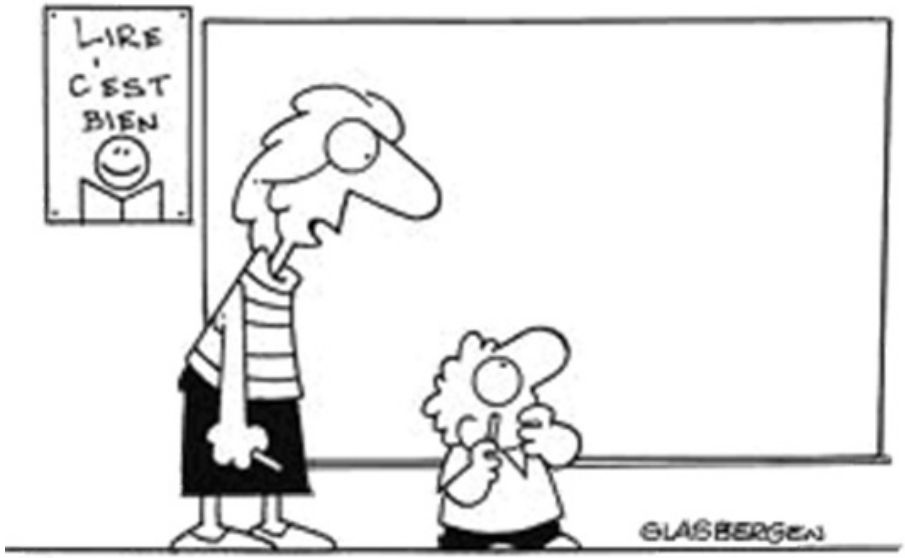
Introduction aux interfaces graphiques avec JavaFX



"Il n'y a pas d'icône à cliquer. C'est un tableau noir."

Au menu

Au menu

Interface Homme-Machine (IHM)

- Regroupe tous les moyens et outils mis en œuvre afin qu'un humain puisse contrôler et communiquer avec une machine
- Mobilise des électroniciens et des mécaniciens (pour la partie matérielle), des informaticiens (pour la partie logicielle), mais aussi des psychologues, des sociologues ou encore des ergonomes (pour la partie utilisateur)
  - Approche multidisciplinaire
- Concevoir une IHM est donc compliquée !

Erratum

- Intitulé du cours abusif
- Se limite à la réalisation d'interfaces graphiques (en Java)
  - Interfaces *WIMP* (*Windows, Icons, Menus and Pointing devices*)
- N'est pas question de la qualité des interfaces :
  - Ergonomie
  - Utilisabilité
  - Adaptabilité
  - Homogénéité
  - Charge cognitive

Interface graphique

- Offre une interaction humaine conviviale
- Boucle et répond à des événements

Modèle

Données + traitements

⟷

Fenêtre

Des composants (*widgets*) qui réagissent *via* des écouteurs (*listeners*)

⟷

Utilisateur

Look vs comportement

- Look :
  - Apparence physique
  - Conception de composants personnalisés
  - Gestion de la mise en page
- Comportement :
  - Interactivité
  - Réponse programmée aux événements

Réalisation d'interfaces graphiques

- Problématique
  - Faire le lien avec le système d'exploitation
  - Ne pas tout ré-implémenter de zéro (bouton, fenêtre, *etc.*)
  - Garder une homogénéité entre les applications
- Requiert l'utilisation de bibliothèques graphiques
  - Bibliothèque d'objets interactifs (les *widgets*) que l'on assemble pour construire l'interface
  - Fonctionnalités pour faciliter la programmation d'applications graphiques interactives (et gérer les entrées)

Bibliothèques graphiques en Java

- AWT (*Abstract Window Toolkit*) – 1996
  - *Package* `java.awt`
  - Utilise au maximum les systèmes de Gestion d'Interface Utilisateur
- Swing – 1997
  - *Package* `javax.swing`
  - Dépend moins de la plate-forme cible et minimise l'utilisation du système graphique sous-jacent
- JavaFX – 2014 (inclus dans JDK8)
  - *Package* `javafx`
  - Successeur de Swing et AWT

JavaFX

- Médias audio et vidéo
- Graphismes 2D et 3D
- Animations et effets
- Syntaxe CSS
- Déploiement Web (*Rich Internet Application*)
- Nouveau langage FXML pour définir des interfaces
- Outil SceneBuilder pour réaliser des interfaces facilement
- *etc.*

Documentation officielle

- Javadoc (à partir de celle de JavaSE)
  - `https://docs.oracle.com/javase/8/javafx/api/toc.htm`
- Documentation et exemples de JavaFX
  - `https://docs.oracle.com/javase/8/javase-clienttechnologies.htm`
- Tutoriel sur les composants de l'interface
  - `https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/`
- Tutoriel sur les *layouts*
  - `https://docs.oracle.com/javase/8/javafx/layout-tutorial/`
- Tutoriel sur la gestion des événements
  - `https://docs.oracle.com/javase/8/javafx/events-tutorial/`

Au menu

Architecture d'une application JavaFX

- `Application` représente une application JavaFX
  - Un processus léger (*thread*) est créé pour exécuter la méthode de démarrage de l'application, traiter les événements d'entrée et exécuter les animations.
- `Stage` est le conteneur de plus haut niveau de l'application
  - S'adapte en fonction du cadre d'utilisation
    - ★ Pour une application lourde, une fenêtre du système d'exploitation
    - ★ Pour une application Web, une fenêtre du navigateur
- `Scene` contient les composants visuels
  - Décrit le graphe de scène (*scene graph*) : hiérarchie de `Node`
- `Node`
  - Formes géométriques (*e.g.*, cercles, lignes), composants d'interface utilisateur (*e.g.*, bouton, menu), conteneurs, *etc.*

Application JavaFX

```
import javafx.application.Application;
import javafx.stage.Stage;

public class MyFirstAppJavaFX extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    ...
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

Application

- Une application JavaFX doit hériter de la classe `javafx.application.Application`
- Il faut ensuite implémenter la méthode abstraite `start`
  - Contient le code de l'application
  - Sera appelée par la méthode `launch` de la classe parent `Application`
- Enfin, dans la méthode `main`, appeler uniquement la méthode `launch`

Application JavaFX

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MyFirstAppJavaFX extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("My first app JavaFX");
    ...
    Scene scene = new Scene(...);
    primaryStage.setScene(scene);
    primaryStage.show();
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

Stage

- Appartient au *package* `javafx.stage`
- Fournit la fenêtre de l'application
- Est instancié automatiquement par l'application en paramètre de la méthode `start`
- Contient une instance de `Scene`
- Méthodes importantes :
  - `setTitle(String)` définit le titre de la fenêtre (affiché selon OS)
  - `setScene(Scene)` spécifie la scène (sa racine) à associer à la fenêtre
    - ★ On peut aussi passer en paramètre la hauteur et la largeur de la scène
  - `show()` affiche la fenêtre à l'écran (et la scène qu'elle contient)
  - `getIcons().add()` définit l'icône à utiliser dans la barre de titre et lorsque la fenêtre est réduite

Stage – suite

- Par défaut, au lancement d'une application, la fenêtre principale (`Stage`) est centrée sur l'écran
- Méthodes pour modifier la position et la taille de cette fenêtre :
  - `setX()` : définit la position horizontale du coin supérieur gauche
  - `setY()` : définit la position verticale du coin supérieur gauche
  - `centerOnScreen()` centre la fenêtre sur l'écran (par défaut)
  - `setMinWidth()` définit la largeur minimale de la fenêtre
  - `setMinHeight()` définit la hauteur minimale de la fenêtre
  - `setMaxWidth()` définit la largeur maximale de la fenêtre
  - `setMaxHeight()` définit la hauteur maximale de la fenêtre
  - `setResizable()` définit si la fenêtre est redimensionnable ou non
  - `sizeToScene()` adapte la taille de la fenêtre à la taille de la scène liée à cette fenêtre
  - `setFullScreen()` place la fenêtre en mode plein-écran ou en mode standard (si paramètre `false`)

Application JavaFX

