Cours IHM-1
JavaFX

4 - Conteneurs Layout-Panes

Jacques BAPST

jacques.bapst@hefr.ch

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Disposition des composants
Généralités
Notions de base

Disposition des composants [1]

- La qualité d'une interface graphique repose sur de nombreux éléments mais la disposition des composants dans la fenêtre figure certainement parmi les plus importants.
- Quand on parle de la ***disposition*** *(**layout**)* d'une interface, on s'intéresse plus particulièrement :
  - A la **taille** des composants
  - A la **position** des composants
    - ⇨ Position dans la fenêtre
    - ⇨ Position relative des éléments les uns par rapport aux autres
  - Aux **alignements** et **espacements** qui favorisent la structuration visuelle et influencent l'esthétique de l'interface
  - Aux **bordures** et aux **marges** (notamment autour des conteneurs)
  - Au **comportement dynamique** de l'interface lorsqu'on redimensionne la fenêtre, lorsqu'on déplace une barre de division *(splitpane)*, lorsque le contenu change dynamiquement, etc.

Disposition des composants [2]

- Avec *JavaFX*, il est possible de dimensionner et de positionner les composants de manière absolue (en pixels, mm, etc.).
- Une **disposition avec des valeurs absolues** peut être utile et même nécessaire dans certaines situations particulières. Cependant, dans la plupart des cas, elle **présente de nombreux inconvénients**, car :
  - La **taille** naturelle **des composants** peut varier, en fonction
    - ⇨ De la langue choisie (libellés, boutons, menus, …)
    - ⇨ De la configuration de la machine cible (paramètres de l'OS)
    - ⇨ Du *look & feel*, thème, *skin*, style (CSS) choisi par l'utilisateur
  - La **taille de la fenêtre** peut également varier
    - ⇨ Par le souhait de l'utilisateur
    - ⇨ Par obligation, pour s'adapter à la résolution de l'écran de la machine cible (pour afficher l'intégralité de l'interface)

Disposition des composants [3]

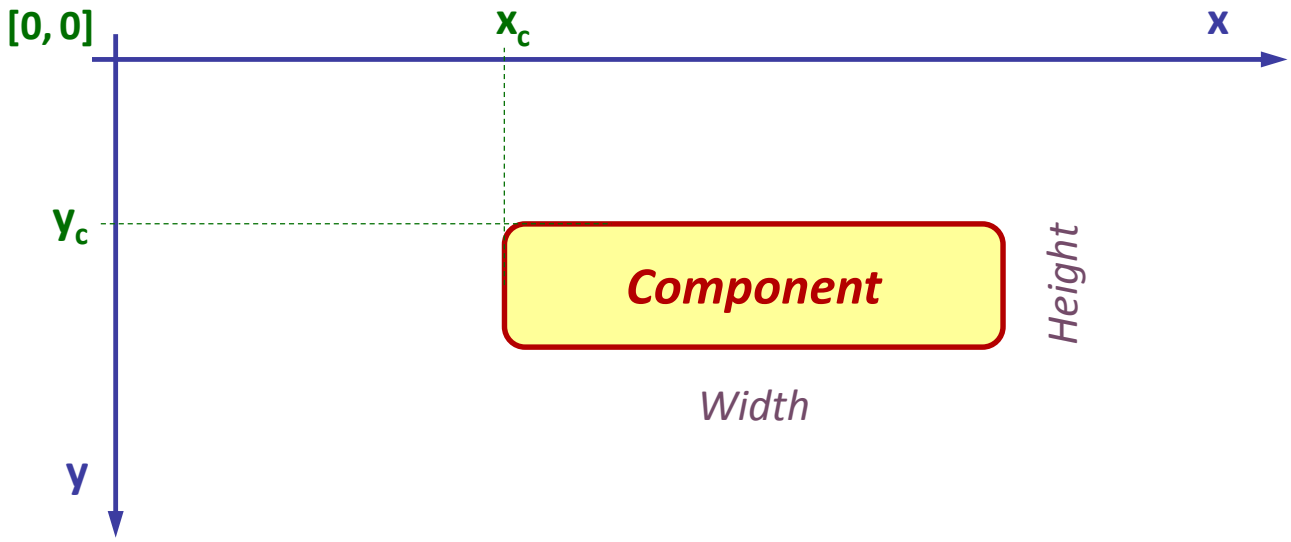
- Pour éviter ces inconvénients on préfère déléguer la disposition des composants à des **gestionnaires de disposition** ***(layout managers)*** qui sont associés à des conteneurs.
- L'idée principale est de définir des **règles de disposition** (des **contraintes**) que le gestionnaire se chargera de faire respecter en fonction du contexte spécifique de la machine cible.
- Avec *JavaFX*, les *layout managers* sont intégrés aux conteneurs *(layout-panes)* et ne sont pas manipulés directement par les programmeurs (seulement au travers des propriétés des conteneurs).
- C'est donc par le choix du *layout-pane* et en fonction des contraintes données que sont déterminées les règles de disposition.
- Plusieurs *layout-panes* sont prédéfinis dans *JavaFX*. Il est également possible de définir ses propres conteneurs avec des règles de disposition spécifiques, mais c'est rarement nécessaire.

Taille des composants

- Les composants *(nodes)* que l'on peut placer dans des conteneurs possèdent des propriétés qui peuvent être prises en compte lors du calcul de leur disposition.
  - `minWidth` : Largeur minimale souhaitée pour le composant
  - `prefWidth` : Largeur préférée (idéale) du composant
  - `maxWidth` : Largeur maximale souhaitée pour le composant
  - `minHeight` : Hauteur minimale souhaitée pour le composant
  - `prefHeight` : Hauteur préférée (idéale) du composant
  - `maxHeight` : Hauteur maximale souhaitée pour le composant
- L'effet de ces propriétés dépend naturellement du type de conteneur *(layout-pane)* utilisé et de ses règles spécifiques de positionnement et de dimensionnement.
  - Elles ne sont donc pas nécessairement prises en compte !

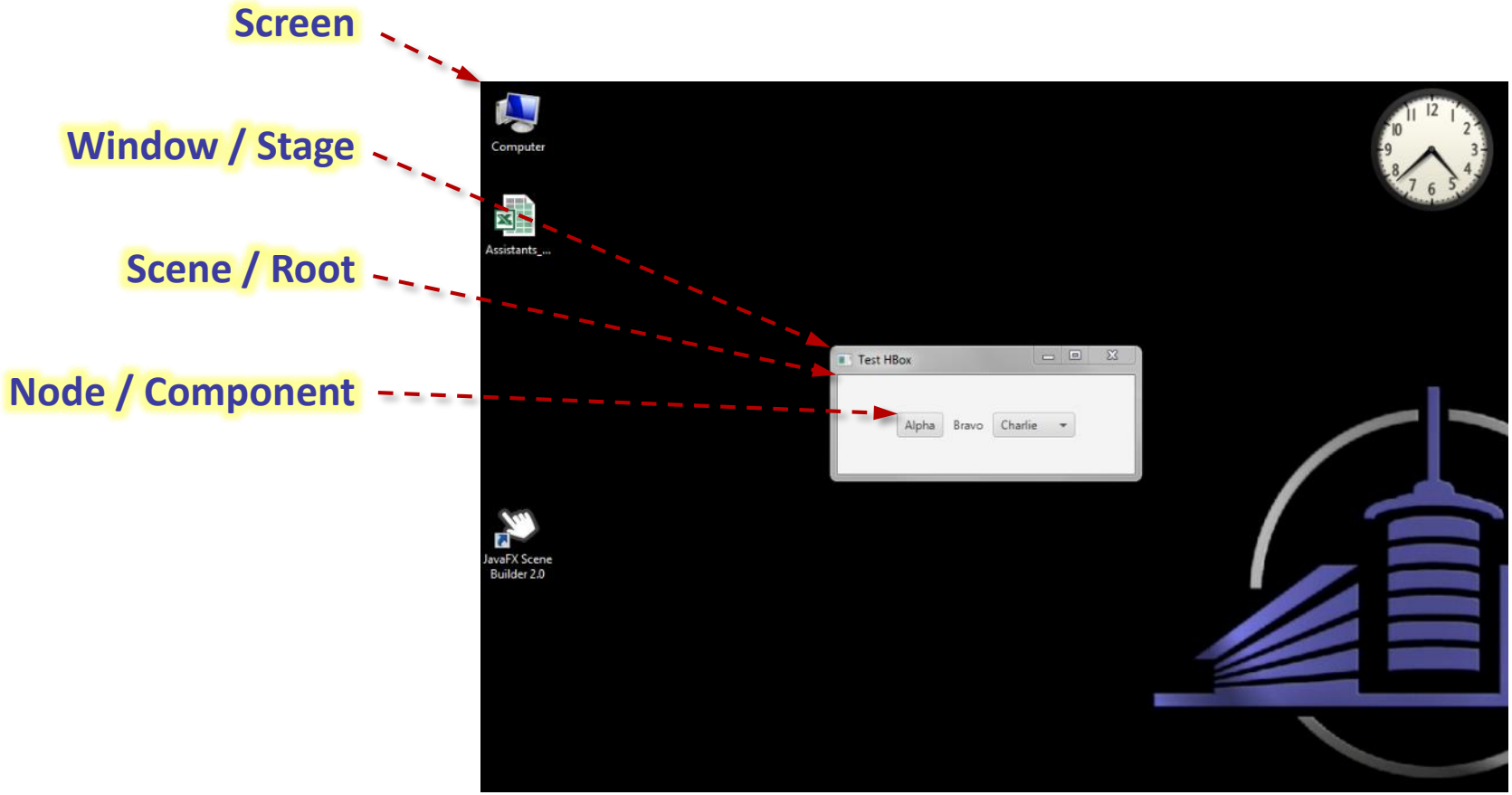
Système de coordonnées [1]

- Le **système de coordonnées** utilisé par *JavaFX* place l'origine dans le coin supérieur gauche du conteneur ou du composant considéré.
- Les valeurs de l'axe *x* croissent vers la droite.
- Les valeurs de l'axe *y* croissent vers le bas (traditionnel en infographie).
- La taille des composants est définie par leur largeur *(width)* et par leur hauteur *(height)*.



