```
tableSetRowSpacings :: TableClass self=> self-> Int -> IO ()
```

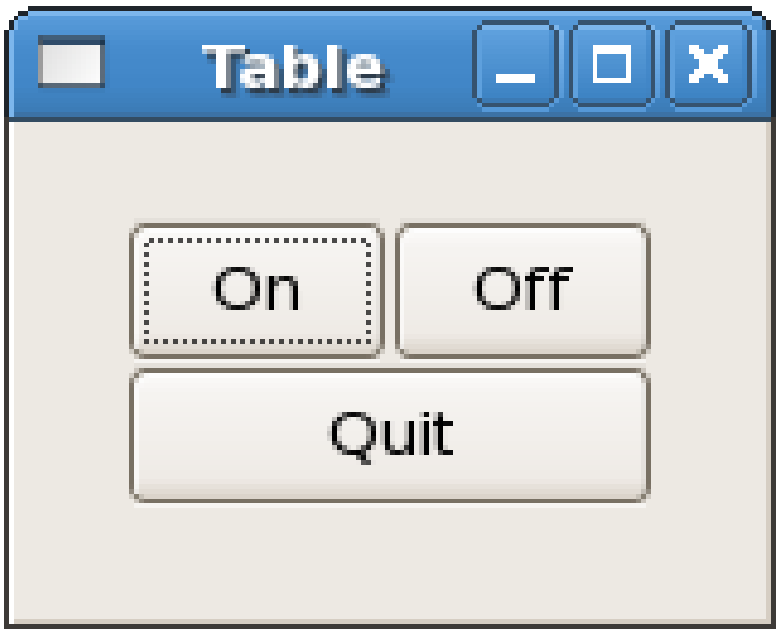
et :

```
tableSetColSpacings :: TableClass self => self -> Int -> IO ()
```

Notez qu'avec ces appels, la dernière ligne et la dernière colonne n'ont pas d'espace.

# 1. Exemple d'empaquetage avec une grille

Maintenant, nous allons faire trois boutons dans une grille 2x2. Les deux premiers boutons seront placés dans la ligne du haut. Un troisième bouton (**Quit**) sera placé dans la ligne du bas, occupant deux colonnes, ce qui ressemblera à quelque chose comme ça :




Voici le code source :

```
import Graphics.UI.Gtk


main :: IO ()
main = do
  initGUI
  window  <- windowNew
  set window [windowTitle := "Table", containerBorderWidth := 20,
              windowDefaultWidth := 150, windowDefaultHeight := 100]
  table   <- tableNew 2 2 True
  containerAdd window table
  button1 <- buttonNewWithLabel "On"
  onClicked button1 (buttonSwitch button1)
  tableAttachDefaults table button1 0 1 0 1
  button2 <- buttonNewWithLabel "Off"
  onClicked button2 (buttonSwitch button2)
  tableAttachDefaults table button2 1 2 0 1
  button3 <- buttonNewWithLabel "Quit"
  onClicked button3 mainQuit
  tableAttachDefaults table button3 0 2 1 2
  onDestroy window mainQuit
  widgetShowAll window
  mainGUI

buttonSwitch :: Button -> IO ()
buttonSwitch b = do
  txt <- buttonGetLabel b
  let newtxt = case txt of
                 "Off" -> "On"
                 "On"  -> "Off"
  buttonSetLabel b newtxt
```

La fonction `buttonSwitch` est connectée aux deux boutons de la ligne du haut. La fonction `buttonGetLabel` est un exemple de la façon d'obtenir les attributs d'un widget. Il y a également une alternative plus générale avec get qui prend un widget et un attribut comme argument. Dans l'exemple ci-dessus, cela serait :

```
txt <- get b buttonLabel
```

Avec le même résultat.

# Tutoriel Gtk2Hs 4.1 - Le widget bouton

## 1. Boutons normaux

Nous avons vu presque tout ce qu'il y a à voir sur le bouton widget. C'est assez simple. Cependant, il y a plus d'une façon de créer un bouton. Vous pouvez voir `buttonNewWithLabel` ou `buttonNewWithMnemonic` pour créer un bouton avec un label, utiliser `buttonNewFromStock` pour créer un bouton contenant une image et un texte, ou utiliser `buttonNew` pour créer un bouton vide. Libre à vous ensuite d'empaqueter ou dessiner dans ce nouveau bouton. Pour faire cela, créez une nouvelle boite, et empaquetez vos objets dans cette boite en utilisant `boxPackStart` (ou boxPackEnd), et ensuite utilisez `containerAdd` pour empaqueter le bouton dans la boite.

`buttonNewWithMnemonic` et `buttonNewFromStock` prennent tous les deux une chaîne de caractère comme premier argument. Utilisez le caractère underscore pour marquer le caractère comme mnémonique (raccourcis au clavier). En appuyant sur **Alt** et cette touche, cela active le bouton. Dans la seconde fonction, la chaîne est un `stockId`, c'est à dire l'identifiant d'une liste prédéfinie d'images avec des labels.

Voici un exemple d'utilisation de `buttonNew` pour créer un bouton avec une image et un label.


```haskell
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  set window [windowTitle := "Pix",
              containerBorderWidth := 10]
  button <- buttonNew
  onClicked button (putStrLn "button clicked")
  box    <- labelBox "info.xpm" "cool button"
  containerAdd button box
  containerAdd window button
  widgetShowAll window
  onDestroy window mainQuit
  mainGUI

labelBox :: FilePath -> String -> IO HBox
labelBox fn txt = do
  box   <- hBoxNew False 0
  set box [containerBorderWidth := 2]
  image <- imageNewFromFile fn
  label <- labelNew (Just txt)
  boxPackStart box image PackNatural 3
  boxPackStart box label PackNatural 3
  return box
```

La fonction labelBox peut être utilisée pour empaqueter des images et des labels dans n'importe quel widget de type conteneur. L'image provient d'un fichier en utilisant `imageNewFromFile` et le label vient de labelNew qui prend comme argument Maybe String. Nothing signifie pas de label.

Le widget Button possède les signaux suivant, qui sont assez explicites :

**`onPressed`** : émis lorsque le bouton est pressé.

**`onReleased`** : émis lorsque le bouton est relâché.

**`onClicked`** : émis lorsque le bouton est pressé puis relâché.

**`onEnter`** : émis lorsque le curseur passe sur le bouton.

**`onLeave`** : émis lorsque le curseur sort du bouton.

# 2. Boutons bascule

Les boutons bascules sont dérivés des boutons normaux et sont très similaires, sauf qu'ils seront toujours dans un état ou dans l'autre, modifiés par un clic. Lorsque vous cliquez dessus, ils se figent à l'état "pressé" et si vous cliquez à nouveau dessus ils se relâchent et restent à l'état "relâché".

Les boutons bascules sont les bases pour les cases à cocher et les boutons radio aussi, les appels utilisés pour les bascules sont hérités par les boutons radio et les cases à cocher. Nous les aborderons plus loin.

Créons un nouveau bouton bascule :

```
toggleButtonNew :: IO ToggleButton

toggleButtonNewWithLabel :: String -> IO Togglebutton

toggleButtonNewWithMnemonic :: String -> IO ToggleButton
```

Comme on pouvait l'imaginer, ils fonctionnent de façon identique aux appels du bouton "normal". Le premier crée un bouton bascule vide, et le dernier un bouton avec un label directement empaqueté dedans. De la même manière on peut créer un mnémonique avec le caractère underscore (tiret bas) devant le caractère voulu.

Pour connaître l'état du widget (ainsi que pour les cases à cocher et les boutons radio), on utilise :

```
toggleButtonGetActive :: ToggleButtonClass self => self -> IO Bool
```

Cela retourne True si le bouton est "pressé", et False s'il est "relaché"

Pour forcer l'état d'un bouton bascule ainsi que les boutons radio et les cases à cocher, on utilise la fonction :

```
toggleButtonSetActive :: ToggleButtonClass self => self -> Bool -> IO ()
```

L'appel ci-dessus peut être utilisé pour définir l'état du bouton bascule et ses descendants, les boutons radio et les cases à cocher. Le premier argument est le bouton dont on veut modifier l'état et le second, l'état du bouton que l'on veut lui imposer, True pour l'état "pressé" et False pour l'état relâché.

Notez que lorsque vous utilisez la fonction `toggleButtonSetActive` et que l'état est modifié, cela cause l'émission des signaux `onClicked` et `onToggled` depuis le bouton.

# 3. Cases à cocher

Les cases à cocher héritent de plusieurs propriétés et fonctions des boutons bascules, mais ont un aspect un peu différent. Plutôt que d'être des boutons avec du texte à l'intérieur, ce sont de petites cases avec du texte à leur droite. Elles sont souvent utilisées dans des applications pour activer ou non certaines options.

Les fonctions de création sont similaires au bouton classique.

```
checkButtonNew :: IO CheckButton

checkButtonNewWithLabel :: String -> IO Checkbutton

checkButtonNewWithMnemonic :: String -> IO CheckButton
```

La fonction `checkButtonNewWithLabel` crée une case à cocher avec un label à côté d'elle.

`CheckButton` est une instance de `ToggleButtonClass` et le signal `onToggled` est utilisé lorsq'un `CheckButton` est coché ou décoché, comme le bouton bascule.

# 4. Boutons radio

Les boutons radio sont similaires aux cases à cocher sauf qu'ils sont regroupés de telle sorte qu'un seul peut être sélectionné à la fois. Ils sont adaptés pour les choix dans une courte liste d'options. Créer un nouveau bouton radio se fait avec un de ces appels :

```
radioButtonNew :: IO RadioButton

radioButtonNewWithLabel :: String -> IO RadioButton

radioButtonNewWithMnemonic :: String -> IO RadioButton

radioButtonNewFromWidget :: RadioButton -> IO RadioButton

radioButtonNewWithLabelFromWidget :: RadioButton -> String -> IO RadioButton

radioButtonNewWithMnemonicFromWidget :: RadioButton -> String -> IO RadioButton
```

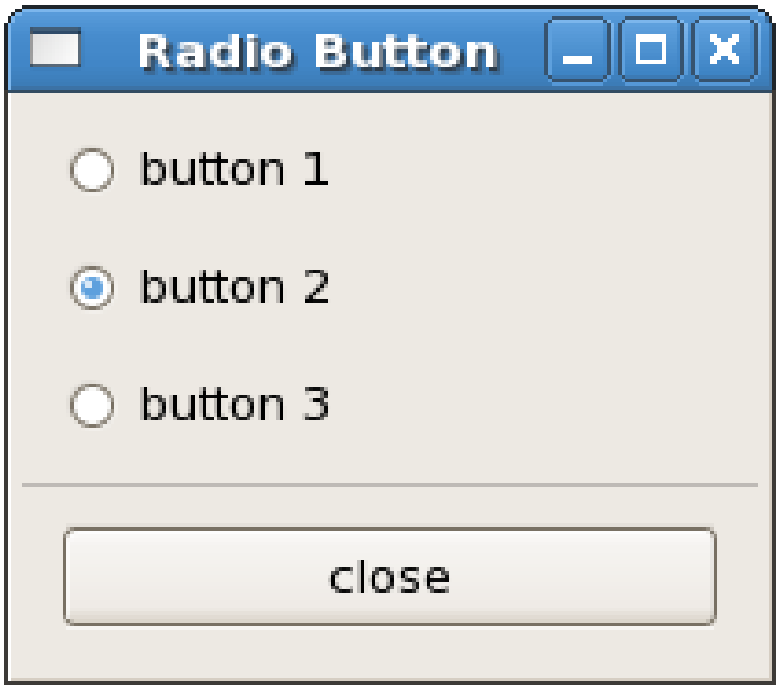
Vous noterez la présence d'un argument supplémentaire dans les trois dernières fonctions. Il est utilisé pour lier les nouveaux boutons à ceux déjà créés plus tôt dans un groupe.

C'est aussi une bonne idée de définir explicitement quel est le bouton qui doit être "pressé" par défaut avec :

```
toggleButtonSetActive :: ToggleButtonClass self => self -> Bool -> IO ()
```

Ceci est décrit dans la section des boutons bascules et fonctionne de la même manière. Une fois que les boutons radio sont groupés ensemble, seulement un bouton du groupe peut être activé à la fois. Si l'utilisateur clique sur un bouton radio, et ensuite sur un autre, le premier bouton radio émettra d'abord un signal `onToggled` (pour signaler le passage à l'état inactif), puis le second émettra un signal `onToggled` (pour signaler le passage à l'état actif).

L'exemple suivant crée un groupe de boutons radio avec trois boutons.Lorsque l'utilisateur presse un des boutons radio, ceux qui sont déclenchés écriront sur la sortie standard stdout en utilisant `putStrLn` dans la fonction `setRadioState` définie plus loin.




```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window  <- windowNew
  set window [windowTitle := "Radio Button", containerBorderWidth := 5,
              windowDefaultWidth := 200, windowDefaultHeight := 150]
```

```
    box1     <- vBoxNew False 0
    containerAdd window box1
    box2     <- vBoxNew False 10
    containerSetBorderWidth box2 10
    boxPackStart box1 box2 PackNatural 0
    button1 <- radioButtonNewWithLabel "button 1"
    boxPackStart box2 button1 PackNatural 0
    button2 <- radioButtonNewWithLabelFromWidget button1 "button 2"
    boxPackStart box2 button2 PackNatural 0
    button3 <- radioButtonNewWithLabelFromWidget button2 "button 3"
    boxPackStart box2 button3 PackNatural 0
    toggleButtonSetActive button2 True
    onToggled button1 (setRadioState button1)
    onToggled button2 (setRadioState button2)
    onToggled button3 (setRadioState button3)
    sep      <- hSeparatorNew
    boxPackStart box1 sep PackNatural 0
    box3     <- vBoxNew False 10
    containerSetBorderWidth box3 10
    boxPackStart box1 box3 PackNatural 0
    closeb   <- buttonNewWithLabel "close"
    boxPackStart box3 closeb PackNatural 0
    onClicked closeb mainQuit
    widgetShowAll window
    onDestroy window mainQuit
    mainGUI

  setRadioState :: RadioButton -> IO ()
  setRadioState b = do
    state <- toggleButtonGetActive b
    label <- get b buttonLabel
    putStrLn ("State " ++ label ++ " now is " ++ (show state))
```

# Tutoriel Gtk2Hs 4.2 - Réglages, curseurs, plages de valeurs

Gtk2Hs a différents widgets qui peuvent être ajustés visuellement par l'utilisateur en utilisant la souris ou le clavier tel que le widget curseur (slider) décrit dans la section correspondante. Il y a aussi quelques widgets qui affichent certaines portions d'une plus grande quantité de données telles le widget text et le widget viewport.

Évidemment, une application a besoin de réagir aux changements que l'utilisateur effectue dans le widget slider. Une des solutions serait que chaque widget émette son propre type de signal quand les réglages ont été faits. Mais vous pouvez aussi vouloir connecter les réglages de plusieurs widgets ensemble de sorte que la modification de l'un modifie l'autre. L'exemple le plus évident est de connecter une barre de défilement à un viewport ou une zone de texte.

L'objet `Adjustment` peut être utilisé pour stocker les paramètres de configuration et les valeurs des widgets réglables, tels que des barres de défilement ou de réglage. Parce que `Adjustment` dérive de GObject et Object, les objets `Adjustment` peuvent émettre des signaux qui peuvent être utilisés non seulement pour permettre au programme de réagir aux réglages de l'utilisateur, mais aussi de propager les valeurs du réglage de manière transparente entre les widgets ajustables.

Plusieurs des widgets qui utilisent les objets Adjustment, comme `ScrolledWindow` peuvent créer leur propre environnement, mais vous pouvez créer le vôtre avec :

```
adjustmentNew :: Double         -- value          - The initial value of the range
              -> Double         -- lower          - The minimum value of the range
              -> Double         -- upper          - The maximum value of the range
              -> Double         -- stepIncrement  - The smaller of two possible increments
              -> Double         -- pageIncrement  - The larger of two possible increments
              -> Double         -- pageSize       - The size of the visible area
              -> IO Adjustment
```

La fonction de création prend toutes les valeurs qui sont contenus dans l'objet : value est la valeur initiale et doit se trouver entre les limites upper et lower du slider. Cliquer sur les flèches augmente la valeur de stepIncrement. Cliquer sur le slider avance de pageIncrement. pageSize est requis pour déterminer si la fin du slider est toujours dans les limites. Vous pouvez définir et obtenir tous les attributs d'un réglage par les méthodes spécifiques ou bien en utilisant les fonctions set et get :

```
onValueChanged :: Adjustment -> IO () -> IO (ConnectId Adjustment)
```

est le signal émis lorsque la valeur ou le réglage change et :

```
onAdjChanged :: Adjustment -> IO () -> IO (ConnectId Adjustment)
```

est le signal émis lorsque un ou plusieurs autres champs actuels sont modifiés.

## 1. Widgets échelle

Les widgets échelle sont utilisés pour permettre à l'utilisateur de sélectionner et manipuler visuellement une valeur dans une plage de valeur en utilisant un curseur. Vous pouvez utiliser un widget de ce type pour être utilisé , par exemple,pour ajuster le grossissement d'une image, contrôler la brillance de la couleur ou spécifier le nombre de minutes d'inactivité avant que l'économiseur d'écran ne se mette en route.

Les fonctions suivantes créent respectivement un widget curseur vertical et horizontal :

```
vScaleNew :: Adjustment -> IO VScale

hScaleNew :: Adjustment -> IO Hscale
```

Il y a également deux constructeurs qui ne prennent pas de réglage.

```
vScaleNewWithRange :: Double ->. Double -> Double -> IO VScale

hScaleNewWithRange :: Double ->. Double -> Double -> IO Hscale
```

Les paramètres Double contiennent les valeurs minimum, maximum et l'incrément. L'incrément est la valeur dont l'échelle est modifiée lorsque les flèches sont utilisées.

Les échelles horizontales et verticales sont des instances de ScaleClass et leurs comportements communs sont définis dans le module Graphics.UI.Gtk.Abstract.Scale.

Les widgets échelle peuvent afficher leur valeur courante en tant que nombre juste à côté. Le comportement par défaut est de montrer la valeur, mais vous pouvez changer cela avec la fonction :

```
scaleSetDrawValue :: ScaleClass self => self -> Bool -> IO ()
```

La valeur affichée par un widget échelle est arrondie par défaut à une décimale. Vous pouvez changer cela avec :

```
scaleSetDigits :: ScaleClass self => self -> Int -> IO ()
```

Au final, la valeur peut être affiché à différentes positions :

```
scaleSetValuePos :: ScaleClass self => self -> PositionType -> IO ()
```

Le type de PositionType est définie comme ceci :

```
data PositionType = PosLeft | PosRight | PosTop | PosBottom
```

Scale hérite lui-même de plusieurs méthodes de sa classe de base qui est Range.

## 1.1. Définition de la politique de mise à jour

La politique de mise à jour d'un widget définit à quel moment, pendant la saisie de l'utilisateur, le champ value va être modifié et émettre le signal onRangeValueChanged. Les politiques de mise à jour sont définies UpdateType qui a trois constructeurs :

**`UpdateContinuous`** : C'est le constructeur par défaut. Le signal onRangeValueChanged est émis en continu à chaque mouvement de slider même de la valeur la plus infime.

**`UpdateDiscontinuous`** : Le signal onRangeValueChanged est émis seulement lorsque l'utilisateur a arrêté de bouger le slider et a relâche le bouton.

**`UpdateDelayed`** : Le signal onRangeValueChanged est émis lorsque l'utilisateur relâche le bouton de la souris ou si le slider n'est plus bougé pendant un certain laps de temps.

La politique de mise à jour du widget peut être définie avec :

```
rangeSetUpdatePolicy :: RangeClass self => self -> UpdateType -> IO ()
```

## 1.2. Définir et accéder aux réglages

Obtenir et fixer les réglages d'un widget échelle peut se faire avec :

```
rangeGetAdjustment :: RangeClass self => self -> IO Adjustment

rangeSetAdjustment :: RangeClass self => self -> Adjustment -> IO ()
```

rangeSetAdjustment ne fait absolument rien si vous lui passez le Adjustment qui est déjà en cours d'utilisation sans vérifier si vous avez changé certains des champs ou non. Si vous passez un nouveau Adjustment, il va déréférencer l'ancien, connecter les signaux appropriés au nouveau et appeler la fonction privée `gtk_range_adjustment_changed()` qui va (ou du moins est supposée) recalculer la taille ou la position du slider et le redessiner si nécessaire. Comme mentioné dans la section des Adjustment, si vous souhaitez réutiliser le même Adjustment quand vous modifiez ses valeurs directement, vous devez émettre le signal changed dessus.

### 1.3. Liens entre touches et souris

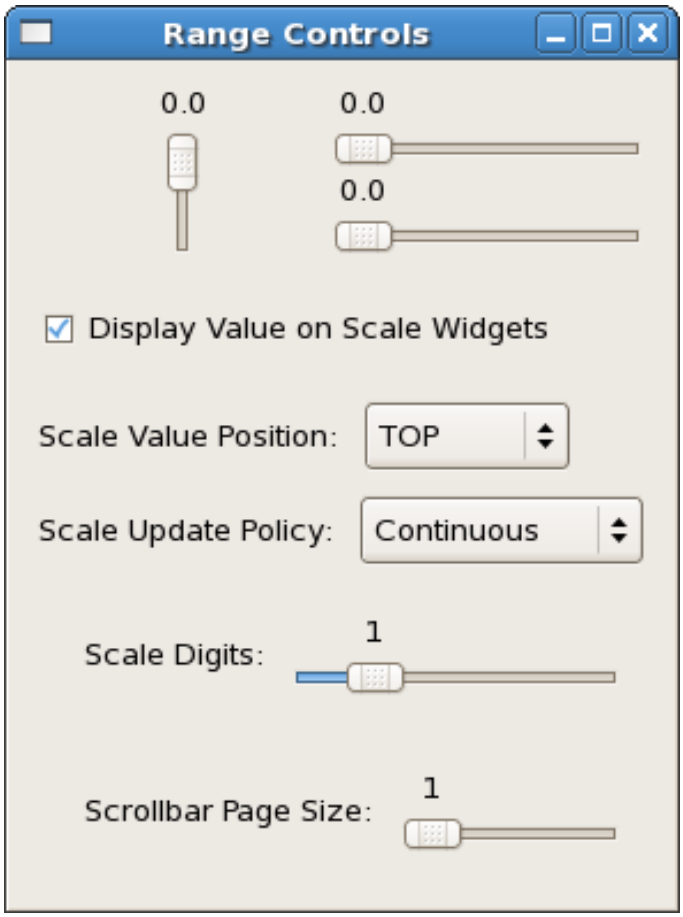
Tous les widgets échelle de Gtk2Hs réagissent aux clics souris plus ou moins de la même façon. Cliquer sur le bouton 1 dessus va ajouter ou soustraire au réglage le stepIncrement de la valeur value, et le slider sera bougé en accord. Cliquer sur bouton 2 va mettre le slider au point sur lequel le bouton a été cliqué.

