JavaFX

Historique des interfaces graphiques

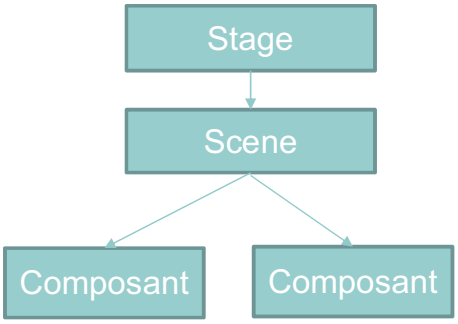
- Le package historique: java.awt (1995)
  - Composants s'appuyant sur les objets natifs de la machine cible
  - Peu d'objets
  - Difficulté d'écrire des applications multiplateformes (Write Once, Run Anywhere)
- Le package javax.swing est apparu dès la version 2 (1996)
  - Composants écrits à 98% en java
  - Apparence ne dépendant plus de la machine cible
  - Modèle / Vue / Contrôleur

JavaFX

- Java FX annoncé en 2007 mais réellement effectif depuis 2011 et la JavaFX2 puis 2014 avec JavaFX 8
- Bibliothèque graphique riche disposant de composants complexes (effets visuels, navigateur Web, images, audio, vidéo, …).
- L'interface graphique peut être conçue de manière déclarative FXML (soit directement, soit par l'outil SceneBuilder) ou de manière procédurale (en code java).
- Application possible de feuilles de style CSS
- Etc.

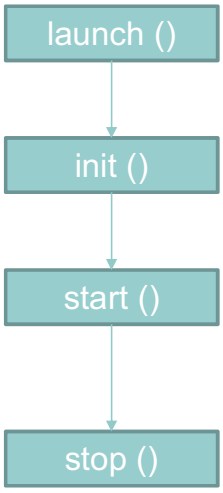
Architecture

- Une application JavaFX étend la classe *javafx.application.Application*
- La fenêtre principale est un objet de type *Stage* fournie par le système au lancement de l'application
- L'interface est représentée par un objet de type *Scene* qu'il faut créer et associer à la la fenêtre Stage
- La Scene est composée des composants graphiques qui constituent l'interface
- La méthode *start ()* permet de réaliser l'interface



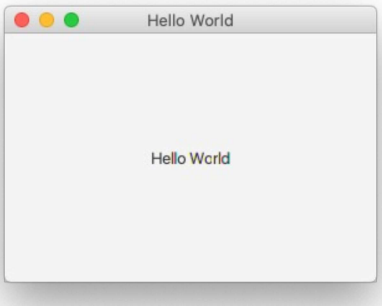
Cycle de vie

- *launch ()* est généralement lancée de la méthode *main ()* ou implicitement s'il n'y a pas de *main ()* (tolérance)
- Appel de la méthode *init ()*. Peut être redéfinie, ne fait rien par défaut.
- Appel de la méthode *start (Stage)*. Méthode abstraite qui DOIT être redéfinie.
- Appel de la méthode *stop ()* lorsque la dernière fenêtre de l'application se ferme ou si l'application appelle *Platform.exit ()* (ne pas utiliser *System.exit ()*). Peut être redéfinie, ne fait rien par défaut.



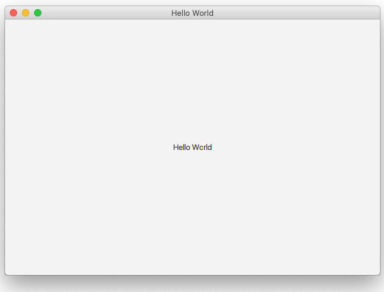
Hello World en mode procédural

```
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{

        primaryStage.setTitle("Hello World");
        BorderPane root = new BorderPane () ;
        Label hello = new Label ("Hello World") ;
        root.setCenter(hello);
        Scene scene = new Scene (root, 300,200) ;
        primaryStage.setScene(scene); ;
        primaryStage.show();
    }


    public static void main(String[] args) {
        launch(args);
    }
}
```