```
public class MyFirstAppJavaFX extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("My first app JavaFX");
    BorderPane root = new BorderPane();
    root.setCenter(new ImageView(...));
    ...
    VBox box = new VBox(10);
    box.getChildren().add(new Button(...));
    ...
    Scene scene = new Scene(root);
    primaryStage.setScene(scene);
    primaryStage.show();
  }
  public static void main(String[] args) {
    launch(args);
  }
}
```

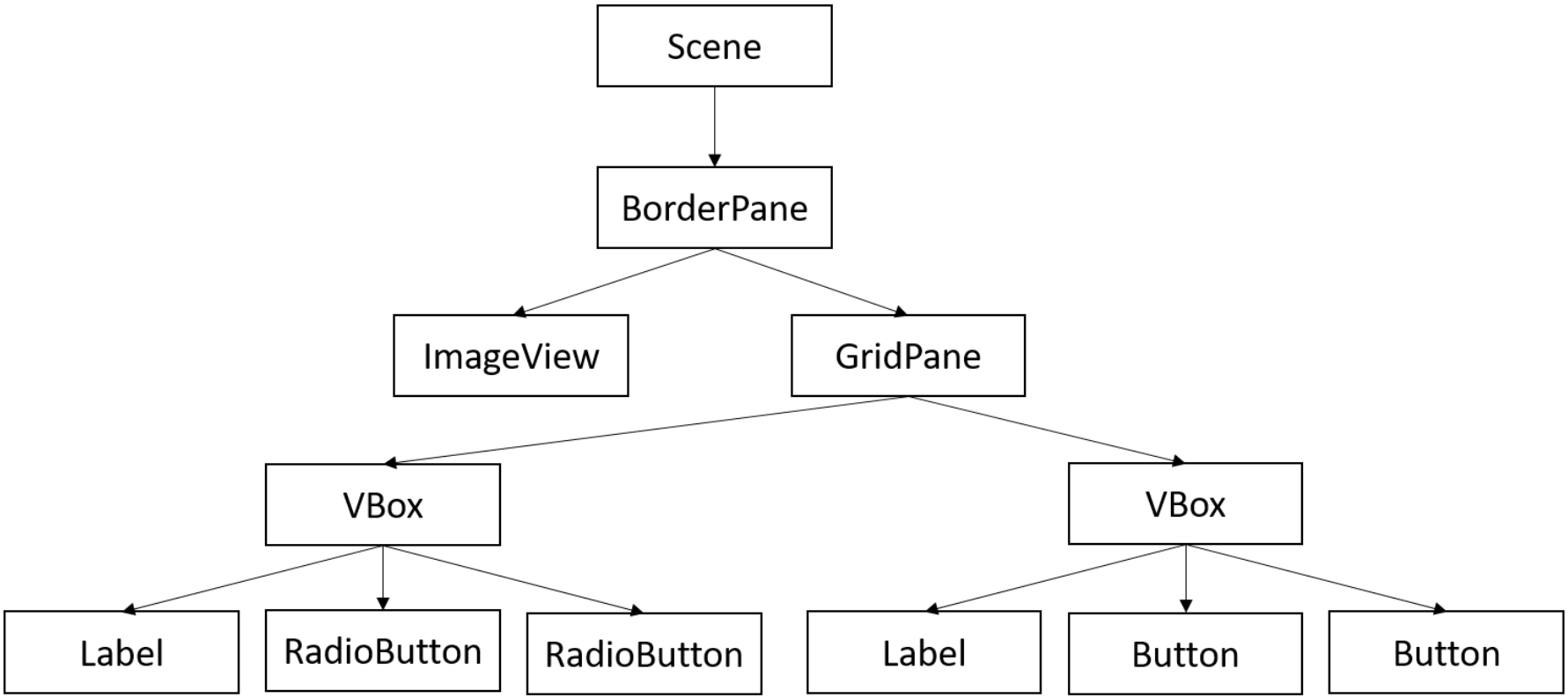
Scene

- Appartient au *package* `javafx.scene`
- Contient les composants visuels
- Décrit le graphe de scène
  - Simple arbre de nœuds (`Node`)

Graphe de scène

- Nœuds de base (feuilles)
  - Représentent généralement des entités atomiques simples
  - Formes géométriques (`Shape`) : cercle, ligne, *etc.*
  - Composants d'interface utilisateur : bouton, menu, *etc.*
  - Contenus graphiques : image, vidéo, *etc.*
- Nœuds intermédiaires (autre que la racine)
  - Conteneurs : panneaux (`Pane`), *etc.*
  - Organisent spatialement leurs enfants à l'écran
- La racine
  - Objet `Scene` qui doit être placé dans un objet `Stage`

Exemple de graphe de scène



Construction du graphe de scène

- Récupérer la liste des enfants d'un conteneur grâce à la méthode `getChildren()`
- Ajouter à cette liste un enfant ou un groupe d'enfants avec les méthodes `add()` ou `addAll()`
- Enlever à cette liste un enfant ou un groupe d'enfants avec les méthodes `remove()` ou `removeAll()`

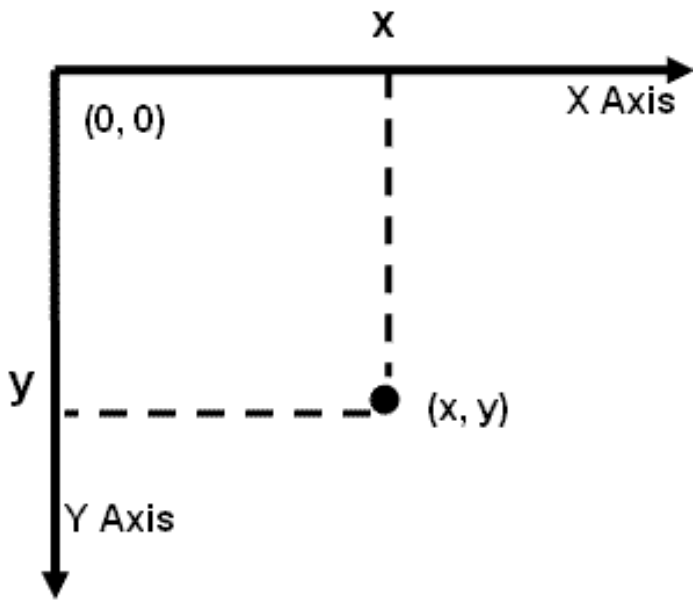
Node

- Cette classe possède un très grand nombre d'attributs permettant de :
  - Positionner le nœud dans le système de coordonnées de son parent
    - ★ Par exemple, en le translatant ou en le tournant
  - Définir le style visuel du nœud
    - ★ Par exemple, son opacité
  - Attacher des gestionnaires d'événements pour répondre aux différents événements liés au nœud qui se produisent
    - ★ Par exemple, son survol avec le pointeur de la souris

.

Système de coordonnées Java

- Le système de coordonnées java est mesuré en pixels, avec l'origine (0, 0) dans son coin supérieur gauche

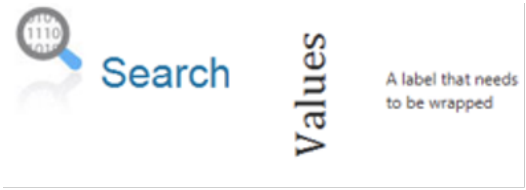


Style CSS JavaFX

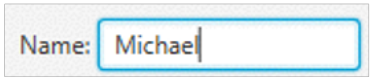
- Est basé sur W3C CSS et permet de personnaliser les nœuds du graphe de scène
- JavaFX utilise le préfixe "-fx-" pour définir ses propriétés CSS
- Une feuille de style utilise les sélecteurs de classe et d'identifiant pour définir des styles
  - Plusieurs classes de style peuvent être appliquées à un seul nœud et un identifiant de style à un nœud unique
  - La syntaxe *.myClass* définit une classe de style
  - La syntaxe *#myId* définit un identifiant de style