Les barres de défilement ne sont pas "focusable" et n'ont pas de touches associées. Les touches associées aux autres widgets échelle (qui sont, bien sur, seulement actives lorsque le widget a le focus) ne font pas de distinctions entre les widgets échelle verticaux et horizontaux.

Tous les widgets échelle peuvent être actionnés avec les flèches, gauche, droite, haute et basse aussi bien qu'avec les touches **Page Up** et **Page Down**. Les flèches bougent le slider en haut et en bas par le stepIncrement, alors que **Page Up** et **Page Down** le modifie par pageIncrement. Les touches **Home** et **End** positionnent le curseur au tout début ou tout à la fin.

## 2. Exemple

Cet exemple met en place une fenêtre avec trois widgets échelle tous connectés au même réglage, et un couple de contrôle pour ajuster certains des paramètres mentionnés ci-dessus, vous pourrez alors voir comment ils affectent le fonctionnement des widgets pour l'utilisateur.




Les trois échelles sont placées de telle sorte que celui qui est vertical est à côté des deux qui sont à l'horizontale. Nous avons donc besoin d'une boîte horizontale pour l'échelle verticale et d'une boite verticale à côté pour les deux échelles horizontales. Les échelles et les boites doivent être empaquetées avec PackGrow de sorte que les échelles soient redimensionnées avec la boite principale qui est-elle même une boite verticale dans la fenêtre.

Les trois échelles sont construites avec le même Adjustment, définissant la valeur initiale à 0.0, la valeur minimale à 0.0, la valeur maximale à 101.0, le pas stepIncrement à 0.1, le pas pageIncrement à 1.0 et la taille de page à 1.0.

```
adj1 <- adjustmentNew 0.0 0.0 101.0 0.1 1.0 1.0
```

L'utilisateur peut controler si les valeurs de l'échelle sont affichées avec un checkButton. Il est empaqueté dans la boite principale et activé par défaut. Un checkButton est un bouton bascule et quand l'utilisateur le coche ou le décoche le signal onToggled est émis. Cela déclenche l'évaluation de la fonction toggleDisplay, qui est définie :

```
toggleDisplay :: ScaleClass self => CheckButton -> [self] -> IO ()
toggleDisplay b scls = sequence_ (map change scls) where
                       change sc = do st <- toggleButtonGetActive b
                                      scaleSetDrawValue sc st
```

La fonction prend un type checkButton comme paramètre et une liste de ScaleClass. Cependant, une liste peut seulement contenir des valeurs du même type, et vScale et hScale sont de types différents. Nous pouvons donc utiliser la fonction sur des listes d'échelles verticales ou sur des listes d'échelles horizontales, mais les listes contenant les deux types engendreront une erreur de typage.

L'utilisateur peut sélectionner positionType en utilisant un widget que nous n'avons pas encore abordé, une ComboBox. Elles permettent une sélection de choix. Celui qui doit être actif est déterminé par un index, qui est 0 ici (c'est à dire le premier) :

```
makeOpt1 :: IO ComboBox
makeOpt1 = do
  cb <- comboBoxNewText
  comboBoxAppendText cb "TOP"
  comboBoxAppendText cb "BOTTOM"
  comboBoxAppendText cb "LEFT"
  comboBoxAppendText cb "RIGHT"
  comboBoxSetActive cb 0
  return cb
```

Une seconde comboBox laisse l'utilisateur sélectionner la politique de mise à jour avec les trois constructeurs UpdateType :

```
makeOpt2 :: IO ComboBox
makeOpt2 = do
  cb <- comboBoxNewText
  comboBoxAppendText cb "Continuous"
  comboBoxAppendText cb "Discontinuous"
  comboBoxAppendText cb "Delayed"
  comboBoxSetActive cb 0
  return cb
```

Les comboBox en elle-même ne font qu'afficher du texte. Pour sélectionner la position et la politique de mise à jour, on définit

```
setScalePos :: ScaleClass self => ComboBox -> self -> IO ()
setScalePos cb sc = do
    ntxt <- comboBoxGetActiveText cb
    let pos = case ntxt of
                (Just "TOP")    -> PosTop
                (Just "BOTTOM") -> PosBottom
                (Just "LEFT")   -> PosLeft
                (Just "RIGHT")  -> PosRight
                Nothing         -> error "setScalePos: no position set"
    scaleSetValuePos sc pos

setUpdatePol :: RangeClass self => ComboBox -> self -> IO ()
setUpdatePol cb sc = do
    ntxt <- comboBoxGetActiveText cb
    let pol = case ntxt of
                (Just "Continuous")    -> UpdateContinuous
                (Just "Discontinuous") -> UpdateDiscontinuous
                (Just "Delayed")       -> UpdateDelayed
                Nothing                -> error "setUpdatePol: no policy set"
    rangeSetUpdatePolicy sc pol
```

Nous n'avons pas utilisé de listes pour gérer les échelles verticales et horizontales, de sorte que chaque échelle horizontale est gérée séparément.

La précision du nombre affiché sur les trois échelles sera ajusté par une autre échelle pour laquelle nous utilisons un autre réglage. La précision maximum est 10 est chaque incrément est de 1. La précision de cette échelle de contrôle est de 1.

```
  adj2 <- adjustmentNew 1.0 0.0 5.0 1.0 1.0 0.0
```

Quand le réglage est modifié, le signal onValueChanged est émis et la fonction définie setDigits est évaluée.

```
setDigits :: ScaleClass self => self -> Adjustment -> IO ()
setDigits sc adj = do val <- get adj adjustmentValue
                      set sc [scaleDigits := (round val)]
```

Nous utilisons ici les fonctions générales set et get sur les attributs ; mais nous pourrions tout aussi bien utiliser les méthodes appropriées. Notez que la valeur de type Double du réglage doit être arrondie dans un type entier (Integral).

Nous utilisons une autre échelle horizontale pour gérer la taille de page des trois échelles d'exemple. Lorsqu'elle est réglée à 0.0, les échelles peuvent atteindre leur maximum de 100.0 et quand elle est réglée à 100.0, les échelles sont fixées à leur valeur la plus faible. Cela implique la modification des échelles par un signal onValueChanged provenant d'un troisième Adjustment :

```
  onValueChanged adj3 $ do val <- adjustmentGetValue adj3
                           adjustmentSetPageSize adj1 val
```

La fonction principale est :

```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window  <- windowNew
  set window [windowTitle := "range controls",
              windowDefaultWidth := 250]
  mainbox <- vBoxNew False 10
  containerAdd window mainbox
  containerSetBorderWidth mainbox 10

  box1 <- hBoxNew False 0
  boxPackStart mainbox box1 PackGrow 0
  adj1 <- adjustmentNew 0.0 0.0 101.0 0.1 1.0 1.0
  vsc  <- vScaleNew adj1
  boxPackStart box1 vsc PackGrow 0

  box2 <- vBoxNew False 0
  boxPackStart box1 box2 PackGrow 0

  hsc1 <- hScaleNew adj1
  boxPackStart box2 hsc1 PackGrow 0
  hsc2 <- hScaleNew adj1
  boxPackStart box2 hsc2 PackGrow 0

  chb <- checkButtonNewWithLabel "Display Value on Scale Widgets"
  boxPackStart mainbox chb PackNatural 10
  toggleButtonSetActive chb True

  box3   <- hBoxNew False 10
  boxPackStart mainbox box3 PackNatural 0
  label1 <- labelNew (Just "Scale Value Position:")
  boxPackStart box3 label1 PackNatural 0
  opt1   <- makeOpt1
  boxPackStart box3 opt1 PackNatural 0

  box4   <- hBoxNew False 10
  boxPackStart mainbox box4 PackNatural 0
  label2 <- labelNew (Just "Scale Update Policy:")
  boxPackStart box4 label2 PackNatural 0
  opt2   <- makeOpt2
  boxPackStart box4 opt2 PackNatural 0

  adj2 <- adjustmentNew 1.0 0.0 5.0 1.0 1.0 0.0

  box5   <- hBoxNew False 0
  containerSetBorderWidth box5 10
  boxPackStart mainbox box5 PackGrow 0
  label3 <- labelNew (Just "Scale Digits:")
  boxPackStart box5 label3 PackNatural 10
  dsc    <- hScaleNew adj2
  boxPackStart box5 dsc PackGrow 0
```

```
    scaleSetDigits dsc 0

    adj3 <- adjustmentNew 1.0 1.0 101.0 1.0 1.0 0.0

    box6   <- hBoxNew False 0
    containerSetBorderWidth box6 10
    boxPackStart mainbox box6 PackGrow 0
    label4 <- labelNew (Just "Scrollbar Page Size:")
    boxPackStart box6 label4 PackNatural 10
    psc    <- hScaleNew adj3
    boxPackStart box6 psc PackGrow 0
    scaleSetDigits psc 0

    onToggled chb $ do toggleDisplay chb [hsc1,hsc2]
                       toggleDisplay chb [vsc]

    onChanged opt1 $ do setScalePos opt1 hsc1
                        setScalePos opt1 hsc2
                        setScalePos opt1 vsc

    onChanged opt2 $ do setUpdatePol opt2 hsc1
                        setUpdatePol opt2 hsc2
                        setUpdatePol opt2 vsc

    onValueChanged adj2 $ do setDigits hsc1 adj2
                             setDigits hsc2 adj2
                             setDigits vsc  adj2

    onValueChanged adj3 $ do val <- adjustmentGetValue adj3
                             adjustmentSetPageSize adj1 val

    widgetShowAll window
    onDestroy window mainQuit
    mainGUI
```

Les fonctions non-standard ayant été énumérées auparavant.

# Tutoriel Gtk2Hs 4.3 - Étiquettes

Les étiquettes sont énormément utilisées par Gtk2Hs et sont relativement simples. Les étiquettes n'émettent pas de signaux. Si vous avez besoin de capturer un signal, placez l'étiquette à l'intérieur d'un widget EventBox (ils permettent de capturer des signaux pour des widgets qui n'ont pas leur propre fenêtre).

Pour créer une étiquette, utilisez :

```
labelNew :: Maybe String -> IO Label

labelNewWithMnemonic :: String -> IO Label
```

Avec la seconde fonction, si un caractère est précédé d'un underscore (tiret bas), il est souligné. Si vous avez besoin d'afficher le caractère underscore dans une étiquette, utilisez __ (deux underscores à la suite). Le premier caractère représente un raccourci clavier appelé mnémonique. Quand cette touche est pressée, le widget activable qui contient l'étiquette (par exemple, un bouton) sera activé. Le mnémonique peut également être affecté à un widget avec labelSetMnemonicWidget.

Pour changer le texte du label après sa création ou pour obtenir le texte de l'étiquette, utilisez les fonctions :

```
labelSetText :: LabelClass self => self -> String -> IO ()

labelGetLabel :: LabelClass self => self -> IO String
```

Ou alors les fonctions génériques set ou get. L'espace nécessaire à la nouvelle chaîne de caractères sera automatiquement ajusté au besoin. Vous pouvez écrire des étiquettes multi-lignes en plaçant des sauts de ligne dans la chaîne de caractères. Si vous avez des chaînes multi-lignes, les lignes peuvent être justifiées en utilisant :

```
labelSetJustify :: LabelClass self => self -> Justification -> IO ()
```

Ou le type Justification reçoit un des constructeurs suivant :
-JustifyLeft
-JustifyRight
-JustifyCenter
-JustifyFill

Le widget label est aussi capable de faire des retours à la ligne automatiques. Cela peut être activé avec :

```
labelSetLineWrap :: LabelClass self => self -> Bool -> IO ()
```

Si vous voulez que votre étiquette soit soulignée, alors vous pouvez définir un motif pour l'étiquette :

```
labelSetPattern :: LabelClass self => self -> [Int] -> IO ()
```

La liste de Int marque les parties soulignées du texte, alternées par les parties non-soulignées. Par exemple, [3, 1, 3] signifie que les trois premiers caractères seront soulignés, le suivant non et les trois autres suivants oui.

Vous pouvez également rendre le texte d'une étiquette sélectionnable, l'utilisateur pourra alors le copier et le coller et utiliser des options de formatage.

En dessous, un petit exemple pour illustrer certaines de ces fonctions. Il utilise le widget Frame pour montrer les styles des étiquettes. Un widget Frame est juste une décoration comme un HSeparator et un VSeparator mais il entoure le widget et est une instance de Container. Le widget encadré doit donc être ajouté avec containerAdd. Un cadre peut lui-même contenir une étiquette pour fournir des informations sur son contenu.


Pour que toutes les étiquettes soient encadrées, on crée une fonction myLabelWithFrameNew pour retourner l'étiquette et son cadre dans un "tuple". Gtk2Hs est vraiment dans le style de Haskell, vous pouvez utiliser tous les types de données et fonctionnalités. Justifier du texte est assez évident mais cela s'applique uniquement aux lignes à l'intérieur de l'étiquette. Pour justifier à droite label2, vous avez besoin de miscSetAlignment comme montré en dessous. Les deux derniers widgets dans la boite horizontale à gauche sont empaquetés avec boxPackEnd au lieu de l'habituel boxPackStart. L'étiquette du bouton montre l'utilisation d'un mnémonique comme raccourci clavier. Appuyer sur **Alt-C** au clavier a le même effet que de cliquer sur le bouton.

```
import Graphics.UI.Gtk

main:: IO ()
main = do
  initGUI
  window  <- windowNew
  set window [windowTitle := "Labels", containerBorderWidth := 10]
  mainbox <- vBoxNew False 10
  containerAdd window mainbox
  hbox    <- hBoxNew True 5
  boxPackStart mainbox hbox PackNatural 0
  vbox1   <- vBoxNew False 10
  vbox2   <- vBoxNew False 0
  boxPackStart hbox vbox1 PackNatural 0
  boxPackStart hbox vbox2 PackNatural 0

  (label1,frame1) <- myLabelWithFrameNew
  boxPackStart vbox1 frame1 PackNatural 0
  labelSetText label1 "Penny Harter"

  (label2,frame2) <- myLabelWithFrameNew
  boxPackStart vbox1 frame2 PackNatural 0
  labelSetText label2 "broken bowl\nthe pieces\nstill rocking"
  miscSetAlignment label2 0.0 0.0
  hsep1           <- hSeparatorNew
  boxPackStart vbox1 hsep1 PackNatural 10

  (label3,frame3) <- myLabelWithFrameNew
  boxPackStart vbox1 frame3 PackNatural 0
  labelSetText label3 "Gary Snyder"

  (label4,frame4) <- myLabelWithFrameNew
  boxPackStart vbox1 frame4 PackNatural 0
  labelSetText label4 "After weeks of watching the roof leak\nI fixed it tonight\nby moving a
      single board"
  labelSetJustify label4 JustifyCenter

  (label5,frame5) <- myLabelWithFrameNew
  boxPackStart vbox2 frame5 PackNatural 0
  labelSetText label5 "Kobayashi Issa"

  (label7,frame7) <- myLabelWithFrameNew
  boxPackEnd vbox2 frame7 PackNatural 0
  labelSetText label7 "only one guy and\nonly one fly trying to\nmake the guest room do"
  labelSetJustify label7 JustifyRight
```

```
  (label6,frame6) <- myLabelWithFrameNew
  boxPackEnd vbox2 frame6 PackNatural 10
  labelSetText label6 "One Guy"
  frameSetLabel frame6 "Title:"
  labelSetPattern label6 [3, 1, 3]

  button      <- buttonNew
  boxPackEnd mainbox button PackNatural 20
  buttonlabel <- labelNewWithMnemonic "Haiku _Clicked"
  containerAdd button buttonlabel

  widgetShowAll window
  onClicked button (putStrLn "button clicked...")
  onDestroy window mainQuit
  mainGUI


myLabelWithFrameNew :: IO (Label,Frame)
myLabelWithFrameNew = do
  label <- labelNew Nothing
  frame <- frameNew
  containerAdd frame label
  frameSetShadowType frame ShadowOut
  return (label, frame)


-- Haikus quoted from X.J. Kennedy, Dana Gioia, Introduction to Poetry, Longman, 1997
```

# Tutoriel Gtk2Hs 4.4 - Flèches et infobulles

Le widget Arrow dessine une pointe de flèche, pointant dans plusieurs directions possibles et avec plusieurs styles possibles. Comme le widget Label, il n'émet pas de signaux :
Il y a seulement deux fonctions pour manipuler un widget Arrow :

```
arrowNew :: ArrowType -> ShadowType -> IO Arrow

arrowSet :: ArrowClass self => self -> ArrowType -> ShadowType -> IO ()
```

Le type ArrowType a cinq constructeurs :
-ArrowUp
-ArrowDown
-ArrowLeft
-ArrowRight
-ArrowNone
Le type ShadowType a également cinq constructeurs :
-ShadowIn
-ShadowOut
-ShadowEtchedIn
-ShadowEtchedOut
-ShadowNone
Les infobulles sont les petites chaînes de texte qui apparaissent quand vous laissez le pointeur de la souris au-dessus d'un bouton ou au-dessus d'autres widgets pendant quelques secondes.
Les widgets qui ne reçoivent pas d'événements (Les widgets qui n'ont pas leur propre fenêtre) ne fonctionnent pas avec les info-bulles.
Le premier appel que vous utilisez crée une nouvelle info-bulle. Vous avez seulement besoin de le faire une fois pour un groupe d'info-bulles.

```
tooltipsNew :: IO Tooltips
```

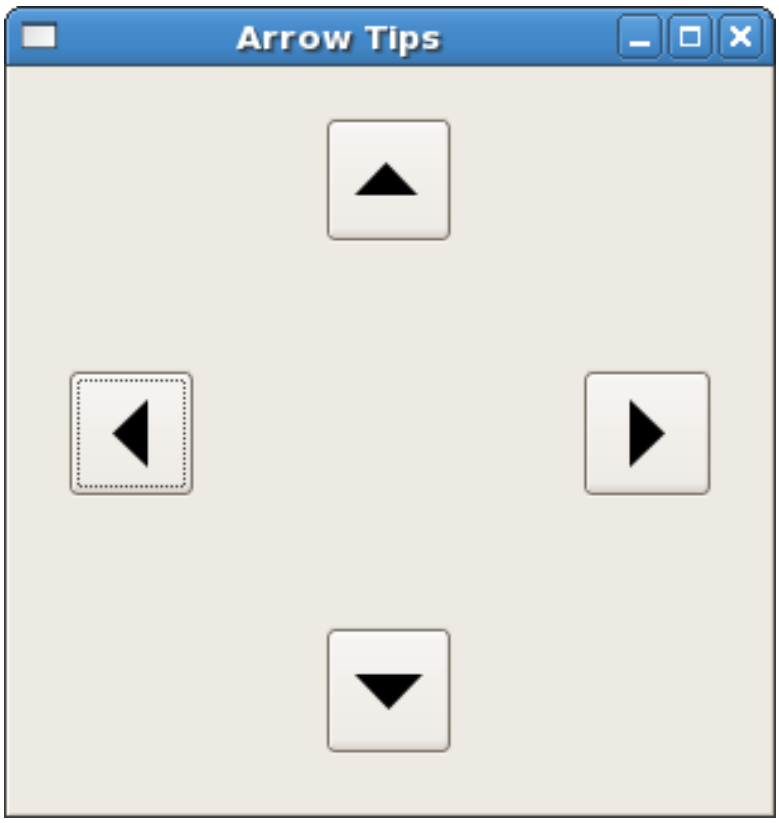
Ensuite, pour tous les widgets, utilisez :

```
tooltipsSetTip :: (TooltipsClass self, WidgetClass widget)
  => self -> widget -> String -> String -> IO ()
```

Le premier argument est l'infobulle que vous avez créée, suivi par le widget auquel vous souhaitez assigner l'infobulle et enfin le texte que vous souhaitez afficher. Le dernier argument est une chaîne de caractères qui peut être utilisée comme identifiant.
Vous pouvez activer ou désactiver l'infobulle associée à un Tooltips avec :

```
tooltipsEnable :: TooltipsClass self => self -> IO ()
tooltipsDisable :: TooltipsClass self => self -> IO ()
```

Voici un exemple pour montrer l'utilisation des flèches et des infobulles




La fenêtre ci-dessus a été redimensionnée par rapport à sa taille d'origine pour montrer comment l'empaquetage dans une grille conserve l'espacement des boutons avec les flèches.

```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  set window [windowTitle := "Arrow Tips",
              windowDefaultWidth := 200,
              windowDefaultHeight := 200, containerBorderWidth := 20]

  table <- tableNew 5 5 True
  containerAdd window table

  button1 <- buttonNew
  button2 <- buttonNew
  button3 <- buttonNew
  button4 <- buttonNew

  tableAttachDefaults table button1 0 1 2 3
  tableAttachDefaults table button2 2 3 0 1
  tableAttachDefaults table button3 4 5 2 3
  tableAttachDefaults table button4 2 3 4 5

  tlt <- tooltipsNew

  arrow1 <- arrowNew ArrowLeft ShadowEtchedIn
  containerAdd button1 arrow1
  tooltipsSetTip tlt button1 "West" "T1"

  arrow2 <- arrowNew ArrowUp ShadowEtchedOut
  containerAdd button2 arrow2
  tooltipsSetTip tlt button2 "North" "T2"

  arrow3 <- arrowNew ArrowRight ShadowEtchedIn
  containerAdd button3 arrow3
  tooltipsSetTip tlt button3 "East" "T3"

  arrow4 <- arrowNew ArrowDown ShadowEtchedOut
  containerAdd button4 arrow4
  tooltipsSetTip tlt button4 "South" "T4"
```

```
    tooltipsEnable tlt
    widgetShowAll window
    onDestroy window mainQuit
    mainGUI
```

# Tutoriel Gtk2Hs 4.5 - Dialogues, Stock Items et barres de progression

Un dialogue est un exemple de widget composé. Il se compose d'une fenêtre, d'une partie supérieure qui est une boite verticale et une zone d'action qui est une boite horizontale. Par défaut, les deux parties sont séparées par un séparateur.