Hello World en mode déclaratif

```
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```



Hello World – fichier fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.BorderPane?>


<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
prefHeight="400.0" prefWidth="600.0" xmlns="http://javafx.com/javafx/8.0.181"
xmlns:fx="http://javafx.com/fxml/1">
  <center>
    <Label text="Hello World" BorderPane.alignment="CENTER" />
  </center>
</BorderPane>
```

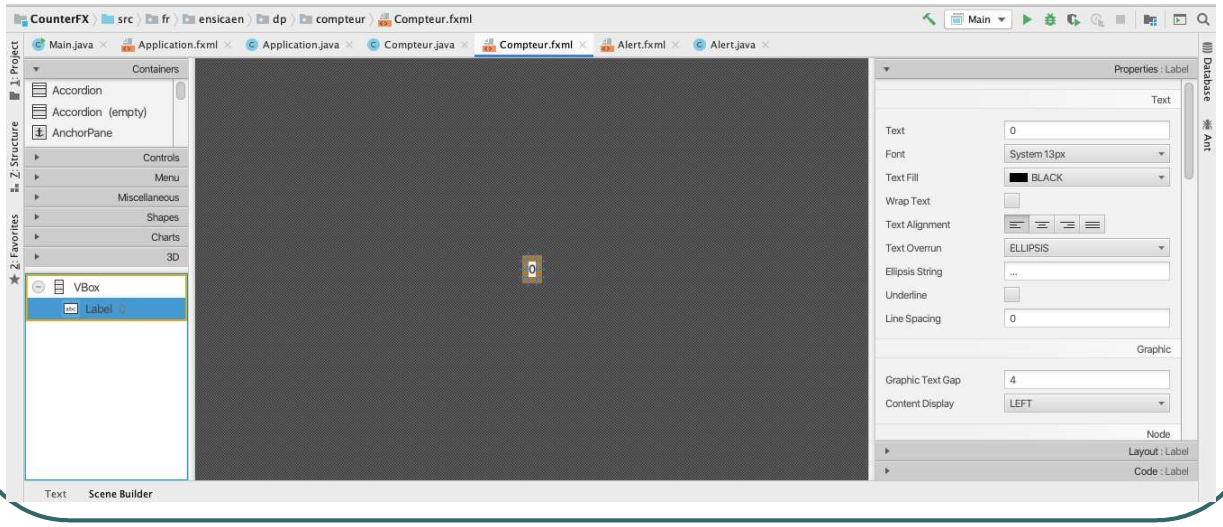
Classes Java

Propriétés

- Une propriété P est un élément d'une classe que l'on peut manipuler avec des accesseurs en lecture et/ou écriture (*getP (), setP ()).*
- Avec JavaFX, une propriété peut également être un objet implémentant l'interface *Property*
- Les propriétés peuvent être liées entre elles (binding)
- Une propriété peut déclencher un événement lorsque sa valeur change.
- Les classes de propriétés implémentent l'interface *Observable* et offrent ainsi la possibilité d'enregistrer les observateurs (listener).
- L'interface fonctionnelle *ChangeListener<T>* et sa méthode *changed ()* sera invoquée en cas de modification de la propriété en recevant l'ancienne et la nouvelle valeur de la propriété.

Création d'un composant

- Créer un composant graphique « Compteur » capable d'afficher une valeur entière et disposant de méthodes d'incrémentation et de décrémentation.