```
Button b = new Button("My button");
b.setStyle("-fx-border-color: blue;");
b.setRotate(45);
//Load and use an external CSS style file in a scene
Scene scene = new Scene(...);
scene.getStylesheets().add("style.css");
b.getStyleClass().add("myClass"); // Add a style class
b.setId("myId"); // Add a style id
```

Widgets JavaFX (1)

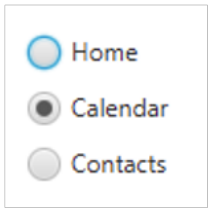


Label & ImageView

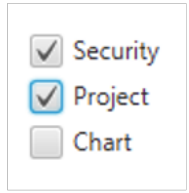


TextField

PasswordField

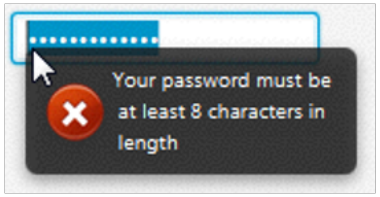


RadioButton

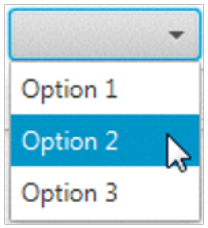


Checkbox

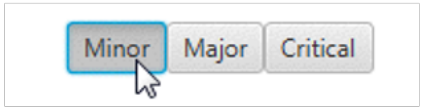
Separator



Tooltip

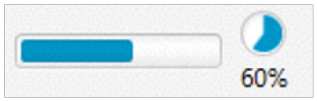


ComboBox



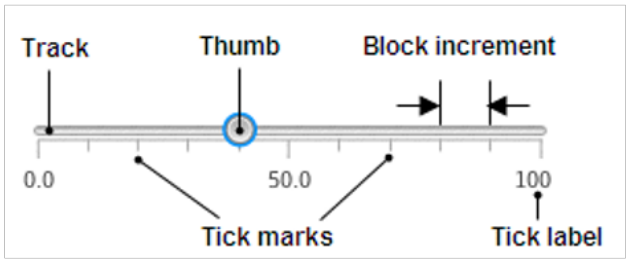
ToggleButton

Button



ProgressBar & ProgressIndicator

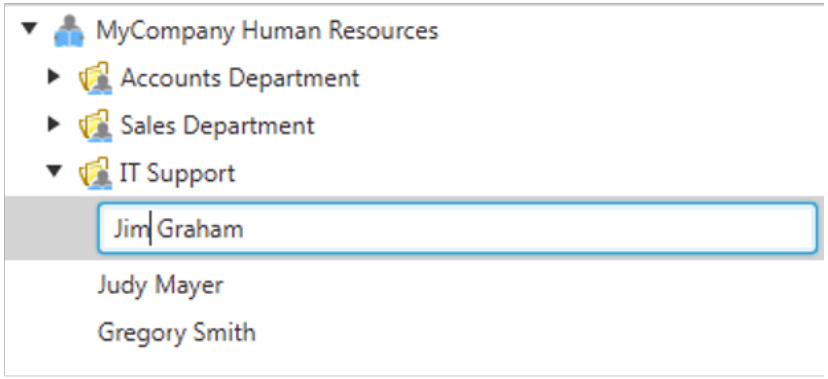
Slider



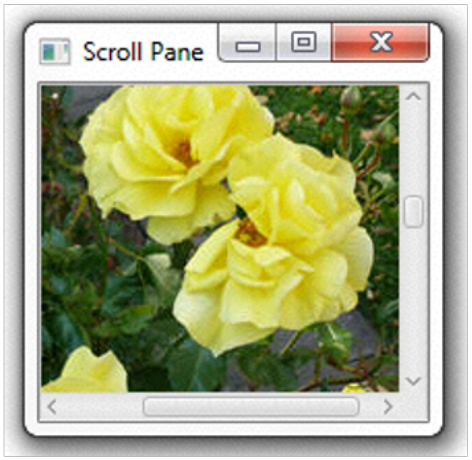
Widgets JavaFX (2)

First Name	Last Name	Email
Emma	White	emma.white@example.com
Emma	Jones	emma.jones@example.com
Ethan	Williams	ethan.williams@example.com
Isabella	Johnson	isabella.johnson@example.com
Jacob	Smith	jacob.smith@example.com
Michael	Brown	michael.brown@example.com

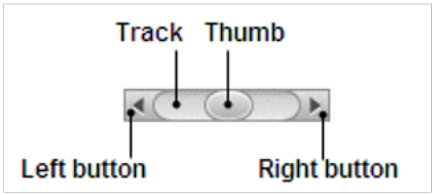
TableView



TreeView



ScrollPane

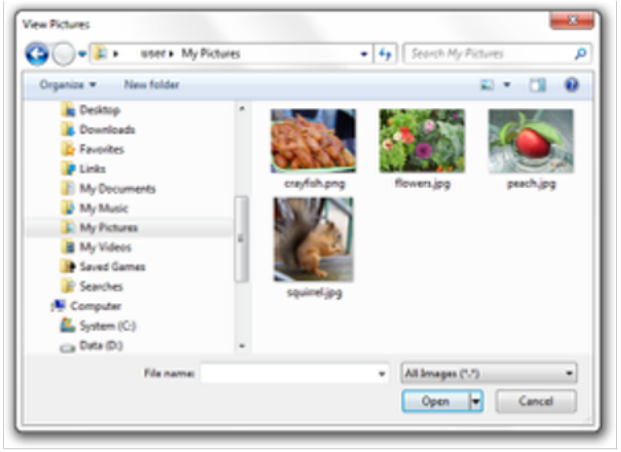


ScrollBar

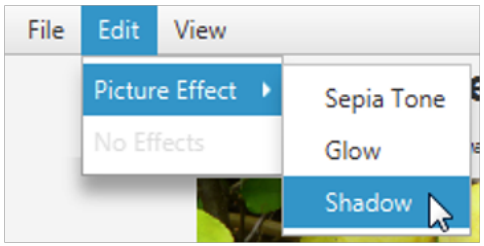
ListView

Widgets JavaFX (3)

ColorPicker



FileChooser



Menu

DatePicker

Boîtes de dialogue JavaFX

- Fenêtres secondaires attachées à une autre fenêtre
- Peuvent être modales
  - Impossibilité de passer à une autre fenêtre de l'application tant que la boîte de dialogue n'est pas fermée
- Différents types de boîtes de dialogue
  - Pour informer, demander une confirmation ou une valeur

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("From␣jml");
alert.setHeaderText(null);
alert.setContentText("JavaFX␣is␣cool!");
alert.initModality(Modality.APPLICATION_MODAL);
alert.showAndWait();
```

