Interfaces graphiques avec JAVAFX

Les objectifs de ce cours

Objectifs principaux

- savoir réaliser un programme avec interface graphique
- savoir ce qu'est la programmation événementielle

Objectifs secondaires

- voir des exemples d'utilisation de classes internes et de lambda expressions

Yannick.Parchemal@parisdescartes.fr

Interfaces graphiques avec JAVAFX

Les séquences de ce cours

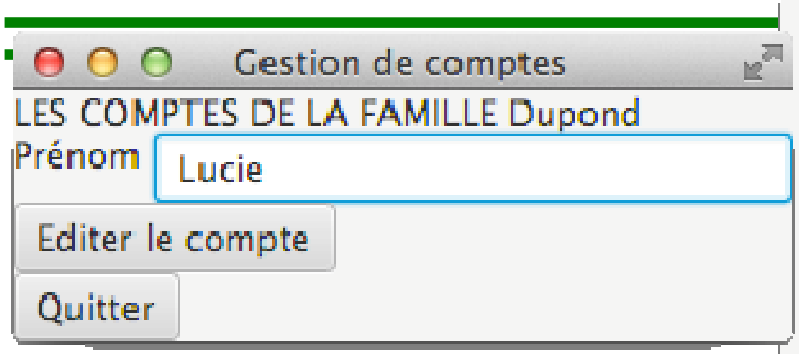
- Introduction
- Premiers pas avec JavaFX
- Programmation événementielle avec JavaFX
- Evénements et gestionnaire d'événements
- Composants : Label, Button et CheckBox
- La classe Pane et ses dérivées
- Composants : RadioButton, TextField et TextArea
- Une fenêtre de dialogue

Pré requis

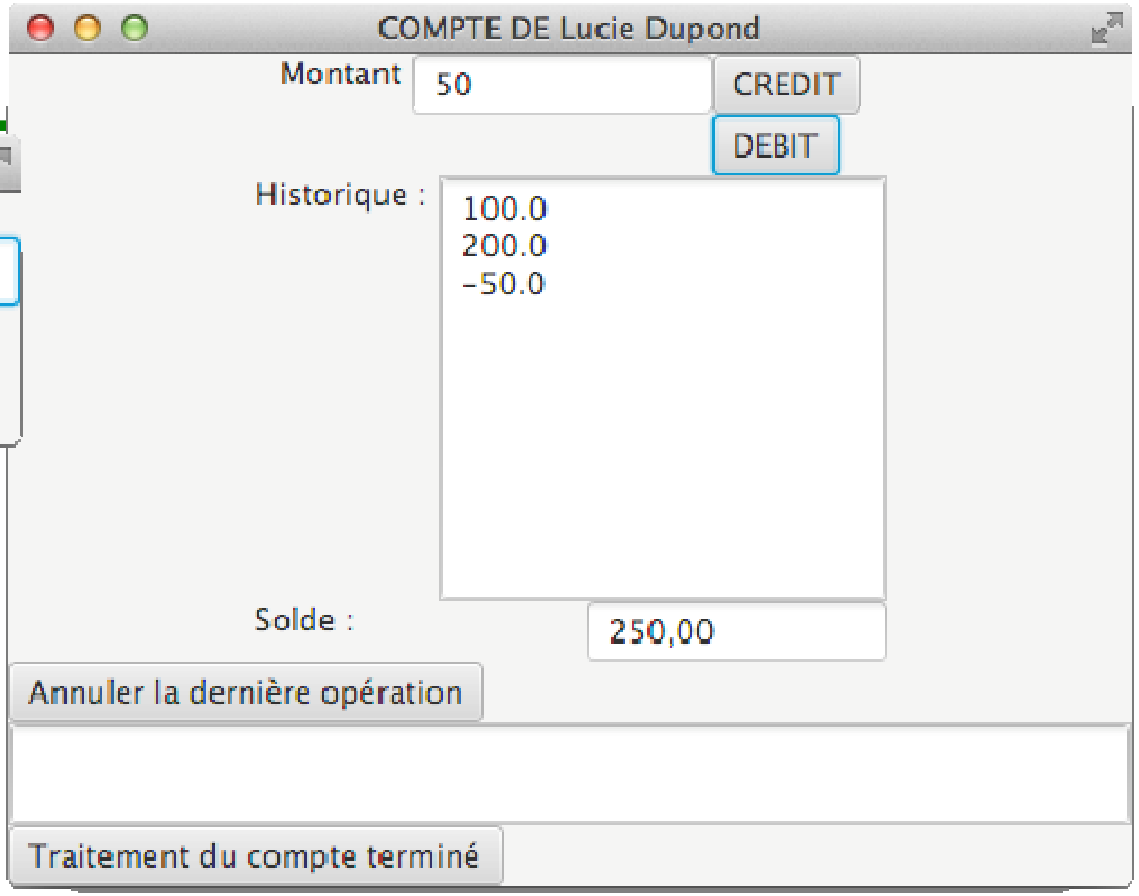
- Classes internes et lambda expressions

Yannick.Parchemal@parisdescartes.fr

Un exemple complet



La fenêtre d'accueil
(PanelFamille)