Le widget Dialog peut être utilisé pour des messages à l'intention de l'utilisateur et d'autres tâches similaires. Les fonctions de bases sont :

```
dialogNew :: IO Dialog

dialogRun :: DialogClass self => self -> IO ResponseID
```

Vous pouvez ajouter des boutons dans la zone d'action avec :

```
dialogAddButton :: DialogClass self => self -> String -> ResponseId -> IO Button
```

N'importe quel widget peut être ajouté d'une façon similaire avec dialogAddActionWidget.

La chaîne String dans dialogAddButton peut être le texte du bouton, mais dans la mesure où les dialogues sont principalement utilisés pour des situations standards, un StockItem sera généralement plus approprié.

StockItem sont des ressources qui sont reconnues dans tout Gtk2Hs, telles que les icônes standards IconSet. Vous pouvez définir votre propre stockitem mais plusieurs très utiles sont listés dans le module Graphics.UI.Gtk.General.StockItems. Ils ont l'identifiant StockId qui est un alias pour String. De cet identifiant, un widget (généralement un bouton) avec le texte de l'icône appropriée est automatiquement sélectionné.

Si vous utilisez un StockId en ajoutant un bouton à une boite de dialogue, vous pouvez également utiliser un constructeur ResponseId prédéfini avec ces boutons (ResponseId n'est pas de type String). Des réponses adaptées peuvent être construites avec ResponseUser Int.

À tout moment, quand un bouton dialogue est pressé, sa réponse est passée à l'application qui l'a appelé par dialogRun. D'après la documentation de Gtk2Hs dialogRun fige la boule principale jusqu'à ce que le dialogue émette le signal de réponse ou qu'il soit détruit.

Les barres de progression sont utilisées pour montrer le statut d'une opération en cours de déroulement.

```
progressBarNew :: IO ProgressBar
```

Bien qu'il n'y ait qu'un seul type, il y a deux façons distinctes d'utiliser une barre de progression. Si on connaît l'avancement de la tâche, la fraction (entre 0.0 inclus et 1.0 inclus) peut être fixée avec :

```
progressBarSetFraction :: ProgressBarClass self => self -> Double -> IO ()
```

Cela entraine le remplissage de la barre de progression avec la quantité spécifiée (entre 0.0 et 1.0). Pour représenter la progression, cette fonction doit être appelée régulièrement pendant l'opération.

Lorsque l'on ne connaît pas le travail déjà accompli, la barre peut être bougée d'avant en arrière avec :

```
progressBarPulse :: ProgressBarClass self => self -> IO ()
```

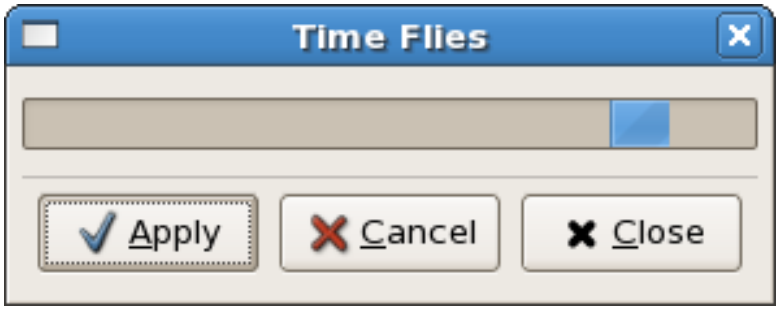
Cette fonction doit aussi être appelée de façon répétée pour montrer l'activité en cours. Il y a plusieurs autres fonctions pour contrôler l'affichage d'une barre de progression, comme l'orientation, le texte, etc… elles sont assez faciles.

La mise en œuvre, en revanche, n'est pas simple car les barres de progression sont généralement utilisées avec des temporisations ou des fonctions similaires pour donner l'impression du multitâche. Avec Haskell, vous pouvez utiliser les threads et les communications pour communiquer entre les threads.

Dans l'exemple qui suit, on simule une activité en utilisant timeoutAdd, qui lance une fonction régulièrement à un intervalle de temps défini en millisecondes. La fonction est passée à timeoutAdd et doit retourner un type IO Bool. Lorsque la fonction renvoie True, le temporisateur est lancé à nouveau. Quand elle renvoie False, il est stoppé. La priorité de timeoutAdd est priorityDefault et est du type Priority

```
timeoutAdd :: IO Bool -> Int -> IO HandlerId
```

Dans l'exemple, nous définissons la fonction showPulse, qui entraîne la pulsation de la barre de progression et retourne toujours IO True. Le pas de la pulsation, la quantité que l'indicateur bouge dans la barre, est fixé à 1.0 avec progressBarSetPulseStep.




L'exemple est un peu atypique dans l'utilisation d'un dialogue dans la mesure où on le garde pour montrer la barre de progression après que l'utilisateur ait pressé le bouton apply. Pour fermer l'application, le dialogue doit être détruit en détruisant la fenêtre. Les boutons close et cancel ne fonctionnent plus après que le bouton apply ait été sélectionné. Si ils sont sélectionnés d'abord au lieu de **Apply**, l'application se fermera.

Si le widget dialogue est détruit, mainQuit est appelée. Comme montré ci-dessus, un Dialog est constitué d'une fenêtre et de deux boites. Les boites sont modifiées par des fonctions spéciales et la barre de progression est empaquetée dans la partie supérieure en utilisant dialogGetUpper. Les boutons dans un dialogue sont visibles par défaut, mais les widgets dans la partie supérieure ne le sont pas. Un dialogue est une instance de WindowClass, et on peut donc définir le titre, la longueur et la hauteur par défaut si on le souhaite.

Quelque chose d'important à noter : Un widget est visible seulement si son parent est visible. Donc, pour afficher la barre de progression, on utilise widgetShowAll sur la boite verticale et pas widgetShow sur la barre de progression.

```
import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI

  dia <- dialogNew
  set dia [windowTitle := "Time Flies"]
  dialogAddButton dia stockApply  ResponseApply
  dialogAddButton dia stockCancel ResponseCancel
  dialogAddButton dia stockClose  ResponseClose

  pr <- progressBarNew
  progressBarSetPulseStep pr 1.0

  upbox <- dialogGetUpper dia
  boxPackStart upbox pr PackGrow 10
  widgetShowAll upbox

  answer <- dialogRun dia
  if answer == ResponseApply
     then do tmhandle <- timeoutAdd (showPulse pr) 500
             return ()
     else widgetDestroy dia

  onDestroy dia mainQuit
  mainGUI

showPulse :: ProgressBar -> IO Bool
showPulse b = do progressBarPulse b
                 return True
```

# Tutoriel Gtk2Hs 4.6 - Zone de saisie de texte et barres d'état

Le widget Entry permet de taper du texte et de l'afficher sur une seule ligne. Un grand nombre de choses peuvent être faites par défaut comme insérer ou remplacer du texte, ...

On crée un nouveau widget Entry avec la fonction :

```
entryNew :: IO Entry
```

Pour remplacer ou récupérer le texte en cours d'édition dans le widget Entry :

```
entrySetText :: EntryClass self => self -> String -> IO ()

entryGetText :: EntryClass self => self -> IO String
```

Si on veut empêcher que le contenu de Entry soit modifié par l'utilisateur, on change son état "éditable". On peut aussi définir : la "visibilité" (pour les mots de passe), le nombre maximum de caractères (0 si illimité), si la zone a un cadre ou non, l'espace à laisser et d'autres attributs. L'autoremplissage du texte est également possible (Voir la documentation de EntryCompletion dans l'API pour son utilisation). Les attributs de Entry qui peuvent évidemment être accessibles avec get et set sont :

```
entryEditable :: EntryClass self => Attr self Bool   -- default True

entryVisibility :: EntryClass self => Attr self Bool   -- default True

entryMaxLength :: EntryClass self => Attr self Int -- 0 if no maximum, limit 66535

entryHasFrame :: EntryClass self => Attr self Bool -- default False

entryWidthChars :: EntryClass self => Attr self Int -- default -1, no space set
```

Le type Entry est une instance de EditableClass et certains attributs et méthodes sont définis ici. Les plus utiles sont :

```
editableInsertText :: EditableClass self => self -> String -> Int -> IO Int

editableDeleteText :: EditableClass self -> Int -> Int -> IO ()

editableSelectRegion :: EditableClass self => self -> Int -> Int -> IO ()

editableDeleteSelection :: EditableClass self -> IO ()
```

Les arguments de type Int indiquent la position de départ ou de fin. L'utilisateur peut aussi copier, coller et couper depuis et vers le presse-papier.

```
editableCutClipboard :: EditableClass self => self -> IO ()

editableCopyClipboard :: EditableClass self => self -> IO ()

editablePasteClipboard :: EditableClass self => self -> IO ()
```

Elles prennent toutes la position courante du curseur. Vous pouvez obtenir et définir cette position avec :

```
editableGetPosition :: EditableClass self => self -> IO Int

editableSetPosition :: EditableClass self => self -> Int
```

Le curseur est affiché avant le caractère indexé (début à 0) dans le widget. La valeur doit être inférieure ou égale au nombre de caractères dans le widget. Une valeur de -1 indique que la position doit être définie après le dernier caractère de la zone de saisie.

La classe Editable a un certain nombre de signaux qui utilisent des fonctions de haut niveau (que nous n'aborderons pas ici). Le widget Entry lui-même a un signal qui est envoyé après que l'utilisateur ait appuyé sur la touche **Enter** :

```
onEntryActivate :: EntryClass ec => ec -> IO () -> IO (ConnectId ec)
```

Il y a aussi des signaux envoyés quand le texte est copié, coupé, collé vers le presse-papier et quand l'utilisateur bascule entre le mode insertion et remplacement.

Les barres d'état sont de simples widgets utilisés pour afficher un message. Elles gardent une liste des messages qui leur sont envoyés de telle sorte que afficher le message courant ré-affichera le précédent message texte. Une barre d'état possède une poignée de redimensionnement par défaut de sorte que l'utilisateur peut la redimensionner.

Afin de permettre aux différents composants d'une application d'utiliser la même barre d'état pour afficher les messages, le widget barre d'état utilise les ContextId qui sont utilisés pour identifier les différents "utilisateurs". Le message au-dessus de la pile est celui qui est affiché quel que soit le contexte qui l'utilise. Les messages sont empilés suivant un processus dernier-entré, premier sorti. Une barre d'état se crée avec :

```
statusbarNew :: IO Statusbar
```

Un nouveau ContextId est généré avec la fonction suivante avec une chaîne de caractères String utilisée comme description du contexte :

```
statusbarGetContextId :: StatusbarClass self => self -> String -> IO ContextId
```

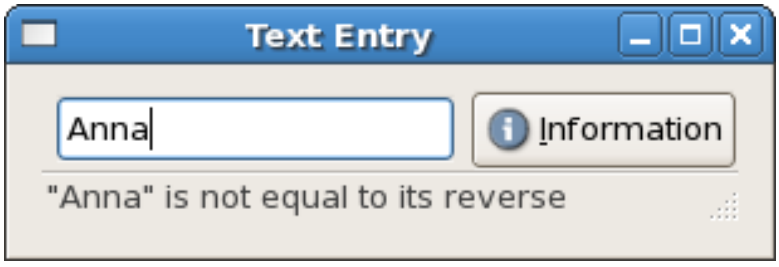
Il y a trois fonctions qui permettent d'intervenir sur les barres d'état.

```
statusbarPush :: StatusbarClass self => self -> ContextId -> String -> IO MessageId

statusbarPop :: StatusbarClass self => self -> ContextId -> IO ()

statusbarRemove :: StatusbarClass self => self -> ContextId -> MessageId -> IO ()
```

La première, statusbarPush, est utilisée pour ajouter un nouveau message à la barre d'état. Elle retourne un MessageId qui peut par la suite être passé à statusbarRemove pour supprimer le message avec le ContextId et le MessageId de la pile de la barre d'état. La fonction statusbarPop enlève le message au-dessus de la pile avec l'identifiant de contexte donné.

Les barres d'état, comme les barres de progression sont utilisées pour afficher des messages à l'utilisateur sur les opérations en cours. Dans l'exemple qui suit, nous allons simuler une telle opération en testant si le texte que l'utilisateur soumet (en pressant la touche **Enter**) est le même que son inverse, et en renvoyant le résultat dans la pile. L'utilisateur peut voir les résultats en appuyant sur le bouton information qui affiche alors les messages de la pile. La première fois, la pile est vide et le bouton est grisé avec :

```
widgetSetSensitivity :: WidgetClass self => self -> Bool -> IO ()
```




```
import Graphics.UI.Gtk

main :: IO ()
main= do
```

```
  initGUI
  window <- windowNew
  set window [windowTitle := "Text Entry", containerBorderWidth := 10]

  vb <- vBoxNew False 0
  containerAdd window vb

  hb <- hBoxNew False 0
  boxPackStart vb hb PackNatural 0

  txtfield <- entryNew
  boxPackStart hb txtfield PackNatural 5
  button <- buttonNewFromStock stockInfo
  boxPackStart hb button PackNatural 0

  txtstack <- statusbarNew
  boxPackStart vb txtstack PackNatural 0
  id <- statusbarGetContextId txtstack "Line"

  widgetShowAll window
  widgetSetSensitivity button False

  onEntryActivate txtfield (saveText txtfield button txtstack id)
  onPressed button (statusbarPop txtstack id)
  onDestroy window mainQuit
  mainGUI

saveText :: Entry -> Button -> Statusbar -> ContextId -> IO ()
saveText fld b stk id = do
    txt <- entryGetText fld
    let mesg | txt == reverse txt = "\"" ++ txt ++ "\""  ++
                                    " is equal to its reverse"
             | otherwise =  "\"" ++ txt ++ "\""  ++
                            " is not equal to its reverse"
    widgetSetSensitivity b True
    msgid <- statusbarPush stk id mesg
    return ()
```

# Tutoriel Gtk2Hs 4.7 - Boutons compteurs

Le widget SpinButton est généralement utilisé pour permettre à l'utilisateur de choisir une valeur dans une plage de données. Il comprend une boite de saisie de texte avec deux boutons fléchés en haut et en bas sur le côté. Cliquer sur un des boutons entraîne l'incrément ou le décrément dans la plage de valeurs possibles. La boite de saisie peut également être éditée directement pour renseigner une valeur spécifique. SpinButton est une instance de EditableClass, les fonctions et attributs qui sont associés à cette classe sont donc disponibles.

Le bouton compteur permet de définir le nombre de décimales et les incréments. Le fait d'appuyer de façon prolongée sur les boutons entraîne une accélération du changement des valeurs en fonction de la durée de pression du bouton.

SpinButton utilise un objet Adjustment pour conserver les informations des valeurs et de la page que le bouton compteur peut prendre. Rappelons qu'un widget Adjustment se crée avec la fonction suivante :

```
adjustmentNew :: Double         -- value
              -> Double         -- lower
              -> Double         -- upper
              -> Double         -- stepIncrement
              -> Double         -- pageIncrement
              -> Double         -- pageSize
              -> IO Adjustment
```

Ces attributs issus de Adjustment sont utilisés par le SpinButton de la façon suivante

**value** : La valeur initiale du SpinButton

**lower** : La valeur minimale

**upper** : La valeur maximale

**stepIncrement** : La valeur qui doit être incrémentée ou décrémenté quand le bouton 1 de la souris est pressé.

**pageIncrement** : La valeur qui doit être incrémentée ou décrémenté quand le bouton 2 de la souris est pressé.

**pageSize** : Inusité

En complément, le bouton 3 de la souris peut être utilisé pour aller directement aux valeurs maximums ou minimum en cliquant sur un des boutons.

Voyons maintenant comment créer un bouton compteur :

```
spinButtonNew :: Adjustment -> Double -> Int -> IO SpinButton
```

Le deuxième argument (climbRate) prend une valeur entre 0.0 et 1.0 et indique à quelle vitesse le bouton compteur est modifié quand une flèche est cliquée. Le troisième argument spécifie le nombre d'emplacements dans lesquels la valeur doit être affichée.

Il y a aussi un constructeur qui permet la création d'un bouton compteur sans avoir à créer manuellement un Adjustment.

```
spinButtonNewWithRange :: Double -> Double -> Double -> IO SpinButton
```

Les trois arguments, tous de type Double spécifient respectivement la valeur minimum, la valeur maximum, l'incrément ajouté ou soustrait par le widget compteur.

Un SpinButton peut être reconfiguré après la création en utilisant la fonction suivante :

```
spinButtonConfigure :: SpinButtonClass self => self -. Adjustment -> Double -> Int
```

Le premier argument le SpinButton qui doit être reconfiguré, les autres arguments sont le climbRate et le nombre de décimales à afficher.

Les attributs du SpinButton qui peuvent être récupérés et changés avec les fonctions génériques get et set sont :

```
spinButtonAdjustment :: SpinButtonClass self => Attr self Adjustment
spinButtonClimbRate  :: SpinButtonClass self => Attr self Double
spinButtonDigits     :: SpinButtonClass self => Attr self Int

spinButtonSnapToTicks :: SpinButtonClass self => Attr self Bool
spinButtonNumeric     :: SpinButtonClass self => Attr self Bool
spinButtonWrap        :: SpinButtonClass self => Attr self Bool

spinButtonValue :: SpinButtonClass self => Attr self Double
```

Les trois premières ont été abordées auparavant. L'attribut spinButtonSnapToTicks détermine si les valeurs erronées sont automatiquement changées vers l'incrément le plus proche (False par défaut). L'attribut spinButtonNumeric détermine si les caractères non-numériques doivent être ignorés (False par défaut) et spinButtonWrap détermine si le bouton compteur doit reboucher sur la plage de valeur lorsque les limites sont dépassées (False par défaut).

L'attribut spinButtonValue est utilisé pour lire la valeur courante ou définir une nouvelle valeur (par défaut 0).

Pour changer la valeur d'un bouton compteur, on peut aussi utiliser :

```
spinButtonSpin :: SpinButtonClass self => self -> SpinType -> Double -> IO ()
```

ou SpinType détermine le type de changement et Double détermine la valeur.

`SpinType` possède les constructeurs suivant :

-`SpinStepForward`
-`SpinStepBackward`
-`SpinPageForward`
-`SpinPageBackward`
-`SpinHome`
-`SpinEnd`
-`SpinUserDefined`

Beaucoup de ces réglages utilisent des valeurs de l'objet Adjustment qui est associé au bouton compteur. SpinStepForward et SpinStepBackward changent la valeur du bouton compteur du nombre d'incréments spécifié, à moins qu'il soit égal à 0, auquel cas, la valeur est remplacée par la valeur de stepIncrement. SpinPageForward et SpinPageBackward modifient simplement la valeur du SpinButton par l'incrément. SpinPageHome et SpinPageEnd mettent la valeur respectivement au maximum ou au minimum de la plage de Adjustment. SpinUserDefined modifie la valeur du bouton compteur par la valeur spécifiée.

Un bouton compteur a également une politique de mise à jour :

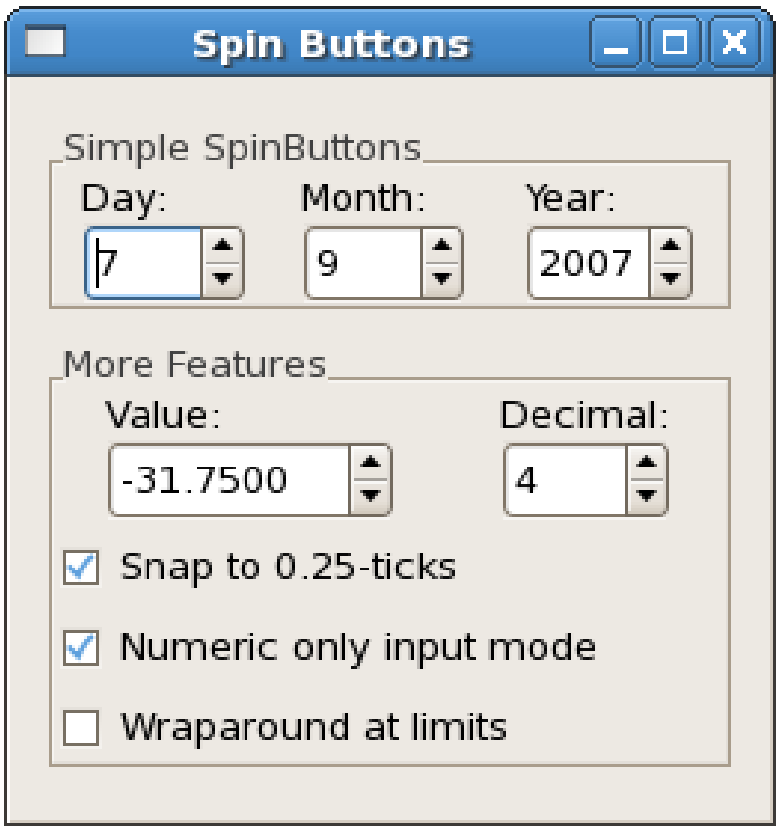
spinButtonUpdatePolicy :: SpinButtonClass self => Attr self SpinButtonUpdatePolicy

Les constructeurs de SpinButtonUpdatePolicy sont soit UdateAlways ou UpdateIfValid. Ces politiques affectent le comportement d'un SpinButton lorsque qu'un texte est inséré et que sa valeur est synchronisée avec celle de Adjustment. Dans le cas de UpdateIfValid, le bouton compteur est changé seulement si le texte rentré est une valeur numérique dans la plage spécifiée par Adjustment. Dans le cas ou cela n'est pas vrai, le texte est réinitialisé à la valeur courante. Dans le cas de UpdateAlways, les erreurs sont ignorées pendant la conversion du texte en valeur numérique.