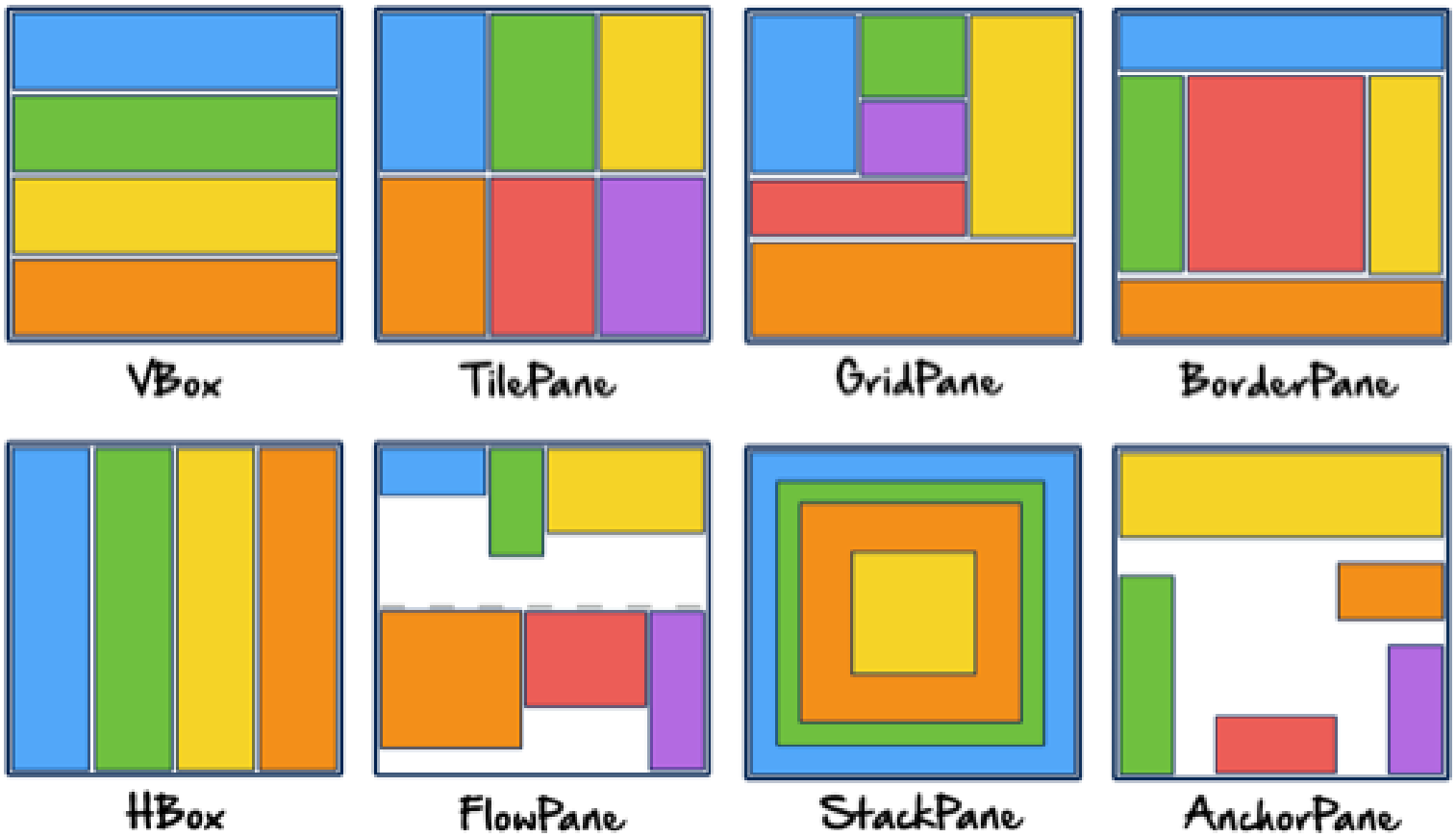
Placement des *widgets*

- Problématique
  - Spécifier les tailles et les positions des composants à la main est possible, mais :
    - ★ Laborieux
    - ★ Potentiellement inadaptées sur une autre configuration (matérielle et/ou logicielle)
    - ★ Besoin de prévoir le redimensionnement de la fenêtre (y compris avec des tailles trop réduites)
  - Il faut être indépendant de la taille des *widgets* et de la fenêtre
- Solution : utiliser des gestionnaires de placement (*layouts*) pour chaque conteneur

Panneau (`Pane`)

- Sert de conteneur à ses nœuds enfants
- Et, souvent, les organise spatialement d'une manière ou d'une autre
- Lors de l'organisation de ses enfants, un panneau peut être amené à redimensionner ceux qui sont redimensionnables
- Pour cela, la classe `Node` fournit un certain nombre de méthodes permettant entre autres au parent de connaître de chacun de ses enfants :
  - La largeur/hauteur minimale (`minWidth`/`minHeight`)
  - La largeur/hauteur maximale (`maxWidth`/`maxHeight`)
  - La largeur/hauteur préférée, *i.e.*, idéale (`prefWidth`/`prefHeight`)
- Les nœuds dont la taille est fixe ont la même valeur minimale, maximale et préférée pour les deux dimensions

Différents types de panneaux JavaFX



FlowPane

- Place les nœuds ligne par ligne horizontalement ou colonne par colonne verticalement
- Les éléments se positionnent automatiquement en fonction de leur taille et de celle du panneau

```
FlowPane pane = new FlowPane(Orientation.HORIZONTAL);
pane.setPadding(new Insets(10, 10, 10, 10));
pane.setVgap(5);
pane.setHgap(5);
pane.setAlignment(Pos.CENTER);
for(int i = 0; i < 10; i++) {
   pane.getChildren().add(new Button("Test " + i));
}
```

BorderPane

- Est doté de 5 zones : une zone centrale et quatre zones latérales (haut, bas, gauche et droite)
- Chaque zone peut être occupée par au plus un nœud
- Toutes les zones, sauf la centrale, sont dimensionnées en fonction du nœud qu'elles contiennent
- La totalité de l'espace restant est attribué à la zone centrale

```
BorderPane pane = new BorderPane();
pane.setTop(new Button("Haut"));
pane.setRight(new Button("Droite"));
pane.setBottom(new Button("Bas"));
pane.setLeft(new Button("Gauche"));
pane.setCenter(new Button("Centre"));
```

GridPane

- Organise ses enfants dans une grille
- La largeur de chaque colonne et la hauteur de chaque ligne sont déterminées en fonction de la taille des nœuds qu'elles contiennent
- Par défaut, les nœuds enfants n'occupent qu'une case de la grille
- Mais il est possible de leur faire occuper une région rectangulaire composée de plusieurs lignes et/ou colonnes contiguës
- Les nœuds enfants peuvent être alignés de différentes manières (à gauche, à droite, au milieu...) dans la case de la grille qu'ils occupent

```
GridPane pane = new GridPane();
for(int i = 0; i < 10; i++) {
  pane.add(new Button("Test " + i), i%5, i/5);
}
```

HBox / VBox

- Place les nœuds dans une seule ligne / colonne

```
HBox hbox = new HBox(10);
for(int i = 0; i < 10; i++) {
  hbox.getChildren().add(new Button("Test " + i));
}
VBox vbox = new VBox(10);
for(int i = 0; i < 10; i++) {
  vbox.getChildren().add(new Button("Test " + i));
}
```

StackPane

- Organise ses enfants de manière à les empiler les uns sur les autres, du premier (placé au bas de la pile) au dernier (placé au sommet)
- Étant donné que ces enfants sont empilés, ils apparaissent visuellement les uns sur les autres
- Tous, sauf le premier, sont donc en général partiellement transparents, faute de quoi ils obscurcissent totalement les nœuds placés sous eux

Conseils et astuces

- Faire un dessin de son interface sur papier afin d'en déduire le graphe de scène à réaliser
- Combiner différents panneaux
  - Des `VBox` et/ou `HBox` dans un `GridPane`
  - Des `FlowPane` dans un `BorderPane`
  - *etc.*
- Regrouper les instructions correspondant à un même composant
- La méthode `sizeToScene()` permet de redimensionner la fenêtre en fonction de son contenu
  - Particulièrement utile si le contenu de la fenêtre change

Au menu

Programmation événementielle

- C'est différent de la programmation procédurale où le code est exécuté dans l'ordre séquentiel des instructions écrites
- Dans la programmation événementielle, le code est exécuté lors de la survenue d'événements (dont les actions de l'utilisateur)
- Boucle événementielle
  - Traite successivement les événements dans leur ordre d'arrivée
    - ★ Les événements sont placés dans une file d'attente