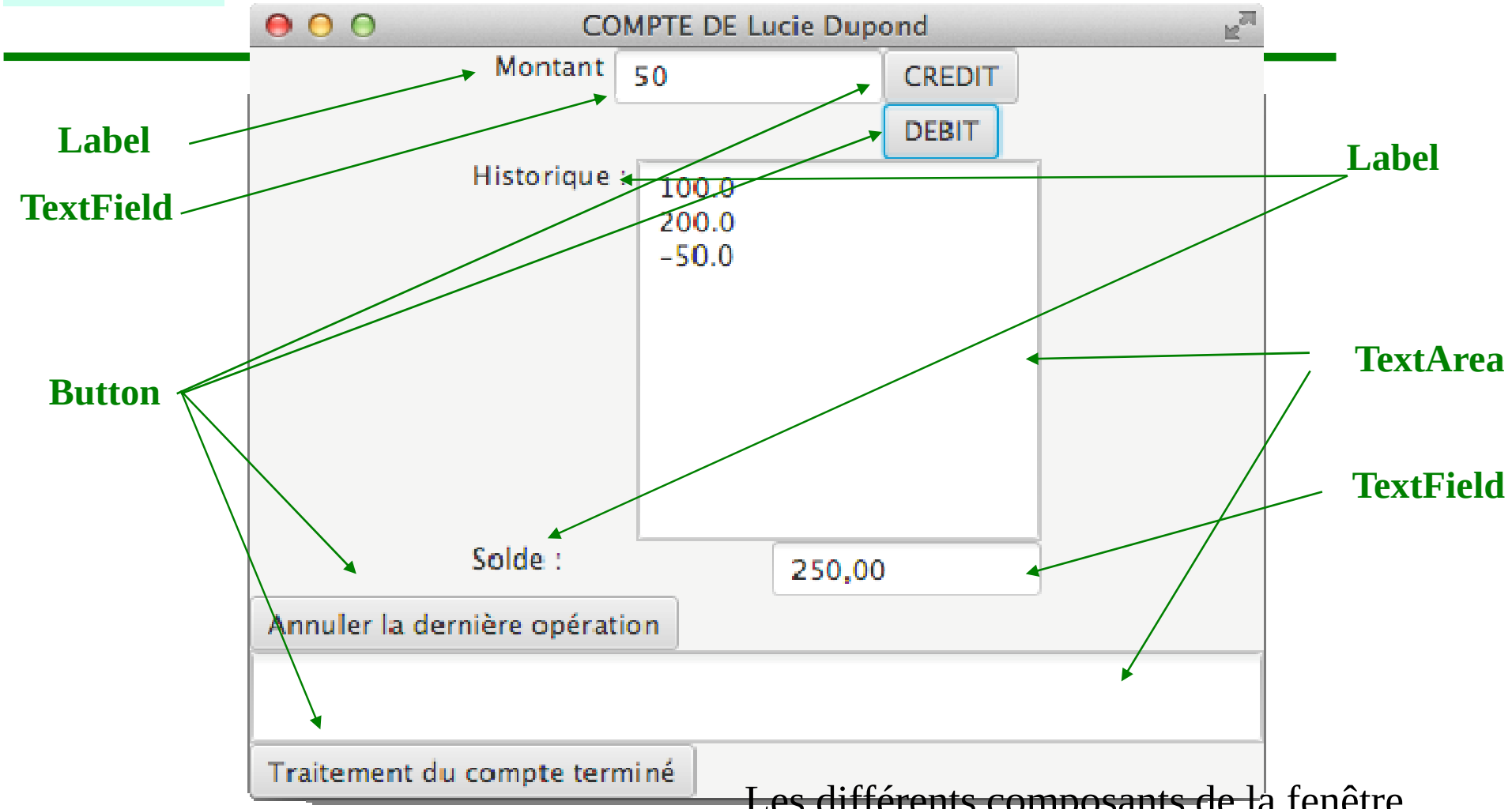


La fenêtre de gestion du compte
(PanelCompte)

Les deux fenêtres de l'exemple

Yannick.Parchemal@parisdescartes.fr

Description d'une fenêtre



DESCRIPTION STATIQUE

DESCRIPTION DYNAMIQUE

Les différents composants de la fenêtre
La répartition de ces composants
à l'intérieur de la fenêtre

Réaction des composants
aux différents **événements**

Premiers pas avec JAVAFX

Description de la séquence

Description du contenu :

1 cours d'environ 15 minutes
2 exercices pratiques

Durée de travail estimée : 45 minutes

Auteur : Yannick.Parchemal@parisdescartes.fr

Objectifs :

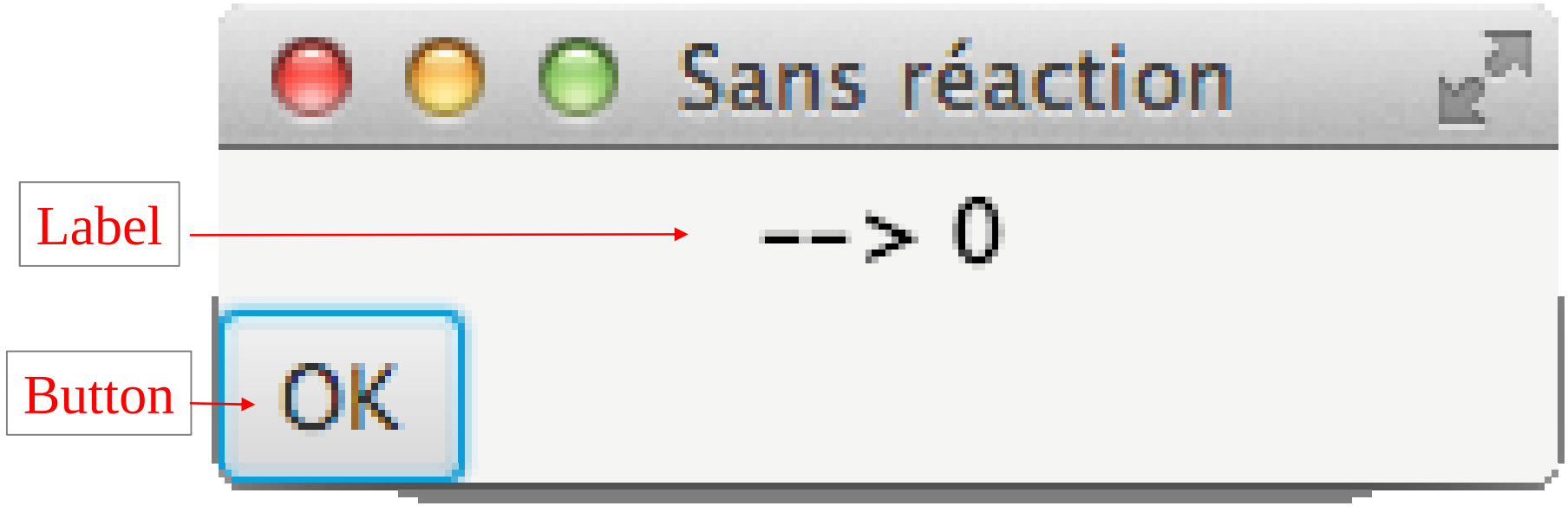
Le cours de cette séquence montre les concepts utilisés pour coder une interface graphique très simple non réactive en javaFX

Connaissances acquises : le rôle de classes incontournables de JavaFX : Application, Stage, Scene mais aussi Button, Label, BorderPane

Compétences acquises : codage d'une interface graphique simple non réactive similaire à celle vue dans le cours