Exemple de composant

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>

<fx:root type="javafx.scene.layout.VBox" xmlns:fx="http://javafx.com/fxml/1"
xmlns="http://javafx.com/javafx/8.0.181">
    <Label fx:id="compteurLabel" text="0" />
</fx:root>
```

Conteneur

Classe

Identifiant

Valeur initiale

Fichier Compteur.fxml

Classe contrôleur

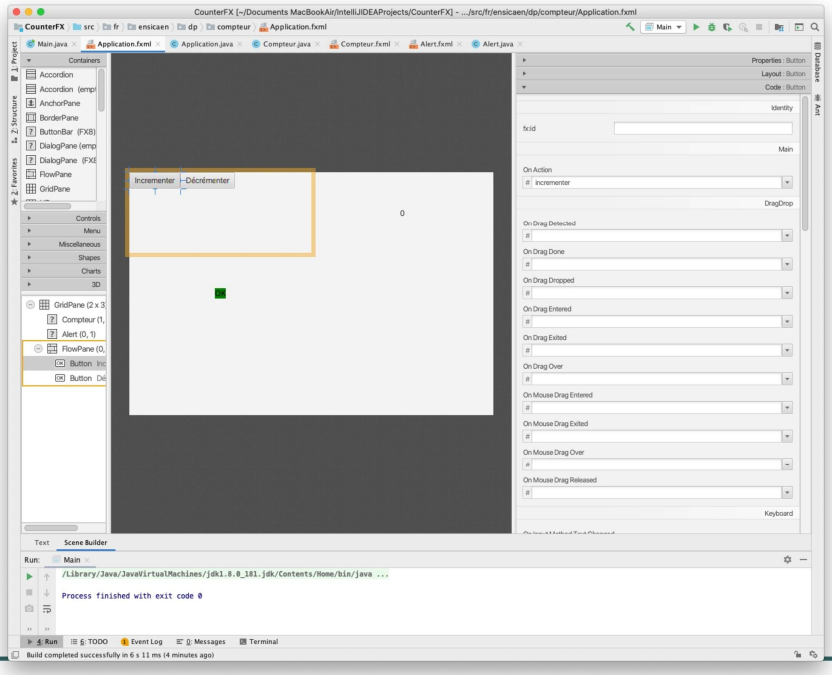
```
public class Compteur extends VBox {
   private IntegerProperty compteur = new SimpleIntegerProperty(0) ;

   @FXML private Label compteurLabel ;

   public Compteur() {
      super () ;
      FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource("Compteur.fxml"));
      fxmlLoader.setRoot(this);        fxmlLoader.setController(this);

      try {  fxmlLoader.load();
      } catch (IOException exception) { throw new RuntimeException(exception); }
   }

   public final ReadOnlyIntegerProperty getCompteurProperty () {  return compteur ; }

   public void previous () {
      compteur.set( compteur.getValue() - 1 );  compteurLabel.setText(compteur.getValue().toString()); ;
   }

   public void next () {
      compteur.set( compteur.getValue() + 1 );  compteurLabel.setText(compteur.getValue().toString()); ;
   }
}
```

Propriété

Association avec le fichier fxml

Définit le contrôleur

Application



Application

```
<children>
<FlowPane prefHeight="200.0" prefWidth="200.0">
    <children>
      <Button mnemonicParsing="false" onAction="#incrementer" text="Incrementer" />
      <Button mnemonicParsing="false" onAction="#decrementer" text="Décrémenter" />
    </children>
  </FlowPane>

<Compteur fx:id="monCompteur" alignment="CENTER" prefHeight="50.0" prefWidth="50.0"
GridPane.columnIndex="1" />
</children>
```

```
public class Application {
    @FXML private Compteur monCompteur ;

    public void decrementer (ActionEvent evt) {
        monCompteur.previous();
    }
    public void incrementer (ActionEvent evt) {
        monCompteur.next();
    }
}
```

Ajout d'un composant Alert

- Modification: déclencher une alerte lorsque la valeur du compteur dépasse un seuil --> création d'un composant »Alert »

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>

<fx:root type="javafx.scene.layout.VBox" xmlns="http://javafx.com/javafx/8.0.181"
xmlns:fx="http://javafx.com/fxml/1">
   <Label fx:id="alertLabel" text="OK" />
</fx:root>
```


Contrôleur Alert (1/2)

```
public class Alert extends VBox implements ChangeListener {
   @FXML private Label alertLabel ;
   private int max = 3 ;

   public Alert() { … }  // Code semblable au code de Compteur

   @FXML
   private void initialize () {
      afficheOK();
   }

   public final int getMax() { return max; }
   public final void setMax(int max) { this.max = max; }


   @Override
   public void changed(ObservableValue observable, Object oldValue, Object newValue) {
      int compteur = (int) newValue ;
      if (compteur > max) afficheAlert();
      else afficheOK();
   }
```

Contrôleur Alert (2/2)

```
 private void afficheOK () {
       alertLabel.setBackground(new Background(new BackgroundFill(Color.GREEN,null,null))) ;
       alertLabel.setTextFill(javafx.scene.paint.Color.web("#000000")); alertLabel.setText("OK");
    }

    private void afficheAlert () {
       alertLabel.setBackground(new Background(new BackgroundFill(Color.RED,null,null))) ;
       alertLabel.setTextFill(javafx.scene.paint.Color.web("#FFFFFF"));
       alertLabel.setText("Alerte");
    }
}
```