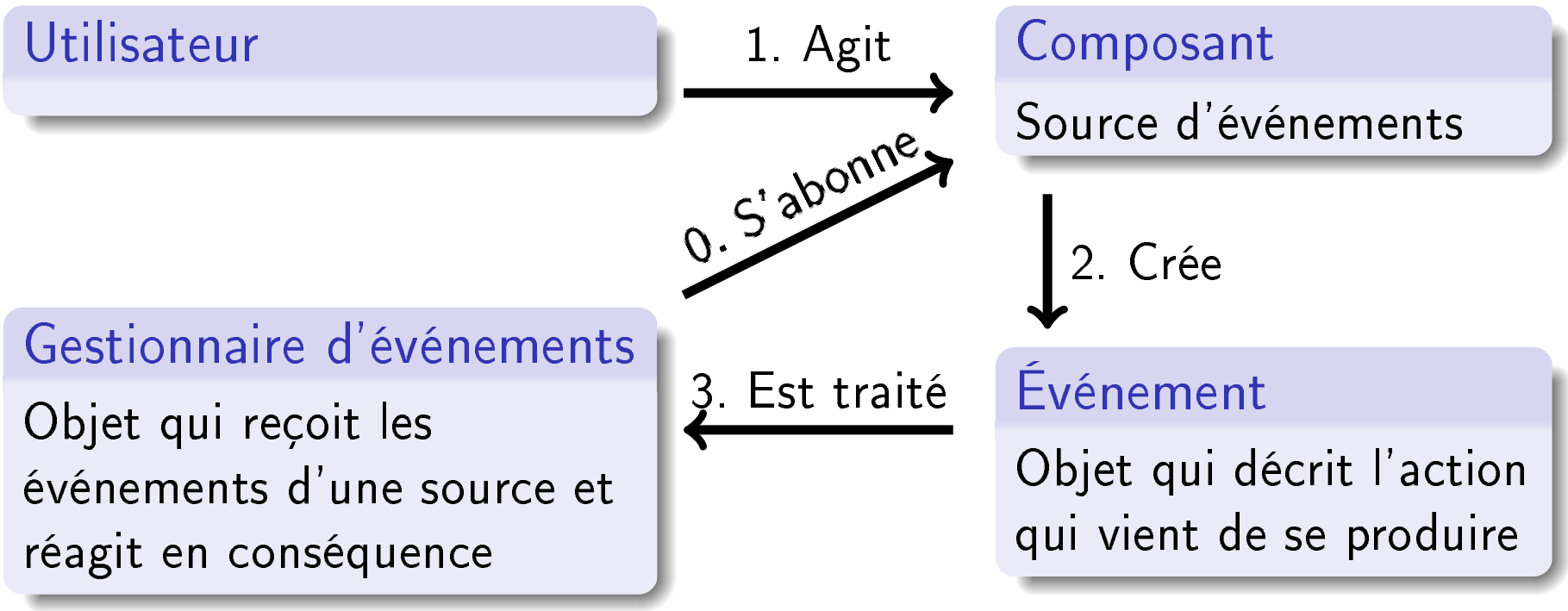
```
tant que le programme n'est pas terminé
  attendre le prochain événement
  traiter cet événement
```

  - Ne doit pas être écrite explicitement, elle est fournie par la bibliothèque de gestion d'interfaces graphiques (ici, JavaFX)

Gestion des événements dans JavaFX

- Dans une application JavaFX, la boucle événementielle s'exécute dans un *thread* séparé, nommé le *thread* d'application JavaFX
  - Démarré automatiquement au lancement de l'application JavaFX
  - Toute manipulation des nœuds du graphe de scène, ainsi que toute création d'objets de type `Scene` ou `Stage`, doivent se faire depuis ce *thread*
- La boucle événementielle est séquentielle
  - Un événement ne peut être traité que lorsque le traitement de l'événement précédent est terminé
  - Par conséquent, une interface graphique est totalement bloquée, tant et aussi longtemps qu'un événement est en cours de traitement
    - ★ Si un calcul lourd doit être effectué en réponse à un événement, mieux vaut réaliser ce traitement dans un autre *thread* (en tâche de fond)

Système interactif



N.B. : plusieurs gestionnaires d'événements peuvent s'abonner à la même source, qui enverra ses événements à chacun de ses abonnés

Événement

- Objet dont la classe hérite de la classe `javafx.event.Event`
- Transporte des informations sur l'événement survenu
  - `type` : événement déclencheur (`EventType`)
    - ★ Ex : `MouseEvent.MOUSE_PRESSED` ou `MouseEvent.MOUSE_RELEASED`
  - `source` : origine de l'événement
  - `target` : nœud sur lequel l'événement agit (`EventTarget`)
    - ★ Ex : pour la souris, c'est le nœud le plus haut placé sous le pointeur
  - Et éventuellement des informations spécifiques à l'événement
- Toutes ces informations sont accessibles *via* des accesseurs, tels que `getType()`, `getSource()`, `getTarget()`, `getX()`, *etc.*

Différents types d'événement

- JavaFX fournit plusieurs sous-classes d'`Event` :
  - `ActionEvent` pour les actions simples sur l'interface (*e.g.*, clic sur un bouton)
  - `KeyEvent` pour les appuis sur les touches du clavier
  - `MouseEvent` pour les déplacements, survols et clics de la souris
  - `ScrollEvent` pour les défilements à l'aide de la molette de la souris, du trackpad ou encore d'un écran tactile
  - `TouchEvent` pour les appuis sur un écran tactile
  - `WindowEvent` pour les événements survenant sur une fenêtre (*e.g.*, fermeture)
  - *etc.*
- Il est également possible de créer ses propres événements en spécialisant la classe `Event` ou l'une de ses sous-classes
  - Transporter des informations spécifiques
  - Obtenir un couplage faible entre des composants logiciels

Gestionnaire d'événements

- Objet implémentant l'interface fonctionnelle et générique `EventHandler<T extends Event>`
  - Le paramètre de type `T` spécifie le type de l'événement géré par le gestionnaire
  - Elle comporte une unique méthode `void handle(T event)`
    - ★ Appelée lorsqu'un événement spécifique du type pour lequel ce gestionnaire est enregistré se produit
- Sert à décrire la réaction à un événement
- Est attaché à une ou plusieurs sources d'événements (souvent un nœud) et ses méthodes sont appelées chaque fois qu'un événement se produit

S'abonner à une source d'événements

- Utiliser la méthode `addEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)` de la classe `Node`
  - `eventType` correspond au type d'événement géré
  - `eventHandler` correspond au gestionnaire d'événements dont la méthode `handle` sera appelée quand l'événement sera déclenché
  - Exemple : *btn*`.addEventHandler(ActionEvent.ACTION, `*handler*`);`
- Ou utiliser les *convenience methods* qui permettent d'enregistrer un gestionnaire d'événements directement auprès de certains composants
  - Syntaxe : `setOn`*XXX*`(EventHandler<T> value)` où *XXX* est le type de l'événement (sans le suffixe `Event`)
  - Exemple :
    - `setOnAction(EventHandler<ActionEvent> value)`
    - `setOnKeyPressed(EventHandler<KeyEvent> value)`
    - `setOnMousePressed(EventHandler<MouseEvent> value)`

Réagir à un événement

- Pour chaque source d'événements (bouton, liste, *etc.*)
  - Définir un gestionnaire d'événements
  - Construire une instance de ce gestionnaire d'événements
  - Ajouter à la source cette instance comme gestionnaire d'événements
- Plusieurs façon de faire / syntaxes possibles
  - Classe interne
  - Classe interne et anonyme
  - Expressions lambda