Pré requis : connaissances du langage Java

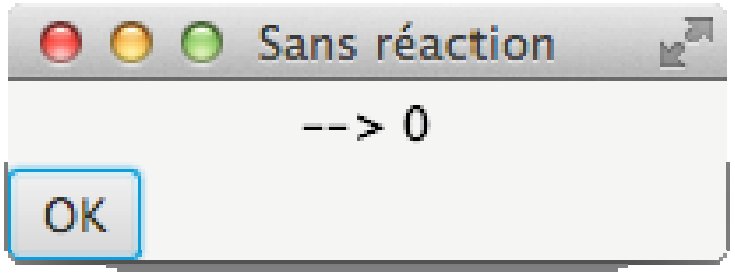
Une application non réactive …



- 2 composants graphiques : un label et un bouton
- il ne se produit rien lorsque l'on clique sur le bouton

Deux composants : un label et un bouton

```
Label label =  new Label("--> 0");

Button button= new Button("OK");
```



Control : synonyme de composant graphique en JavaFX

Button — le nom complet est javafx.scene.control.Button

Label — le nom complet est javafx.scene.control.Label

Un panneau pour disposer les composants

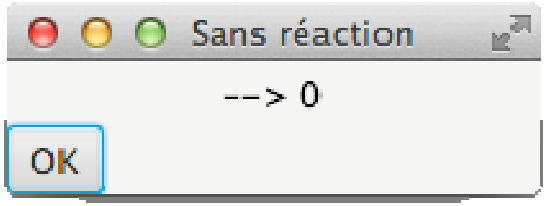
BorderPane pane= new BorderPane( );

Label label = new Label("--> 0");

Button button= new Button("OK");

pane.setCenter(label);

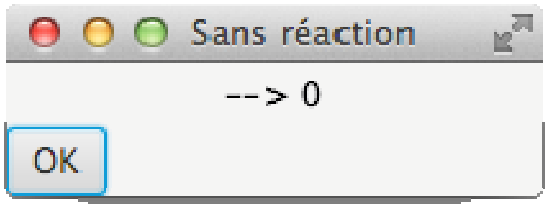
pane.setBottom(button);



BorderPane le nom complet est javafx.scene.layout.BorderPane

Application, Scene et Stage

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
public class PremiereAppli extends Application {
    @Override
        public void start(Stage stage) {

        stage.setTitle("Sans réaction");
        BorderPane pane= new BorderPane( );
        Label label =  new Label("--> 0");
        Button button= new Button("OK");
        pane.setCenter(label);
        pane.setBottom(button);
        Scene scene = new Scene(pane, 200, 50);
        stage.setScene(scene);
        stage.show( );
}
…
```



Quelques classes de JAVAFX

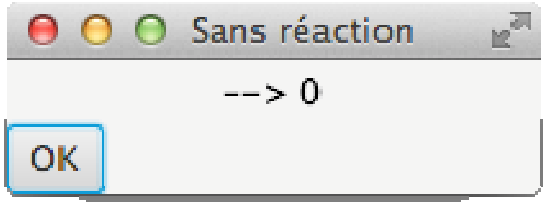
Terme Java	En français	rôle
la classe **Application**	Application	La classe principale d'une application JAVAFX est une extension de Application
la classe **Stage**	Théatre (scène)	Cela correspond à la fenêtre de l'application
la classe **Scene**	Scène (la scène jouée)	c'est dans une Scene que vont être mis tous les composants graphiques (boutons, zones de texte, listes …)
la classe **Pane**	Panneau (de fenêtre)	La classe mère des classes de panneaux. Dans un Pane, on va pouvoir mettre des composants
la classe **BorderPane**		Un pane dans lequel on peut mettre cinq composants (aux bords : top bottom left right et au centre : center)
la classe **Label**	Etiquette	Une étiquette (non éditable)
la classe **Button**	Bouton	Un bouton clickable

Et la méthode main ?

```
public class PremiereAppli extends Application {

        public void start(Stage stage) {
        ……
  }
// toutes les applications javafx ont une méthode main
// effectuant cet appel à launch
        public static void main(String [ ] args) {
// la méthode launch est une méthode static de Application qui
//              - crée le Stage correspondant à la fenêtre principale
//      - crée une instance de PremiereAppli
//      - applique la méthode start sur cette instance
            launch(args);// ou Application.launch(args);
        }
}
```

Une meilleure écriture de notre premier exemple

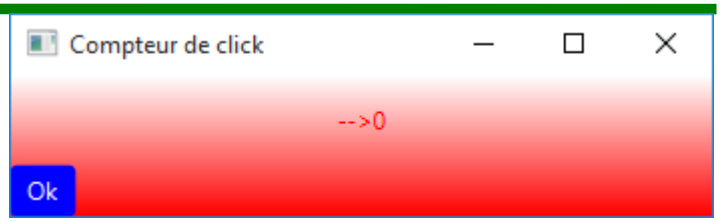
```
public class PremiereAppli extends Application {

    @Override
    public void start(Stage stage) {
                                    // on fixe un titre pour la fenêtre
                                    stage.setTitle("Sans réaction");
                // le pane qui va contenir les 2 composants
                Pane pane = new PanelPremiereAppli( );
        // on crée une scene contenant ce pane de 200 pixels de large par 50 pixels de haut
                        Scene scene = new Scene(pane, 200,50);
                        // c'est cette scène qui doit être montrée
                        stage.setScene(scene);
                        stage.show( );
}


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



La classe PanelPremièreAppli

```
public class PanelPremiereAppli extends BorderPane{

        public PanelPremiereAppli ( ){

                Button btn = new Button("OK");

                Label label =new Label("--> 0");

                this.setBottom(btn);
                this.setCenter(label);

        }
}
```

Utilisation de fichiers css



Lien un fichier css à une Scene

```
scene.getStylesheets( ).add(
                    getClass( ).getResource("application.css").toExternalForm());
// application.css doit se trouver dans le même répertoire que le fichier .class
```

application.css

```
.label{
    -fx-text-fill: red;
}
.button {
      -fx-background-color:blue;
      -fx-text-fill: white;
}
.root {
      -fx-background-color: linear-gradient(#ffffff, #ff0000);
}
```

→ Consulter JavaFX CSS Reference Guide

Programmation événementielle avec JAVAFX

Description de la séquence

Description du contenu :

1 cours d'environ 20 minutes
2 exercices pratiques

Durée de travail estimée : 60 minutes

Auteur : Yannick.Parchemal@parisdescartes.fr

Objectifs :

Le cours de cette séquence montre sur un exemple l'utilisation de la programmation événementielle pour programmer des interfaces graphiques.

