UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

Développement des Interfaces Homme Machine (IHM)

Nourhène BEN RABAH

Docteur en Informatique

2019/2020

Adresse mail : Nourhene.Ben-Rabah@univ-paris1.fr

Sommaire

Introduction: Patron de conception

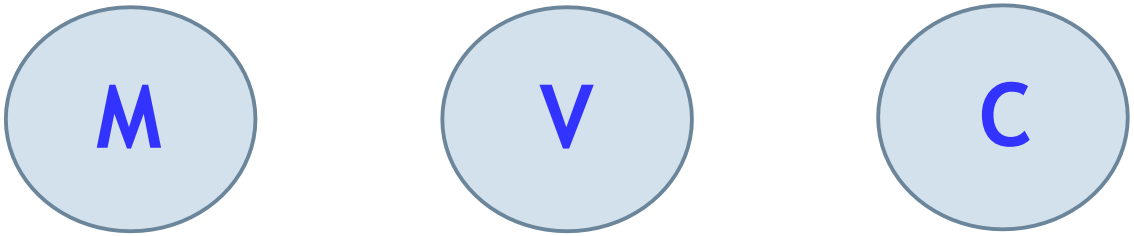
- Comment peut-on structurer le code des applications interactives (celles qui comportent une interface utilisateur) ??

  ⇨ Utiliser un **patron de conception** (Design Pattern)

  - ✓ Un ensemble de bonnes pratiques diffusées sous formes de classes pour la résolution d'un ou de plusieurs problèmes de conception,
  - ✓ Il décrit une solution standard, utilisable lors de la conception des différents applications,
  - ✓ Représente une idée et non une implémentation particulière,
  - ✓ Plusieurs patrons de conception existent: Singleton, Façade…

Introduction: Le patron de conception MVC

- Il offre une séparation claire des responsabilités au sein d'une application,
- Le code des applications est divisé en entités distinctes :



- Ces entités communiquent entre elles au moyen de divers mécanismes (invocation de méthodes, génération et réception d'événements, etc),

- Il a été introduit avec le langage Smalltalk-80 dans le but de simplifier le développement ainsi que la maintenance des applications, en répartissant et en découplant les activités dans différents sous-systèmes (plus ou moins) indépendants,

Introduction: Le patron de conception MVC

- Le principe de base du patron MVC est relativement simple, on divise le code du système interactif en trois parties distinctes :
  1. Le ou les **modèles** (Models) qui se chargent de la gestion des données (accès, transformations, calculs, etc.). Le modèle enregistre (directement ou indirectement) l'état du système et le tient à jour,
  2. Les **vues** (Views) qui comprennent tout ce qui touche à l'interface utilisateur (composants, fenêtres, boîtes de dialogue) et qui a pour tâche de présenter les informations (visualisation). Les vues participent aussi à la détection de certaines actions de l'utilisateur (clic sur un bouton, déplacement d'un curseur, geste swipe, saisie d'un texte, …),
  3. Les **contrôleurs** (Controllers) qui sont chargés de réagir aux actions de l'utilisateur (clavier, souris, gestes) et à d'autres événements internes (activités en tâches de fond, timer) et externes (réseau, serveur),

Le modèle

- Le modèle (Model) est responsable de la gestion des données qui caractérisent l'état du système et son évolution,
- Dans certaines situations (simples) le modèle peut contenir lui même les données mais, la plupart du temps, il agit comme un intermédiaire (proxy) et gère l'accès aux données qui sont stockées dans une base de données, un serveur d'informations, le cloud, …
- Il offre également les méthodes et fonctions permettant de gérer, transformer et manipuler ces données,
- Les informations gérées par le modèle doivent être indépendantes de la manière dont elles seront affichées. Le modèle doit pouvoir exister indépendamment de la représentation visuelle des données,