Classe interne

- Est définie à l'intérieur d'une autre classe (dite externe ou englobante)
- Est un membre de la classe externe dans laquelle elle est imbriquée
  - Peut être déclarée `public`, `protected` , `private` ou *package* selon les mêmes règles de visibilité appliquées à un membre de la classe externe
  - Peut être déclarée `static`
    - La classe interne statique est accessible *via* le nom de la classe externe
    - Toutefois, une classe interne statique ne peut pas accéder aux membres non statiques de la classe externe
- Peut accéder aux variables et méthodes d'instance de la classe externe
  - Pas besoin de passer la référence de la classe externe au constructeur de la classe interne
  - *ClasseExterne*`.this` permet de distinguer les membres non statiques de la classe externe de ceux de la classes interne (`this`)
- Est compilée dans une classe nommée *ClasseExterne*`$`*ClasseInterne*`.class`

Réagir à un événement

Classe interne

```
public class MyFirstAppJavaFX extends Application {
   private Button b;
   @Override
   public void start(Stage primaryStage) throws Exception {
      ...
      b = new Button("clique moi");
      b.setOnAction(new MyBtnHandler());
      ...
   }
   class MyBtnHandler implements EventHandler<ActionEvent> {
      @Override
      public void handle(ActionEvent event) {
         System.out.println(b.getText());
      }
   }
}
```

Classe interne et anonyme

- Est une classe interne sans nom
  - Écrite à l'endroit de son instanciation
  - Étant interne, elle peut accéder aux variables et méthodes d'instance de la classe externe
  - Étant anonyme, son code peut être rapproché de la définition du composant qu'il traite (lisibilité $\nearrow$)
- Attention à la syntaxe ! (toujours impossible d'instancier une interface)
- Doit toujours étendre une super-classe ou implémenter une interface
  - Mais ne peut pas avoir de clause `extends` ou `implements` explicite
  - Doit implémenter toutes les méthodes abstraites de la super-classe ou de l'interface
  - Utilise toujours le constructeur sans argument de sa super-classe pour créer une instance
  - Ou le constructeur `Object()` si implémente une interface
- Est compilée dans une classe nommée *ClasseExterne*`$`*n*`.class`, où *n* est le nombre de classes internes

Réagir à un événement

Classe interne et anonyme

```
public class MyFirstAppJavaFX extends Application {
   private Button b;
   @Override
   public void start(Stage primaryStage) throws Exception {
      ...
      b = new Button("clique moi");
      b.setOnAction(new EventHandler<ActionEvent>() {
         @Override
         public void handle(ActionEvent event) {
            System.out.println(b.getText());
         }
      });
      ...
   }
}
```

Expressions lambda

- Introduites dans Java 8
- Peuvent être considérées comme une méthode anonyme avec une syntaxe concise
- Syntaxe de base :
  - `(type1 param1, type2 param2, ...) -> expression`
  - `(type1 param1, type2 param2, ...) -> { statements; }`
  - Le type de données d'un paramètre peut être explicitement déclaré ou implicitement déduit par le compilateur
  - Les parenthèses peuvent être omises s'il n'y a qu'un seul paramètre sans type de données explicite

Réagir à un événement

Expression lambda (Java 8)

```
public class MyFirstAppJavaFX extends Application {
  private Button b;
  @Override
  public void start(Stage primaryStage) throws Exception {
    ...
    b = new Button("clique moi");
    b.setOnAction(e -> System.out.println(b.getText()));
    ...
  }
}
```

Liaison de données (1)

- JavaFX introduit un nouveau concept, appelé *binding property*, qui permet de lier des données entre elles
- Ces données doivent être encapsulées dans une `Property`
- La modification d'une `Property` entraîne la modification des `Property` qui lui sont associées
- Les attributs des composants JavaFX sont en général des `Property`
  - Elles sont donc liables à d'autres `Property`
    - ★ Celles d'un autre composant de l'API JavaFX
    - ★ Ou celles créées par le développeur
  - Particulièrement utile pour :
    - ★ Coder certains comportements de l'interface
    - ★ Lier des paramètres de l'application à des composants de l'interface

Liaison de données (2)

- Le *binding* lie les valeurs de deux `Property`
- Cette liaison peut être unidirectionnelle ou bidirectionnelle
- Liaison unidirectionnelle : *`property1`*`.bind(`*`property2`*`)`
  - La valeur de *property2* est systématiquement recopiée dans celle de *property1* en cas de changement
- Liaison bidirectionnelle : *`property1`*`.bindBidirectional(`*`property2`*`)`
  - Toute nouvelle valeur de l'une des deux `Property` est recopiée dans l'autre
- Le *binding* permet aussi d'effectuer certaines opérations mathématiques dépendant du type de la valeur à recopier (`add`, `subtract`, `multiply`, *etc.*)
  - Ex : `btn.prefHeightProperty().bind(stage.heightProperty().divide(2));`

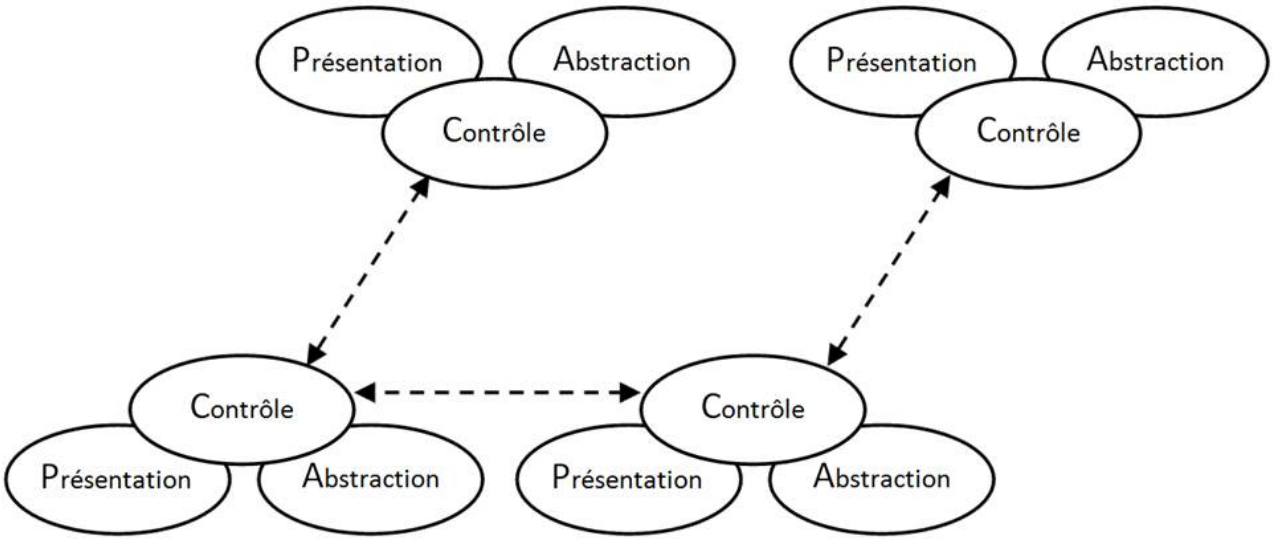
Au menu

Problématique

- Ajouter une IHM dans une application peut être très intrusif
  - Mélange du code des interfaces graphiques avec celui du noyau applicatif (*i.e.*, le code qui modélise les données et fournit des traitements sur ces données)
    - ★ Lisibilité, réutilisation, extensibilité et maintenance ↘
- On aimerait :
  - Séparer l'IHM du reste de l'application
  - Avoir un code modulaire (*i.e.*, la possibilité d'ajouter/retirer/substituer un module sans affecter les autres modules)

Présentation, Abstraction, Contrôle (PAC)

- Modèle abstrait d'architecture logicielle pour les IHM
- Similaire à MVC
- Il organise le système interactif comme une hiérarchie de composants, chacun composé de trois types de facette



- Contrairement à MVC, la partie visualisation n'interagit pas directement avec le modèle !