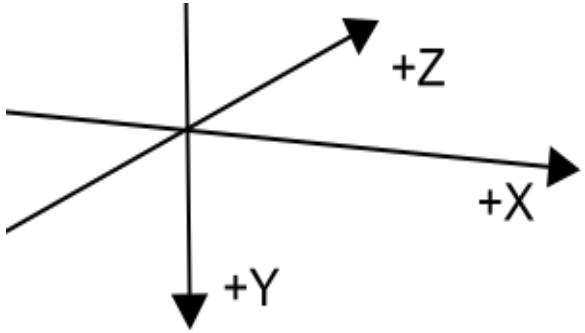
Système de coordonnées [2]

- Coordonnées d'origine de différents éléments d'une application *JavaFX*.



Système de coordonnées [3]

- En réalité, toute la librairie graphique est basée sur un système de coordonnées en trois dimensions (3D).
- La troisième dimension est représentée par l'axe *z* dont les valeurs augmentent lorsqu'on s'éloigne de l'observateur.



- Pour la gestion des interfaces "classiques" (de type WIMP), on ne s'occupe généralement pas de l'axe *z* et on travaille uniquement dans le plan *x*/*y*.
- Dans les API, pratiquement toutes les coordonnées sont données avec des nombres à virgule (généralement des `double`).
  - Le système n'est donc pas nécessairement lié aux pixels de l'écran
  - Des transformations sont ainsi facilement possibles (translation, homothétie, rotation, …) en utilisant l'algèbre linéaire
  - Lors du rendu un alignement sur les pixels peut être automatiquement effectué (voir méthode `snapToPixel()`)

Fenêtre principale [1]

- Par défaut, au lancement d'une application, la fenêtre principale *(primary stage)* est centrée sur l'écran.
- Différentes méthodes peuvent être invoquées sur l'objet `Stage` pour influencer la position et la taille de cette fenêtre :
  - `setX()` : Position en x du coin supérieur gauche
  - `setY()` : Position en y du coin supérieur gauche
  - `centerOnScreen()` : Centrage sur l'écran (par défaut)
  - `setMinWidth()` : Fixe la largeur minimale de la fenêtre
  - `setMinHeight()` : Fixe la hauteur minimale de la fenêtre
  - `setMaxWidth()` : Fixe la largeur maximale de la fenêtre
  - `setMaxHeight()` : Fixe la hauteur maximale de la fenêtre
  - `setResizable()` : Détermine si la fenêtre est redimensionnable
  - `sizeToScene()` : Adapte la taille de la fenêtre à la taille de la scène liée à cette fenêtre (utile si l'on change dynamiquement le contenu du graphe de scène)

Fenêtre principale [2]

- Autres méthodes de l'objet `Stage` :
  - `setTitle()` : Définit le titre de la fenêtre (affiché selon OS)
  - `setFullScreen()` : Place la fenêtre en mode plein-écran ou en mode standard (si paramètre `false`) (selon OS)
  - `getIcons().add()` : Définit l'icône dans la barre de titre
  - `setAlwaysOnTop()` : Place la fenêtre toujours au dessus des autres (généralement à éviter)
  - `setScene()` : Définit la scène (sa racine) qui est associée à la fenêtre
  - `show()` : Affiche la fenêtre à l'écran (et la scène qu'elle contient)
  - `showAndWait()` : Affiche la fenêtre à l'écran et attend que la fenêtre soit fermée pour retourner (méthode bloquante). Cette méthode n'est pas applicable à la fenêtre principale *(primary stage)*.
  - . . . *Et beaucoup d'autres (consulter la documentation)*

Fenêtre principale [3]

- Pour déterminer la **taille de l'écran**, on peut utiliser la classe `Screen` et rechercher le rectangle qui englobe la zone utilisable de l'écran (ou l'intégralité de la surface de l'écran).
- La méthode `getVisualBounds()` prend en compte l'espace occupé par certains éléments du système d'exploitation (barre de tâche, menu, etc.).
- La méthode `getBounds()` retourne par contre un rectangle qui représente la surface totale de l'écran.

```
Screen screen = Screen.getPrimary();
Rectangle2D bounds  = screen.getVisualBounds();

double screenWidth  = bounds.getWidth();
double screenHeight = bounds.getHeight();

double screenDpi    = screen.getDpi();   // Dot per inch
```

Fenêtre principale [4]

- La fenêtre principale, appelée *primary stage* est celle qui est passée en paramètre (par le système) à la méthode `start()`.
- Dans une application *JavaFX,* il est possible de créer d'autres fenêtres indépendantes (d'autres objets `Stage`) et de les gérer.

```
private void createStage() {
  Stage secondStage = new Stage();
  secondStage.setTitle("Second Stage");

  root2 = . . .   // Création de la scène de la deuxième fenêtre

  secondStage.setScene(new Scene(root2));
  secondStage.show();
}

@Override
public void start(Stage primaryStage) {
  . . .
  createStage();
  . . .
}
```

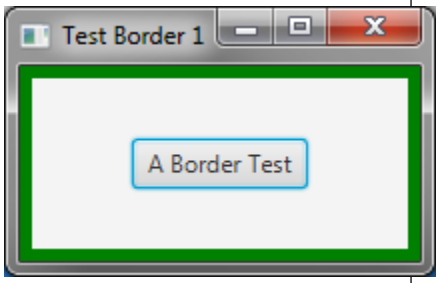
Bordures [1]

- Des **bordures** peuvent être appliquées à toutes les sous-classes de `Region`, notamment autour des conteneurs *(layout-panes)* et autour des composants (moins fréquent).
- C'est la propriété **`border`** (héritée de `Region`) qui est utilisée pour définir les caractéristiques de la bordure.
- Les classes **`Border`** et **`BorderStroke`** permettent de créer des bordures et de les assigner à la propriété `border`. **`Border`** est immuable et permet ainsi de créer des **objets réutilisables** (on peut appliquer une même bordure à plusieurs conteneurs ou composants).
- Pour les bordures, *JavaFX* suit le modèle de conception proposé pour les feuilles de style *CSS3*. Cette spécification est relativement complexe et permet la gestion de bordures assez sophistiquées.
- Les exemples ci-après illustrent une utilisation simple des bordures pour entourer un conteneur (de type `HBox`).

Bordures [2]

- Exemple 1 : bordure rectangulaire

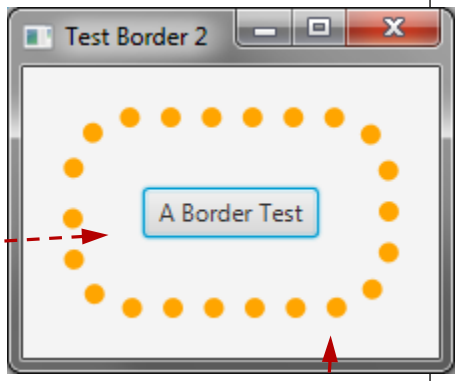
```
primaryStage.setTitle("Test Border 1");

Button btnA = new Button("A Border Test");
HBox   root = new HBox();
root.getChildren().add(btnA);

root.setPadding(new Insets(30, 50, 30, 50));
root.setAlignment(Pos.CENTER);

Border border1 = new Border(
                    new BorderStroke(Color.GREEN,
                                     BorderStrokeStyle.SOLID,
                                     CornerRadii.EMPTY,
                                     new BorderWidths(6),
                                     new Insets(0)           ));
root.setBorder(border1);

primaryStage.setScene(new Scene(root));
primaryStage.show();
```



Bordures [3]

- Exemple 2 : bordure pointillée, aux coins arrondis

```
primaryStage.setTitle("Test Border 2");

Button btnA = new Button("A Border Test");
HBox   root = new HBox();
root.getChildren().add(btnA);

root.setPadding(new Insets(30));
root.setAlignment(Pos.CENTER);

Border border2 = new Border(
                    new BorderStroke(Color.ORANGE,
                                     BorderStrokeStyle.DOTTED,
                                     new CornerRadii(30),
                                     new BorderWidths(10),
                                     new Insets(20)
                                     ));
root.setBorder(border2);

primaryStage.setScene(new Scene(root));
primaryStage.show();
```



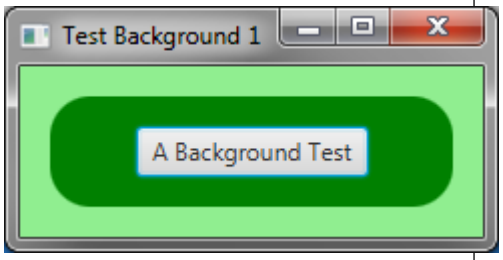
Arrière-plans [1]

- Des couleurs et/ou des images d'arrière-plan peuvent être appliquées à toutes les sous-classes de `Region`, et donc aussi bien aux conteneurs *(layout-panes)* qu'aux composants (*controls*).
- C'est la propriété **`background`** (héritée de `Region`) qui est utilisée pour définir les caractéristiques de l'arrière-plan.
- Les classes **`Background`**, **`BackgroundFill`** et **`BackgroundImage`** permettent de créer des arrière-plans qui peuvent être composés d'une superposition de remplissage *(fill)* et/ou d'images. Ces classes sont immuables et permettent ainsi de créer des **objets réutilisables** (on peut appliquer un même arrière-plan à plusieurs conteneurs ou composants).
- Les exemples qui suivent illustrent l'application d'arrière-plans à des conteneurs (de type `HBox`).