Au final, vous pouvez demander explicitement qu'un SpinButton se mette à jour lui-même :

```
spinButtonUpdate :: SpinButtonClass self => self -> IO ()
```

On peut maintenant prendre un nouvel exemple. Voici une capture d'écran après avoir modifié certains réglages :




Les boutons compteurs ont tous été créés avec la fonction suivante qui utilise spinButtonNewWithRange.stepIncrement est toujours fixé à 1.0 et n'est donc pas un paramètre de myAddSpinButton.

```
myAddSpinButton :: HBox -> String -> Double -> Double -> IO SpinButton
myAddSpinButton box name min max = do
    vbox  <- vBoxNew False 0
    boxPackStart box vbox PackRepel 0
    label <- labelNew (Just name)
    miscSetAlignment label 0.0 0.5
    boxPackStart vbox label PackNatural 0
    spinb <- spinButtonNewWithRange min max 1.0
    boxPackStart vbox spinb PackNatural 0
    return spinb
```

Dans la fonction main, on utilise un des boutons compteurs qui existe déjà mais en lui donnant un nouveau paramétrage avec spinButtonConfigure. L'ancienne limite de -1000.0 à 1000.0 est dorénavant remplacée par -100.0 et 100.0 (Notez les parenthèses autour des valeurs négatives). La valeur d'origine est définie à 0.0 et l'incrément à 0.25. L'incrément de page est défini à 10.0. La taille de page (qui n'est pas utilisée ici) est fixée à 0.0.

Le nouveau signal ici est onValueSpinned, qui est émis quand l'utilisateur change la valeur du bouton compteur. Ici, il est utilisé pour contrôler le nombre de décimales qui sont affichées dans le bouton spinLarge. Notez qu'il est nécessaire d'arrondir la valeur pour convertir le Double en en type Integral (entier).

Dans cet exemple, nous avons utilisé les fonctions get et set sur les attributs plutôt que les fonctions spécifiques qui sont également disponibles. C'est la façon de programmer qu'il est recommandé d'utiliser avec Gtk2Hs car dans l'avenir certaines fonctions risqueront d'être obsolètes.

```
import Graphics.UI.Gtk

main:: IO ()
main = do
    initGUI
    window  <- windowNew
    mainbox <- vBoxNew False 0
    set window [windowTitle := "Spin Buttons", containerBorderWidth := 10,
                windowDefaultWidth := 250, windowDefaultHeight := 200,
                containerChild := mainbox]
    hbox1   <- hBoxNew False 0
    frame1  <- frameNew
    set frame1 [frameLabel := "Simple SpinButtons", containerChild := hbox1,
```

```
                frameLabelYAlign := 0.8, frameShadowType := ShadowOut]
    boxPackStart mainbox frame1 PackNatural 5

    spinD <- myAddSpinButton hbox1 "Day:" 1.0 31.0
    spinM <- myAddSpinButton hbox1 "Month:" 1.0 12.0
    spinY <- myAddSpinButton hbox1 "Year:" 2000.0 2100.0
    set spinY [spinButtonValue := 2007]

    vbox1  <- vBoxNew False 5
    frame2 <- frameNew
    set frame2 [frameLabel := "More Features", containerChild := vbox1,
                frameLabelYAlign := 0.8, frameShadowType:= ShadowOut]
    boxPackStart mainbox frame2 PackNatural 5

    hbox2 <- hBoxNew False 0
    boxPackStart vbox1 hbox2 PackNatural 0

    spinLarge <- myAddSpinButton hbox2 "Value:" (-1000.0) 1000.0
    adj       <- adjustmentNew 0.0 (-100.0) 100.0 0.25 10.0 0.0
    spinButtonConfigure spinLarge adj 0.0 2
    spnctl    <- myAddSpinButton hbox2 "Decimal:" 0.0 10.0
    set spnctl [spinButtonValue := 2.0]

    tsnap <- checkButtonNewWithLabel "Snap to 0.25-ticks"
    boxPackStart vbox1 tsnap PackNatural 0

    tnumr <- checkButtonNewWithLabel "Numeric only input mode"
    boxPackStart vbox1 tnumr PackNatural 0

    twrap <- checkButtonNewWithLabel "Wraparound at limits"
    boxPackStart vbox1 twrap PackNatural 0

    widgetShowAll window

    onValueSpinned spnctl $ do newdig <- get spnctl spinButtonValue
                               set spinLarge [spinButtonDigits := (round newdig)]

    onToggled tsnap $ do st <- get tsnap toggleButtonActive
                         set spinLarge [spinButtonSnapToTicks := st]

    onToggled tnumr $ do st <- get tnumr toggleButtonActive
                         set spinLarge [spinButtonNumeric := st]

    onToggled twrap $ do st <- get twrap toggleButtonActive
                         set spinLarge [spinButtonWrap := st]

    onDestroy window mainQuit
    mainGUI
```

# Tutoriel Gtk2Hs 5.1 - Calendrier

Le widget calendrier permet d'afficher et de récupérer des informations relatives à la date. C'est un widget facile à créer et à utiliser. Utilisez :

```
calendarNew:: IO Calendar
```

Par défaut, la date du jour est affichée. Pour récupérer la date d'un calendrier, utilisez :

```
calendarGetDate :: CalendarClass self => self -> IO (Int, Int, Int)
```

La sortie est de type (année,mois,jour). Notez que les mois commencent à 0. Il faut donc ajouter 1 pour que ce soit correct. Les attributs associés sont :

```
calendarYear :: CalendarClass self => Attr self Int
calendarMonth :: CalendarClass self => Attr self Int
calendarDay :: CalendarClass self => Attr self Int
```

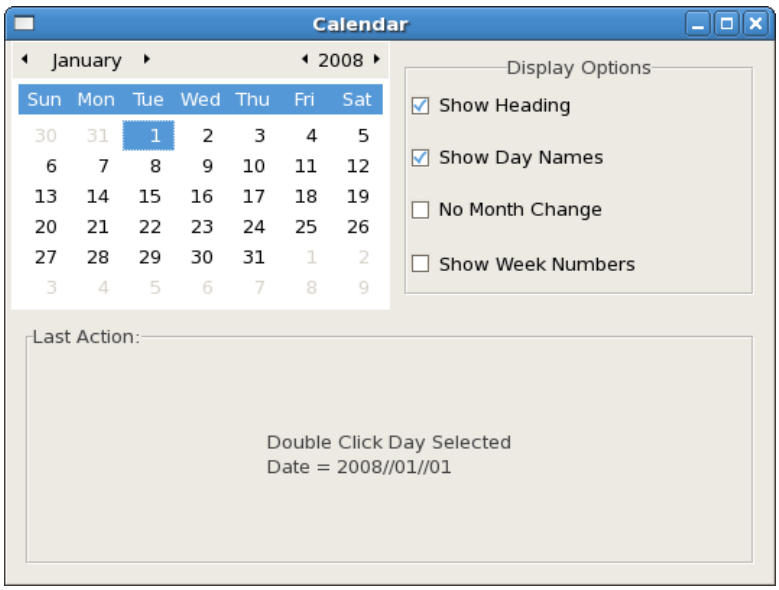
Le widget calendrier a quelques options qui permettent de modifier l'apparence du widget. On modifie ces options avec la fonction : calendarSetDisplayOptions. Pour récupérer les réglages, on utilise : calendarGetDisplayOptions.

```
calendarSetDisplayOptions :: CalendarClass self => self -> [CalendarDisplayOptions] -> IO ()
calendarGetDisplayOptions :: CalendarClass self => self -> IO [CalendarDisplayOptions]
```

`CalendarDisplayOptions` a les constructeurs suivant :
-`CalendarShowHeading` Cette option indique que le mois et l'année doivent être affichés quand le calendrier s'affiche.
-`CalendarShowDayNames` Cette option indique que les 3 premières lettres de chaque jour doivent être affichées (Lun, Mar, ...).
-`CalendarNoMonthChange` Cette option empêche l'utilisateur de modifier le mois affiché. Cela peut être utile si vous avez seulement besoin d'afficher un mois en particulier.
-`CalendarShowWeekNumbers` Cette option indique que le numéro de la semaine doit être indiqué en bas à gauche du calendrier.
-`CalendarWeekStartMonday` Cette option indique que le calendrier doit démarrer le Lundi et pas le Dimanche (par défaut). Cela affecte seulement l'ordre dans lequel les jours sont affichés de gauche à droite.
Ces options peuvent aussi être définies et récupérées avec les fonctions génériques `get` et `set`.
Finalement, n'importe quel nombre de jours dans le mois peuvent être "marqués". Un jour marqué est mis en surbrillance dans le calendrier. Les fonctions suivantes sont fournies pour manipuler les jours marqués :

```
calendarMarkDay :: CalendarClass self => self -> Int -> IO Bool
calendarUnmarkDay :: CalendarClass self => self -> Int -> IO Bool
calendarClearMarks :: CalendarClass self => self -> IO ()
```

Notez que les "marquages" sont persistants quand on navigue à travers les mois et les années.
Le widget calendrier peut envoyer des signaux indiquant les modifications de la date. Ces signaux sont :
-onDaySelected
-onDaySelectedDoubleClick
L'exemple suivant montre l'utilisation d'un widget calendrier :




```haskell
import Graphics.UI.Gtk

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Calendar",
                 windowDefaultWidth:= 200,
                 windowDefaultHeight:= 100]
     mainbox <- vBoxNew True 0
     containerAdd window mainbox

     hbox1 <- hBoxNew True 0
     boxPackStart mainbox hbox1 PackGrow 0

     cal <-calendarNew
     boxPackStart hbox1 cal PackGrow 0

     vbox1 <- vBoxNew True 0
     frame1 <- frameNew
     set frame1 [frameLabel := "Display Options",
                 containerBorderWidth := 10,
                 frameLabelYAlign := 0.5,
                 frameLabelXAlign := 0.5,
                 containerChild := vbox1 ]
     boxPackStart hbox1 frame1 PackGrow 0
     headingopt <- addDisplayOpt vbox1 "Show Heading"
     daynameopt <- addDisplayOpt vbox1 "Show Day Names"
     monchngopt <- addDisplayOpt vbox1 "No Month Change"
     weeknumopt <- addDisplayOpt vbox1 "Show Week Numbers"

     set headingopt [toggleButtonActive := True]
     set daynameopt [toggleButtonActive := True]

     reslabel <- labelNew Nothing
     showMess cal reslabel "Nothing Done Yet"
     frame2 <- frameNew
     set frame2 [frameLabel := "Last Action:",
                 containerBorderWidth := 10,
                 containerChild := reslabel]
     boxPackStart mainbox frame2 PackGrow 0

     mySetOnToggled headingopt cal calendarShowHeading
     mySetOnToggled daynameopt cal calendarShowDayNames
     mySetOnToggled monchngopt cal calendarNoMonthChange
     mySetOnToggled weeknumopt cal calendarShowWeekNumbers

     onDaySelected cal (showMess cal reslabel "Day Selected")
     onDaySelectedDoubleClick cal
           (showMess cal reslabel "Double Click Day Selected")
```

```
      widgetShowAll window
      onDestroy window mainQuit
      mainGUI


addDisplayOpt :: VBox -> String -> IO CheckButton
addDisplayOpt box lbl = do
          cb <- checkButtonNewWithLabel lbl
          boxPackStart box cb PackGrow 5
          return cb

mySetOnToggled :: CheckButton -> Calendar ->
                  Attr Calendar Bool ->
                  IO (ConnectId CheckButton)
mySetOnToggled cb cl att = onToggled cb $ do
          cbstate <- get cb toggleButtonActive
          set cl [att := cbstate]

showMess :: Calendar -> Label -> String -> IO ()
showMess cal lbl str = do
          (year, month, day) <- calendarGetDate cal
          labelSetText lbl $ str ++ "\n" ++ "Date = " ++
                       (show year) ++ "//" ++
                       (myshow (month +1))  -- month is 0 to 11
                        ++ "//" ++ (myshow day)
                             where myshow n | n <= 9 = '0':(show n)
                                            | otherwise = show n

{- Commented out for platform specific testing:
These signals all seem to be implemented as onDaySelected.
The platform was: Gtk2Hs 0.9.12 on Fedora Core 6

      onMonthChanged cal (showMess cal reslabel "Month Changed")
      onNextMonth cal (showMess cal reslabel "Next Month Selected")
      onNextYear cal (showMess cal reslabel "Next Year Selected")
      onPrevMonth cal (showMess cal reslabel "Previous Month
Selected")
      onPrevYear cal (showMess cal reslabel "Previous Year
Selected")
-}
```

# Tutoriel Gtk2Hs 5.2 - Sélection de fichiers

Les fichiers et les dossiers sont essentiels dans tous les programmes et Gtk possède beaucoup de composants pour faciliter leur manipulation. La sélection de fichiers et de dossiers par l'utilisateur est implémentée au travers de l'interface FileChooser. Il y a 4 modes de base pour le type FileChooserAction. Ses constructeurs sont :

`-FileChooserActionOpen` Utilisé pour permettre à l'utilisateur d'ouvrir un fichier.

`-FileChooserActionSave` Utilisé pour permettre à l'utilisateur de sauver un fichier.

`-FileChooserActionSelectFolder` Utilisé pour permettre à l'utilisateur de sélectionner un dossier.

`-FileChooserActionCreateFolder` Utilisé pour permettre à l'utilisateur de créer un dossier.

L'interface FileChooser a des attributs, des méthodes et des signaux mais ce n'est pas à proprement parler un widget. Il y a trois widgets qui utilisent l'interface de différentes manières : FileChooserWidget , FileChooserButton et FileChooserDialog. Comme vous le verrez dans l'exemple plus loin, un widget pour sauver un fichier ou sélectionner un dossier peut aussi contenir un bouton pour créer un dossier. Par conséquent, le constructeur FileActionCreateFolder ne sera probablement jamais utilisé dans vos programmes.

Il est important de noter que, bien que les widgets n'ouvrent pas et ne sauvent pas eux-mêmes les fichiers, la création des dossiers, au contraire, se fait par les widgets.

Notre premier exemple utilisera FileChooserWidget qui peut être dans le mode Ouvrir ou Sauver.

```
fileChooserWidgetNew :: FileChooserAction -> IO FileChooserWidget
```

On utilise FileChooserActionOpen ici, et quand l'utilisateur choisit un fichier en double-cliquant dessus ou en appuyant sur la touche Entrée, le signal onFileActived est émis. On utilise alors :

```
fileChooserGetFilename :: FileChooserClass self => self -> IO (Maybe FilePath)
```

A partir du chemin (FilePath), le programme peut alors ouvrir le fichier. Le format du chemin varie selon la plateforme et est déterminé par la variable d'environnement G_FILENAME_ENCODING.

Vous pouvez paramétrer si l'utilisateur peut sélectionner plusieurs fichiers ou non avec :

```
fileChooserSetselectMultiple :: FileChooserClass self => self -> Bool -> IO ()
```

et avec FileChooserWidget, vous pouvez facilement ajouter une case à cocher pour laisser l'utilisateur choisir. Placer un widget de ce type se fait avec :

```
fileChooserSetExtraWidget :: (FileChooserClass self, WidgetClass extraWidget)=> self ->
    extraWidget -> IO ()
```

Une autre fonctionnalité est l'utilisation de filtres pour afficher seulement les fichiers d'un certains type, soit en spécifiant leur type MIME, soit en spécifiant un motif, soit en spécifiant un format. Les filtres sont documentés dans Graphics.UI.Gtk.Selectors.FileFilter.

Le morceau de code qui suit montre l'utilisation des filtres. La dernière ligne ajoute les filtres dans le widget sélecteur de fichiers et, tout comme les widgets complémentaires, le placement se fait automatiquement.

```
  hsfilt <- fileFilterNew
  fileFilterAddPattern hsfilt "*.hs"
  fileFilterSetName hsfilt "Haskell Source"
  fileChooserAddFilter fch hsfilt
```

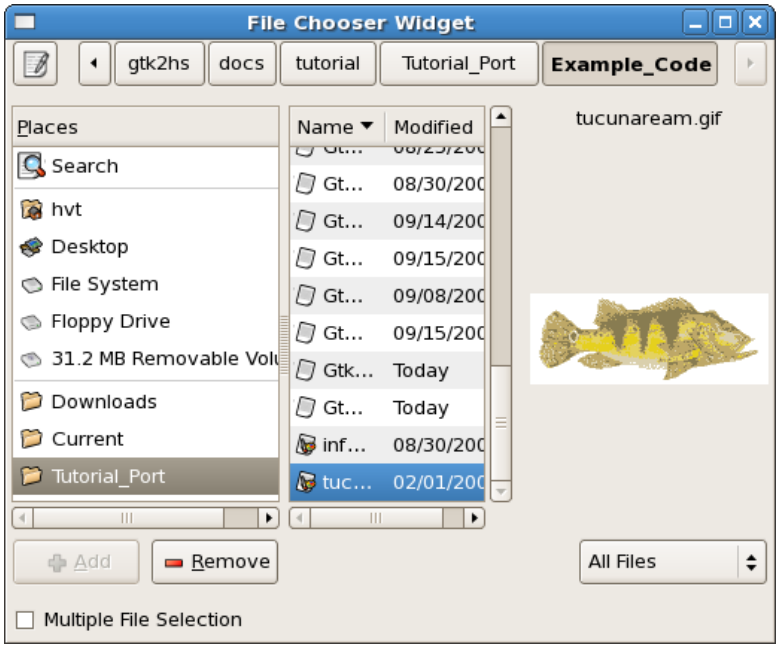
Vous pouvez également ajouter un widget de prévisualisation avec :

```
fileChooserSetPreviewWidget :: (FileChooserClass self, WidgetClass previewWidget) => self ->
    previewWidget -> IO ()
```

Dans l'exemple, ce widget est utilisé pour afficher les fichiers images. L'exemple utilise un widget Image (documenté dans Graphics.UI.Gtk.Display.Image) comme utilisé auparavant dans le chapitre 4.1. Nous avions utilisé alors imageNewFromFile pour ajouter des images sur un bouton ; ici on construit un widget Image vide. Pour le mettre à jour, on utilise le signal onUpdatePreview qui est émis à chaque fois que l'utilisateur change sa sélection de fichier en bougeant la souris ou en appuyant sur les touches de raccourcis. Ce signal est plus général qu'il n'en a l'air, mais ici il n'est utilisé que pour la prévisualisation. Le bout de code est :

```
   onUpdatePreview fch $
      do file <- fileChooserGetPreviewFilename fch
         case file of
              Nothing -> putStrLn "No File Selected"
              Just fpath -> imageSetFromFile img fpath
```

Il y a des fonctions et des attributs pour contrôler l'affichage ; par exemple, ce qui se passe quand on sélectionne un fichier qui n'est pas un fichier graphique, mais elles ne sont pas strictement nécessaires. Dans l'exemple suivant, les fichiers non-graphiques sont simplement ignorés ou indiqués par une icône standard. Voilà a quoi tout cela ressemble :




Notez que l'utilisateur peut aussi ajouter et supprimer des marque-pages et que FileChooser possède également des fonctions pour les gérer. Mais cette fonctionnalité n'est pas abordée dans l'exemple qui suit :

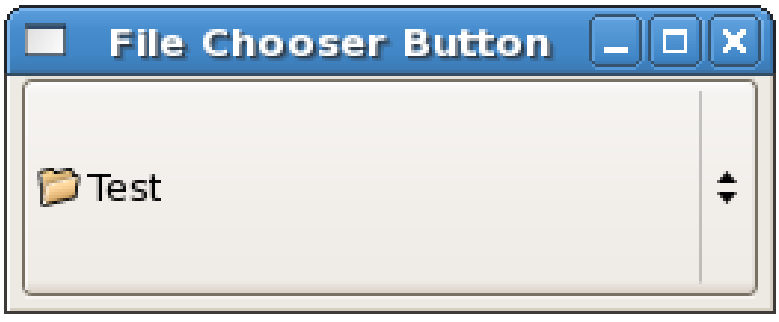
```
import Graphics.UI.Gtk

main :: IO ()
main = do
   initGUI
   window <- windowNew
   set window [windowTitle := "File Chooser Widget",
               windowDefaultWidth := 500,
               windowDefaultHeight := 400 ]

   fch <- fileChooserWidgetNew FileChooserActionOpen
   containerAdd window fch

   selopt <- checkButtonNewWithLabel "Multiple File Selection"
   fileChooserSetExtraWidget fch selopt

   hsfilt <- fileFilterNew
   fileFilterAddPattern hsfilt "*.hs"
   fileFilterSetName hsfilt "Haskell Source"
   fileChooserAddFilter fch hsfilt

   nofilt <- fileFilterNew
   fileFilterAddPattern nofilt "*.*"
   fileFilterSetName nofilt "All Files"
   fileChooserAddFilter fch nofilt
```

```
    img <- imageNew
    fileChooserSetPreviewWidget fch img


    onUpdatePreview fch $
         do file <- fileChooserGetPreviewFilename fch
            case file of
                 Nothing -> putStrLn "No File Selected"
                 Just fpath -> imageSetFromFile img fpath


    onFileActivated fch $
         do dir <- fileChooserGetCurrentFolder fch
            case dir of
                 Just dpath -> putStrLn
                                 ("The current directory is: " ++ dpath)
                 Nothing -> putStrLn "Nothing"
            mul <- fileChooserGetSelectMultiple fch
            if mul
               then do
                 fls <- fileChooserGetFilenames fch
                 putStrLn
                   ("You selected " ++ (show (length fls)) ++ "files:")
                 sequence_ (map putStrLn fls)
               else do
                 file <- fileChooserGetFilename fch
                 case file of
                      Just fpath -> putStrLn ("You selected: " ++ fpath)
                      Nothing -> putStrLn "Nothing"

    onToggled selopt $ do state <- toggleButtonGetActive selopt
                          fileChooserSetSelectMultiple fch state

    widgetShowAll window
    onDestroy window mainQuit
    mainGUI
```

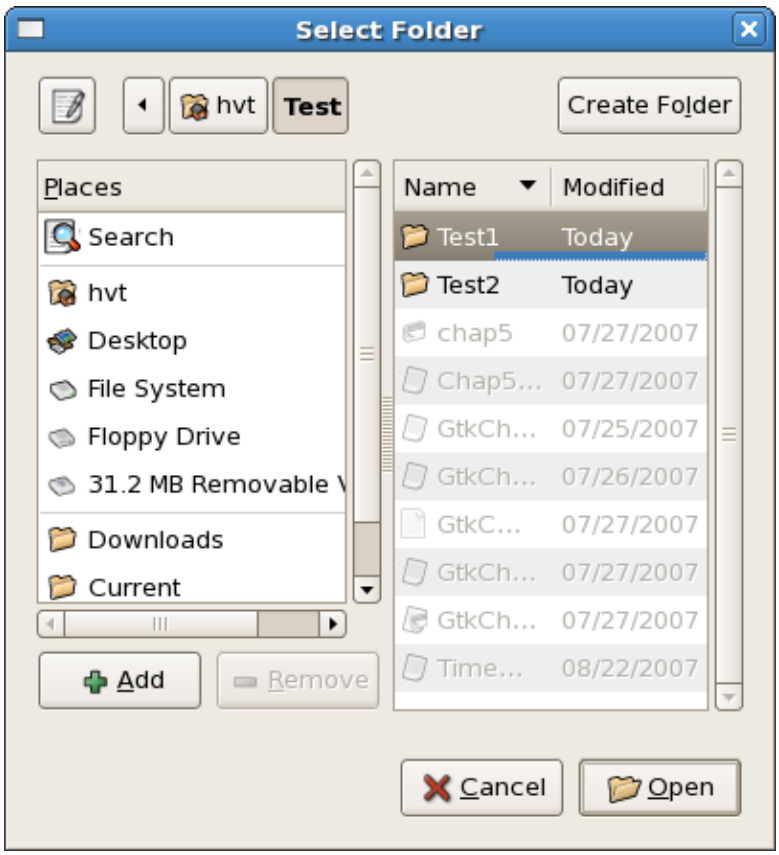
La seconde façon d'utiliser l'interface FileChooser est un bouton de sélection FileChooserButton.

```
fileChooserButtonNew :: String FileChooserAction -> String -> IO FileChooserButton
```

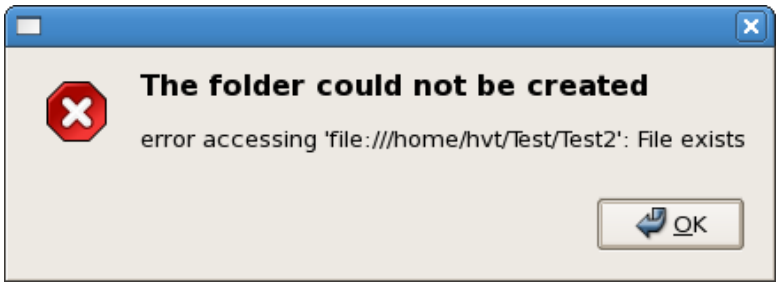
Le paramètre String est le nom du dialogue qui s'affiche quand l'utilisateur sélectionne l'option "Autre..." après avoir pressé sur le bouton. Dans l'exemple ci-dessous, on a construit un bouton sélecteur de fichiers avec FileChooserActionSelectFolder. Voici a quoi ressemble le bouton après avoir sélectionné le répertoire "Test".