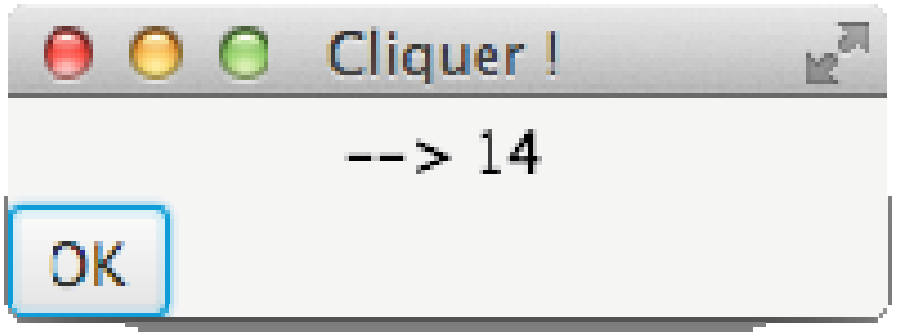
Connaissances acquises : concepts d'événement et de handler.

Compétences acquises : codage d'une interface graphique similaire à celle vue en cours

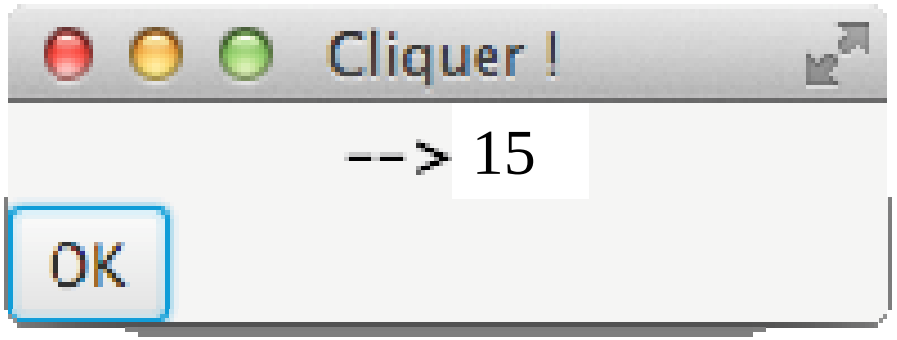
Pré requis : (séquence "premiers pas avec javaFX")

- connaître les classes incontournables de JavaFX : Application, Stage, Scene
- avoir étudié une interface graphique rudimentaire et non réactive

Une fenêtre qui réagit



Compte le nombre de fois que l'on a cliqué sur le bouton OK



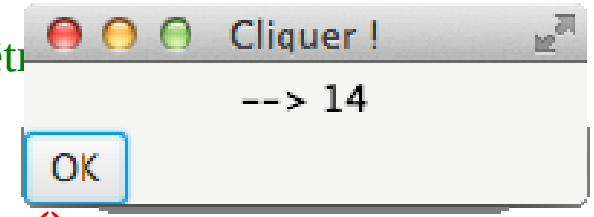
La classe Compteur

Une classe utile pour notre exemple

```
package up5.mi.pary.jc.util;

public class Compteur {

        private int value;

        public Compteur( ) {
                this.value=0;
        }

        public void incrementer(int incr){
                this.value+=incr;
        }

        public int getValue( ){
                return this.value;
        }

}
```

AppliClick : l'application JavaFX du compteur de click

```
public class AppliClick extends Application {

    @Override
    public void start(Stage stage) {
                                    // on fixe un titre pour la fenêtr
                                    stage.setTitle("Cliquer !");
                                    // création du compteur
                            Compteur compteur = new Compteur();
                    // création du PanelClick qui contient le bouton et le label
                Pane pane = new PanelClick(compteur);
        // on crée une scene contenant ce pane de 200 pixels de large par 50 pixels de haut
                            Scene scene = new Scene(pane, 200,50);
                            // c'est cette scène qui doit être montrée
                            stage.setScene(scene);
                            stage.show( );
}

  // la méthode main : ce sera toujours la même
                public static void main(String[ ] args) {
                            launch(args);
                }}
```



La classe PanelClick (1)

```
public class PanelClick extends BorderPane{

public PanelClick(Compteur compteur){

    Button button= new Button("OK");
    Label label = new Label("--> "+compteur.getValue( ));

    this.setBottom(button);
    this.setCenter(label);

}
```

De l'action de l'utilisateur à la realisation de la tâche

1 **Action de l'utilisateur sur le clavier ou la souris**

2 **Création d'un événement par javafx**

Exemples de classes d'événement (sous classes de Event) :

- clicks boutons : ActionEvent
- mouvement de la sourie : MouseEvent
- clavier : KeyEvent

Le composant où a eu lieu l'action de l'utilisateur est appelé source de l'événement

3 **Réalisation de la tâche grâce à un *handler***

(un handler est une instance de javafx.event.EventHandler)

Evénements : lexique

Terme	En français	rôle
Evénement		A chaque action de l'utilisateur sur le clavier ou la souris, un événement est crée par l'API JavaFX
la classe Event (javafx.event)	Evénement	La classe Event est la classe mère de toutes les classes d'événement
la source d'un événement		La source d'un événement est le composant graphique sur lequel porte l'action de l'utilisateur
la classe ActionEvent (javafx.event)	Evénement de type action	Les événements créés lors d'un click sur un bouton sont des instances de ActionEvent. ActionEvent est une classe dérivée de Event

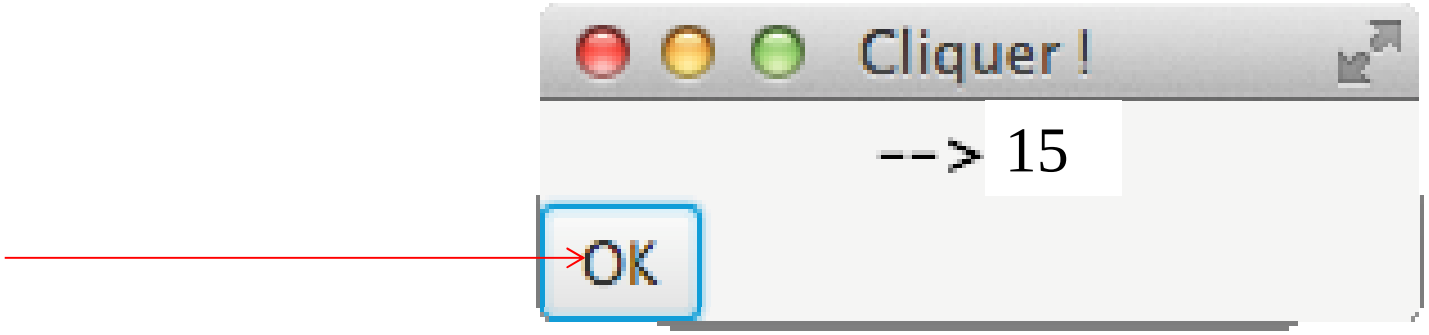
Les Event Handlers
(ou gestionnaire d'événements)