Arrière-plans [2]

- Exemple avec deux remplissages *(fill)* d'arrière-plan superposés :

```
Button btnA = new Button("A Background Test");
HBox   root = new HBox();
root.getChildren().add(btnA);

root.setPadding(new Insets(30, 50, 30, 50));
root.setAlignment(Pos.CENTER);

Background bgGreen = new Background(
                         new BackgroundFill(Color.LIGHTGREEN,
                                            CornerRadii.EMPTY,
                                            null                ),
                         new BackgroundFill(Color.GREEN,
                                            new CornerRadii(20),
                                            new Insets(15)    ));

root.setBackground(bgGreen);
```



Arrière-plans [3]

- Exemple avec la répétition d'une image d'arrière-plan :

```
Button btnA = new Button("A Background Test");
Image  img  = new Image("/resources/warn_1.png");
HBox   root = new HBox();
root.getChildren().add(btnA);

root.setPadding(new Insets(30, 50, 30, 50));
root.setAlignment(Pos.CENTER);

Background bgImg = new Background(
                       new BackgroundImage(
                                              img,
                                              BackgroundRepeat.ROUND,
                                              BackgroundRepeat.ROUND,
                                              BackgroundPosition.CENTER,
                                              BackgroundSize.DEFAULT    ));

root.setBackground(bgImg);
```

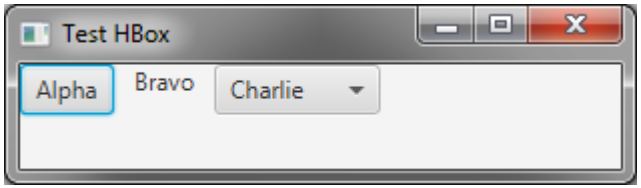
Conteneurs
Layout-Panes de base
(disponibles par défaut)

Layout-Panes

- Dans la création des graphes de scène, les **conteneurs** (appelés ***layout-panes*** ou parfois simplement ***layouts*** ou ***panes***) jouent donc un rôle important dans la structuration et la disposition des composants qui seront placés dans les interfaces.
- En fonction du design adopté (phase de conception de l'interface), il est important de réfléchir au choix des conteneurs qui permettront au mieux de réaliser la mise en page souhaitée.
- Les pages qui suivent décrivent sommairement les *layouts-panes* qui sont prédéfinis dans *JavaFX*.
- Seules les caractéristiques principales sont mentionnées dans ce support de cours. Ce n'est en aucun cas un manuel de référence.
- Il faut donc impérativement consulter la documentation disponible (*Javadoc*, *tutorials*, …) pour avoir une description détaillée de l'API (constantes, propriétés, constructeurs, méthodes, …).

HBox [1]

- Le layout **HBox** place les composants sur une ligne horizontale. Les composants sont ajoutés à la suite les uns des autres (de gauche à droite).



- L'alignement des composants enfants est déterminé par la propriété `alignment`, par défaut `TOP_LEFT` (type énuméré `Pos`).
- L'espacement horizontal entre les composants est défini par la propriété `spacing` (0 par défaut). La valeur de cette propriété peut être passée en paramètre au constructeur (`new HBox(8)`).
- Si possible, le conteneur respecte la taille préférée des composants. Si le conteneur est trop petit pour afficher tous les composants à leur taille préférée, il les réduit jusqu'à `minWidth`.

HBox [2]

- L'ajout des composants enfants dans le conteneur s'effectue en invoquant d'abord la méthode générale `getChildren()` qui retourne la liste des enfants du conteneur et en y ajoutant ensuite le composant considéré (méthodes `add()` ou `addAll()`) :
  - `HBox root = new HBox();`
  - `root.getChildren().add(btnA);`
  - `root.getChildren().addAll(btnOk, btnQuit);`

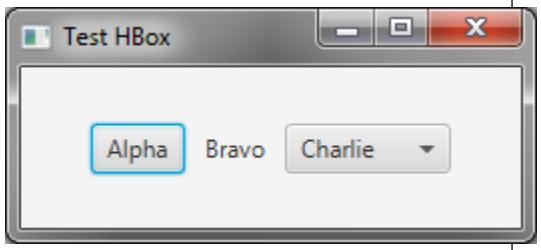
- Des méthodes statiques de `HBox` peuvent être invoquées pour appliquer des contraintes de positionnement :
  - `hgrow()` : permet d'agrandir le composant passé en paramètre jusqu'à sa taille maximale selon la priorité (`Priority`) donnée
  - `margin()` : fixe une marge (objet de type `Insets`) autour du composant passé en paramètre (zéro par défaut `Insets.EMPTY`)

HBox [3]

- Exemple (déclaration des composants et code de la méthode `start()`)

```
private HBox              root;
private Button            btnA   = new Button("Alpha");
private Label             lblB   = new Label("Bravo");
private ComboBox<String>  cbbC   = new ComboBox<>();
```

```
primaryStage.setTitle("Test HBox");

root = new HBox(10);                    // Horizontal Spacing : 10
root.setAlignment(Pos.CENTER);
root.getChildren().add(btnA);
root.getChildren().add(lblB);

cbbC.getItems().addAll("Charlie", "Delta");
cbbC.getSelectionModel().select(0);
root.getChildren().add(cbbC);

primaryStage.setScene(new Scene(root, 300, 100));
primaryStage.show();
```



HBox [4]

- La propriété `padding` permet de définir l'**espace** (**marge**) entre le bord du conteneur et les composants enfants.
  - Un paramètre de type `Insets` est passé en paramètre, il définit les espacements dans l'ordre suivant : *Top, Right, Bottom, Left* ou une valeur identique pour les quatre côtés si l'on passe un seul paramètre

```
root.setPadding(new Insets(10, 5, 10, 5));
root.setPadding(new Insets(10));
```

- Une **couleur de fond** peut être appliquée au conteneur en définissant la propriété `style` qui permet de passer en chaîne de caractères, un style CSS.
  - On définit la propriété CSS `-fx-background-color`.

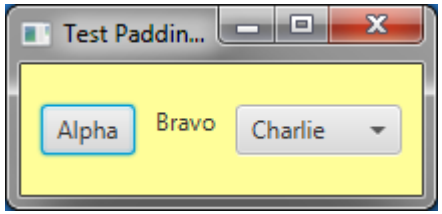
```
root.setStyle("-fx-background-color: #FFFD33");
```

On peut aussi utiliser `setBorder()` *discuté au début du chapitre*

HBox [5]

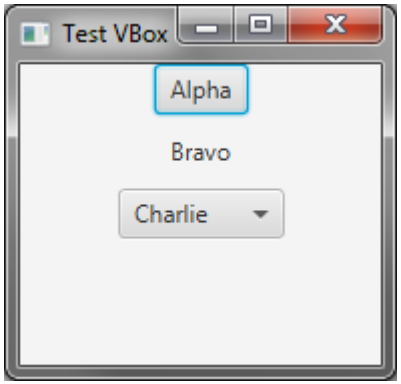
- Exemple (conteneur avec *padding* et couleur de fond)

```
primaryStage.setTitle("Test Padding+Background");

root = new HBox(10);

//--- Top, Right, Bottom, Left Spacing
root.setPadding(new Insets(20, 10, 20, 10));
root.setStyle("-fx-background-color: #FFFE99");

root.getChildren().add(btnA);
root.getChildren().add(lblB);

cbbC.getItems().addAll("Charlie", "Delta");
cbbC.getSelectionModel().select(0);
root.getChildren().add(cbbC);

primaryStage.setScene(new Scene(root));
primaryStage.show();
```



VBox [1]

- Le layout **VBox** place les composants verticalement, sur une colonne. Les composants sont ainsi ajoutés à la suite les uns des autres (de haut en bas).



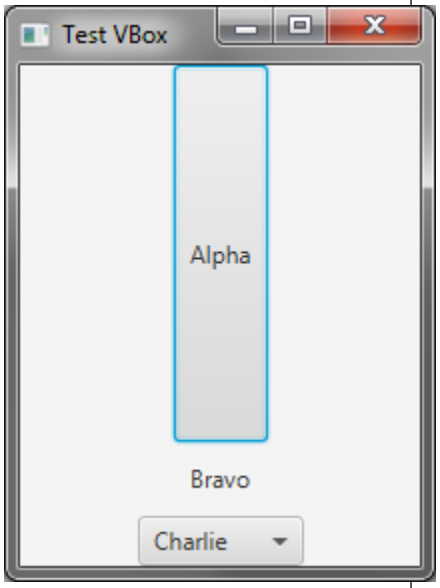
- Toutes les propriétés et méthodes décrites pour le conteneur `HBox` s'appliquent également au conteneur `VBox` avec seulement quelques adaptations assez évidentes, par exemple la propriété `hgrow` devient `vgrow`. La différence essentielle est donc simplement que le sens de l'empilement des composants enfants est vertical et non horizontal.
- Ce conteneur ne sera donc pas décrit plus en détail ici.