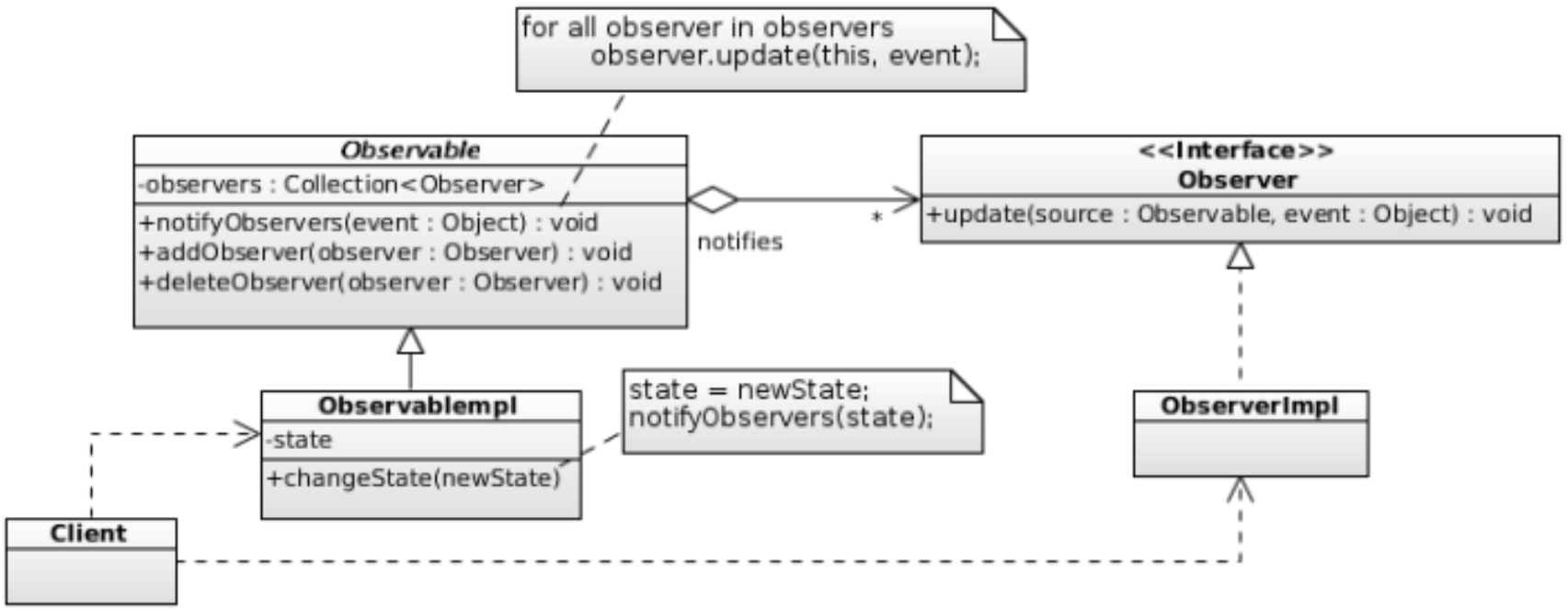
Trois types de facette

- Présentation
  - Purement IHM
  - Peut correspondre à un affichage ou à une saisie de données
- Abstraction
  - Noyau applicatif
  - Modélise la représentation et le traitement des données
- Contrôle
  - Lien entre les deux autres types de facette
  - Garantit la cohérence entre les données du modèle et leurs représentations
  - Traduit les actions de l'utilisateur en opérations sur le modèle

$\Rightarrow$ Utilisation du patron de conception *Observateur* de Java pour réaliser et connecter les différentes facettes

Patron de conception *Observateur*

- Il définit une interdépendance de type un à plusieurs, de telle sorte que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour



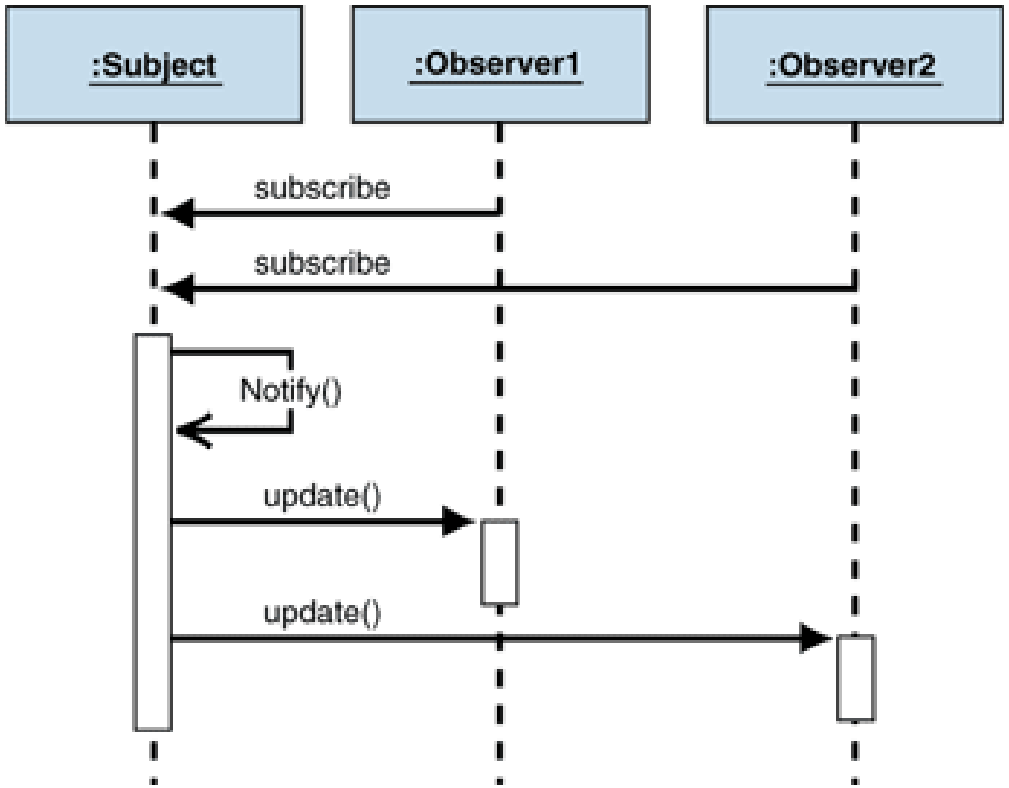
Classe abstraite `Observable`

- Elle fournit une interface pour ajouter/supprimer dynamiquement des observateurs
- Ses méthodes :
  - `addObserver`/`deleteObserver` : ajoute/supprime un observateur
  - `setChanged` : spécifie que l'objet a été modifié
  - `notifyObservers` : notifie les observateurs d'un changement
- Objet observé
  - Il a sa classe qui étend la classe `Observable`
  - Il connaît ses observateurs (un nombre quelconque d'observateurs peut l'observer)
  - Après chaque modification, il appelle ses méthodes `setChanged` puis `notifyObservers` pour notifier ses observateurs que son état a changé

Interface `Observer`

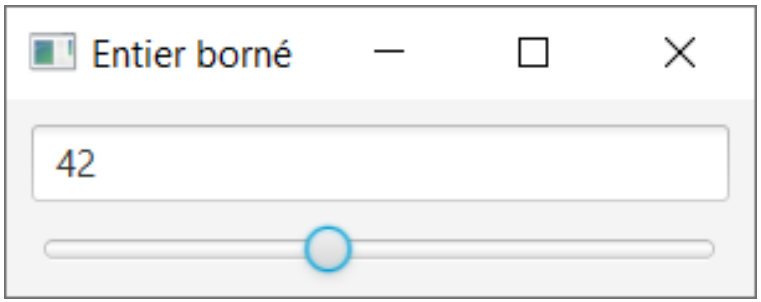
- Elle définit une interface de mise à jour pour les objets qui doivent être notifiés de changement dans un objet observé
- Sa méthode :
  - `update` : met à jour l'objet pour conserver la cohérence de son état avec celui de l'objet observé
- Objet observateur
  - Il a sa classe qui implémente l'interface `Observer`
  - Il doit être ajouté à la liste des observateurs de l'objet observé
  - Il gère une référence sur l'objet qu'il observe
    - ★ Pour l'interroger sur son état afin d'adapter le sien
    - ★ Pour se supprimer de sa liste d'observateurs