Modification de l'application:

```
@FXML private Alert monAlerte ;

@FXML
private void initialize () {
   monCompteur.getCompteurProperty().addListener(monAlerte);
}
```

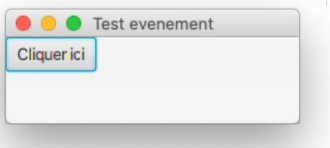

Java FX & Evénement

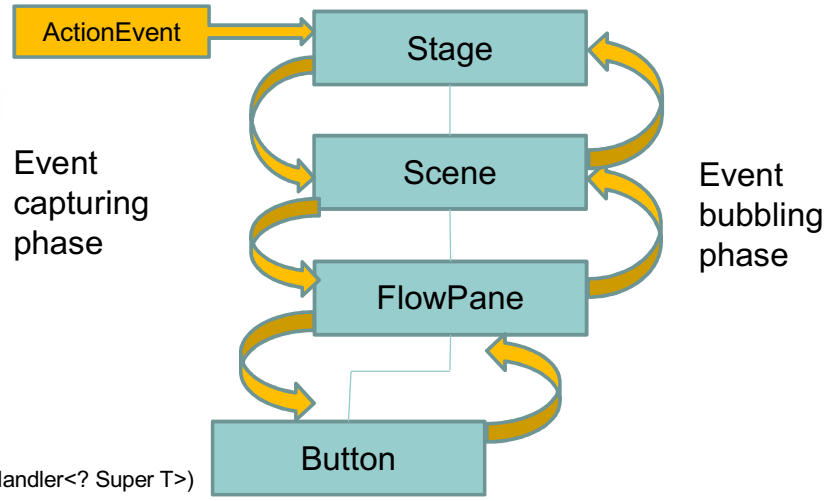
- Les événements sont représentés par des objets de la classe *javafx.event.Event* ou une des ses sous-classes.
- Chaque objet de type événement comprend au moins les informations suivantes:
  - Le type de l'événement (*EventType*). Le type permet de classifier différents événements à l'intérieur d'une même classe. Un type possède un nom (consultable par *getName ()*) et un type parent (consultable par *getSuperType ()*) . Le type racine est *Event.ANY*.
  - La source de l'événement (*Object* consultable par *getSource ()*)
  - La cible de l'événement (*EventTarget* consultable avec *getTarget ()*)

Propagation des événements

- Target selection
  Le système détermine la cible de l'événement en fonction du type d'événement.
- Event capturing phase
  L'événement est propagé en descendant du graphe de scène de la racine jusqu'à la cible. Chaque nœud a la possibilité de réagir à l'événement.
- Event bubbling phase
  L'événement remonte le graphe de la cible jusqu'à la racine. Chaque nœud a une deuxième opportunité de réagir à l'événement.

Exemple de propagation





Event capturing phase
addEventFilter (EventType<T>, EventHandler<? Super T>)

Event bubbling phase
addEventHandler (EventType<T>, EventHandler<? Super T>)

JavaFX et Threads

- Une application JavaFX met en œuvre différents threads:
  - Thread principal qui exécute la méthode *main ()*
  - Le thread JavaFX-Launcher qui exécute le code de la méthode *init ()* (classe *Application*)
  - Le thread JavaFX-Application qui exécute le code des méthodes *start (), stop (),* le code des gestionnaires d'événements (contrôleur) et le code chargé du rendu graphique de l'interface
  - Eventuellement les « worker threads » pour exécuter le code « lent »

JavaFX et Threads

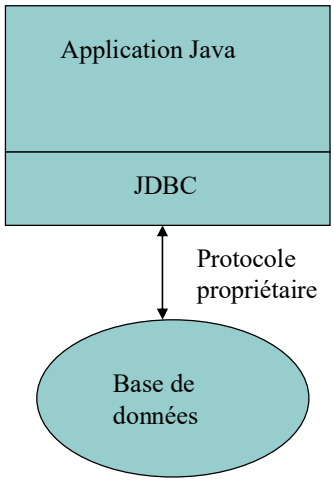
- Les traitements longs sont déportés dans les « worker threads »
- Mais seul le thread « JavaFX-Application » a le droit de modifier l'interface graphique.
- Pour concilier ces deux exigences, il faut utiliser la méthode statique *runLater (Runnable)* de la classe *Platform.*