La vue

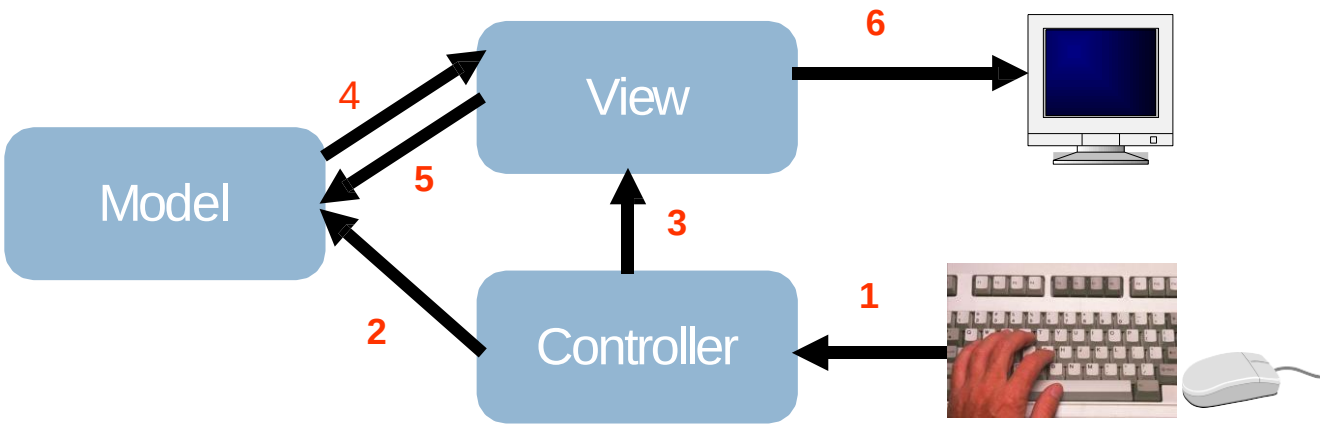
- **La vue (View)** est chargée de la représentation visuelle des informations en faisant appel à des écrans, des fenêtres, des composants, des conteneurs (layout), des boîtes de dialogue, etc,
- Plusieurs vues différentes peuvent être basées sur le même modèle (plusieurs représentations possibles d'un même jeu de données). La vue intercepte certaines actions de l'utilisateur et les transmet au contrôleur pour qu'il les traite (souris, clavier, gestes, …),
- La mise à jour de la vue peut être déclenchée par un contrôleur ou par un événement signalant un changement intervenu dans les données du modèle par exemple (mode asynchrone),
- La représentation visuelle des informations affichées peut dépendre du Look-and-Feel adopté (ou imposé) et peut varier d'une plateforme à l'autre. L'utilisateur peut parfois modifier lui même le thème de présentation des informations.

Contrôleur

- **Le contrôleur (Controller)** est chargé de réagir aux différentes actions de l'utilisateur ou à d'autres événements qui peuvent survenir.
- Le contrôleur définit le comportement de l'application et sa logique (comment elle réagit aux sollicitations, business logic).
- Dans les applications simples, le contrôleur gère la synchronisation entre la vue et le modèle (rôle de chef d'orchestre).
- Le contrôleur est informé des événements qui doivent être traités et sait d'où ils proviennent.
- La plupart des actions étant interceptées (ou en lien) avec la vue, il existe un couplage assez fort entre la vue et le contrôleur.
- Le contrôleur communique généralement avec le modèle et avec la vue. C'est le sens des transferts et le mode de communication qui caractérisent différentes variantes de l'architecture MVC.

Interactions MVC

1. L'utilisateur effectue une action, le contrôleur est averti.
2. Le contrôleur peut demander des modifications au modèle.
3. Le contrôleur peut demander la vue pour mettre à jour.
4. Le modèle peut notifier la vue s'il a été modifié.
5. La vue peut avoir besoin d'interroger le modèle pour les données actuelles.
6. La vue affiche les mises à jour pour l'utilisateur.