(Event) Handler

Les actions sont réalisées grâce à des handlers

handle

"handle" est la méthode du handler chargée de réaliser les actions




```
public void handle(ActionEvent event) {
                    // incrémentation du compteur
                    // mise à jour du label
}
```

La classe PanelClick : associer un handler au bouton

```
public class PanelClick extends BorderPane{


public PanelClick(Compteur compteur){


    Button button= new Button("OK");
    Label label = new Label("--> "+compteur.getValue( ));
        // l'instruction qui va permettre de réagir au click sur le bou
        button.setOnAction(new ClickHandler(label,compteur));

                                // La méthode setOnAction de la classe
Button
                                // permet d'associer le handler au bouton
    this.setBottom(button);
    this.setCenter(label);

}
```

ClickHandler

```
class ClickHandler implements EventHandler<ActionEvent> {
  private Label label;
  private Compteur compteur;

public ClickHandler (Label label,Compteur compteur){
                this.label=label;
                this.compteur=compteur;
        }
        @Override
/** Méthode appelée
   * lorsqu'un événement ActionEvent a été créé suite
   * à une action sur un composant auquel ce handler est associé */
        public void handle(ActionEvent event) {
                this.compteur.incrementer(1);
                this.label.setText("--> "+this.compteur.getValue());
        }
}
```

L'interface EventHandler

```
/** interface des gestionnaires d'événements */
public interface EventHandler<T extends Event> extends java.util.EventListener
{
/**Appelé lorsque un événement a pour source un composant
  *  auquel ce handler est associé
  *  @param  event  l'événement provoquant cet appel
  */
   public void handle(T event) ;
}
```

Handler : lexique

Terme	En français	rôle
Event Handler ou Handler	Gestionnaire d'événements	Un handler est un objet permettant, grâce à sa méthode "handle" de réaliser la tâche correspond à une action clavier ou souris sur un composant donné
l'interface EventHandler (javafx.event)		C'est l'interface implémentée par toutes les classes de handlers. Elle déclare la méthode "handle"
l'interface EventListener (java.util)		L'interface de tous les handlers (javafx et aussi autres que javafx)

Programmation événementielle avec JAVAFX

Activités associées

Les activités associées sont sur le site du cours.
Il est nécessaire de faire ces activités pour pouvoir considérer la séquence comme terminée.

Classes internes membres statiques et d'instance

Description de la séquence

Description du contenu :
Cette séquence traite des classes internes membres en Java. Après avoir situé ces classes dans le cadre plus générales des classes internes en Java, les connaissances à acquérir sont étudiées par l'étude d'une classe interne membre statique et d'une classe interne membre d'instance.

Durée de travail estimée : 60 minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs : savoir définir et utiliser des classes internes membres

Connaissances acquises : classes membres d'instance et statiques
Compétences acquises : comprendre un programme utilisant des classes membres. Savoir définir et utiliser de telles classes
Pré requis : classes et instances en Java

Classe, classe interne, classe interne membre

Classe

Les membres d'une classe sont

- des attributs (d'instance ou statique),
- des constructeurs,
- des méthodes (d'instance ou statique)
- et peuvent être aussi des classes appelées classes internes

Classe interne

C'est une classe définie dans une autre classe

classe interne membre

- statique
- d'instance

- définie au même niveau que les autres membres
- a un niveau de visibilité (private, package, protected, public)

classe locale

définie dans une méthode ou dans un constructeur

Les classes membres sont souvent appelées classes internes…

Classes membres internes

```
public class UneClasseEnglobante {

        private static int att1 ;

        private int att2 ;

        public int meth(String s){…}

    public static class UneClasseMembreStatique {
            private int b ;
        …
            }
    public class UneClasseMembreInstance {
            private int c ;
        …
        }
}
```

Nom pleinement qualifié (ou nom complet) des classes membres internes

```
package yp.test;
public class UneClasseEnglobante {
                .....
        public static class UneClasseMembreStatique {
                                ...
        }
    public class UneClasseMembreInstance {
                                ...
            }
}
```

nom pleinement qualifié des classes :

yp.test.UneClasseEnglobante;
yp.test.UneClasseEnglobante.UneClasseMembreStatique;
yp.test.UneClasseEnglobante.UneClasseMembreInstance;

import possible :

```
import yp.test.UneClasseEnglobante;
import yp.test.UneClasseEnglobante.UneClasseMembreStatique;
import yp.test.UneClasseEnglobante.UneClasseMembreInstance;
```

Classes membres statiques

Les classes membres statiques ont accès à tous les membres statiques de la classe englobante

```
package test;
public class UneClasseEenglobante {

        private static int base=10 ;

            public static class UneClasseMembreStatique {
                    private int total ;
                    public UneClasseMembreStatique(int val){
                            this.total=val+base;
            }
                    public int getTotal (){
                return this.total;}
        }
```

Exemple d'utilisation (dans une autre classe) :

```
import test.UneClasseEnglobante.UneClasseMembreStatique ;
…
UneClasseMembreStatique ucms = new  UneClasseMembreStatique(5):
System.out.println(ucms.getTotal( )); // 15
```

Les classes membres statiques permettent
de bien packager ses classes : c'est leur principale utilité

Classes membres d'instance

```
public class Banque {

        public Banque(String nom){ …}
        public String toString( ){…}

        public  class Compte {
                public Compte(String nomTitulaire){…}
                public void ajouterOperation(int montant){…}
                public String toString( ){..}
}
```

une instance d'une classe membre d'instance
est toujours liée à une instance de la classe englobante

```
Banque banque= new Banque("YPB");
Compte compte1 = banque.new Compte("Dupond");
compte1.ajouterOperation(100);
Compte compte2 = banque.new Compte("Durand");
compte2.ajouterOperation(300);
System.out.println(compte1.toString()); //Compte :Dupond 100 YPB
System.out.println(banque.toString());  //Banque YPB capital 400
```