Interaction Java/bases de données

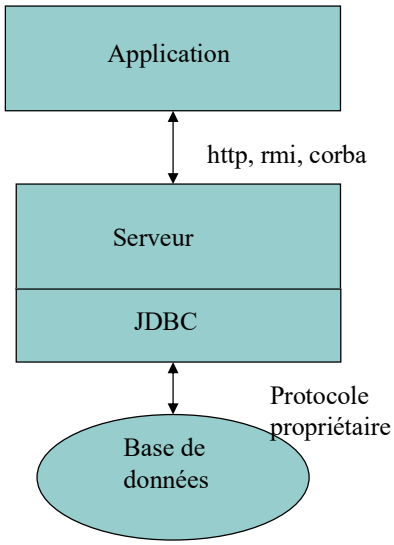
JDBC: Java DataBase Connectivity

- C'est une API java qui permet aux applications java de communiquer avec les gestionnaires de base de données dans un langage universel (comparable à ODBC).
- Les applications peuvent ainsi être indépendantes de la base de données utilisées.
- Un pilote JDBC permet:
  - Etablir une connexion avec une base de données.
  - Envoyer des requêtes SQL.
  - Traiter les résultats.

Architecture d'utilisation



Modèle 2/3



Modèle 3/3

Classes de connexion

java.sql.Driver	Interface devant être implémentée par les classes de chargement des pilotes JDBC.
java.sql.DriverManager	Un objet *DriverManager* va tenter de localiser le pilote JDBC et charger les classes correspondantes.
java.sql.Connection	Un objet *Connection* représente le lien entre l'application et la base de données. Toutes les requêtes SQL transmises et le retour des résultats s'effectueront à travers cet objet.

Exemple de connexion

```
private Connection Conn ;

try {
            Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    }
        catch (Exception e)
        {
            System.err.println(" Probleme avec le driver JDBC: "  + e);
            return ;
        }

      try {
            Conn = DriverManager.getConnection("jdbc:mysql://xxx.ecole.ensicaen.fr/",<login>,<password>) ;
        }
          catch (SQLException e)
         {
            System.err.println("Probleme ouverture: "  + e) ;
            return ;
         }
```


Classes d'accès à la base de données

java.sql.Statement	Classe à utiliser pour les requêtes SQL élémentaires. Quelques méthodes: *public ResultSet executeQuery (String sql) throws SQLException* *public int executeUpdate (String sql) throws SQLException* *public boolean execute(String sql) throws SQLException*
java.sql.ResultSet	Une instance de cette classe contient une rangée de données extraite de la base par une requête SQL et offre plusieurs méthodes chargées d'en isoler les colonnes. La notation suivante est utilisée: <type> get<type> (int | String) Exemple: *String getString (« title »)* A un instant donné, un objet *ResultSet* ne peut contenir plus d'une rangée mais propose une méthode *next()* permettant de référencer la rangée suivante.
java.sql.PreparedStatement	Cette classe est utilisée pour pouvoir envoyer au gestionnaire de base de données une requête SQL pour interprétation mais non pour exécution. Cette requête peut contenir des paramètres qui seront renseignés ultérieurement.

Exemple de code

```
Statement stmt = conn.createStatement () ;

ResultSet rs = stmt.executeQuery (« SELECT a,b,c FROM Table1 ») ;

while (rs.next ())
{
    int x = rs.getInt ("a") ;
    String s = rs.getString ("b") ;
    float f = rs.getFloat ("c") ;
}
```


Aspect transactionnel

- Par défaut, les opérations sur la base de données sont en mode *auto-commit*. Dans ce mode, chaque opération est validée unitairement pour former la transaction.
- Pour rassembler plusieurs opérations en une seule transaction:
  - *connection.setAutoCommit(false);*
  - *connection.commit () ;*
- Retour en arrière:
  - *connection.rollback ();*