Voici a quoi ressemble le dialogue :




Comme on peut le voir, il y a un bouton "créer un dossier" en haut à droite de la fenêtre de dialogue et il peut être utilisé pour créer un nouveau dossier. Voici ce qui arrive quand on essaie de créer un dossier déjà existant :




Créer ou recréer un dossier déjà existant peut causer des problèmes, c'est pour cette raison que Gtk2Hs se charge d'avertir automatiquement l'utilisateur. Quand l'utilisateur sélectionne un répertoire existant, le signal onCurrentFolderChanged est émit et le programme peut lancer l'action appropriée. Créer un dossier le sélectionne automatiquement, on peut donc utiliser onCurrentFolderChanged pour travailler sur ce dossier. Voici un petit exemple :

```
import Graphics.UI.Gtk

main :: IO ()
main = do
     initGUI
     window <- windowNew
     set window [windowTitle := "File Chooser Button",
         windowDefaultWidth := 250, windowDefaultHeight := 75 ]
     fchd <- fileChooserButtonNew "Select Folder"
     FileChooserActionSelectFolder
     containerAdd window fchd

     onCurrentFolderChanged fchd $
          do dir <- fileChooserGetCurrentFolder fchd
             case dir of
                  Nothing -> putStrLn "Nothing"
                  Just dpath -> putStrLn ("You selected:\n" ++ dpath)
```

```
    widgetShowAll window
    onDestroy window mainQuit
    mainGUI
```

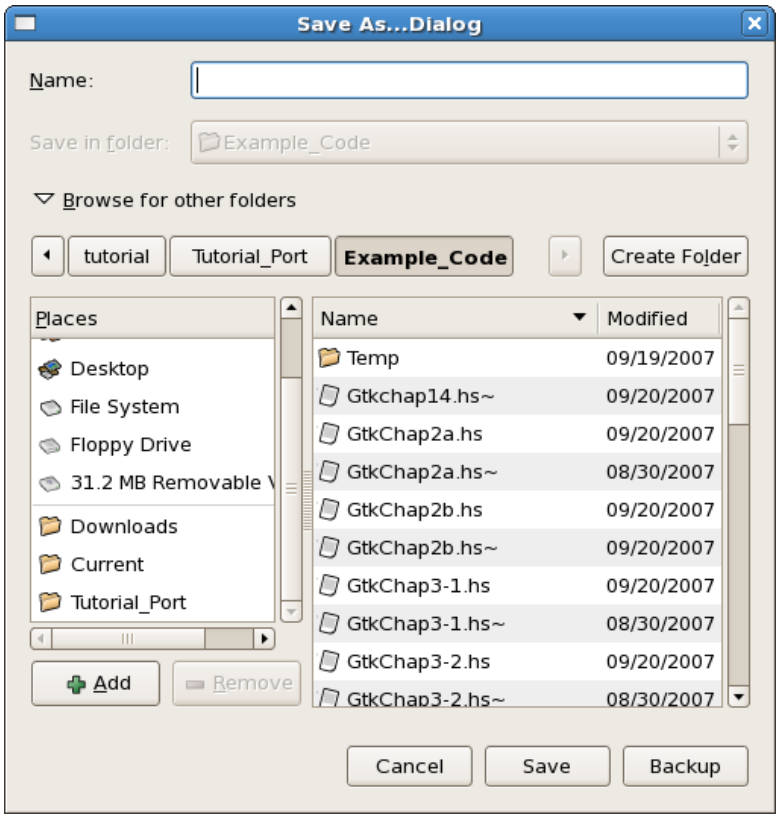
La troisième manière d'utiliser une interface FileChooser se fait avec un FileChooserDialog. Elle peut être construite en mode ouvrir ou sauver et est habituellement lancée depuis un menu ou une barre d'outils.

FileChooserDialog implémente à la fois FileChooser et Dialog. Rappelons (chapitre 4.5) qu'un dialogue est un widget composite avec des boutons généralement implémentés avec dialogRun qui produit des sorties de type ResponseId. FileChooserDialog est construit avec :

```
fileChooserDialogNew ::
  Maybe String                  -- title of the dialog or default
  -> Maybe Window               -- parent window of the dialog or nothing
  -> FileChooserAction          -- open or save mode
  -> [(String, ResponseId)]     -- list of buttons and their response codes
  -> IO FileChooserDialog
```

Tout ce que l'on a faire est de spécifier les noms des boutons et leurs sorties (ResponseId) dans le quatrième argument et ils seront implémentés automatiquement.

L'exemple lance un widget FileChooserActionSave et ce dialogue possède trois boutons. Voici a quoi il ressemble :




Comme vous pouvez le voir, il y a un bouton en haut à droite pour créer un dossier. Comme dans l'exemple précédent, tenter de créer un dossier déjà existant entraîne un message d'erreur. Toutefois, on peut réécrire sur un fichier (autorisé par défaut). Vous pouvez faire confirmer la réécriture d'un fichier par l'utilisateur avec :

```
fileChooserSetDoOverwriteconfirmation :: FileChooserClass self => self -> Bool -> IO ()
```

Comme expliqué auparavant, ni enregistrement, ni écriture n'est faite par le widget FileChooserDialog ; le programme obtient seulement le chemin d'accès

Voici le code du troisième exemple :

```
import  Graphics.UI.Gtk

main :: IO ()
main = do
    initGUI
```

```
    fchdal <- fileChooserDialogNew (Just "Save As...Dialog") Nothing
                                    FileChooserActionSave
                                    [("Cancel", ResponseCancel),
                                     ("Save", ResponseAccept),
                                     ("Backup", ResponseUser 100)]

    fileChooserSetDoOverwriteConfirmation fchdal True
    widgetShow fchdal
    response <- dialogRun fchdal
    case response of
         ResponseCancel -> putStrLn "You cancelled..."
         ResponseAccept -> do nwf <- fileChooserGetFilename fchdal
                              case nwf of
                                   Nothing -> putStrLn "Nothing"
                                   Just path -> putStrLn ("New file path is:\n" ++ path)
         ResponseUser 100 -> putStrLn "You pressed the backup button"
         ResponseDeleteEvent -> putStrLn "You closed the dialog window..."

    widgetDestroy fchdal
    onDestroy fchdal mainQuit
    mainGUI
```

# Tutoriel Gtk2Hs 5.3 - Sélection de fontes et de couleurs

La sélection de fontes et de couleurs se fait à peu près comme la sélection de fichiers. Il y a trois manières de les implémenter, en tant que widget, en tant que dialogue ou en tant que boutons. Les valeurs sélectionnées par l'utilisateur sont récupérées par le biais d'attributs ou de fonctions appropriées. Nous allons d'abord aborder la sélection de fontes, on utilise :

```
fontSelectionNew :: IO FontSelection
fontSelectionDialogNew :: String -> IO FontSelectionDialog
fontButtonNew :: IO FontButton
```

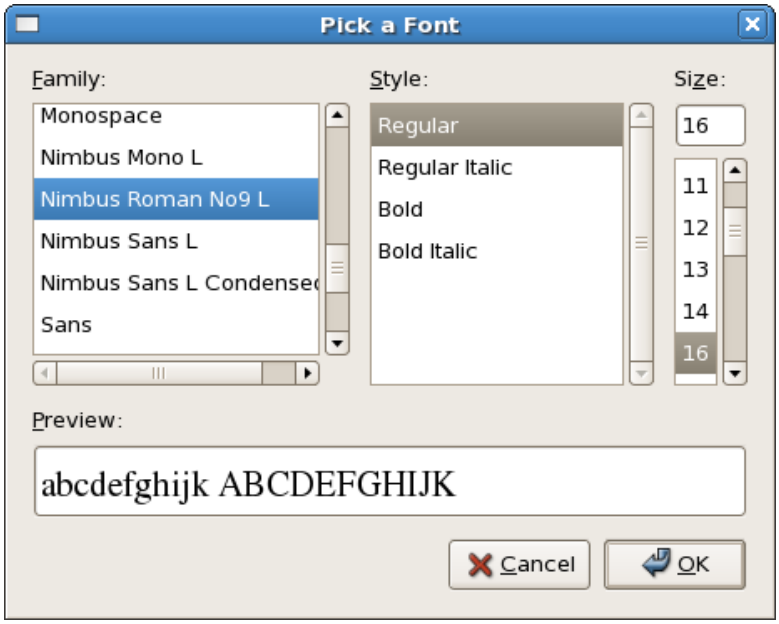
Le paramètre String est le titre de la fenêtre de dialogue. Il y a une série d'attributs et des fonctions pour gérer l'affichage de ces widgets. Toutes assez simples. Avec un dialogue, vous pouvez utiliser les types ResponseId ; et avec vous utiliserez :

```
onFontSet:: FontButtonClass self => self -> IO () -> IO (ConnectId self)
```

Vous utiliserez alors la fonction suivante pour obtenir le nom de la fonte sélectionnée :

```
fontButtonGetFontName :: FontButtonClass self => self -> IO String
```

Le nom de la fonte sera quelque chose comme "Courier Italic 10" ou "URW Gothic L Semi-Bold Oblique 16", tout ce qui est disponible sur votre système. Comme vous pouvez le voir sur l'image suivante, l'utilisateur peut sélectionner une famille, un style, une taille.




Chercher et obtenir des informations sur les fontes est documenté dans Graphics.UI.Gtk.Pango.Font. Beaucoup de fonctionnalités avancées sont supportées, mais l'utilisateur de base aura seulement besoin de savoir comment obtenir une description (FontDescription) à partir du nom d'une fonte.

```
fontDescriptionFromString :: String -> IO FontDescription
```

Une fois que l'on a la description (FontDescription) on peut utiliser :

```
widgetModifyFont:: WidgetClass self => self -> Maybe FontDescription -> IO ()
```

La sélection d'une couleur est analogue à la sélection d'une fonte. Vous avez trois possibilités :

```
colorSelectionNew :: IO Color Selection
colorSelectionDialogNew: :: String -> IO ColorSelectionDialog
colorButtonNew :: IO Color Button
```

Avec un bouton ColorButton, utilisez

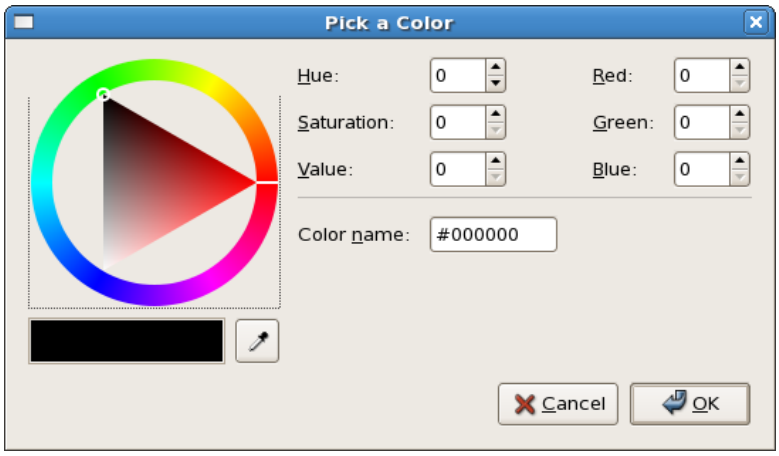
```
onColorSet :: ColorButtonClass self => self -> IO () -> IO (ConnectId self)
```

puis :

```
colorButtonGetColor :: ColorButtonClass self => self -> IO Color
```

Il y a aussi une fonction (et un attribut) pour obtenir la valeur Alpha (l'opacité) si cette fonctionnalité a été activée.

La fenêtre par défaut de sélection de couleur, ressemble à cela :



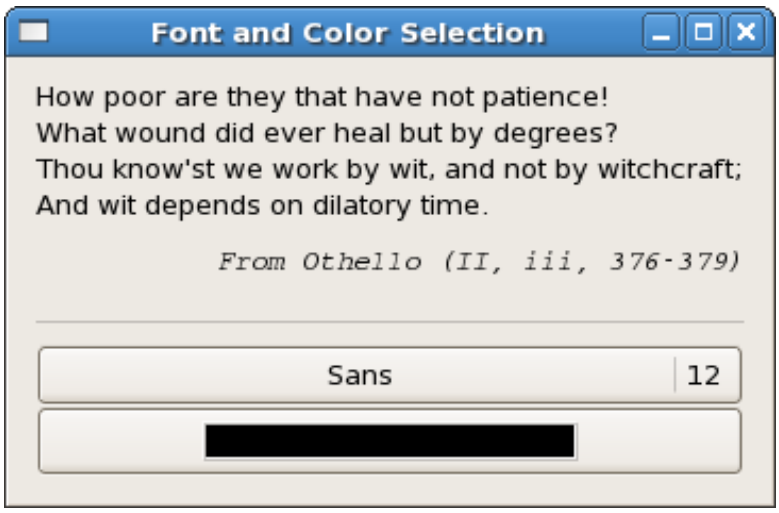

Une couleur est un type de données composées de trois entiers Int, pouvant aller de 0 à 65535, qui spécifient respectivement les composantes Rouge, Vert et Bleu. Il y a des fonctions pour définir les couleurs d'avant-plan, d'arrière-plan, les textes et les couleurs d'un widget et ces fonctions prennent un paramètre StateType Il y a : StateNormal, StateActive, StatePreLight, StateSelected et StateInsensitive et varient suivant que le widget est actif, le pointeur de la souris est sur le widget, si le widget est sélectionné, .... L'affichage des widgets a plusieurs fonctionnalités, mais pour changer la couleur d'une étiquette, vous pouvez simplement utiliser StateNormal et la couleur Color que l'utilisateur a choisi :

```
widgetModifyFg :: WidgetClass self => self -> StateType -> Color -> IO ()
```

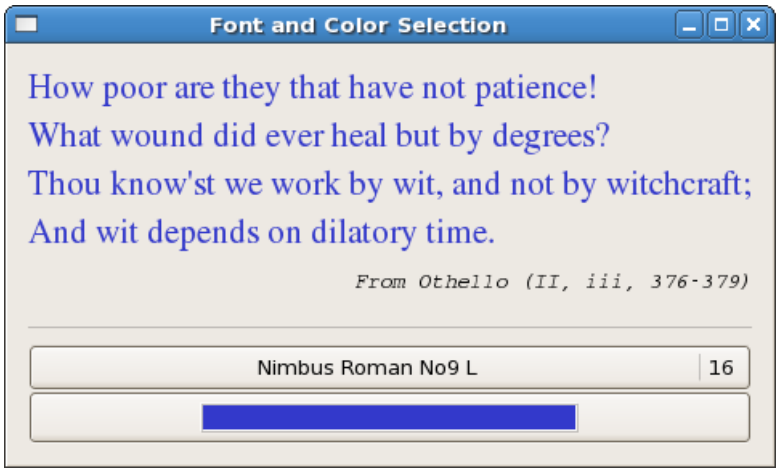
Si on ne connaît pas l'état StateType du widget, on peut utiliser cette fonction :

```
widgetGetState :: WidgetClass w => w -> IO StateType
```

Voici un exemple de sélection de fonte et de couleur.




La fenêtre se redimensionne automatiquement pour s'adapter à la fonte la plus large.




```
import Graphics.UI.Gtk

main :: IO ()
main = do
     initGUI
     window <- windowNew
     set window [windowTitle := "Font and Color Selection",
                 containerBorderWidth := 10 ]
     vb <- vBoxNew False 0
     containerAdd window vb

     qtlab <- labelNew (Just "How poor are they that have not
patience!\nWhat wound did ever heal but by degrees?\nThou know'st
we work by wit, and not by witchcraft;\nAnd wit depends on dilatory
time.")
     boxPackStart vb qtlab PackGrow 0

     srclab <- labelNew (Just "From Othello (II, iii, 376-379)")
     srcfont <- fontDescriptionFromString "Courier Italic 10"
     widgetModifyFont srclab (Just srcfont)
     miscSetAlignment srclab 1.0 0.5
     boxPackStart vb srclab PackNatural 10

     sep <- hSeparatorNew
     boxPackStart vb sep PackGrow 10

     fntb <- fontButtonNew
     boxPackStart vb fntb PackGrow 0

     colb <- colorButtonNew
     boxPackStart vb colb PackGrow 0

     onFontSet fntb $ do name <- fontButtonGetFontName fntb
```

```
                        fdesc <- fontDescriptionFromString name
                        widgetModifyFont qtlab (Just fdesc)
                        putStrLn name

    onColorSet colb $ do colour <- colorButtonGetColor colb
                         widgetModifyFg qtlab StateNormal colour
                         putStrLn (show  colour)

    widgetShowAll window
    onDestroy window mainQuit
    mainGUI

instance Show Color where
        show (Color r g b) = "Red: " ++ (show r) ++
                             " Green: " ++ (show g) ++
                             " Blue: " ++ (show b)
```

# Tutoriel Gtk2Hs 5.4 - Bloc-notes

Le widget Notebook est un ensemble de "pages" qui se superposent l'une sur l'autre. Chaque page est différente mais une seule sera visible à la fois. Les pages contiennent d'autres widgets que vous devez spécifier.

Pour créer un widget bloc-notes :

```
NotebookNew :: IO Notebook
```

Une fois que le bloc-notes a été créé, il y a un certain nombre de fonctions et d'attributs qu'on peut utiliser pour le personnaliser. Les attributs suivant définissent la position des touches et si elles sont visibles ou non.

```
notebookTabPos :: NotebookClass self => Attr self PositionType
notebookShowTabs :: NotebookClass self => Attr self Bool
```

PositionType possède les constructeurs suivant : PosLeft, PosRight, PosTop (par défaut) et PosBottom.

Maintenant intéressons-nous à la façon d'ajouter des pages au bloc-notes. Il y a trois façons d'ajouter : Ajouter au début (prepend), ajouter à la fin (append) et l'insertion.

```
noteBookAppendPage :: (NotebookClass self, WidgetClass child)
  => self
  -> child          -- the widget to use as the contents of the page
  -> String         -- the label text
  -> IO Int         -- the index (page number) of the new page (starting at 0)
```

La fonction notebookPrependPage a exactement le même type. Bien sur il retournera 0 comme index. La fonction notebookInsertPage prend comme paramètre additionnel, l'index ou placer la page. Vous pouvez supprimer des pages avec notebookRemovePage.

Un Notebook est un widget conteneur qui peut contenir d'autres widgets comme par exemple des boites horizontales et verticales. De cette façon, on peut construire des pages assez complexes et les remplir avec les fonctions standards de remplissage.

Les fonctions listées pour ajouter et insérer des pages fonctionnent seulement pour des onglets à texte. Toutes les trois ont des versions qui permettent d'utiliser un menu déroulant et avec lesquelles on peut utiliser n'importe quel widget comme étiquette.

```
notebookAppendPageMenu ::
  (NotebookClass self, WidgetClass child, WidgetClass tabLabel, WidgetClass menuLabel)
  => self
  -> child           -- the widget to use as the contents of the page
  -> tabLabel        -- the widget to use as the label of the page
  -> menuLabel       -- the widget to use as the label of the popup menu
  -> IO Int          -- the index (page number) of the new page (starting at 0)
```

notebookPrependPageMenu et notebookInsertPageMenu placent la nouvelle page en premier ou à la place indiquée.

Certains attributs utiles sont :

```
notebookScrollable :: NotebookClass self => Attr self Bool
notebookCurrentPage :: NotebookClass self => Attr self Int
notebookEnablePopup :: NotebookClass self => Attr self Bool
```

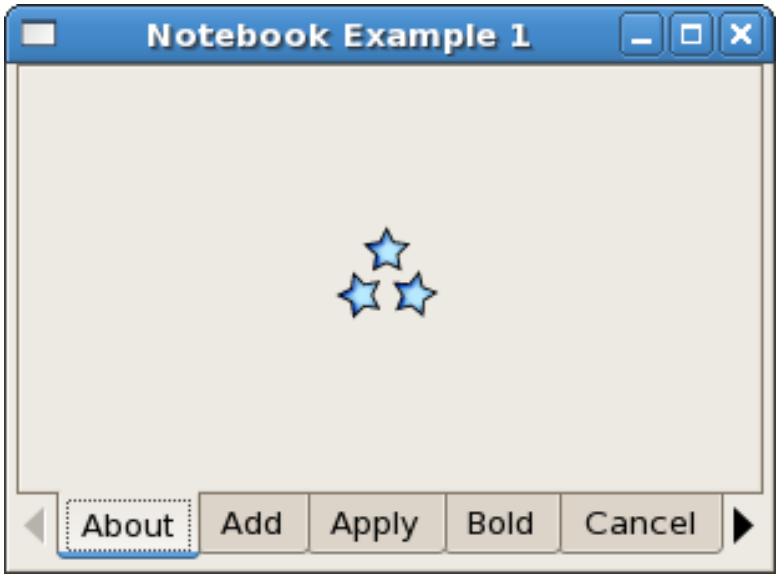
Si il y a un grand nombre de pages, vous pouvez utiliser notebookScrollable. Utilisez notebookCurrentPage ou la fonction notebookSetCurrentPage pour ouvrir le bloc-note à une autre page que la première. L'attribut notebookEnablePopup détermine si l'utilisateur, en cliquant sur le bouton droit de la souris sur un onglet, verra un menu déroulant de toutes les pages disponibles.