VBox [2]

- Exemple (avec agrandissement vertical d'un composant)

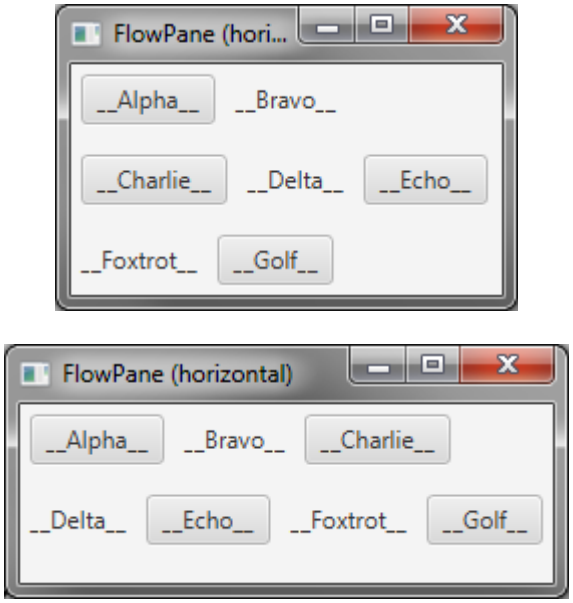
```
private VBox              root;
private Button            btnA   = new Button("Alpha");
private Label             lblB   = new Label("Bravo");
private ComboBox<String>  cbbC   = new ComboBox<>();
```

```
primaryStage.setTitle("Test VBox");

VBox.setVgrow(btnA, Priority.ALWAYS);
btnA.setMaxHeight(Double.MAX_VALUE);

root = new VBox(10);   // Vertical Spacing : 10
root.setAlignment(Pos.TOP_CENTER);
root.getChildren().add(btnA);
root.getChildren().add(lblB);

cbbC.getItems().addAll("Charlie", "Delta");
cbbC.getSelectionModel().select(0);
root.getChildren().add(cbbC);

primaryStage.setScene(new Scene(root, 180, 150));
primaryStage.show();
```

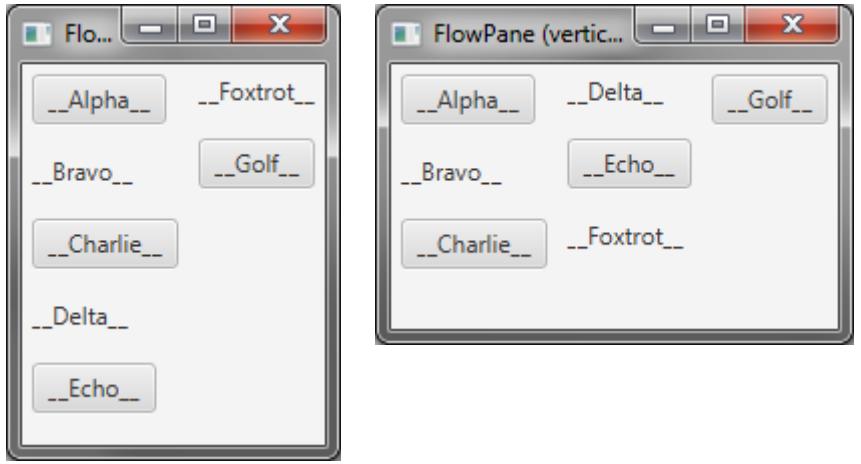


FlowPane [1]

- Le layout **`FlowPane`** place les composants sur une ligne horizontale ou verticale et passe à la ligne ou à la colonne suivante *(wrapping)* lorsqu'il n'y a plus assez de place disponible.
- Un des paramètres du constructeur (de type `Orientation`) détermine s'il s'agit d'un `FlowPane` horizontal (par défaut) ou vertical.



FlowPane horizontal



FlowPane vertical

FlowPane [2]

- L'ajout des composants enfants dans un conteneur `FlowPane` s'effectue en invoquant `getChildren().add(`*node*`)` ou `addAll(`*n, …*`)`
- Quelques propriétés importantes du conteneur `FlowPane` :
  - `hgap` : Espacement horizontal entre les composants ou colonnes (peut aussi être passé au constructeur)
  - `vgap` : Espacement vertical entre les composants ou lignes (peut aussi être passé au constructeur)
  - `padding` : Espacement autour du conteneur (marge)
  - `alignment` : Alignement global des composants dans le conteneur
  - `rowValignment` : Alignement vertical dans les lignes (si horizontal-pane)
  - `columnHalignment` : Alignement horizontal dans les colonnes (si vertical-pane)
  - `prefWrapLength` : Détermine la largeur préférée (si horizontal-pane) ou la hauteur préférée (si vertical-pane)
  - `orientation` : Orientation du *FlowPane* (peut aussi être passé au constructeur)

FlowPane [3]

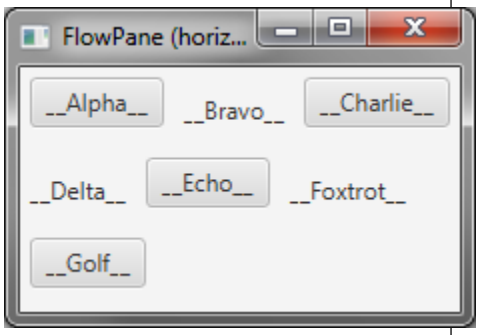
- Exemple (déclaration du conteneur et des composants)

```
private FlowPane  root;

private Button    btnA    = new Button("__Alpha__");
private Label     lblB    = new Label("__Bravo__");
private Button    btnC    = new Button("__Charlie__");
private Label     lblD    = new Label("__Delta__");
private Button    btnE    = new Button("__Echo__");
private Label     lblF    = new Label("__Foxtrot__");
private Button    btnG    = new Button("__Golf__");

. . .
```

FlowPane [4]

- Exemple (code de la méthode `start()`)

```
primaryStage.setTitle("FlowPane (horizontal)");

root = new FlowPane();
root.getChildren().add(btnA);     // Ajout
root.getChildren().add(lblB);     // des
root.getChildren().add(btnC);     // composants
root.getChildren().add(lblD);
root.getChildren().add(btnE);
root.getChildren().add(lblF);
root.getChildren().add(btnG);

root.setPadding(new Insets(5));   // Marge extérieure
root.setHgap(10);                 // Espacement horiz. entre composants
root.setVgap(15);                 // Espacement vertical entre lignes

root.setPrefWrapLength(250);      // Largeur préférée du conteneur

root.setRowValignment(VPos.BOTTOM); // Alignement vertical dans lignes
primaryStage.setScene(new Scene(root));
primaryStage.show();
```



TilePane [1]

- Le layout **`TilePane`** place les composants dans une grille alimentée soit horizontalement (par ligne, de gauche à droite) soit verticalement (par colonne, de haut en bas).
- Un des paramètres du constructeur (de type `Orientation`) détermine s'il s'agit d'un `TilePane` horizontal (par défaut) ou vertical.
- On définit pour la grille un certain nombre de colonnes (propriété `prefColumns`) si l'orientation est horizontale ou un certain nombre de lignes (propriété `prefRows`) si l'orientation est verticale.
- Toutes les **cellules** de cette grille (les tuiles) ont la **même taille** qui correspond à la plus grande largeur préférée et à la plus grande hauteur préférée parmi les composants placés dans ce conteneur.
- Le conteneur `TilePane` est très proche du conteneur `FlowPane`. La différence principale réside dans le fait que toutes les cellules ont obligatoirement la même taille (ce qui n'est pas le cas pour `FlowPane`).

TilePane [2]

- Quelques propriétés importantes du conteneur `TilePane` :
  - `hgap` : Espacement horizontal entre les composants ou colonnes (peut aussi être passé au constructeur).
  - `vgap` : Espacement vertical entre les composants ou lignes (peut aussi être passé au constructeur).
  - `padding` : Espacement autour du conteneur (marge).
  - `alignment` : Alignement global de la grille dans le conteneur (par défaut `TOP_LEFT`).
  - `tileAlignment` : Alignement des composants à l'intérieur de chaque tuile (par défaut `CENTER`). Peut être redéfini individuellement (par la propriété `alignment` de chaque composant).
  - `prefColumns` : Nombre préféré de colonnes (par défaut 5) si horizontal. Détermine la largeur préférée du conteneur.
  - `prefRows` : Nombre préféré de lignes (par défaut 5) si vertical. Détermine la hauteur préférée du conteneur.

TilePane [3]

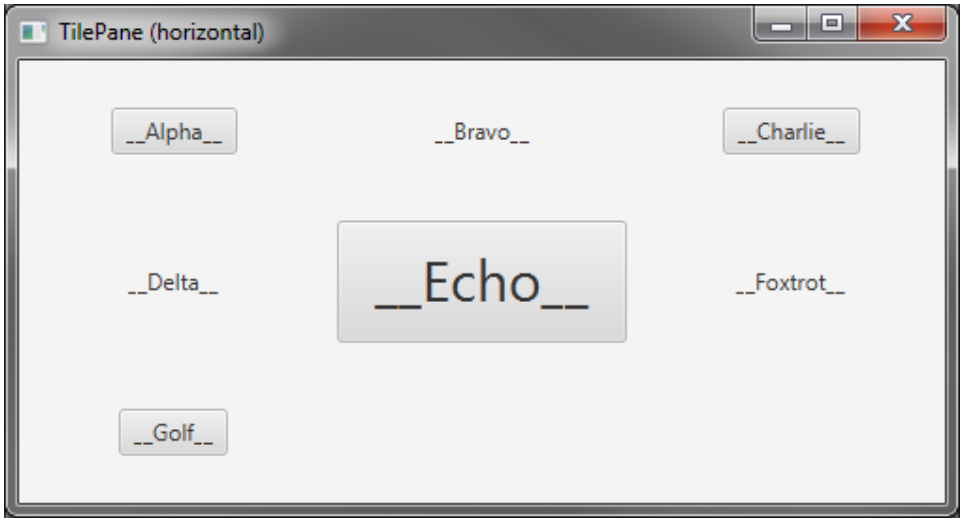
- Suite des propriétés :
  - `prefTileWidth` : Détermine la largeur préférée des tuiles (cellules). Change la valeur calculée par défaut (`USE_COMPUTED_SIZE`).
  - `prefTileHeight` : Détermine la hauteur préférée des tuiles (cellules). Change la valeur calculée par défaut (`USE_COMPUTED_SIZE`).

Remarque : Les propriétés `prefColumns` et `prefRows` ne reflètent pas nécessairement le nombre de colonnes ou le nombre de lignes actuels de la grille.

Ces propriétés servent uniquement à calculer la taille préférée du conteneur. Si la taille du conteneur change, le nombre de lignes et de colonnes sera adapté à l'espace à disposition (*wrapping* automatique des tuiles).