Accès aux membres de la classe englobante

la classe membre d'instance a accès à tous les membres de la classe englobante

```
public class Banque {
        private String nom;  private long capital=0;

        public Banque(String nom){this.nom = nom;}
        public String toString( ){
                return "Banque "+this.nom+" capital "+this.capital;
                }
        public  class Compte {
                private String nomTitulaire;  private int solde=0;

                public Compte(String nomTitulaire){
                        this.nomTitulaire=nomTitulaire;
                }
                public void ajouterOperation(int montant){
                        this.solde+=montant;
                        Banque.this.capital+=montant;
                        }
                public String toString( ){
                        return "Compte :"+nomTitulaire+" "+solde+"
"+Banque.this.nom;
                }}
```

this implicite

```
public class Banque {
        private String nom;  private long capital=0;

        public Banque(String nom){ this.nom = nom;}// this. obligatoire
        public String toString( ){
                return "Banque "+this.nom+" capital "+this.capital;
                }
        public  class Compte {
                private String nomTitulaire;  private int solde=0;

                public Compte(String nomTitulaire){
                        this.nomTitulaire=nomTitulaire; // this. obligatoire
                }
                public void ajouterOperation(int montant){
                        this.solde+=montant;
                        Banque.this.capital+=montant;
                        }
                public String toString( ){
                        return "Compte :"+this.nomTitulaire+" "+solde+"
"+Banque.this.nom;
                }}
```

Classes internes membres d'instances : syntaxe spécifique

Création d'une instance d'une classe interne membre d'instance

<instance de la classe englobante>.new

exemple :
Banque banque= **new Banque("YPB");**
Compte compte1 = banque.**new Compte("Dupond");**

Accès à partir de la classe interne à l'instance de la classe englobante

<nom de la classe englobante>.this

```
public class Banque {
        private long capital=0;
        …
        public  class Compte {
                ...  Banque.this.capital+=montant;// ou capital+=montant;
```

Classes internes membres : lexique

Terme	
Classe interne	Une classe interne est une classe définie dans une autre classe
Classe interne membre	Les classes internes membres sont définies à l'intérieur d'une classe au même niveau que les autres membres de la classe (constructeurs, méthodes, attributs). Ce peut être des membres d'instance ou statiques
Classe englobante	La classe dans laquelle est définie une classe interne est appelée la classe englobante

Classes internes membres

Les activités associées sont sur le site du cours.
Il est nécessaire de faire ces activités pour pouvoir considérer la séquence comme terminée.

Classes locales et lambda expressions

Description de la séquence

Description du contenu :

Durée de travail estimée : 60 minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs : étude des classes locales

Connaissances acquises : classes locales et lambda expressions
Compétences acquises : comprendre un programme utilisant des classes locales. Savoir définir et utiliser de telles classes
Pré requis : classes internes membres

Classe, classe interne, classe interne membre

Classe

Les membres d'une classe sont
- **des attributs (d'instance ou statique),**
- **des constructeurs,**
- **des méthodes (d'instance ou statique)**
- **et peuvent être aussi des classes appelées classes internes**

Classe interne

C'est une classe définie dans une autre classe

classe interne membre

- **définie au même niveau que les autres membres**

classe locale

classe locale (nommée)

classe locale anonyme

- **définie dans une méthode ou dans un constructeur**

lambda expression

- **une syntaxe simplifiée pour la definition de classes locales anonymes**

La méthode sort de java.util.List

En vue de l'exemple suivant …

```
public interface List <E> { …
/** Sorts this list according to the order induced by the specified Comparator.
 *   All elements in this list must be mutually comparable using the specified comparator
 * @params   c the Comparator used to compare list elements. A null value indicates that the
elements' natural ordering should be used
* @ since 1.8
**/
default void sort(Comparator<? super E> c)
```

```
public interface Comparator<T> { …
/**
* @param o1 - the first object to be compared.
* @param o2 - the second object to be compared.
* @ return a negative integer, zero, or a positive integer as the first argument is less than
*                          , equal to, or greater than the second.
**/
int compare(T o1, T o2)
```

La méthode sort de java.util.List

Exemple d'utilisation de la méthode sort de java.util.List :

```
public class Critere implements Comparator<Integer>{

                public Critere (boolean ordreCroissant){
                        this.ordreCroissant = ordreCroissant;
                        }
                @Override
                public int compare(Integer int1, Integer int2) {
                        return (this.ordreCroissant?1:-1)*
int1.compareTo(int2);
                }
}

// dans une fonction …
List<Integer> list= Arrays.asList(3,17,81,9,12);
list.sort(new Critere(false)); // list vaut alors [81, 17, 12, 9, 3]
```

Classes locales

```
public static void trier(List<Integer> list,boolean ordreCroissant){

        class Critere implements Comparator<Integer>{

                @Override
                public int compare(Integer int1, Integer int2) {
                        return (ordreCroissant?1:-1)* int1.compareTo(
                }
        }
        list.sort(new Critere());
}

public static void main(String [] args){

        List<Integer> list= Arrays.asList(3,17,81,9,12);
        trier(list,false);
        System.out.println(list);// [81, 17, 12, 9, 3]
        trier(list,true);
        System.out.println(list);// [3, 9, 12, 17, 81]
}
```

Classes locales anonymes

Classe locale nommée

```
public static void trier(List<Integer> list,boolean ordreCroissant){
        class Critere implements Comparator<Integer>{
                @Override
                public int compare(Integer int1, Integer int2) {
                        return (ordreCroissant?1:-1)* int1.compareTo(int2);
                }
        }
        list.sort(new Critere());
}
```

Classe locale anonyme

C'est une classe locale sans nom définie et instanciée sur la même instruction.

```
public static void trier(List<Integer> list,boolean ordreCroissant){
        list.sort(new Comparator<Integer>( ){
                @Override
                public int compare(Integer int1, Integer int2) {
                        return (ordreCroissant?1:-1)* int1.compareTo(int2);
                }
                );
}
```