Interactions MVC: : Clique Bouton

Utilisateur

1: Evènement clique

Interface utilisateur

View

Décisions

Contrôleur

Stockage et données intermédiaires

Modèle

Interactions MVC: : Clique Bouton

Utilisateur

1: Evènement clique

Interface utilisateur

2: Action interprété

Décisions

Stockage et données intermédiaires

View

Contrôleur

Modèle

Interactions MVC: : Clique Bouton

Utilisateur

1: Evènement clique

Interface utilisateur

2: Action interprété

Décisions

3: Opérations

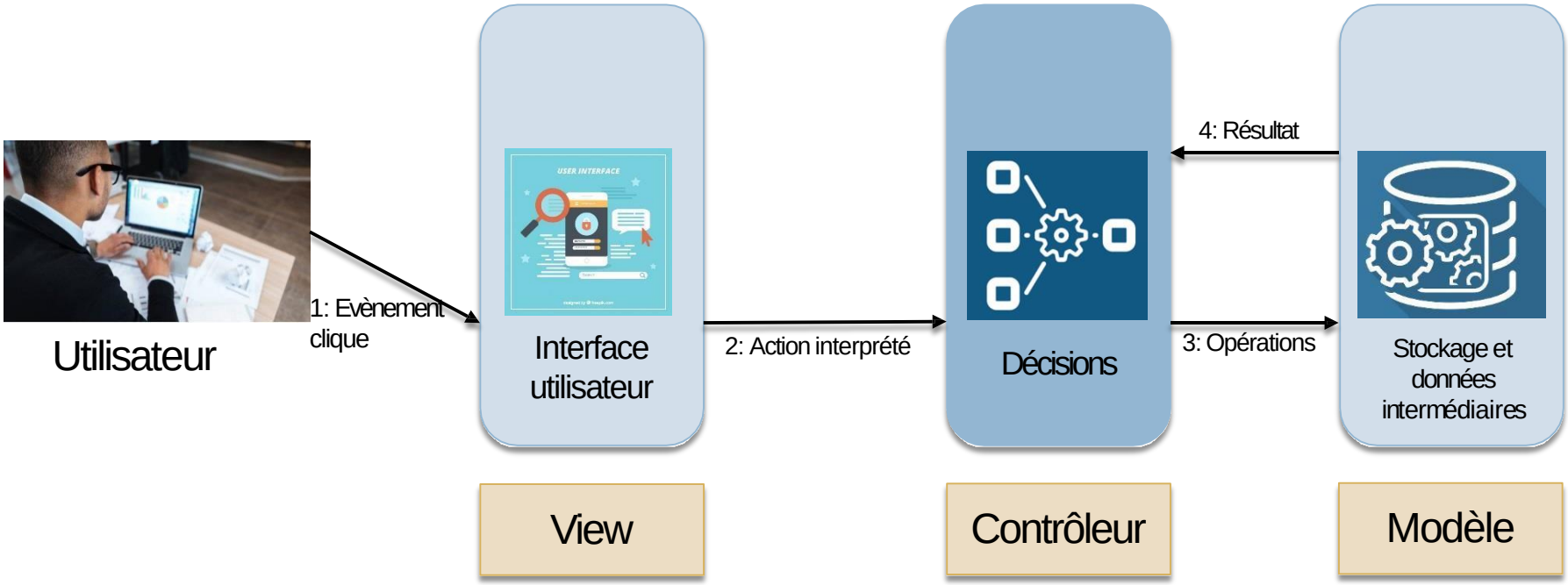
Stockage et données intermédiaires

View

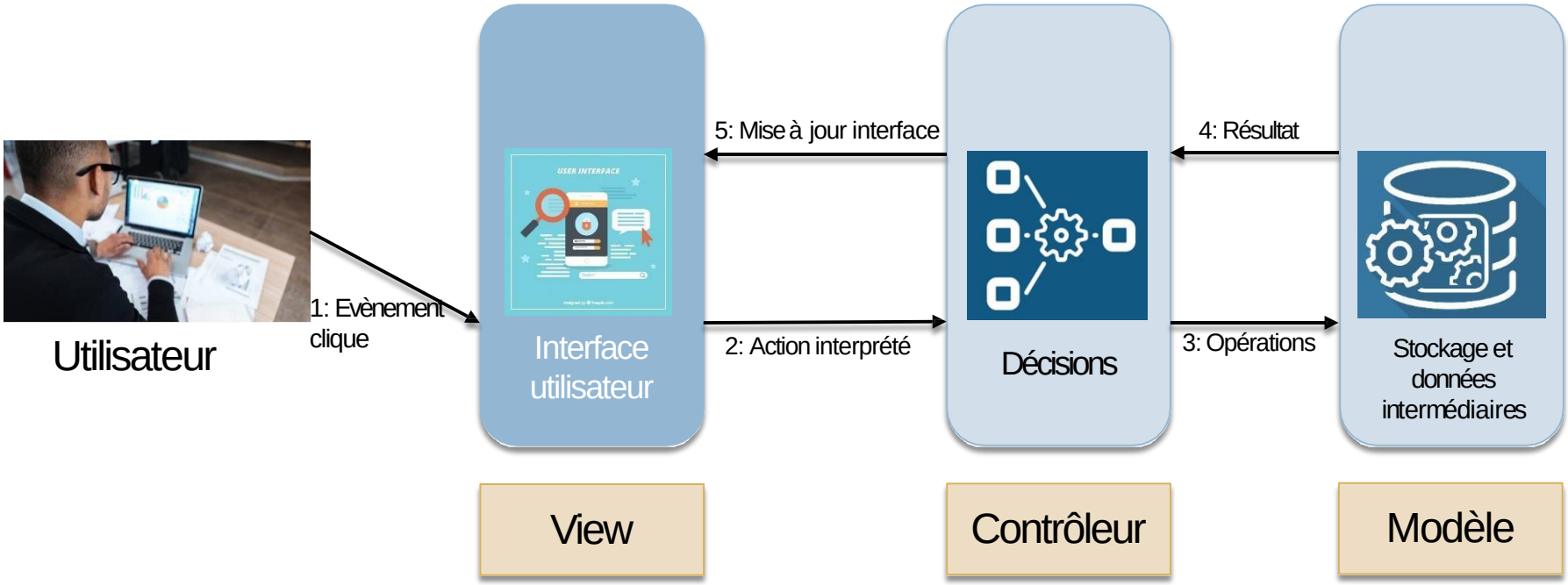
Contrôleur

Modèle

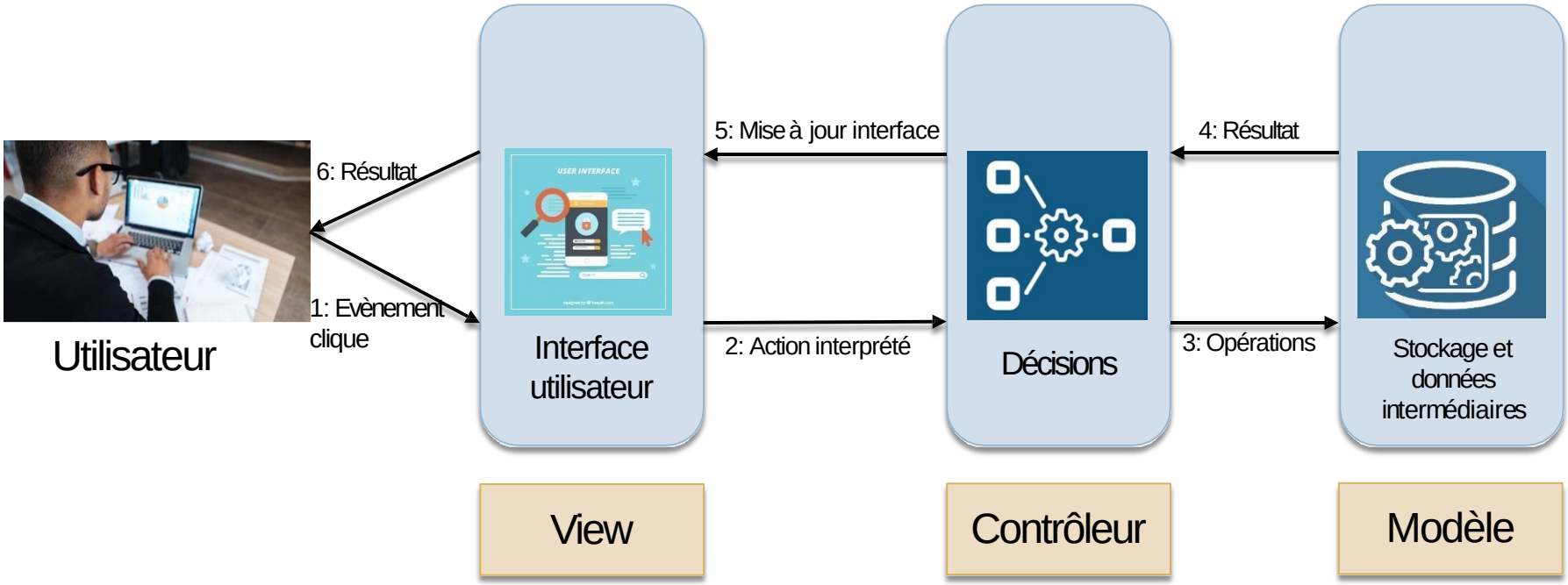
Interactions MVC: : Clique Bouton



Interactions MVC: : Clique Bouton



Interactions MVC: Clique Bouton



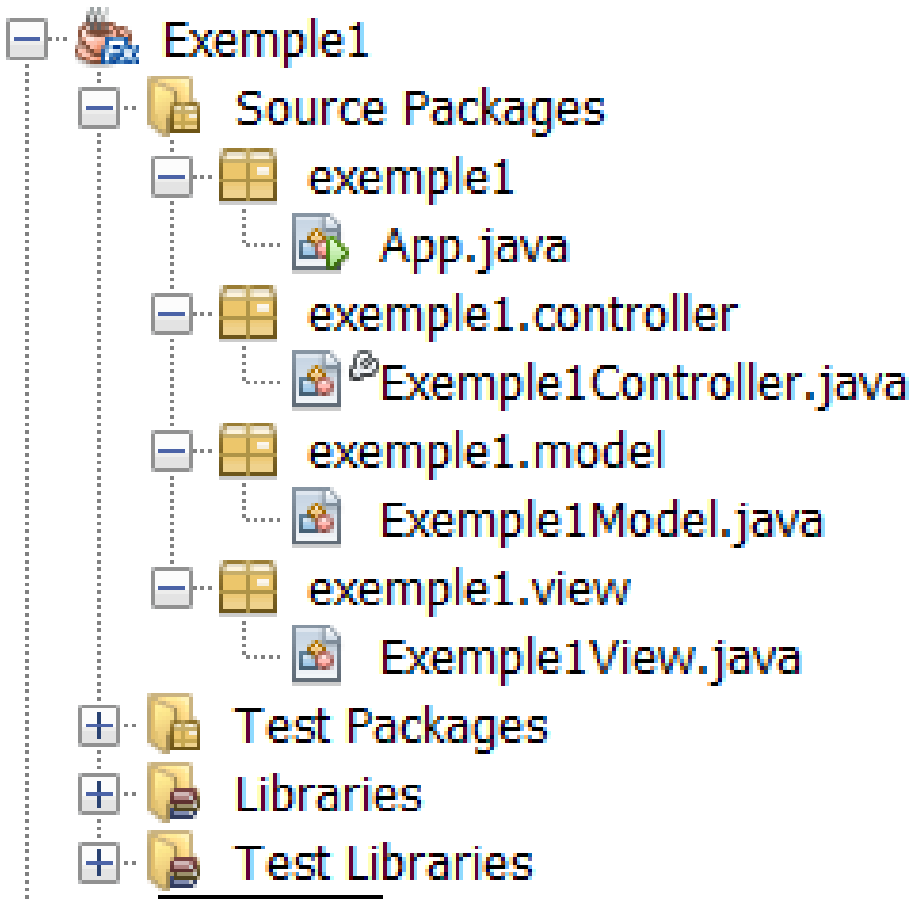
Exercice : Say Hello

Exercice 1:

- Implémenter une interface en JavaFX qui affiche un message Hello suivi du nom saisie dans le champ texte,

Indication: Lorsque je saisis un mot et je clique sur OK , le modèle se met à jour,

Correction Exercice1



Correction Exercice1: Class Model


```
package exemple1.model;

public class Exemple1Model {

    private String nom ;

    public Exemple1Model(){ this.nom = new String() ; }

    public String getNom() { return nom; }

    public void setNom(String nom) { this.nom = nom; }
}
```

Correction Exemple1: Class Controller


```
package exemple1.controller;

import exemple1.model.Exemple1Model;
import exemple1.view.Exemple1View;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class Exemple1Controller implements EventHandler<ActionEvent>{

    private final Exemple1Model model;
    private final Exemple1View view;

    public Exemple1Controller(Exemple1Model model , Exemple1View view){
        this.model = model;
        this.view = view;
    }

    @Override
    public void handle(ActionEvent event) {
       String message ;
       message = view.getNomTextField().getText();
       if(!message.isEmpty()){
           model.setNom(message);
           view.getResultLabel().setText("Hello "+message);
       }
    }
}
```

Correction Exercice1: Class View


```
package exemple1.view;

import exemple1.controler.Exemple1Controler;
import exemple1.model.Exemple1Model;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class Exemple1View {

    private final Label nomLabel ;
    private final TextField nomTextField ;
    private final Button validerButton;
    private final Label resultLabel;
    private final Scene scene ;
    private final GridPane rootPane ;
    private final Exemple1Model model;
```

Correction Exercice1: Class View


```
public Exemple1View(Stage primaryStage){
    model = new Exemple1Model();

    rootPane = new GridPane();
    nomTextField = new TextField();
    nomLabel = new Label("Nom :");
    validerButton = new Button("Valider");
    resultLabel = new Label("Hello World !!");

    validerButton.setOnAction(new Exemple1Controller(model , this));
    resultLabel.setMinWidth(100);

    rootPane.add(nomLabel, 0, 0);
    rootPane.add(nomTextField, 1, 0);
    rootPane.add(resultLabel, 0, 1);
    rootPane.add(validerButton, 1, 1);

    scene = new Scene(rootPane , 350, 100);
    primaryStage.setScene(scene);
}

public Label getNomLabel() { return nomLabel;}

public TextField getNomTextField() { return nomTextField;}

public Button getValiderButton() { return validerButton;}

public Label getResultLabel() { return resultLabel;}
}
```