Un widget Notebook a son propre signal :

```
onSwitchPage :: NotebookClass nb => nb -> (Int -> IO ()) -> IO (ConnectId nb)
```

La fonction que vous devez fournir prend un index de page (renvoyé par onSwitchPage) et doit exécuter quelques opérations.

Les exemples montrent un catalogue d'icônes StockItem.




Les StockItem ont été abordés sommairement dans le chapitre 4.5. Rappelons que un StockItem est reconnu par GTK+ (et Gtk2Hs). La fonction suivante produit une liste de tous les identifiants reconnus.

```
stockListIds :: IO [StockId]
```

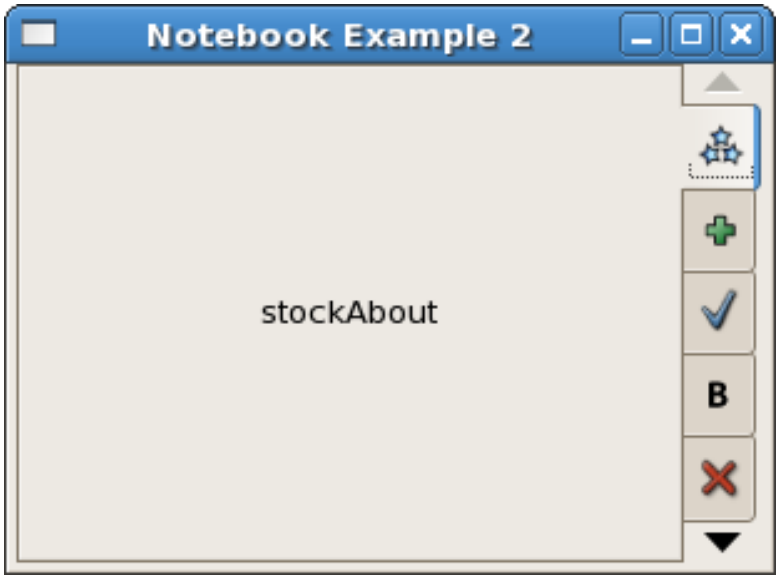
Un StockId est une chaîne de caractères String et dans Gtk2Hs a la forme stockCopy , stockDialogError, etc... L'exemple définit une fonction tabName pour convertir les identifiants GTK+ dans la liste des identifiants StockId pour les onglets du bloc-note. La fonction myNewPage utilise imageNewFromStock pour passer de l'icône à un widget Image qui est ensuite ajouté à la page. Elle retourne l'index de la page (Mais cela ne sert pas dans le programme). Pour obtenir une liste de toutes les pages utilisez séquence au lieu de séquence_

Notez que la taille de l'icône, en pixels, peut être restreinte. La valeur par défaut est 4 et la valeur utilisée ici est 6.

```
import Graphics.UI.Gtk
import Data.Char (toUpper)

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Notebook Example 1", windowDefaultWidth := 300,
                 windowDefaultHeight := 200 ]

     ntbk <- notebookNew
     containerAdd window ntbk
     set ntbk [notebookScrollable := True, notebookTabPos := PosBottom]

     stls <- stockListIds
     sequence_ (map (myNewPage ntbk) stls)

     onSwitchPage ntbk (putStrLn . ((++)"Page: ") . show)

     widgetShowAll window
     onDestroy window mainQuit
     mainGUI

tabName :: StockId -> String
tabName st = (drop 3) (conv st) where
                  conv (x:[]) = x:[]
                  conv (x:y:ys) | x == '-' = (toUpper y):(conv ys)
                                | otherwise = x: (conv (y:ys))

myNewPage :: Notebook -> StockId -> IO Int
myNewPage noteb stk =
          do img <- imageNewFromStock stk 6
```

```
            pagenum <- notebookAppendPage noteb img (tabName stk)
            return pagenum
```

Une deuxième façon de montrer le catalogue est de mettre les icônes dans les onglets du bloc-note.




```
import Graphics.UI.Gtk
import Data.Char (toUpper)

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Notebook Example 2", windowDefaultWidth := 300,
                 windowDefaultHeight := 200 ]

     ntbk <- notebookNew
     containerAdd window ntbk
     set ntbk [notebookScrollable := True, notebookEnablePopup := True,
                   notebookTabPos := PosRight ]

     stls <- stockListIds
     sequence_ (map (myNewPage ntbk) stls)

     onSwitchPage ntbk (putStrLn . ((++)"Page: ") . show)

     widgetShowAll window
     onDestroy window mainQuit
     mainGUI

tabName :: StockId -> String
tabName st = (drop 3) (conv st) where
                  conv (x:[]) = x:[]
                  conv (x:y:ys) | x == '-' = (toUpper y):(conv ys)
                                | otherwise = x: (conv (y:ys))

myNewPage :: Notebook -> StockId -> IO Int
myNewPage noteb stk =
          do img <- imageNewFromStock stk 4
             let nmstr = tabName stk
             men <- labelNew (Just ((take 1) nmstr))
             cont <- labelNew (Just ("stock" ++ nmstr))
             pagenum <- notebookAppendPageMenu noteb cont img men
             return pagenum
```

# Tutoriel Gtk2Hs 6.1 - Fenêtres avec défilement

Les fenêtres avec défilement sont utiles pour créer une zone déroulante avec un autre widget à l'intérieur. Vous pouvez insérer n'importe quel type de widget dans une fenêtre déroulante qui sera accessible quelle que soit sa taille en utilisant les barres de défilement.

La fonction suivante sert à créer une nouvelle fenêtre avec défilement.

```
scrolledWindowNew :: Maybe Adjustment -> Maybe Adjustment -> IO ScrolledWindow
```

Le premier argument est le réglage pour la direction horizontale et le deuxième est pour le réglage vertical. Ils sont, la plupart du temps définis à Nothing.

```
scrolledWindowSetPolicy :: ScrolledWindowClass self => self -> PolicyType -> PolicyType -> IO ()
```

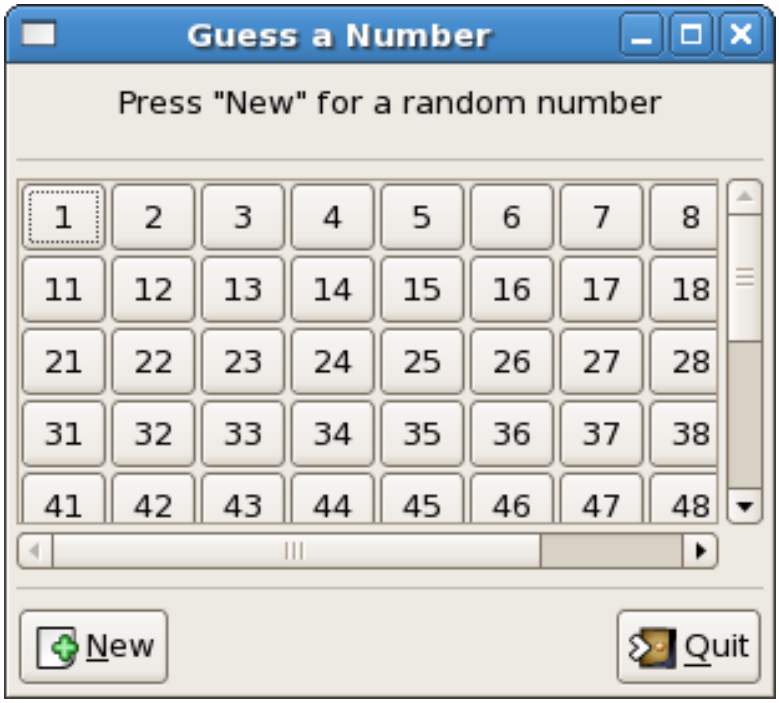
Cette fonction sert à définir la politique d'affichage des barres de défilement horizontales et verticales. Le constructeur PolicyAlways affiche toujours les barres de défilement, PolicyNever ne les affichent jamais et PolicyAutomatic les affichent seulement si la taille de la page est plus grande que la fenêtre. La valeur par défaut est PolicyAlways.

Pour placer un objet dans la fenêtre avec défilement, on utilise containerAdd si cet objet a une fenêtre qui lui est associée. Si il n'en a pas, un Viewport est nécessaire, mais on peut en ajouter un automatiquement avec :

```
scrolledWindowAddWithViewport :: (ScrolledWindowClass self, WidgetClass child) => self -> child
    -> IO ()
```

Voici un exemple qui empile un tableau de 100 boutons bascules dans une fenêtre avec défilement. Cet exemple est tiré de 'Yet Another Haskell Tutorial' par Hal Daumé III. Ce tutoriel est librement disponible sur le site internet de Haskell. Ce programme laisse l'utilisateur trouver un nombre entre 1 et 100 en lui indiquant si le nombre qu'il indique est trop grand, trop petit ou correct. Le nombre aléatoire est généré avec la fonction randomRIO du module System.Random.

Cet exemple implémente ce programme avec une interface graphique.



Dans la fenêtre principale, on utilise une boite verticale pour empaqueter une étiquette (pour guider l'utilisateur) un séparateur horizontal, une fenêtre avec défilement, un séparateur horizontal et une boite horizontale pour deux boutons. La fenêtre avec défilement est empaquetée avec PackGrow pour qu'elle puisse être redimensionnée avec la fenêtre principale. Les boutons Play et Quit sont empaquetés aux bords opposés de la boite horizontale.

Les 100 boutons sont créés avec :

```
buttonlist <- sequence (map numButton [1..100])
```

La fonction numButton est définie par :

```
numButton :: Int -> IO Button
numButton n = do
        button <- buttonNewWithLabel (show n)
        return button
```

Chaque bouton a le numéro approprié comme étiquette.

À l'intérieur de la fenêtre avec défilement, on crée un tableau de 10 par 10 pour les 100 boutons. Pour positionner les boutons, on utilise la fonction cross.

```
cross :: [Int] -> [Int] -> [(Int,Int)]
cross row col = do
        x <- row
        y <- col
        return (x,y)
```

La fonction attachButton prend un tableau, un bouton et un couple de coordonnées pour placer le bouton dans le tableau. (Voir Chapitre 3.3 pour plus d'informations sur l'empaquetage des tableaux).

```
attachButton :: Table -> Button -> (Int,Int) -> IO ()
attachButton ta bu (x,y) = tableAttachDefaults ta bu y (y+1) x (x+1)
```

Le morceau de code suivant empaquette tous les boutons dans le tableau avec buttonlist décrit précédemment.

```
let places = cross [0..9] [0..9]
sequence_ (zipWith (attachButton table) buttonlist places)
```

Chaque fois que l'utilisateur appuie le bouton Play, un nombre aléatoire est généré qui peut ensuite être comparé au choix de l'utilisateur. Mais le gestionnaire de signaux de Gtk2Hs onClicked prend un bouton et une fonction sans paramètre de type IO (). Il faut donc quelque chose qui ressemble à des variables globales qui sont apportées par le module Data.IORef. On peut alors utiliser les morceaux de code suivant dans différentes fonctions pour initialiser, écrire et lire le nombre aléatoire.

```
snippet 1    --  randstore <- newIORef 50
snippet 2    --  writeIORef rst rand
snippet 3    --  rand <- readIORef rst
```

Le premier reçoit une variable de type IORef Int et l'initialise à 50. Le second est implémenté dans la fonction randomButton :

```
randomButton :: ButtonClass b => Label -> IORef Int -> b -> IO (ConnectId b)
randomButton inf rst b =
    onClicked b $ do rand <- randomRIO (1::Int, 100)
                     writeIORef rst rand
                     set inf [labelLabel := "Ready"]
                     widgetModifyFg inf StateNormal (Color 0 0 65535)
```

la fonction actionButton implémente la lecture de randstore. Elle compare alors le nombre obtenu de l'étiquette du bouton qui a été cliqué et affiche l'information appropriée sur info.

Enfin, il faut surveiller tous les 100 boutons pour trouver celui qui a été pressé.

```
     sequence_ (map (actionButton info randstore) buttonlist)
```

Le code ci-dessus est similaire aux autres combinaisons de sequence et map que l'on a déjà utilisé mais dans le cas présent, seul un des 100 signaux sera déclenché lorsque l'utilisateur presse ce bouton en particulier.

Le code complet de l'exemple est :

```
import Graphics.UI.Gtk
import Data.IORef
import System.Random (randomRIO)

main:: IO ()
main= do
     initGUI
     window <- windowNew
     set window [ windowTitle := "Guess a Number",
                  windowDefaultWidth := 300, windowDefaultHeight := 250]
     mb <- vBoxNew False 0
     containerAdd window mb

     info <- labelNew (Just "Press \"New\" for a random number")
     boxPackStart mb info PackNatural 7
     sep1 <- hSeparatorNew
     boxPackStart mb sep1 PackNatural 7

     scrwin <- scrolledWindowNew Nothing Nothing
     boxPackStart mb scrwin PackGrow 0

     table <- tableNew 10 10 True
     scrolledWindowAddWithViewport scrwin table

     buttonlist <- sequence (map numButton [1..100])
     let places = cross [0..9] [0..9]
     sequence_ (zipWith (attachButton table) buttonlist places)

     sep2 <- hSeparatorNew
     boxPackStart mb sep2 PackNatural 7
     hb <- hBoxNew False 0
     boxPackStart mb hb PackNatural 0
     play <- buttonNewFromStock stockNew
     quit <- buttonNewFromStock stockQuit
     boxPackStart hb play PackNatural 0
     boxPackEnd hb quit PackNatural 0

     randstore <- newIORef 50
     randomButton info randstore play

     sequence_ (map (actionButton info randstore) buttonlist)

     widgetShowAll window
     onClicked quit (widgetDestroy window)
     onDestroy window mainQuit
     mainGUI

numButton :: Int -> IO Button
numButton n = do
        button <- buttonNewWithLabel (show n)
        return button

cross :: [Int] -> [Int] -> [(Int,Int)]
cross row col = do
        x <- row
        y <- col
        return (x,y)

attachButton :: Table -> Button -> (Int,Int) -> IO ()
attachButton ta bu (x,y) =
             tableAttachDefaults ta bu y (y+1) x (x+1)

actionButton :: ButtonClass b => Label -> IORef Int -> b -> IO (ConnectId b)
actionButton inf rst b =
  onClicked b $ do label <- get b buttonLabel
                   let num = (read label):: Int
                   rand <- readIORef rst
                   case compare num rand of
                     GT -> do set inf [labelLabel :=  "Too High"]
                              widgetModifyFg inf StateNormal (Color 65535 0 0)
                     LT -> do set inf [labelLabel := "Too Low"]
                              widgetModifyFg inf StateNormal (Color 65535 0 0)
```

```
                    EQ -> do set inf [labelLabel := "Correct"]
                             widgetModifyFg inf StateNormal (Color 0 35000 0)

randomButton :: ButtonClass b => Label -> IORef Int -> b -> IO (ConnectId b)
randomButton inf rst b =
    onClicked b $ do rand <- randomRIO (1::Int, 100)
                     writeIORef rst rand
                     set inf [labelLabel := "Ready"]
                     widgetModifyFg inf StateNormal (Color 0 0 65535)
```

# Tutoriel Gtk2Hs 6.2 - Boites d'événements et boites à boutons

Dans Gtk2Hs, un événement est quelque chose qui est envoyé à un widget par la boucle principale, habituellement en résultat d'une action effectuée par l'utilisateur. Le widget répond en retour en émettant un signal et c'est ce signal qui commande au programme de faire quelque chose. Pour le programmeur d'application Gtk2Hs, un événement est juste un type de données Haskell avec des champs nommés. Certains d'entre eux sont décrits dans la section Graphics.UI.Gtk.Gdk.Events de la documentation de l'API. Regardons par exemple le signal :

```
onButtonPress :: WidgetClass w => w -> (Event -> IO Bool) -> IO (ConnectId w)
```

Ce signal ne doit pas être confondu avec le signal émis quand un widget de type Button a été pressé. Le bouton dont on parle ici est le bouton de la souris . Le signal est émis quand un bouton de la souris est pressé quand celle-ci se trouve au dessus de ce widget.Le paramètre est une fonction qui prend un événement (qui doit avoir le constructeur Button) et retourne une valeur booléenne et une action d'entrée sortie IO. Les champs suivants pour Button sont tirés de l'API :

```
eventSent :: Bool
eventClick :: Click
eventTime :: TimeStamp
eventModifier :: [Modifier]
eventButton :: MouseButton
eventXRoot, eventYRoot :: Double
```

Le premier est utilisé pour le retour. Il apparaît dans tous les constructeurs Event tels que Motion, Expose, Key, Crossing, Focus, Configure, Scroll, WindowState, Proximity. De Event vous pouvez extraire toutes les informations concernant les actions de l'utilisateur. Voici un petit exemple :

```
onButtonPress eb
                (\x -> if (eventButton x) == LeftButton
                          then do widgetSetSensitivity eb False
                                  return (eventSent x)
                          else return (eventSent x))
```

Ici, le paramètre eb est le widget sous la souris et la fonction anonyme est du type décrit précédemment. Quelque chose est fait si le bouton gauche de la souris a été pressé et quand eventSent retourne la valeur booléenne attendue. Si un autre bouton de la souris est pressé, rien ne se produit et seul le booléen est retourné.

Maintenant, certains widgets n'ont pas de fenêtre associée et se contente de dessiner dans leur widget "parent". À cause de cela, ils ne peuvent pas recevoir d'événements et si ils sont mal dimensionnés, ils n'afficheront pas correctement le contenu dans la zone dédiée. Le widget EventBox fournit une fenêtre X pour ces widgets "enfants". Il s'agit d'une sous-classe de Bin qui possède également sa propre fenêtre et qui est une sous-classe de ContainerClass.

Pour créer une nouvelle EventBox, utilisez :

```
eventBoxNew :: IO EventBox
```

Pour ajouter un enfant, on utilise simplement :

```
containerAdd :: (ContainerClass self, WidgetClass widget) => self -> widget -> IO ()
```

La fenêtre peut être visible ou invisible et la boite d'événement peut être au dessus ou en dessous de l'enfant. Ceci est défini par :

```
eventBoxVisibleWindow :: Attr EventBox Bool     -- default True
```

```
eventBoxAboveChild :: Attr EventBox Bool        -- default False
```

Si vous voulez juste capturer des événements, alors la fenêtre sera rendue invisible. Si eventBox est au dessus de l'enfant, tous les événements iront au premier. Si elle est en dessous les événements iront au widget "enfant" en premier.

Une boite à boutons est une simple boite qui peut être utilisée pour empaqueter des boutons d'une façon standard. Il y en a de deux sortes, les horizontales et les verticales, et on les construit avec :

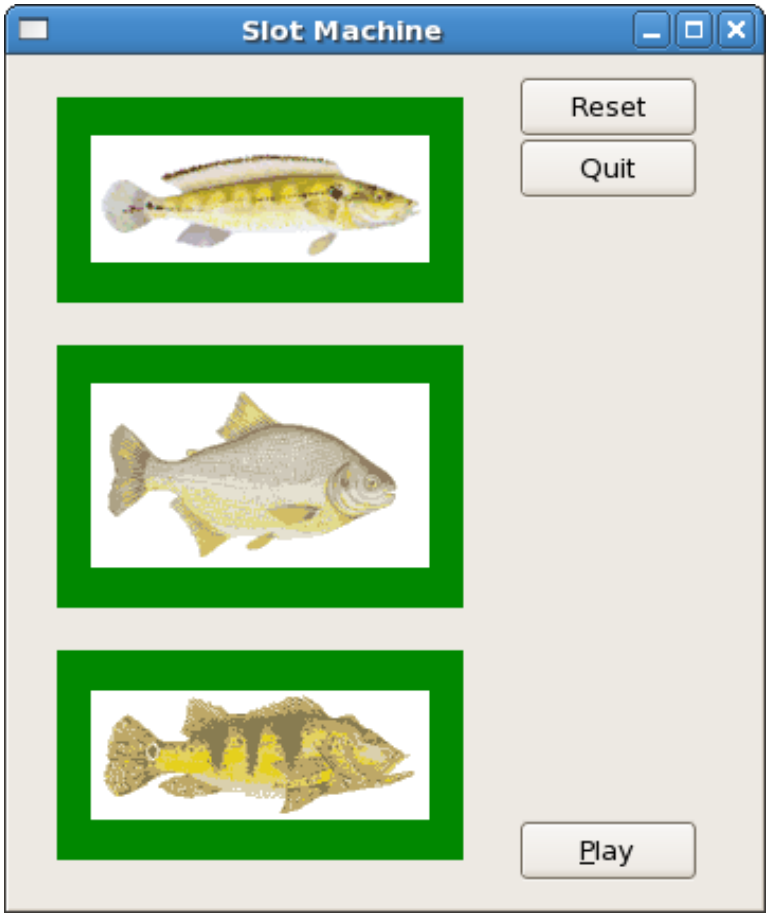
```
hbuttonBoxNew :: IO HButtonBox
vButtonBoxNew :: IO VButtonBox

buttonBoxSetLayout :: ButtonBoxClass self => self -> ButtonBoxStyle -> IO ()
```

Le style est un de ceux-ci : ButtonBoxDefaultStyle, ButtonBoxSpread, ButtonBoxEdge, ButtonBoxStart, ButtonBoxEnd. On n'empaquette pas les boutons comme dans une boite horizontale ou verticale standard, mais en utilisant la fonction containerAdd à la place.