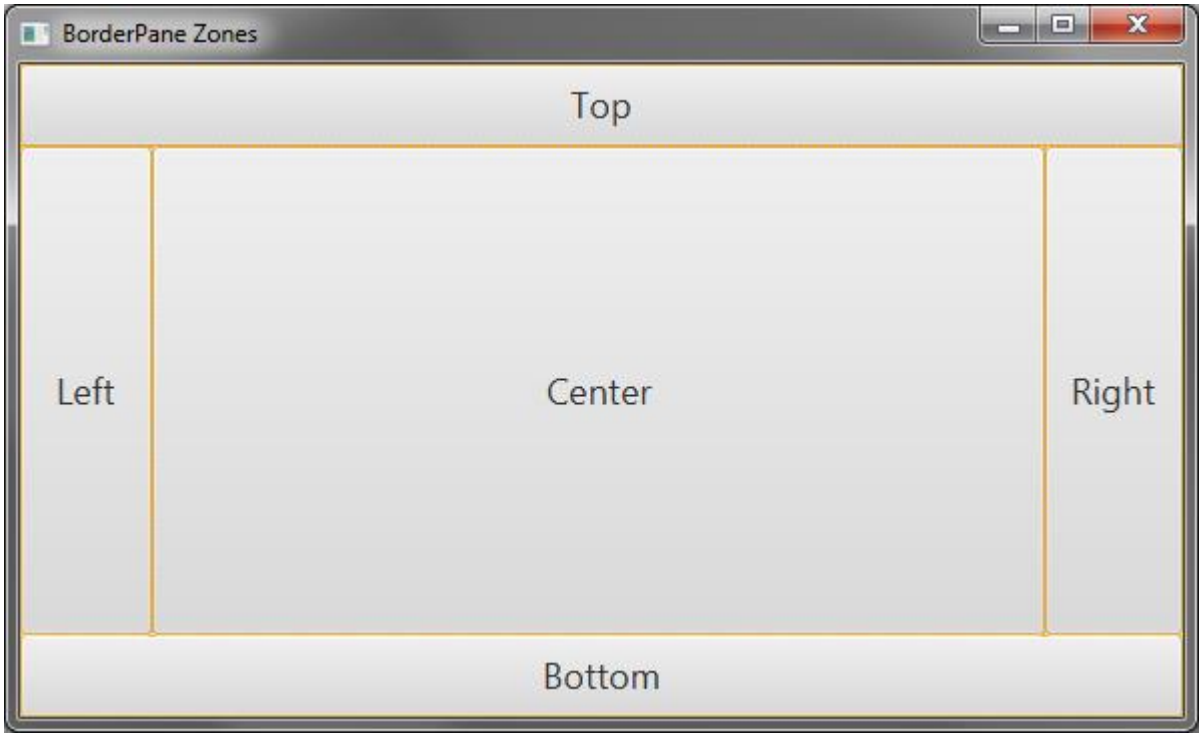
TilePane [4]

- L'ajout des composants enfants dans un conteneur `TilePane` s'effectue en invoquant l'une des deux méthodes :
  - `getChildren().add(`*node*`)`
  - `getChildren().addAll(`*node1*`, `*node2*`, `*node3*`, …)`
- Exemple avec des libellés (`Label`) et des boutons (`Button`)
  - La police de caractères du bouton "`__Echo__`" a été agrandie et détermine la taille de toutes les tuiles (cellules)



BorderPane [1]

- Le conteneur **BorderPane** permet de placer les composants enfants dans cinq zones : *Top, Bottom, Left, Right* et *Center*.
- Un seul objet Node (composant, conteneur, …) peut être placé dans chacun de ces emplacements.

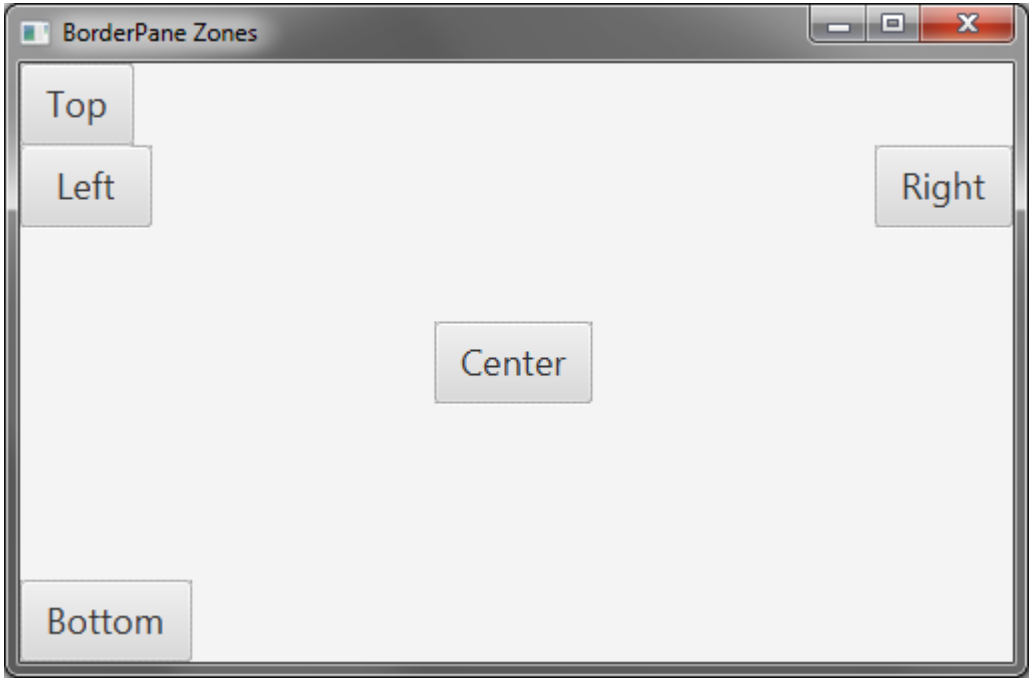


BorderPane [2]

- Les composants placés dans les zones ***Top*** et ***Bottom*** : `setTop()` `setBottom()`
  - Gardent leur hauteur préférée
  - Sont éventuellement agrandis horizontalement jusqu'à leur largeur maximale ou réduit à leur taille minimale en fonction de la largeur du conteneur.
- Les composants placés dans les zones ***Left*** et ***Right*** : `setLeft()` `setRight()`
  - Gardent leur largeur préférée
  - Sont éventuellement agrandis verticalement jusqu'à leur hauteur maximale ou réduit à leur taille minimale en fonction de la hauteur restante entre les (éventuelles) zones *Top* et *Bottom* du conteneur.
- Le composant placé dans la zone ***Center*** : `setCenter()`
  - Est éventuellement agrandi (jusqu'à sa taille maximale) ou réduit (à sa taille minimale) dans les deux directions (largeur et hauteur) pour occuper l'espace qui reste au centre du conteneur.

BorderPane [3]

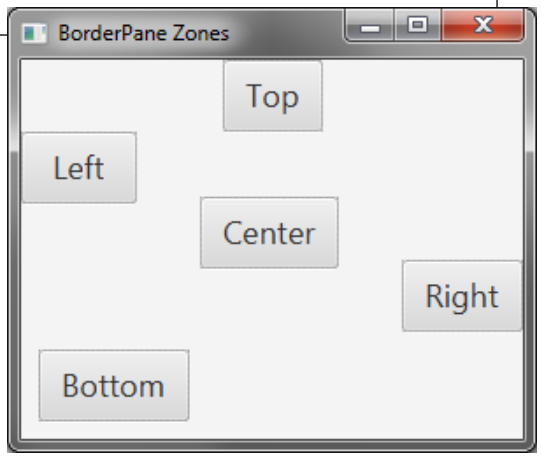
- Certaines zones peuvent être laissées libres (sans composant). Elles n'occupent alors aucune place.
- Si les composants n'occupent pas tout l'espace disponible dans leur zone, leur **point d'ancrage par défaut** (alignement) respectera la disposition suivante :



BorderPane [4]

- Des méthodes statiques de `BorderPane` peuvent être invoquées pour appliquer des contraintes de positionnement :
  - `alignment()` : permet de modifier l'alignement par défaut du composant passé en paramètre (point d'ancrage)
  - `margin()` : fixe une marge (objet de type `Insets`) autour du composant passé en paramètre (par défaut `Insets.EMPTY`)

```
BorderPane.setAlignment(btnTop,    Pos.CENTER);
BorderPane.setAlignment(btnRight, Pos.BOTTOM_CENTER);

BorderPane.setMargin(btnBottom, new Insets(10));
```

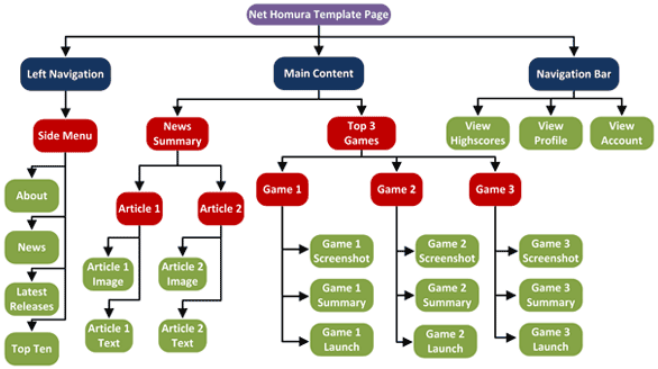


Remarque : Par défaut, la taille maximale des boutons (`Button`) est égale à leur taille préférée. Pour qu'ils s'agrandissent, il faut modifier la propriété `maxWidth` :

```
btnT.setMaxWidth(Double.MAX_VALUE);
```

BorderPane [5]

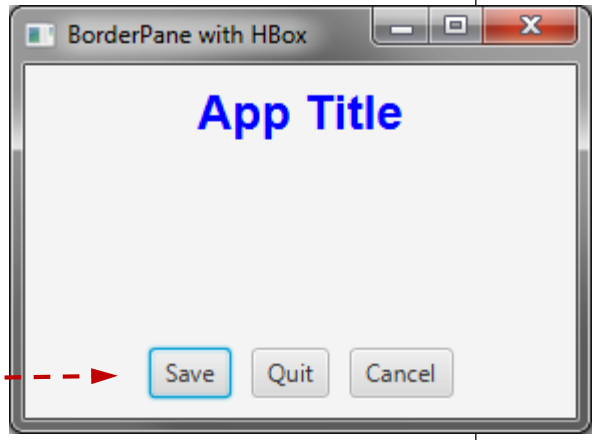
- Le conteneur `BorderPane` est fréquemment utilisé comme conteneur racine du graphe de scène car il correspond à une division assez classique de la fenêtre principale d'une application (barre de titre, barre d'état, zone d'options, zone principale, etc.).
- Pour placer plusieurs composants dans les zones du `BorderPane`, il faut y ajouter des nœuds de type conteneur et ajouter ensuite les composants dans ces conteneurs imbriqués.
- Il est donc très fréquent d'imbriquer plusieurs conteneurs pour obtenir la disposition désirée des composants de l'interface.
- Le **graphe de scène** représente donc un arbre d'imbrication dont la hauteur (nombre de niveaux) dépend du nombre de composants et de la complexité de la structure de l'interface graphique.