Classes locales

Elles sont définies dans des fonctions à l'emplacement d'une instrcution

Comme pour les classes internes membres d'instance

- une instance d'une classe locale est toujours liée à une instance de la classe englobante
- la classe locale a accès à tous les membres de la classe englobante

Accès aux variables de la fonction

la classe locale a accès à
- **toutes les variables "final" de sa zone de portée**

Accessibilité de la classe locale

- **les règles d'accessibilité sont les mêmes que pour une variable**
  - **accessibles dans le bloc où elle est définie juste après sa définition**

Lambda expression

Classe locale anonyme

```
public static void trier(List<Integer> list,boolean ordreCroissant){

        list.sort(new Comparator<Integer>( ){
            @Override
            public int compare(Integer int1, Integer int2) {
                return (ordreCroissant?1:-1)* int1.compareTo(int2);
            }
            );
}
```

Les lambda expression sont une autre syntaxe utilisable pour les classes locales anonymes lorsque la classe anonyme n'a qu'une seule méthode héritée (en dehors des méthodes de la classe Object)

Lambda expression

```
public static void trier(List<Integer> list,boolean ordreCroissant){
        list.sort( (int1,int2) -> return (ordreCroissant?1:-1)* int1.compareTo(int2));
}
```

Utiliser classes internes et lambda expressions pour les handlers

Description de la séquence

Description du contenu :

Durée de travail estimée : 60 minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs : connaître les différents niveaux où une classe peut être définie

Connaissances acquises :
Compétences acquises : savoir utiliser ces différentes catégories de classes dans ses propres programmes
Pré requis : classes internes et lambda expressions

Classes internes non statiques

```
public class PanelCompteur extends BorderPane{
        private Compteur compteur;
        private Label label;
        public PanelCompteur (Compteur compteur){
                this.compteur=compteur;
                Button button = new Button("OK");
                this.label =new Label("--> 0");
                button.setOnAction(new ClickHandlerI( ));
                this.setBottom(button);
                this.setCenter(this.label);
        }
        class ClickHandlerI implements EventHandler<ActionEvent> {
                @Override
                public void handle(ActionEvent event) {
                        PanelCompteur.this.compteur.incrementer(1);
                        PanelCompteur.this.label.setText("--> "+
                                PanelCompteur.this.compteur.getValue());
                }}
}
```

Classes internes non statiques

```
public class PanelCompteur extends BorderPane{
        private Compteur compteur;
        private Label label;
        public PanelCompteur (Compteur compteur){
                        …
        }
        class ClickHandlerI implements EventHandler<ActionEvent> {
                @Override
                public void handle(ActionEvent event) {
                        PanelCompteur.this.compteur.incrementer(1);
                        PanelCompteur.this.label.setText("--> "+
                                PanelCompteur.this.compteur.getValue());
                }}
}
```

Accès à l'instance de la classe englobante :
<nomDeLaClasseEnglobante>.this
Accès à un membre de la classe englobante
<nomDeLaClasseEnglobante>.this.<nom de l'attribut>
<nomDeLaClasseEnglobante>.this.<nom de la méthode>(…)

Accès aux membres de la classe englobante

On peut aussi, s'il n'y a pas de confusion possible mettre
<nom de l'attribut> au lieu de PanelCompteur.this.<nom de l'attribut>

```
public class  PanelCompteur extends BorderPane{
 ……..
  class ClickHandlerI implements EventHandler<ActionEvent> {
                    @Override
                    public void handle(ActionEvent event) {
                              compteur.incrementer(1);
                              label.setText("--> "+ compteur.getValue( ));
```

remarque

On ne peut pas écrire this.label car label n'est pas un attribut de la classe interne.

Classes locales

```
public class PanelCompteur extends BorderPane{
        private Compteur compteur;
        private Label label;
        public PanelCompteur (Compteur compteur){
                this.compteur=compteur;
                Button button = new Button("OK");
                this.label =new Label("--> 0");
                class ClickHandlerI implements
EventHandler<ActionEvent> {
                        @Override
                        public void handle(ActionEvent event) {

        PanelCompteur.this.compteur.incrementer(1);
                                PanelCompteur.this.label.setText("--> "+
                                PanelCompteur.this.compteur.getValue());
                        }}
                button.setOnAction(new ClickHandlerI( ));
                this.setBottom(button);
                this.setCenter(this.label);
        }
```

Classes locales

```
public class PanelClickCL extends BorderPane{

        public PanelClickCL(Compteur compteur){
                Button button = new Button("OK");
                Label label =new Label("--> 0");

                class EcouteurClickL implements EventHandler<ActionEvent> {
                        @Override
                        public void handle(ActionEvent event) {
                                compteur.incrementer(1);
                                label.setText("--> "+compteur.getValue( ));
                        }
                }

                button.setOnAction(new EcouteurClickL( ));

                this.setBottom(button);
                this.setCenter(label);

}
```

Classes locales anonymes

**Une classe locale peut être anonyme.
Elle est alors définie "sur place".**

```
public class PanelCompteur extends BorderPane{
        private Compteur compteur;
        private Label label;
        public PanelCompteur (Compteur compteur){
                this.compteur=compteur;
                Button button = new Button("OK");
                this.label =new Label("--> 0");
                button.setOnAction(new EventHandler<ActionEvent>( ) {
                   @Override
                   public void handle(ActionEvent event) {
                           PanelCompteur.this.compteur.incrementer(1);
                           PanelCompteur.this.label.setText("--> "+
                                PanelCompteur.this.compteur.getValue());
                    }}
                );
                this.setBottom(button);
                this.setCenter(this.label);
}
```

Classes locales anonymes (2)

```
new EventHandler<ActionEvent>( ) {
                    @Override
                    public void handle(ActionEvent event) {
                        PanelCompteur.this.compteur.incrementer(1);
                        PanelCompteur.this.label.setText("--> "+
                            PanelCompteur.this.compteur.getValue());
                     }
}
```

La définition d'une classe anonyme est la même que pour une classe locale.
Elle va toujours de pair avec la création d'une instance de la classe.
Restrictions :

- pas de constructeurs
- impossibilité de créer des instances d'une même classe anonyme à 2 endroits différents dans le programme.