La seconde fonctionnalité d'une boite à boutons est que l'on peut définir un ou plusieurs boutons pour être dans un groupe secondaire. Ceux-ci seront traités différemment lorsque la boite à boutons est redimensionnée. Par exemple, un bouton d'aide peut être isolé (visuellement) des autres. La fonction est :

```
buttonBoxSetChildSecondary :: (ButtonBoxClass self, WidgetClass child) => self -> child -> Bool
    -> IO ()
```

Cet exemple montre l'utilisation des boites d'événements et des boites à bouton :




Les boutons sont empaquetés dans une boite à bouton vertical avec le bouton play comme deuxième "enfant". Ce bouton contient aussi un mnémonique avec la combinaison de touches Alt-P. Les images sont placées dans des boites d'événements avec des fenêtres visibles et leur couleur de fond est définie avec :

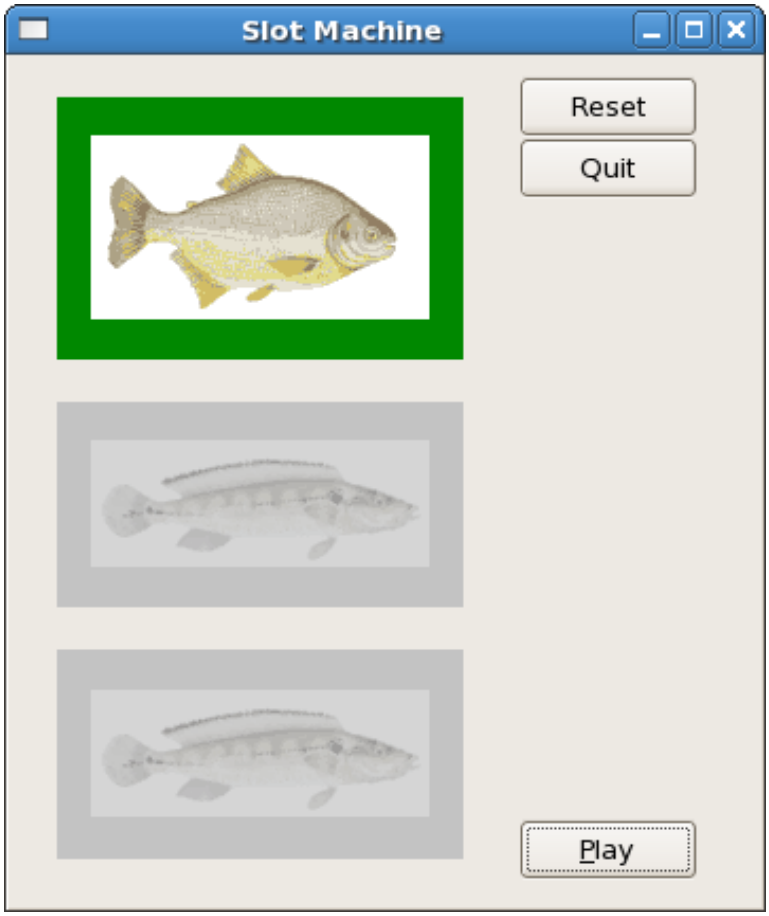
```
widgetModifyBg eb StateNormal (Color 0 35000 0)
```

Comme mentionné dans le chapitre 5.3, StateType peut être StateNormal, StateActive, StatePrelight, StateSelected ou StateInsensitive.

Lorsque l'utilisateur clique le bouton gauche de la souris quand elle est au dessus d'une boite d'évènement, elle sera définie à StateInsensitive avec :

```
widgetSetSensitivity :: WidgetClass self => self -> Bool -> IO ()
```

Cela change le StateType à StateInsensitive et le widget ne répondra plus à aucun événement utilisateur. De plus, son apparence change. Dans l'exemple, on définit également la couleur de fond à gris clair.




```
import Graphics.UI.Gtk
import System.Random (randomRIO)

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Slot Machine",
                 containerBorderWidth := 10,
                 windowDefaultWidth := 350,
                 windowDefaultHeight := 400]
     hb1 <- hBoxNew False 0
     containerAdd window hb1
     vb1 <- vBoxNew False 0
     boxPackStart hb1 vb1 PackGrow 0
     vbb <- vButtonBoxNew
     boxPackStart hb1 vbb PackGrow 0
     resetb <- buttonNewWithLabel "Reset"
     containerAdd vbb resetb
     quitb <- buttonNewWithLabel "Quit"
     containerAdd vbb quitb
     playb <- buttonNewWithMnemonic "_Play"
     containerAdd vbb playb
     set vbb [buttonBoxLayoutStyle := ButtonboxStart,
              (buttonBoxChildSecondary playb) := True ]
```

```
      let picfiles = ["./jacunda.gif", "./pacu.gif", "./tucunaream.gif"]
      evimls <- sequence (map (initEvent vb1) picfiles)
      tips <- tooltipsNew
      sequence_ $ map ((myTooltip tips) . fst) evimls

      onClicked playb (play evimls picfiles)

      onClicked resetb $ sequence_ (zipWith reSet evimls picfiles)

      onClicked quitb (widgetDestroy window)
      widgetShowAll window
      onDestroy window mainQuit
      mainGUI

initEvent :: VBox -> FilePath -> IO (EventBox, Image)
initEvent vb picfile = do
              eb <- eventBoxNew
              boxPackStart vb eb PackGrow 0
              slot <- imageNewFromFile picfile
              set eb[containerChild := slot, containerBorderWidth := 10 ]
              widgetModifyBg eb StateNormal (Color 0 35000 0)
              widgetModifyBg eb StateInsensitive (Color 50000 50000 50000)
              onButtonPress eb
                 (\x -> if (eventButton x) == LeftButton
                           then do widgetSetSensitivity eb False
                                   return (eventSent x)
                           else return (eventSent x))
              return (eb, slot)

reSet :: (EventBox, Image) -> FilePath -> IO ()
reSet (eb, im) pf = do widgetSetSensitivity eb True
                       imageSetFromFile im pf

play :: [(EventBox, Image)] -> [FilePath] -> IO ()
play eilist fplist =
   do let n = length fplist
      rands <- sequence $ replicate n (randomRIO (0::Int,(n-1)))
      sequence_ (zipWith display eilist rands) where
                    display (eb, im) rn = do
                                  state <- widgetGetState eb
                                  if state == StateInsensitive
                                     then return ()
                                     else imageSetFromFile im (fplist !! rn)

myTooltip :: Tooltips -> EventBox -> IO ()
myTooltip ttp eb = tooltipsSetTip ttp eb "Click Left Mouse Button to Freeze" ""
```

# Tutoriel Gtk2Hs 6.3 - Le conteneur de disposition

Jusqu'à maintenant, nous avons empaqueté des widgets soit avec des boites verticales et horizontales, soit avec des tableaux. Vous pouvez, aussi positionner des widgets dans toutes les positions que vous voulez en utilisant un widget Fixed ou Layout. L'utilisation du widget Fixed n'est pas recommandé car il ne se redimensionne pas très bien.

Le conteneur disposition est similaire au conteneur fixé,à la différence qu'il implémente une zone de défilement infinie (en réalité limitée à 2^32).

Un widget disposition se crée avec :

```
layoutNew :: Maybe Adjustment -> Maybe Adjustment -> IO Layout
```

Comme vous pouvez le voir, on peut spécifier optionnellement les objets d'ajustement que le conteneur de disposition utilisera pour le défilement.

Vous pouvez ajouter et déplacer des widgets dans le conteneur disposition en utilisant les fonctions suivantes :

```
layoutPut :: (LayoutClass self, WidgetClass childWidget)=> self -> childWidget -> Int -> Int ->
    IO ()
layoutMove :: (LayoutClass self, WidgetClass childWidget)=> self -> childWidget -> Int -> Int ->
     IO ()
```

Le premier paramètre est la position en x, le deuxième la position en y. Le côté en haut à gauche est à (0,0), x augmente de gauche à droite et y de haut en bas.

La taille du conteneur Layout peut être définie en utilisant la fonction :

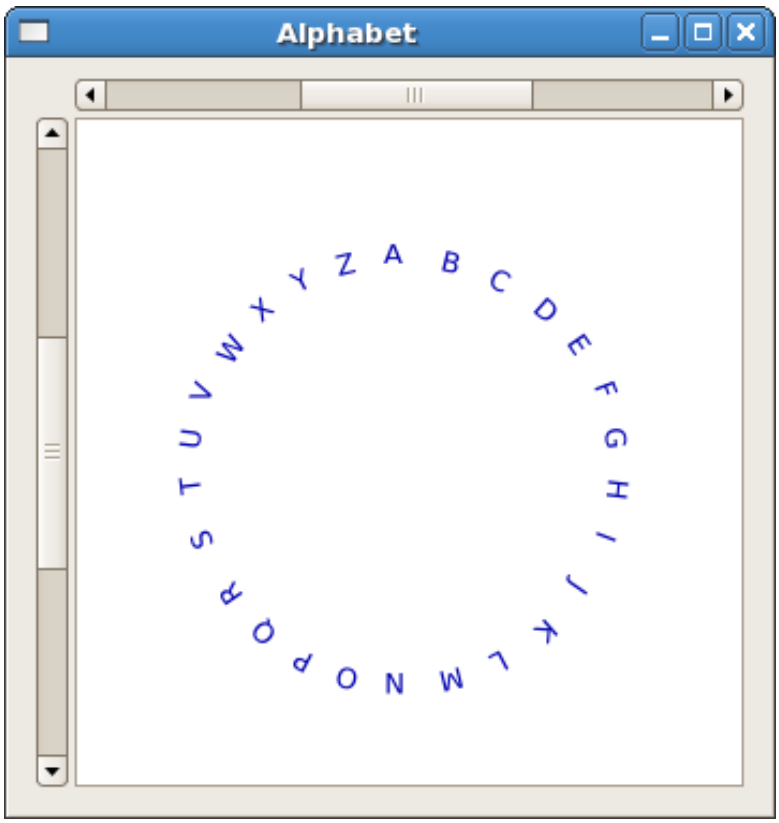
```
layoutSetSize :: LayoutClass self => self  -> Int -> Int -> IO ()
```

Le premier paramètre est la largeur de la totalité de la zone pouvant défiler, le second, la hauteur.

Dans l'exemple, on place une liste d'étiquettes, chacune avec une lettre majuscule dans un cercle autour d'un centre. Les étiquettes sont positionnées perpendiculairement au rayon au moyen de la fonction :

```
labelSetAngle :: labelClass self => self -> Double -> IO ()
```

L'angle est en degrés, pris dans le sens trigonométrique (anti-horaire).




Le widget disposition est placé dans une fenêtre avec défilement au moyen de containerAdd car elle n'a pas besoin de view port (comme vu dans le chapitre 6.1). Les étiquettes sont positionnées en utilisant les coordonnées angulaires qui sont converties en coordonnées cartésiennes avec les fonctions sin et cos. Celles-ci prennent des radians comme arguments (entre 0 et 2*pi). Dans l'exemple, la largeur et la hauteur sont paramétrables, comme l'est la liste à afficher. De plus, dans la fonction main, les coins de Layout sont marqués, de sorte que vous puissiez facilement modifier sa taille si vous le souhaitez.

```
import Graphics.UI.Gtk

main :: IO ()
main = do
     initGUI
     window <- windowNew
     set window [windowTitle := "Alphabet" , windowDefaultWidth := 350,
             windowDefaultHeight := 350 , containerBorderWidth := 10]
     sw <- scrolledWindowNew Nothing Nothing
     set sw [scrolledWindowPlacement := CornerBottomRight,
             scrolledWindowShadowType := ShadowEtchedIn,
             scrolledWindowHscrollbarPolicy := PolicyAutomatic,
             scrolledWindowVscrollbarPolicy := PolicyAutomatic ]
     containerAdd window sw

     layt <- layoutNew Nothing Nothing
     layoutSetSize layt myLayoutWidth myLayoutHeight
     widgetModifyBg layt StateNormal (Color 65535 65535 65535)
     containerAdd sw layt

     upleft  <- labelNew (Just "+(0,0)")
     layoutPut layt upleft 0 0
     upright <- labelNew (Just ("+(" ++ (show (myLayoutWidth - 50)) ++",0)"))
     layoutPut layt upright (myLayoutWidth -50)  0
     dwnright <- labelNew (Just ("+(0," ++ (show (myLayoutHeight -20)) ++ ")"))
     layoutPut layt dwnright 0 (myLayoutHeight -20)
     dwnleft <- labelNew (Just ("+(" ++ (show(myLayoutWidth -70)) ++ "," ++
                                (show (myLayoutHeight -20)) ++ ")"))
     layoutPut layt dwnleft (myLayoutWidth -70) (myLayoutHeight - 20)

     labels <- sequence $ map (labelNew . Just) txtls
     sequence_ $ map (\x -> widgetModifyFg x StateNormal (Color 0 0 45000)) labels

     let wnums = zip labels [0..]
     sequence_ $ map (myLayoutPut layt) wnums
```

```
    widgetShowAll window
    onDestroy window mainQuit
    mainGUI

-- parameters
myLayoutWidth :: Int
myLayoutWidth = 800

myLayoutHeight :: Int
myLayoutHeight = 800

txtls :: [String]
txtls = map (\x -> x:[]) ['A'..'Z']
-- end parameters

step :: Double
step = (2 * pi)/(fromIntegral (length txtls))

ox :: Int
ox =  myLayoutWidth `div` 2

oy :: Int
oy = myLayoutHeight `div` 2

radius :: Double
radius = 0.25 * (fromIntegral ox)

angle :: Int -> Double
angle num = 1.5 * pi + (fromIntegral num) * step

num2x :: Int -> Int
num2x n = ox + relx where
            relx = round $ radius * (cos (angle n))

num2y :: Int -> Int
num2y n = oy + rely where
            rely = round $ radius * (sin (angle n))

myLayoutPut :: Layout -> (Label, Int) -> IO ()
myLayoutPut lt (lb, n) = do
        layoutPut lt lb (num2x n) (num2y n)
        labelSetAngle lb (letterAngle n)

letterAngle :: Int -> Double
letterAngle n = (270 - degree) where
                  degree = (angle n) * (180.0 /pi)
```

# Tutoriel Gtk2Hs 6.4 - Fenêtres à volets et cadres d'apparences

Les widgets fenêtres à volets sont utiles quand on veut diviser une zone en deux parties, avec la taille relative de ces deux parties contrôlées par l'utilisateur. Une rayure est dessinée entre les deux portions, avec une poignée que l'utilisateur peut déplacer pour changer le ratio. La division peut être soit horizontale HPaned, soit verticale VPaned.

Pour créer une nouvelle fenêtre à volets, utilisez :

```
hPanedNew :: IO HPaned
vPanedNew :: IO VPaned
```

Définissez la position de la division avec :

```
panedSetPosition :: PanedClass self => self -> Int -> IO ()
```

Après avoir créé la fenêtre à volets, vous devez ajouter des widgets dans chacune des moitiés

```
panedAdd1 :: (PanedClass self, WidgetClass child) => self -> child -> IO ()
panedAdd2 :: (PanedClass self, WidgetClass child) => self -> child -> IO ()
```

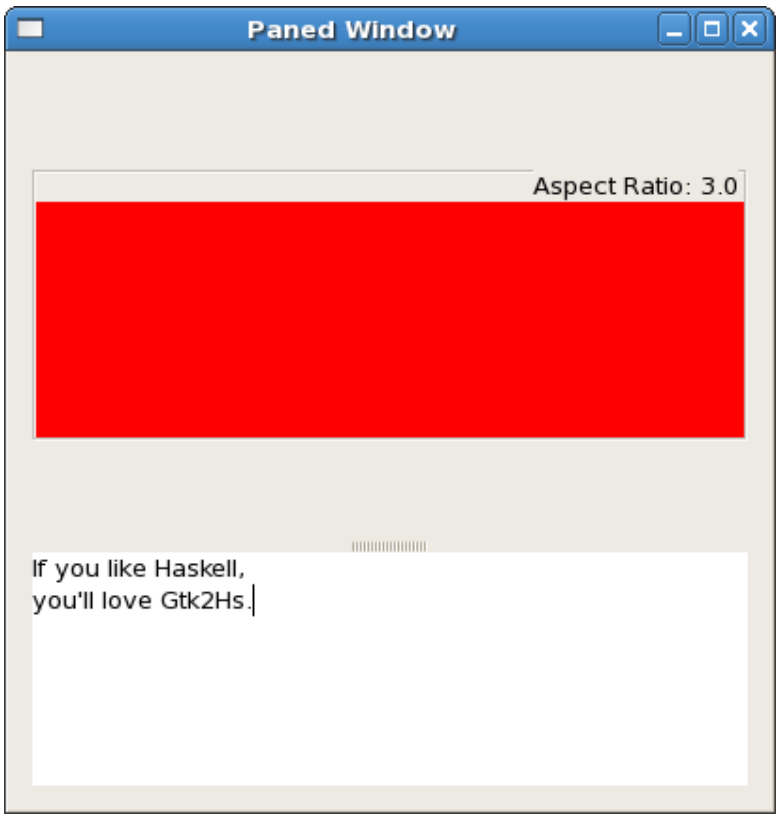
La première fonction ajoute dans la moitié du dessus (ou de droite), la deuxième dans la moitié du dessous (ou de gauche) de la fenêtre à volets. Si vous ne voulez pas que les widgets "enfants" soient redimensionnés avec la fenêtre à volet, utilisez : panedPack1 et panedPack2 à la place.

Un cadre d'apparences est un cadre auquel vous pouvez associer un ratio largeur/hauteur constant. Ce ratio ne sera pas modifié quand le cadre sera redimensionné. On le crée avec :

```
aspectFrameNew :: Float -> Float -> Maybe Float -> IO AspectFrame
```

Le premier paramètre définit l'alignement horizontal du widget "enfant" (entre 0.0 et 1.0). Le second fait la même chose pour l'alignement vertical. Optionnellement, vous pouvez définir le rapport d'aspect avec le troisième paramètre. Dans la mesure ou un widget AspectFrame est un widget Frame, vous pouvez aussi lui ajouter une étiquette.

Dans l'exemple suivant, on crée une fenêtre à volet vertical avec un cadre d'apparence dans la moitié supérieure.




On crée un widget DrawingArea dans le widget AspectFrame. Un widget DrawingArea est une zone blanche qui peut être utilisée pour dessiner à l'intérieur, mais ici, nous définissons juste la couleur de fond pour montrer l'utilisation d'un AspectFrame. Dans la moitié inférieure du widget VPaned on crée un widget TextView. C'est un éditeur et visualiseur de texte multi-lignes avec quelques caractéristiques intéressantes. Dans le cas présent, nous allons juste prendre le texte dans la zone et compter les caractères lorsque l'utilisateur édite du texte.

```
import Graphics.UI.Gtk

main :: IO ()
main = do
     initGUI
     window <- windowNew
     set window [windowTitle := "Paned Window", containerBorderWidth := 10,
                 windowDefaultWidth := 400, windowDefaultHeight := 400 ]

     pw <- vPanedNew
     panedSetPosition pw 250
     containerAdd window pw
     af <- aspectFrameNew 0.5 0.5 (Just 3.0)
     frameSetLabel af "Aspect Ratio: 3.0"
     frameSetLabelAlign af 1.0 0.0
     panedAdd1 pw af

     da <- drawingAreaNew
     containerAdd af da
     widgetModifyBg da StateNormal (Color 65535 0 0)

     tv <- textViewNew
     panedAdd2 pw tv
     buf <- textViewGetBuffer tv

     onBufferChanged buf $ do cn <- textBufferGetCharCount buf
                              putStrLn (show cn)

     widgetShowAll window
     onDestroy window mainQuit
     mainGUI
```

# Tutoriel Gtk2Hs 7.1 - Menus et barres d'outils

Il y a des API spécifiques pour les menus et les barres d'outils, mais il est conseillé de les utiliser ensemble avec UIManager pour définir les actions que vous pouvez alors disposer dans des menus et des barres d'outils. Chaque action peut être associée avec plusieurs widgets. De cette façon, vous pouvez gérer l'activation de l'action au lieu de gérer les entrées des menus et des barres d'outils. Vous pouvez activer ou désactiver le menu et la barre d'outils avec l'action :

```
actionNew
  :: String              --  name : a unique name for the action
  -> String              --  label : what will be displayed in menu items and on buttons
  -> Maybe String        --  tooltip : a tooltip for the action
  -> Maybe String        --  stockId : the stock item to be displayed
  -> IO Action
```

Comme vous pouvez le voir, une action peut être n'importe quoi. Quand l'utilisateur active une action en cliquant sur un widget qui lui est associé ou par un raccourci clavier (voir plus loin), un signal est émis est vous définissez ce qui se produit alors avec :

```
onActionActivate :: ActionClass self => self -> IO () -> IO (ConnectId self)
```

Une Action a des méthodes et des attributs. Par exemple vous pouvez cacher une action ou la rendre inactive avec :

```
actionSetVisible :: ActionClass self => self -> Bool -> IO ()
actionSetSensitive :: ActionClass self => self -> Bool -> IO ()
```

Cependant, les actions sont groupées ensemble et une action ne peut être visible (activable) que si le groupe auquel elle appartient est visible (activable). On crée un nouveau groupe d'action avec :

```
actionGroupNew :: String -> IO ActionGroup
```

L'argument est le nom du ActionGroup et il est utilisé quand on associe des raccourcis claviers avec les actions. Pour ajouter des actions à un groupe, quand aucun raccourci clavier n'est utilisé et sans stock item :

```
actionGroupAddAction ActionClass action => ActionGroup -> action -> IO ()
```

Si un raccourci clavier est utilisé, ou un stock item :

```
actionGroupAddActionWithAccel :: ActionClass action => ActionGroup -> action -> Maybe String ->
    IO ()
```

Si vous utilisez un stock item, le paramètre Maybe String doit être Nothing. Si vous n'utilisez pas un stock item et que vous n'utilisez pas un raccourci, utilisez Just "". D'autre part, la chaîne de caractères doit être dans un format qui permet son analyse (voir plus loin). Vous pouvez définir la visibilité et le caractère activable d'un ActionGroup avec :

```
actionGroupSetVisible :: ActionGroup -> Bool -> IO ()
actionGroupSetSensitive :: ActionGroup -> Bool -> IO ()
```

Comme nous l'avons dit, une action dans un groupe n'est visible que si son attribut et l'attribut de son groupe sont définis.

Maintenant, vous pouvez utiliser les actions en les liant à un ou plusieurs widgets, par exemple dans un menu ou une barre d'outils. Bien sur vous pouvez l'attacher à un seul widget mais l'idée est de les réutiliser. Vous pouvez le faire avec une chaîne de caractères au format XML.