BorderPane et HBox

```
private BorderPane root      = new BorderPane();
private HBox       btnPanel  = new HBox(10);
private Label      lblTitle  = new Label("App Title");
private Button     btnSave   = new Button("Save");
private Button     btnQuit   = new Button("Quit");
private Button     btnCancel = new Button("Cancel");
```

```
lblTitle.setFont(Font.font("SansSerif", FontWeight.BOLD, 24));
lblTitle.setTextFill(Color.BLUE);
root.setTop(lblTitle);

//--- Build Buttons Panel
btnPanel.getChildren().add(btnSave);
btnPanel.getChildren().add(btnQuit);
btnPanel.getChildren().add(btnCancel);
btnPanel.setAlignment(Pos.CENTER);
root.setBottom(btnPanel);

BorderPane.setMargin(lblTitle, new Insets(10, 0, 10, 0));
BorderPane.setMargin(btnPanel, new Insets(10, 0, 10, 0));
BorderPane.setAlignment(lblTitle, Pos.CENTER);

primaryStage.setScene(new Scene(root));
```

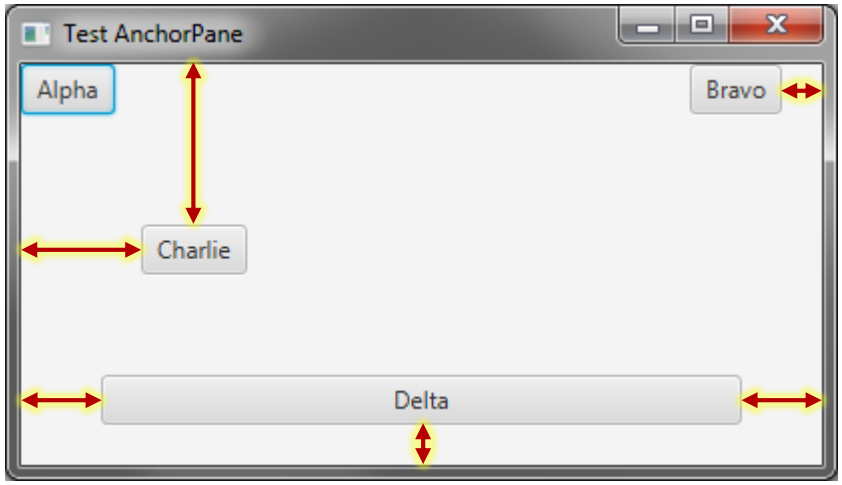


AnchorPane [1]

- Le conteneur **AnchorPane** permet de positionner (ancrer) les composants enfants à une certaine distance des côtés du conteneur (*Top, Bottom, Left* et *Right*).
- Un composant peut être ancré, simultanément, à plusieurs côtés.
- Plusieurs composants peuvent être ancrés à un même côté.
- Ce conteneur ressemble par certains aspects à `BorderPane` mais il y a quelques différences essentielles :
  - Le conteneur `AnchorPane` n'est pas divisé en zones, les composants sont simplement liés (ancrés) par rapport aux bords du conteneur.
  - Un composant peut être ancré à plusieurs bords et même à des bords opposés (*left* et *right* par exemple). Dans ce cas, le composant pourra être étiré ou comprimé pour respecter les contraintes d'ancrage.
  - Plusieurs composants peuvent être ancrés à un même bord. Il pourront dans ce cas, être partiellement ou totalement superposés.

AnchorPane [2]

- Par défaut (sans contrainte), les composants sont ancrés en haut et à gauche *(Top - Left)*.
- Exemple de positionnement avec les contraintes représentées par des flèches :



AnchorPane [3]

- L'ajout des composants enfants dans un conteneur `AnchorPane` s'effectue en invoquant l'une des deux méthodes :
  - `getChildren().add(node)`
  - `getChildren().addAll(node1, node2, node3, …)`
- Pour définir les **contraintes d'ancrage**, il faut invoquer des méthodes statiques de `AnchorPane` :
  - `topAnchor()` : définit la distance (`double`) par rapport au haut
  - `rightAnchor()` : définit la distance (`double`) par rapport au côté droit
  - `bottomAnchor()` : définit la distance (`double`) par rapport au bas
  - `leftAnchor()` : définit la distance (`double`) par rapport au côté gauche
- Exemple de contraintes d'ancrage :

```
AnchorPane.setTopAnchor   (btnOk,   10.0);
AnchorPane.setRightAnchor (btnOk,    0.0);
AnchorPane.setBottomAnchor(btnQuit, 15.0);
```

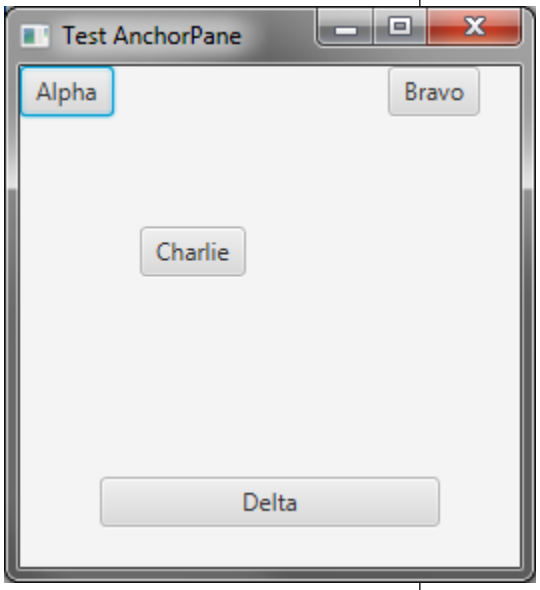
AnchorPane [4]

- Les distances définies dans les contraintes d'ancrage sont données par rapport aux côtés du conteneur. Si le conteneur possède une bordure (`setBorder()`) et/ou un *padding* (`setPadding()`), les dimensions de ces éléments seront pris en compte et s'ajouteront à la distance donnée en contrainte.
- Le layout `AnchorPane` respecte, si possible, la taille préférée des composants enfants.
- Cependant, selon les contraintes données, et si les composants enfants sont redimensionnables (`isResizable()`), leur taille devra être adaptée pour respecter les contraintes données. Dans ce cas, le layout `AnchorPane` ne tiendra pas compte des tailles minimales et maximales des composants lors de leur redimensionnement.
- Si des composants se superposent, c'est l'ordre des ajouts dans le conteneur qui déterminera leur visibilité (le dernier ajouté sera au-dessus).

AnchorPane [5]

```
private AnchorPane root   = new AnchorPane();

private Button     btnA   = new Button("Alpha");
private Button     btnB   = new Button("Bravo");
private Button     btnC   = new Button("Charlie");
private Button     btnD   = new Button("Delta");
```

```
primaryStage.setTitle("Test AnchorPane");

root.getChildren().addAll(btnA, btnB, btnC, btnD);

//--- Contraintes
AnchorPane.setRightAnchor (btnB, 20.0);
AnchorPane.setTopAnchor   (btnC, 80.0);
AnchorPane.setLeftAnchor  (btnC, 60.0);
AnchorPane.setBottomAnchor(btnD, 20.0);
AnchorPane.setLeftAnchor  (btnD, 40.0);
AnchorPane.setRightAnchor (btnD, 40.0);

primaryStage.setScene(new Scene(root, 250, 250));
primaryStage.show();
```



StackPane [1]

- Le conteneur **`StackPane`** empile les composants enfants les uns au dessus des autres dans l'ordre d'insertion : les premiers "au fond", les derniers "au-dessus" *(back-to-front)*.
- Si nécessaire, les composants enfants sont redimensionnés pour s'adapter à la taille du conteneur (la valeur de la propriété `maxSize` est respectée, mais pas celle de `minSize` !).
- Si les composants enfants n'occupent pas toute la place disponible, ils seront alignés selon la valeur de la propriété `alignment` du conteneur (`CENTER` par défaut).
  - Cet alignement par défaut peut être modifié par les contraintes d'alignement spécifiques des composants enfants (voir page suivante)
- L'ajout des composants enfants dans un conteneur de type `StackPane` s'effectue en invoquant l'une des deux méthodes :
  - `getChildren().add(`*`node`*`)`
  - `getChildren().addAll(`*`node1, node2, node3, …`*`)`

StackPane [2]

- Des méthodes statiques de `StackPane` peuvent être invoquées pour appliquer des contraintes de positionnement aux composants :
  - `alignment()` : permet de modifier l'alignement par défaut du composant passé en paramètre
  - `margin()` : fixe une marge (objet de type `Insets`) autour du composant passé en paramètre (par défaut `Insets.EMPTY`)

```
StackPane.setAlignment(icon,  Pos.CENTER_LEFT);
StackPane.setAlignment(label, Pos.CENTER_RIGHT);

StackPane.setMargin(label, new Insets(5));
```

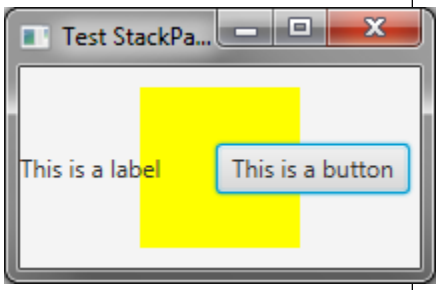
- Le conteneur `StackPane` peut être utile pour créer un composant complexe à partir d'éléments existants en les superposant (placer du texte au dessus d'une forme ou d'une image, combiner des éléments graphiques, etc.).