Correction Exercice1: Class Main


```
package exemple1;

import exemple1.view.Exemple1View;
import javafx.application.Application;
import javafx.stage.Stage;

public class App extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("MVC Exemple 1");
        new Exemple1View(primaryStage);
        primaryStage.show();
    }

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

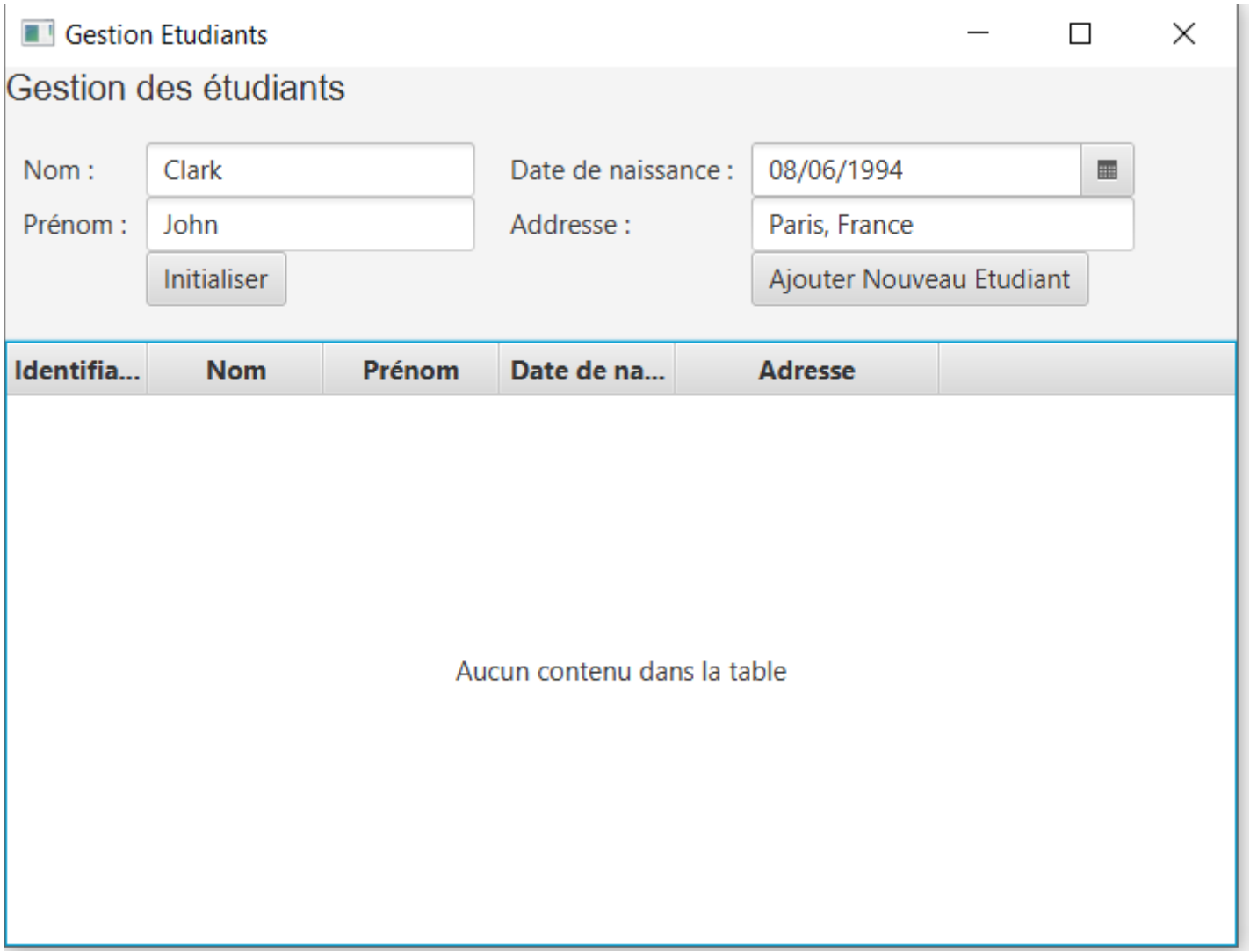
Exercice 2: Gestion des étudiants

Exercice 2:

- Implémenter une interface en JavaFX qui permet de créer des étudiants,
- Afficher la liste des étudiants dans un tableau,

- Indications:
  - Utiliser le composant TableView,
  - Utiliser les Events pour rafraichir le contenu du tableau à chaque fois qu'un étudiant est créé,
  - Formulaire de création des étudiants en GridPane

Exercice 2: Gestion des étudiants



Exercice 2: Gestion des étudiants