Les éléments XML autorisés sont ui, menubar, menu, menuitem, toolbar, toolitem et popup. Les éléments menuitem et toolitem requièrent une action, et ils doivent être associés au nom (unique) que vous avez donné à l'action lorsque vous l'avez créée. Les éléments de la barre de menus et d'outils peuvent aussi avoir des actions qui leur sont associées, mais elles sont optionnelles. Tous les éléments peuvent avoir des noms, et ils sont également optionnels. Les noms sont requis pour distinguer des widgets du même type et avec le même chemin, par exemple deux barres d'outils juste sous la racine (l'élément ui).

En complément vous avez les séparateurs, les espaces réservés et les accélérateurs. Les séparateurs apparaissent comme des lignes dans les barres d'outils et de menus. Les espaces réservés peuvent être utilisés pour grouper des éléments et des sous-arbres et les accélérateurs définissent des raccourcis clavier. Le livre de référence GTK+ signale que les accélérateurs ne doivent pas être confondus avec les mnémoniques. Les mnémoniques sont activés par une lettre dans l'étiquette, les accélérateurs sont activés par une combinaison que vous avez spécifiée.

**Note :** Unfortunately the accelerators for action menus and toolbars do not seem to work as advertised. Whether this is due to GTK+, Gtk2Hs, the platform, or because I've missed something, is not clear to me. You'll just have to try it out !

La section Graphics.UI.Gtk.ActionMenuToolbar.UIManager dans l'API contient un document type pour la chaine XML, ainsi que des informations complémentaires sur le formatage.

Voici un exemple d'une String XML, qui est utilisé dans l'exemple. Les slashes au début et à la fin de chaque ligne sont nécessaires pour indiquer à GHCI et GHC que la chaîne continue ici et les guillemets dans la définition XML doivent aussi être échappés. Les indentations n'ont pas d'utilité ici :

```
uiDecl=  "<ui>\
\           <menubar>\
\            <menu action=\"FMA\">\
\              <menuitem action=\"NEWA\" />\
\              <menuitem action=\"OPNA\" />\
\              <menuitem action=\"SAVA\" />\
\              <menuitem action=\"SVAA\" />\
\              <separator />\
\              <menuitem action=\"EXIA\" />\
\            </menu>\
\           <menu action=\"EMA\">\
\              <menuitem action=\"CUTA\" />\
\              <menuitem action=\"COPA\" />\
\              <menuitem action=\"PSTA\" />\
\           </menu>\
\            <separator />\
\            <menu action=\"HMA\">\
\              <menuitem action=\"HLPA\" />\
\            </menu>\
\           </menubar>\
\           <toolbar>\
\            <toolitem action=\"NEWA\" />\
\            <toolitem action=\"OPNA\" />\
\            <toolitem action=\"SAVA\" />\
\            <toolitem action=\"EXIA\" />\
\            <separator />\
\            <toolitem action=\"CUTA\" />\
\            <toolitem action=\"COPA\" />\
\            <toolitem action=\"PSTA\" />\
\            <separator />\
\            <toolitem action=\"HLPA\" />\
\           </toolbar>\
\         </ui>" </pre>"
```

Tous les attributs d'action sont des chaînes qui sont définies plus tôt, quand les actions sont créées (Voir le code source complet plus loin).

Maintenant, cette définition ou déclaration doit être traitée par un gestionnaire UIManager pour en créer un :

```
uiManagerNew :: IO UIManager
```

Pour ajouter la chaîne XML :

```
uiManagerAddUiFromString :: UIManager -> String -> IO MergeId
```

Ensuite, les actions qui ont été créées auparavant doivent être insérées :

```
uiManagerInsertActionGroup :: UIManager -> ActionGroup -> Int -> IO ()
```

Si vous avez seulement un groupe d'action, la position sera 0, autrement vous devrez spécifier l'index dans la liste que vous avez déjà créée.

Vous pouvez maintenant avoir tous les widgets que vous voulez à partir de votre UIManager et du chemin (incluant les noms si nécessaire) dans votre définition en XML.

```
uiManagerGetWidget :: UIManager -> String -> IO (Maybe Widget)
```

A partie de la définition ci-dessus, on peut avoir une barre de menus et une barre d'outils avec :

```
maybeMenubar <- uiManagerGetWidget ui "/ui/menubar"
     let menubar = case maybeMenubar of
                        (Just x) -> x
                        Nothing -> error "Cannot get menubar from string."
     boxPackStart box menubar PackNatural 0

     maybeToolbar <- uiManagerGetWidget ui "/ui/toolbar"
     let toolbar = case maybeToolbar of
                        (Just x) -> x
                        Nothing -> error "Cannot get toolbar from string."
     boxPackStart box toolbar PackNatural 0
```

L'empaquetage a été inclus dans le morceau de code, pour montrer qu'il doit malgré tout être fait par vous. Voici l'exemple complet :

Une action a été paramétrée pour être inactive afin de montrer la marche à suivre. Un accélérateur a également été ajouté à l'action de sortie, qui prend le stockitem stockQuit mais affiche Ctrl+E comme accélérateur. Selon le manuel de référence de GTK+, les touches accélératrices sont : , F1, .

```
import Graphics.UI.Gtk

main :: IO ()
main = do
     initGUI
     window <- windowNew
     set window [windowTitle := "Menus and Toolbars",
                 windowDefaultWidth := 450, windowDefaultHeight := 200]

     box <- vBoxNew False 0
     containerAdd window box

     fma <- actionNew "FMA" "File" Nothing Nothing
     ema <- actionNew "EMA" "Edit" Nothing Nothing
     hma <- actionNew "HMA" "Help" Nothing Nothing

     newa <- actionNew "NEWA" "New"     (Just "Just a Stub") (Just stockNew)
     opna <- actionNew "OPNA" "Open"    (Just "Just a Stub") (Just stockOpen)
     sava <- actionNew "SAVA" "Save"    (Just "Just a Stub") (Just stockSave)
     svaa <- actionNew "SVAA" "Save As" (Just "Just a Stub") (Just stockSaveAs)
     exia <- actionNew "EXIA" "Exit"    (Just "Just a Stub") (Just stockQuit)

     cuta <- actionNew "CUTA" "Cut"   (Just "Just a Stub") (Just stockCut)
     copa <- actionNew "COPA" "Copy"  (Just "Just a Stub") (Just stockCopy)
     psta <- actionNew "PSTA" "Paste" (Just "Just a Stub") (Just stockPaste)

     hlpa <- actionNew "HLPA" "Help"  (Just "Just a Stub") (Just stockHelp)

     agr <- actionGroupNew "AGR"
     mapM_ (actionGroupAddAction agr) [fma, ema, hma]
     mapM_ (\ act -> actionGroupAddActionWithAccel agr act Nothing)
       [newa,opna,sava,svaa,cuta,copa,psta,hlpa]

     actionGroupAddActionWithAccel agr exia (Just "e")

     ui <- uiManagerNew
     uiManagerAddUiFromString ui uiDecl
     uiManagerInsertActionGroup ui agr 0

     maybeMenubar <- uiManagerGetWidget ui "/ui/menubar"
     let menubar = case maybeMenubar of
```

```
                        (Just x) -> x
                        Nothing -> error "Cannot get menubar from string."
    boxPackStart box menubar PackNatural 0

    maybeToolbar <- uiManagerGetWidget ui "/ui/toolbar"
    let toolbar = case maybeToolbar of
                        (Just x) -> x
                        Nothing -> error "Cannot get toolbar from string."
    boxPackStart box toolbar PackNatural 0

    actionSetSensitive cuta False

    onActionActivate exia (widgetDestroy window)
    mapM_ prAct [fma,ema,hma,newa,opna,sava,svaa,cuta,copa,psta,hlpa]

    widgetShowAll window
    onDestroy window mainQuit
    mainGUI

uiDecl=  "<ui>\
\           <menubar>\
\            <menu action=\"FMA\">\
\              <menuitem action=\"NEWA\" />\
\              <menuitem action=\"OPNA\" />\
\              <menuitem action=\"SAVA\" />\
\              <menuitem action=\"SVAA\" />\
\              <separator />\
\              <menuitem action=\"EXIA\" />\
\            </menu>\
\           <menu action=\"EMA\">\
\              <menuitem action=\"CUTA\" />\
\              <menuitem action=\"COPA\" />\
\              <menuitem action=\"PSTA\" />\
\           </menu>\
\            <separator />\
\            <menu action=\"HMA\">\
\              <menuitem action=\"HLPA\" />\
\            </menu>\
\           </menubar>\
\           <toolbar>\
\            <toolitem action=\"NEWA\" />\
\            <toolitem action=\"OPNA\" />\
\            <toolitem action=\"SAVA\" />\
\            <toolitem action=\"EXIA\" />\
\            <separator />\
\            <toolitem action=\"CUTA\" />\
\            <toolitem action=\"COPA\" />\
\            <toolitem action=\"PSTA\" />\
\            <separator />\
\            <toolitem action=\"HLPA\" />\
\           </toolbar>\
\          </ui>" </pre>"


prAct :: ActionClass self => self -> IO (ConnectId self)
prAct a = onActionActivate a $ do name <- actionGetName a
                                  putStrLn ("Action Name: " ++ name)
```

# Tutoriel Gtk2Hs 7.2 - Menus contextuels, actions radios et bascules

Les menus sont normalement simplement ajoutés à une fenêtre, mais ils peuvent aussi être affichés temporairement en réponse à un clic de la souris. Par exemple, un menu de contexte peut être affiché quand l'utilisateur clique sur le bouton droit de la souris.

On peut obtenir un menu contextuel en utilisant le nœud popup. Par exemple :

```
uiDecl = "<ui> \
\          <popup>\
\            <menuitem action=\"EDA\" />\
\            <menuitem action=\"PRA\" />\
\            <menuitem action=\"RMA\" />\
\            <separator />\
\            <menuitem action=\"SAA\" />\
\          </popup>\
\        </ui>"
```

Construire un menu contextuel nécessite les mêmes étapes qu'un menu ou une barre de menus. Une fois que l'on a créé les actions et mis celles-ci dans un ou plusieurs groupes, on crée l'interface graphique, on ajoute la chaîne de caractère XML puis les groupes. On extrait alors les widgets. Dans l'exemple ci-dessus, nous avons créé 4 actions avec les noms listés ci-dessus. Un menu contextuel ne s'affichant pas pendant une capture d'écran, il n'y a pas d'image disponible.

Les menus contextuels ne s'empaquettent pas et pour les afficher, il faut utiliser la fonction :

```
menuPopup :: MenuClass self => self -> Maybe (MouseButton,TimeStamp)
```

Ceci est documenté dans Graphics.UI.Gtk.MenuComboToolbar.Menu dans la documentation de l'API. Dans l'exemple, on fait apparaître le menu en cliquant le bouton droit de la souris, et le second argument peut être Nothing. La fonction est la même q'avec la boite d'évènement du chapitre 6.2. Mais ici, on peut utiliser la fenêtre elle-même au lieu de la boite d'évènement.

```
onButtonPress window (\x -> if (eventButton x) == RightButton
                            then do menuPopup (castToMenu pop) Nothing
                                    return (eventSent x)
                            else return (eventSent x))
```

Le seul bémol est que le widget retourné par le ui manager est du type Widget est que la fonction menuPopup prend un argument d'un type qui est une instance de MenuClass. Il faut donc utiliser :

```
castToMenu :: GObjectClass obj => obj -> Menu
```

Cette fonction est également documentée dans la section Graphics.UI.Gtk.MenuComboToolbar.Menu. Le listing complet de l'exemple est :

```
import Graphics.UI.Gtk

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Click Right Popup",
                 windowDefaultWidth := 250,
                 windowDefaultHeight := 150 ]
```

```
    eda <- actionNew "EDA" "Edit" Nothing Nothing
    pra <- actionNew "PRA" "Process" Nothing Nothing
    rma <- actionNew "RMA" "Remove" Nothing Nothing
    saa <- actionNew "SAA" "Save" Nothing Nothing

    agr <- actionGroupNew "AGR1"
    mapM_ (actionGroupAddAction agr) [eda,pra,rma,saa]

    uiman <- uiManagerNew
    uiManagerAddUiFromString uiman uiDecl
    uiManagerInsertActionGroup uiman agr 0

    maybePopup <- uiManagerGetWidget uiman "/ui/popup"
    let pop = case maybePopup of
                  (Just x) -> x
                  Nothing -> error "Cannot get popup from string"

    onButtonPress window (\x -> if (eventButton x) == RightButton
                                   then do menuPopup (castToMenu pop) Nothing
                                           return (eventSent x)
                                   else return (eventSent x))

    mapM_ prAct [eda,pra,rma,saa]

    widgetShowAll window
    onDestroy window mainQuit
    mainGUI

uiDecl = "<ui> \
\          <popup>\
\            <menuitem action=\"EDA\" />\
\            <menuitem action=\"PRA\" />\
\            <menuitem action=\"RMA\" />\
\            <separator />\
\            <menuitem action=\"SAA\" />\
\          </popup>\
\        </ui>"

prAct :: ActionClass self => self -> IO (ConnectId self)
prAct a = onActionActivate a $ do name <- actionGetName a
                                  putStrLn ("Action Name: " ++ name)
```

Il y a une autre façon d'utiliser les actions, sans les créer explicitement avec le type ActionEntry :

```
data ActionEntry = ActionEntry {
                                  actionEntryName :: String
                                , actionEntryLabel :: String
                                , actionEntryStockId :: (Maybe String)
                                , actionEntryAccelerator :: (Maybe String)
                                , actionEntryTooltip :: (Maybe String)
                                , actionEntryCallback :: (IO ())
                                }
```

L'utilisation de ces champs a été décrite dans le chapitre 7.1. La fonction actionEntryCallback doit être fournie par le programmeur et sera éxécutée quand cette action est activée.

Pour ajouter une liste d'entrée a une action, on utilise :

```
actionGroupAddActions :: ActionGroup -> [ActionEntry] -> IO ()
```

Le groupe est alors inséré en utilisant uiManagerInsertActionGroup comme précédemment

Des fonctions similaires éxistent pour RadioAction et ToggleAction. les actions radios permettent à l'utilisateur de choisir à partir d'un ensemble de possibilités dont une seule peut être activée. C'est pour cette raison qu'il est logique de les définir toutes ensemble :

```
data RadioActionEntry = RadioActionEntry {
                                           radioActionName :: String
                                         , radioActionLabel :: String
                                         , radioActionStockId :: (Maybe String)
                                         , radioActionAccelerator :: (Maybe String)
                                         , radioActionTooltip :: (Maybe String)
```

```
                                          , radioActionValue :: Int
                                          }
```

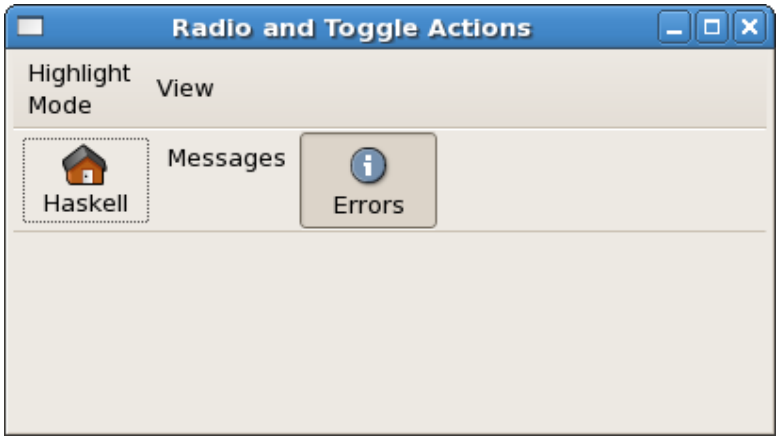
Les cinq premiers champs sont encore utilisés. radioActionValue identifie chacune des sélections possibles. Un ajout à un groupe se fait avec :

```
actionGroupAddRadioActions :: ActionGroup -> [RadioActionEntry] -> Int -> (RadioAction -> IO ())
     -> IO ()
```

Le paramètre Int est la valeur de l'action à activer initialement, ou -1 si il ne doit en y avoir aucune.
La fonction de type (RadioAction -> IO ()) est éxécuté à chaque fois qu'une action est activée.
Les actions toggle ont une valeur Bool est chacune peut être définie ou non. Le widget ToggleActionEntry est définie par :

```
data ToggleActionEntry = ToggleActionEntry {
                                             toggleActionName :: String
                                           , toggleActionLabel :: String
                                           , toggleActionStockId :: (Maybe String)
                                           , toggleActionAccelerator :: (Maybe String)
                                           , toggleActionTooltip :: (Maybe String)
                                           , toggleActionCallback :: (IO ())
                                           , toggleActionIsActive :: Bool
                                          }
```

L'exemple ci-dessous présente l'utilisation des actions à bascule ainsi que des actions radio.
**Note :** The toggleActionCallback function has the wrong value on my platform ; the workaround is, of course, to use the not function.




Les boutons radio pourraient servir à controler un mode de surbrillance comme dans l'éditeur de texte gedit. Le premier menu a un bouton et deux sous-menus qui contiennent les éléments restant. En outre, un des boutons radio se trouve dans une barre d'outils. Cette disposition est controlée par la première définition XML.
Les actions bascule sont des items dans un autre menu, et deux de ceux là sont aussi placés dans la barre d'outils. Cet agencement est déterminé par la seconde définition XML.
Ce qui est intéressant, c'est que le uiManager peut ajouter ces définitions ui juste en les ajoutant comme montré ci-dessous. Vous pouvez donc définir vos menus dans des modules séparés et les combiner facilement dans le module principal. D'après la documentation, le ui manager est assez efficace pour cela, et bien sur vous pouvez aussi utiliser les noms dans les définitions XML pour différencier les chemins. Mais rappelons que la chaine String qui décrit un nom d'action doit être unique pour chaque action.
Il est également possible de supprimer des menus et des barres d'outils en utilisant les fonctions MergeId et uiManagerRemoveUi. Dans ce sens, vous pouvez gérer les menus et les barres d'outils de façon dynamique.

```
import Graphics.UI.Gtk

main :: IO ()
main= do
     initGUI
     window <- windowNew
     set window [windowTitle := "Radio and Toggle Actions",
                 windowDefaultWidth := 400,
                 windowDefaultHeight := 200 ]
```

```
      mhma <- actionNew "MHMA" "Highlight\nMode" Nothing Nothing
      msma <- actionNew "MSMA" "Source"          Nothing Nothing
      mmma <- actionNew "MMMA" "Markup"          Nothing Nothing

      agr1 <- actionGroupNew "AGR1"
      mapM_ (actionGroupAddAction agr1) [mhma,msma,mmma]
      actionGroupAddRadioActions agr1 hlmods 0 myOnChange

      vima <- actionNew "VIMA" "View" Nothing Nothing

      agr2 <- actionGroupNew "AGR2"
      actionGroupAddAction agr2 vima
      actionGroupAddToggleActions agr2 togls

      uiman <- uiManagerNew
      uiManagerAddUiFromString uiman uiDef1
      uiManagerInsertActionGroup uiman agr1 0

      uiManagerAddUiFromString uiman uiDef2
      uiManagerInsertActionGroup uiman agr2 1

      mayMenubar <- uiManagerGetWidget uiman "/ui/menubar"
      let mb = case mayMenubar of
                    (Just x) -> x
                    Nothing -> error "Cannot get menu bar."

      mayToolbar <- uiManagerGetWidget uiman "/ui/toolbar"
      let tb = case mayToolbar of
                    (Just x) -> x
                    Nothing -> error "Cannot get tool bar."

      box <- vBoxNew False 0
      containerAdd window box
      boxPackStart box mb PackNatural 0
      boxPackStart box tb PackNatural 0

      widgetShowAll window
      onDestroy window mainQuit
      mainGUI

hlmods :: [RadioActionEntry]
hlmods = [
      RadioActionEntry "NOA" "None"    Nothing Nothing Nothing 0,
      RadioActionEntry "SHA" "Haskell" (Just stockHome)  Nothing Nothing 1,
      RadioActionEntry "SCA" "C"       Nothing Nothing Nothing 2,
      RadioActionEntry "SJA" "Java"    Nothing Nothing Nothing 3,
      RadioActionEntry "MHA" "HTML"    Nothing Nothing Nothing 4,
      RadioActionEntry "MXA" "XML"     Nothing Nothing Nothing 5]

myOnChange :: RadioAction -> IO ()
myOnChange ra = do val <- radioActionGetCurrentValue ra
                   putStrLn ("RadioAction " ++ (show val) ++ " now active.")

uiDef1 = " <ui> \
\           <menubar>\
\              <menu action=\"MHMA\">\
\                 <menuitem action=\"NOA\" />\
\                 <separator />\
\                 <menu action=\"MSMA\">\
\                    <menuitem action= \"SHA\" /> \
\                    <menuitem action= \"SCA\" /> \
\                    <menuitem action= \"SJA\" /> \
\                 </menu>\
\                 <menu action=\"MMMA\">\
\                    <menuitem action= \"MHA\" /> \
\                    <menuitem action= \"MXA\" /> \
\                 </menu>\
\              </menu>\
\           </menubar>\
\            <toolbar>\
\              <toolitem action=\"SHA\" />\
\            </toolbar>\
\           </ui> "

togls :: [ToggleActionEntry]
togls = let mste = ToggleActionEntry "MST" "Messages" Nothing Nothing Nothing (myTog mste) False
```

```
            ttte = ToggleActionEntry "ATT" "Attributes" Nothing Nothing Nothing (myTog ttte)
                False
            erte = ToggleActionEntry "ERT" "Errors" (Just stockInfo) Nothing Nothing (myTog erte
                )  True
        in [mste,ttte,erte]

myTog :: ToggleActionEntry -> IO ()
myTog te = putStrLn ("The state of " ++ (toggleActionName te)
                     ++ " (" ++ (toggleActionLabel te) ++ ") "
                     ++ " is now " ++ (show $ not (toggleActionIsActive te)))
uiDef2 = "<ui>\
\          <menubar>\
\            <menu action=\"VIMA\">\
\             <menuitem action=\"MST\" />\
\             <menuitem action=\"ATT\" />\
\             <menuitem action=\"ERT\" />\
\            </menu>\
\          </menubar>\
\            <toolbar>\
\              <toolitem action=\"MST\" />\
\              <toolitem action=\"ERT\" />\
\            </toolbar>\
\         </ui>"
```