Diagramme de séquence



Exemple

- Application manipulant une valeur entière bornée
  - Un `TextField` qui affiche la valeur courante et peut permettre de saisir une autre valeur
  - Un `Slider` qui permet de modifier l'entier en faisant glisser son curseur



- Interactions liées :
  - Si des traitements modifient la valeur de l'entier, les deux composants doivent être mis à jour
  - Si une valeur est saisie dans le `TextField`, il faut vérifier qu'elle figure entre les bornes, et modifier le modèle et le `Slider` en conséquence
  - Si le curseur du `Slider` est déplacé, il faut mettre à jour le modèle et le `TextField`

Problème

- Rien n'est indépendant : composants et modèle liés entre eux
- Risque d'oublier une mise à jour dans le traitement de l'un des événements
  - Introduction d'incohérence entre la vue et le modèle
- Remplacer le `TextField` par un `Spinner` $\Rightarrow$ revoir tout le code
- Solution : approche PAC

Abstraction

```
import java.util.Observable;
public class EntierBorne extends Observable { // est observable
  private int valeur, min, max;
  public EntierBorne(int valeur, int min, int max) {
    this.valeur = valeur;
    this.min = min;
    this.max = max;
  }
  public int getValeur() {
    return this.valeur;
  }
  public int getMin() {
    return this.min;
  }
  public int getMax() {
    return this.max;
  }
  public void setValeur(int valeur) {
    this.valeur = Math.max( Math.min(valeur, this.max), this.min);
    this.setChanged();              // methodes appelees
    this.notifyObservers(null); // a chaque modification
  }
}
```

Présentation

```
public class IHMEntierBorne extends Application {
  private EntierBorne abstraction;
  private TextField textField; // facette de presentation
  private Slider slider; // facette de presentation

  @Override
  public void start(Stage primaryStage) throws Exception {
    abstraction = ...;
    primaryStage.setTitle("Entier borne");
    int valeur = abstraction.getValeur();
    int min = abstraction.getMin();
    int max = abstraction.getMax();
    VBox root = new VBox(10);
    textField = new TextField(""+valeur);
    slider = new Slider(min, max, valeur);
    root.getChildren().addAll(textField, slider);
    [...]
```

- Notez que les 2 composants n'ont pas de référence vers l'abstraction (ils ne pourront donc pas communiquer directement avec le modèle)

Contrôle - Slider

```
public class CtrlSlider implements Observer, ChangeListener<Number> {
  private EntierBorne entier;
  private Slider slider;
  public CtrlSlider(EntierBorne entier, Slider slider) {
    this.entier = entier;
    this.slider = slider;
  }
  public void update(Observable arg0, Object arg1) {
    slider.setValue(entier.getValeur());
  }
  public void changed(ObservableValue<? extends Number> observable,
                      Number oldValue, Number newValue) {
    entier.setValeur((int)slider.getValue());
  }
}
```

- Constructeur : des références vers son abstraction et sa facette de présentation pour pouvoir y accéder
- Implémente `Observer` pour pouvoir être ajouté à la liste des observateurs de l'abstraction
- Implémente `ChangeListener` pour pouvoir écouter le `Slider`

Contrôle - TextField

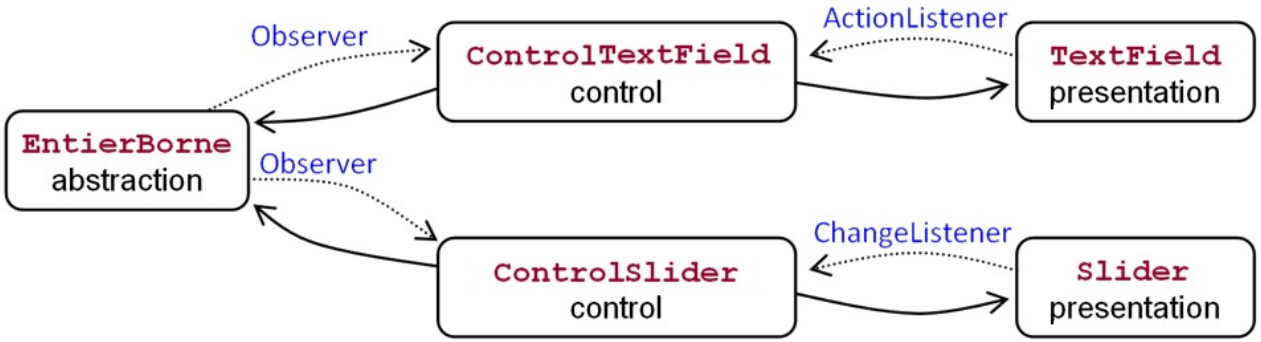
```
public class CtrlTextField implements Observer, ChangeListener<String>
  private EntierBorne entier;
  private TextField textField;
  public CtrlTextField(EntierBorne entier, TextField textField) {
    this.entier = entier;
    this.textField = textField;
  }
  public void update(Observable o, Object arg) {
    textField.setText("" + entier.getValeur());
  }
  public void changed(ObservableValue<? extends String> observable,
                      String oldValue, String newValue) {
    int valeur = Integer.MIN_VALUE;
    try {
      valeur = Integer.parseInt(textField.getText());
    } catch (Exception e) { }
    if (valeur < this.entier.getMin() || valeur > entier.getMax()) {
      Alert a = new Alert(AlertType.WARNING, "La valeur doit etre
entre " + entier.getMin() + " et " + entier.getMax(), ButtonType.OK);
      a.setTitle("Valeur hors bornes");
      a.showAndWait();
      textField.setText("" + entier.getValeur());
    } else { entier.setValeur(valeur); }}}
```

Présentation - suite

```
        [...]
        CtrlTextField ctf = new CtrlTextField(abstraction, // (3)
                                              textField); // (2)
        textField.textProperty().addListener(ctf); // (1)
        abstraction.addObserver(ctf); // (4)
        CtrlSlider cs = new CtrlSlider(abstraction, // (3)
                                       slider); //(2)
        slider.valueProperty().addListener(cs); // (1)
        abstraction.addObserver(cs); // (4)
        [...]
    }
}
```

- Communication contrôle–présentation :
  1. Le contrôle est un écouteur du composant (*i.e.*, il réagira aux actions de l'utilisateur sur ce composant)
  2. Passage du composant en paramètre du constructeur du contrôle afin qu'il puisse agir sur le composant si les données du modèle changent
- Communication contrôle–modèle :
  3. Passage de l'abstraction en paramètre du constructeur du contrôle
  4. Ajout du contrôle à la liste des observateurs de l'abstraction

En résumé



- Si on change la valeur du `TextField` :
  - Le `ControlTextField` vérifie si la valeur saisie est entre les bornes
  - Si ce n'est pas le cas, elle affiche une boîte de dialogue et met à jour le `TextField`
  - Sinon, elle modifie l'entier borné
    - ★ L'entier borné notifie ses observateurs d'une modification
    - ★ La méthode `update` du `ControlSlider` est appelée et celle-ci met à jour le `Slider`
- Symétriquement, si on bouge le `Slider` :
  - L'entier borné est mis à jour par le `ControlSlider`, ce qui notifie le `ControlTextField` qui actualise le `TextField`

Bénéfices

- Séparation nette entre la partie purement IHM et le noyau applicatif
- Composants indépendants à présent
  - `Slider` n'a plus à se soucier du `Textfield`
  - Les modifications sont propagées *via* les contrôleurs
- Robustesse améliorée
  - Il est plus difficile d'oublier une mise à jour
- Meilleure modularité
  - Il est possible d'ajout/retirer une partie de l'interface sans impacter les autres parties