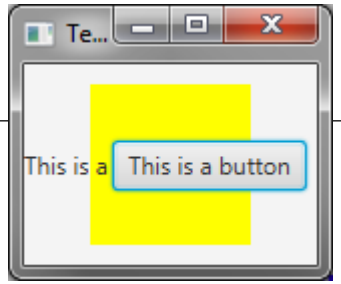
StackPane [3]

- Exemple (état initial et après redimensionnement de la fenêtre) :

```
private StackPane  root  = new StackPane();

private Label      lblA  = new Label("This is a label");
private Button     btnS  = new Button("This is a button");
private Rectangle  rect  = new Rectangle(80, 80, Color.YELLOW);
```

```
primaryStage.setTitle("Test StackPane");

StackPane.setAlignment(lblA, Pos.CENTER_LEFT);
StackPane.setAlignment(btnS, Pos.CENTER_RIGHT);
StackPane.setMargin(btnS, new Insets(5));

root.getChildren().addAll(rect, lblA, btnS);

primaryStage.setScene(new Scene(root, 200, 100));
primaryStage.show();
```



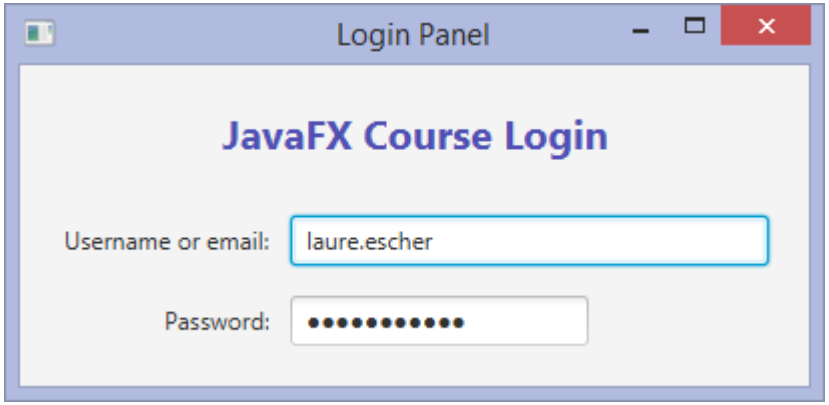


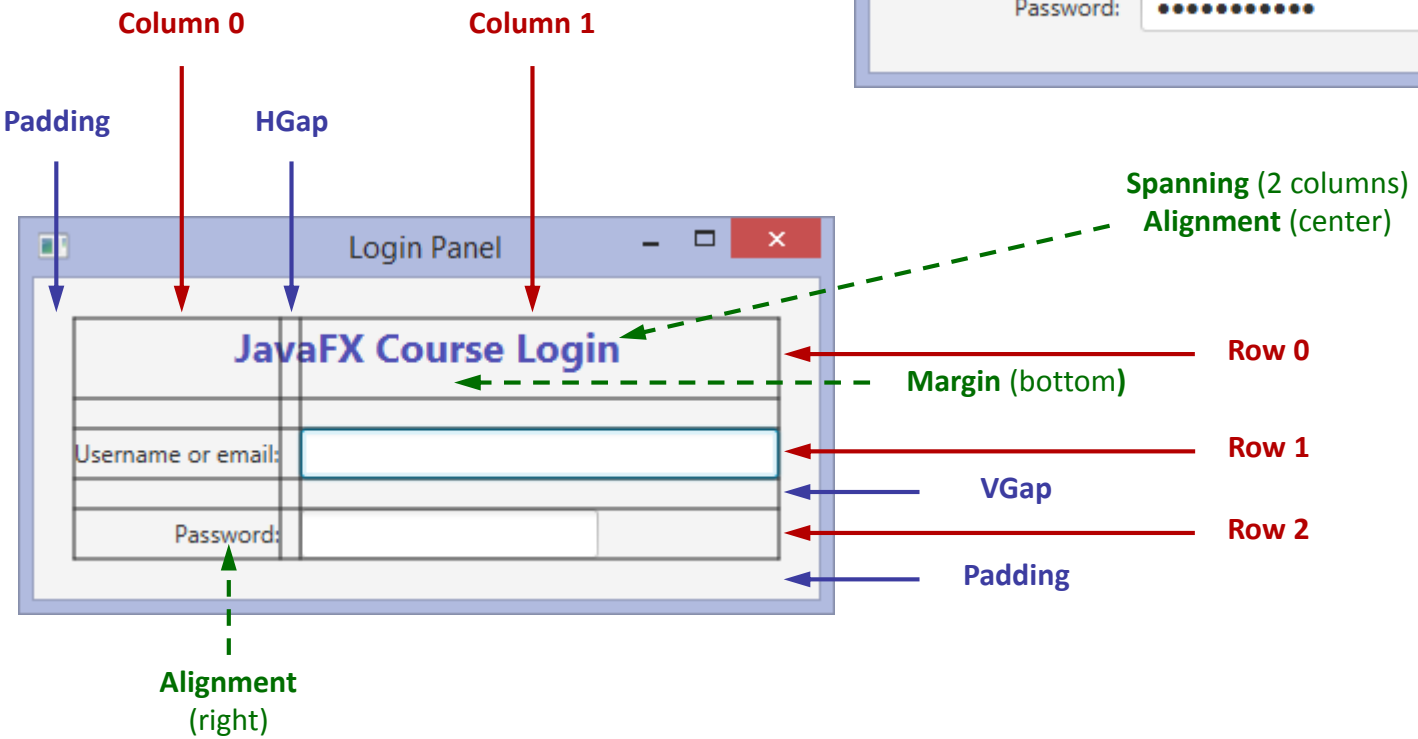
GridPane [1]

- Le conteneur **`GridPane`** permet de disposer les composants enfants dans une grille flexible (arrangement en lignes et en colonnes), un peu à la manière d'une table HTML.
- La grille peut être irrégulière, la hauteur des lignes et la largeur des colonnes de la grille ne sont pas nécessairement uniformes.
- La zone occupée par un composant peut s'étendre *(span)* sur plusieurs lignes et/ou sur plusieurs colonnes.
- Le nombre de lignes et de colonnes de la grille est déterminé automatiquement par les endroits où sont placés les composants.
- Par défaut, la hauteur de chaque ligne est déterminée par la hauteur préférée du composant le plus haut qui s'y trouve.
- Par défaut, la largeur de chaque colonne est déterminée par la largeur préférée du composant le plus large qui s'y trouve.

GridPane [2]

- Exemple simple avec illustration du découpage (structure de la grille).





GridPane [3]

- Pour placer les composants dans un conteneur `GridPane`, on utilise la méthode **`add()`** qui permet de passer en paramètre le composant ainsi que les contraintes principales de placement :
  - Indice de la colonne et de la ligne (numérotation commence à zéro)
  - Nombre de colonnes et de lignes de *spanning* (par défaut : 1)
  - Par exemple, pour l'interface de la page précédente :

```
root.add(lblTitle,    0, 0, 2, 1);  // Title (2 cols spanning)
root.add(lblUsername, 0, 1);        // Username label
root.add(tfdUsername, 1, 1);        // Username text-field
root.add(lblPassword, 0, 2);        // Password label
root.add(pwfPassowrd, 1, 2);        // Password text-field
```

- Les composants peuvent être ajoutés dans n'importe quel ordre.
- On peut ajouter dans un `GridPane` n'importe quel objet de type `Node`, donc également des conteneurs (⇨ imbrication).

GridPane [4]

- Quelques propriétés importantes du conteneur `GridPane` :
  - `hgap` : Espacement horizontal entre les colonnes
  - `vgap` : Espacement vertical entre les lignes
  - `alignment` : Alignement de la grille dans le conteneur (si elle n'occupe pas tout l'espace)
  - `padding` : Espacement autour de la grille (marge)
  - `gridLinesVisible` : Affichage des lignes de construction de la grille. Très utile pour la mise au point *(debugging)* de l'interface.

GridPane [5]

- Il est possible de placer plusieurs composants dans une même cellule de la grille.
  - Ils s'afficheront comme dans un conteneur `StackPane`
    - ⇨ Les composants seront empilés, le dernier ajouté sera "au-dessus"
- Les contraintes de placement des composants peuvent être modifiées dynamiquement (durant le cours du programme) :
  - *Position* : Modifier les propriétés `columnIndex` et `rowIndex`
  - *Spanning* : Modifier les propriétés `columnSpan` et `rowSpan`
- La taille des lignes et des colonnes de la grille peut être gérée
  - Par des **contraintes de lignes et de colonnes** qui s'appliquent pour tous les composants placés dans la ligne ou la colonne concernée
  - Par des **contraintes individuelles**, appliquées aux composants placés dans la grille

  Les contraintes individuelles sont prioritaires sur celles appliquées globalement aux lignes et aux colonnes.

GridPane [6]

- Les **contraintes globales** de lignes/colonnes sont définies dans des objets de type :
  - `RowConstraints` : Pour les lignes
  - `ColumnConstraints` : Pour les colonnes
- Les contraintes globales sont ensuite associées aux lignes/colonnes du `GridPane` en les ajoutant dans une liste, avec les méthodes :
  - `getRowConstraints.add(`*row_constraint*`)`
  - `getColumnConstraints.add(`*column_constraint*`)`
  - L'ordre des ajouts correspond à l'ordre des lignes/colonnes
- Exemple :

```
ColumnConstraints ctCol0 = new ColumnConstraints(50, 100, 200,
                                                 Priority.ALWAYS,
                                                 HPos.CENTER,
                                                 true            );
root.getColumnConstraints().add(ctCol0);
```

GridPane [7]

- Les contraintes de lignes **`RowConstraints`** possèdent les propriétés suivantes :
  - `minHeight` : Hauteur minimale souhaitée pour la ligne
  - `prefHeight` : Hauteur préférée (idéale) pour la ligne
  - `maxHeight` : Hauteur maximale souhaitée pour la ligne
  - `percentHeight` : Hauteur de la ligne en pourcent de la hauteur de la grille (est prioritaire sur *min-*, *pref-* et *maxHeight*)
  - `valignment` : Alignement par défaut des composants dans la ligne (de type énuméré `VPos` : `TOP`, `CENTER`, `BOTTOM`, `BASELINE`)
  - `vgrow` : Priorité d'agrandissement vertical (de type énuméré `Priority` : `ALWAYS`, `SOMETIMES`, `NEVER`)
  - `fillHeight` : Booléen indiquant si le composant doit s'agrandir (`true`) jusqu'à sa hauteur maximale ou alors garder sa hauteur préférée (`false`). Par défaut : `true`.

GridPane [8]

- Les contraintes de colonnes **`ColumnConstraints`** possèdent les propriétés suivantes :
  - `minWidth` : Largeur minimale souhaitée de la colonne
  - `prefWidth` : Largeur préférée (idéale) de la colonne
  - `maxWidth` : Largeur maximale souhaitée de la colonne
  - `percentWidth`: Largeur de la colonne en pourcent de la largeur de la grille (est prioritaire sur *min-*, *pref-* et *maxWidth*)
  - `halignment` : Alignement par défaut des composants dans la colonne (de type énuméré `HPos` : `LEFT`, `CENTER`, `RIGHT`)
  - `hgrow` : Priorité d'agrandissement horizontal (de type énuméré `Priority` : `ALWAYS`, `SOMETIMES`, `NEVER`)
  - `fillWidth` : Booléen indiquant si le composant doit s'agrandir (`true`) jusqu'à sa largeur maximale ou alors garder sa largeur préférée (`false`). Par défaut : `true`.

GridPane [9]

- On peut également appliquer des **contraintes individuelles** aux composants placés dans un `GridPane`. Ces contraintes sont prioritaires sur les contraintes de lignes et colonnes.
- Les contraintes sont appliquées en invoquant des méthodes statiques de `GridPane` qui permettent de gérer les propriétés suivantes :
  - `halignment` : Alignement horizontal du composant passé en paramètre (`HPos` : `LEFT`, `CENTER`, `RIGHT`)
  - `valignement` : Alignement vertical du composant passé en paramètre (`VPos` : `TOP`, `CENTER`, `BOTTOM`, `BASELINE`)
  - `hgrow` : Priorité d'agrandissement horizontal (`Priority` : `ALWAYS`, `SOMETIMES`, `NEVER`)
  - `vgrow` : Priorité d'agrandissement vertical (`Priority` : `ALWAYS`, `SOMETIMES`, `NEVER`)
  - `margin` : Marge autour du composant (de type `Insets`)

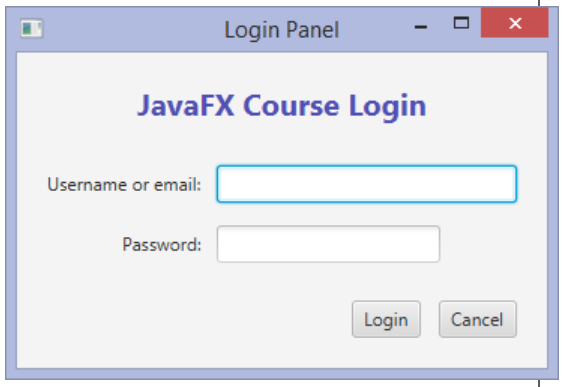
GridPane et HBox [1]

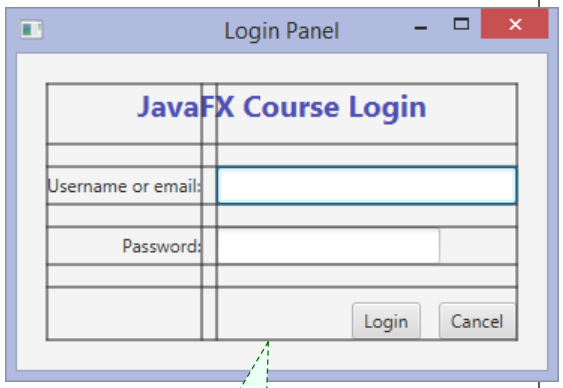
```
private GridPane      root        = new GridPane();
private HBox          btnPanel    = new HBox(12);

private Label         lblTitle    = new Label("JafaFX Course Login");
private Label         lblUsername = new Label("Username or email:");
private TextField     tfdUsername = new TextField();
private Label         lblPassword = new Label("Password:");
private Button        btnLogin    = new Button("Login");
private Button        btnCancel   = new Button("Cancel");
private PasswordField pwfPassword = new PasswordField();
```

```
primaryStage.setTitle("Login Panel");

//--- Title
lblTitle.setFont(Font.font("System", FontWeight.BOLD, 20));
lblTitle.setTextFill(Color.rgb(80, 80, 180));
root.add(lblTitle,    0, 0, 2, 1);
GridPane.setHalignment(lblTitle, HPos.CENTER);
GridPane.setMargin(lblTitle, new Insets(0, 0, 10,0));
```

GridPane et HBox [2]

```
//--- Username (label and text-field)
tfdUsername.setPrefColumnCount(20);
root.add(lblUsername, 0, 1);
root.add(tfdUsername, 1, 1);
GridPane.setHalignment(lblUsername, HPos.RIGHT);

//--- Password (label and text-field)
pwfPassword.setPrefColumnCount(12);
root.add(lblPassword, 0, 2);
root.add(pwfPassword, 1, 2);
GridPane.setHalignment(lblPassword, HPos.RIGHT);
GridPane.setFillWidth(pwfPassword, false);

//--- Button panel
btnPanel.getChildren().add(btnLogin);
btnPanel.getChildren().add(btnCancel);
btnPanel.setAlignment(Pos.CENTER_RIGHT);
root.add(btnPanel, 1, 3);
GridPane.setMargin(btnPanel, new Insets(10, 0, 0,0));
```





setGridLinesVisible(true)

GridPane et HBox [3]

```
//--- Column global constraints
ColumnConstraints ctCol0 = new ColumnConstraints();  // No constraint
ColumnConstraints ctCol1 = new ColumnConstraints(50, 200, 400,
                                                 Priority.ALWAYS,
                                                 HPos.LEFT,
                                                 true);
root.getColumnConstraints().add(ctCol0);
root.getColumnConstraints().add(ctCol1);

//--- GridPane properties
root.setAlignment(Pos.CENTER);
root.setPadding(new Insets(20));
root.setHgap(10);
root.setVgap(15);

primaryStage.setMinWidth(300);
primaryStage.setMinHeight(200);

//  root.setGridLinesVisible(true);  // Uncomment to display grid lines

primaryStage.setScene(new Scene(root));
primaryStage.show();
```

Résumé des conteneurs [1]

Classe	Description
HBox, **VBox**	Place les composants horizontalement (sur une ligne) ou verticalement (dans une colonne).
FlowPane *(horizontal)*	Place les composants horizontalement sur une ligne et passe à la ligne suivante s'il n'y a plus assez de place dans le conteneur *(line-wrapping)*.
FlowPane *(vertical)*	Place les composants verticalement (de haut en bas), en colonne et passe à la colonne suivante s'il n'y a plus assez de place dans le conteneur *(column-wrapping)*.
TilePane *(horizontal)*	Place les composants dans une grille dont les cellules (les tuiles) ont toutes la même taille. Les composants sont ajoutés horizontalement, ligne par ligne.
TilePane *(horizontal)*	Place les composants dans une grille dont les cellules (les tuiles) ont toutes la même taille. Les composants sont ajoutés verticalement, colonne par colonne.

Résumé des conteneurs [2]

Classe	Description
BorderPane	Dispose de cinq emplacements pour placer les composants : *Top, Bottom, Left, Right, Center.*
AnchorPane	Place les composants en respectant une contrainte de distance par rapport à un ou plusieurs bords du conteneur.
StackPane	Place les composants les uns au dessus des autres (empilement). Le dernier ajouté est placé *au-dessus* des autres.
GridPane	Place les composants dans une grille potentiellement irrégulière (par défaut, la taille des lignes et des colonnes est déterminée par le plus grand composant qui y est placé. Les composants sont ajoutés en donnant l'indice de la colonne et de la ligne (la numérotation commence à zéro). La zone d'un composant n'est pas limitée à une seule cellule, elle peut s'étendre sur plusieurs colonnes et plusieurs lignes *(spanning)*. Un composant peut s'agrandir pour occuper toute sa zone.

Suppression de composants

- Pour **supprimer** un ou plusieurs composants d'un conteneur, il faut invoquer la méthode `remove(node)` ou `removeAll(node1, node2, …)` sur la liste des composants du conteneur (retournée par la méthode `getChildren()`).
- Si l'on souhaite **remplacer** un composant par un autre, il faut supprimer l'ancien composant et ajouter ensuite le nouveau.
- Après une modification dynamique du graphe de scène (ajout ou suppression de composants), il est souvent utile d'invoquer la méthode `sizeToScene()` pour forcer le redimensionnement de la fenêtre principale *(stage)* en prenant en compte les changements effectués.

```
. . .
root.getChildren().remove(btnSave);    // Replace button 'Save'
root.getChildren().add(cbxMail);       // by checkbox 'Mail'

primaryStage.sizeToScene();            // Resize window (pack)
. . .
```