Classes anonymes, accès aux variables

```
public class PanelCompteur extends BorderPane{

        public PanelCompteur (final Compteur paramCompteur){

                Button button = new Button("OK");
                final Label label =new Label("--> 0");
                button.setOnAction(new EventHandler<ActionEvent>( ) {
                   @Override
                   public void handle(ActionEvent event) {
                           paramCompteur.incrementer(1);
                           label.setText("--> "+
                                   paramCompteur.getValue());
                    }}
                );
                this.setBottom(button);
                this.setCenter(label);
}
```

Lambda expression : Une simplification d'écriture

```
button.setOnAction(new EventHandler<ActionEvent>( ) {
                        @Override
                        public void handle(ActionEvent event) {
                                paramCompteur.incrementer(1);
                                label.setText("--> "+ paramCompteur.getValue());
                         }}
                    );
```

Simplification possible dans le cas d'interface avec une seule méthode

```
button.setOnAction((event) -> {
                                paramCompteur.incrementer(1);
                                label.setText("--> "+ paramCompteur.getValue());
                        }
                    );
```

Lambda expression : Comment le compilateur s'y retrouve …

```
button.setOnAction((event) -> {
                        paramCompteur.incrementer(1);
                        label.setText("--> "+ paramCompteur.getValue());
                    }
                );
```

L'interface est déduite de la signature de setOnAction
La méthode définie est l'unique méthode de l'interface
Le type de event est déduit de la signature de la méthode (ici handle)

```
button.setOnAction(new EventHandler<ActionEvent>( ) {
                    @Override
                    public void handle(ActionEvent event) {
                        paramCompteur.incrementer(1);
                        label.setText("--> "+ paramCompteur.getValue());
                     }}
                );
```

Evénements et gestionnaires d'événements

Description de la séquence

Description du contenu :

Durée de travail estimée : 60 minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs :

Connaissances acquises :
Compétences acquises :
Pré requis :

Les gestionnaires d'événements (handlers)

1. Action de l'utilisateur sur le clavier ou la souris dans un composant (*control*)

↓

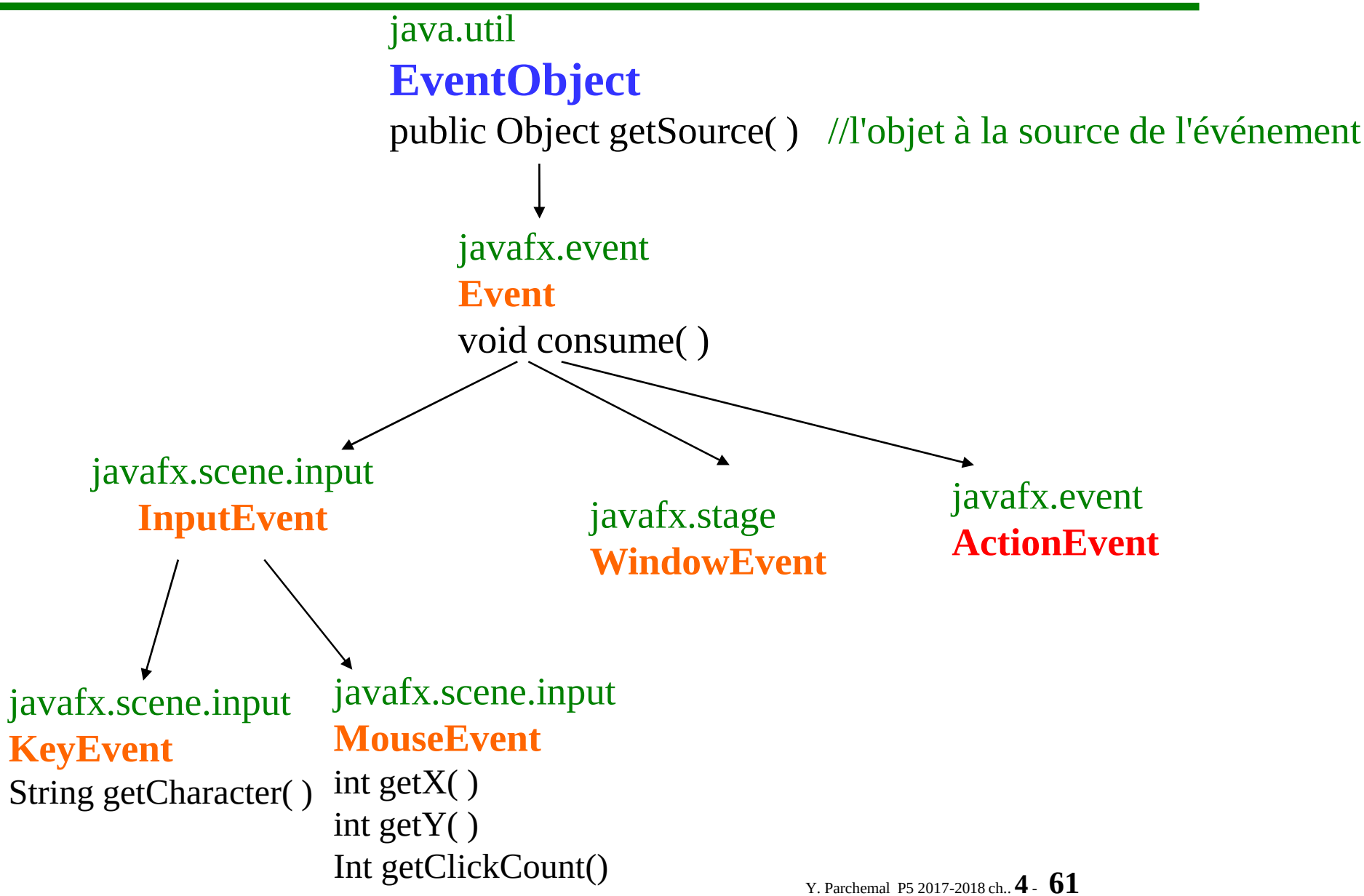
2. Génération automatique d'un événement par javafx

↓

3. Réalisation de la tâche
grâce au *handler (gestionnaire d'événements)*
associé (par le programme)
à cette action pour ce composant

Les handlers sont des objets qui permettent d'effectuer les tâches demandées par l'utilisateur de l'interface graphique.

Quelques classes d'événements de javafx



Association d'un handler à un composant pour un type d'événement
La méthode setOnAction de la classe Button

public final void setOnAction(EventHandler<ActionEvent> handler)

button.setOnAction(handler); →

'handler' devient le handler associé à ce bouton pour la gestion des click
Si je clique sur button, un événement ActionEvent sera créé.
et la méthode handle sera appliqué au 'handler' avec en paramètre l'événement créé

setOnAction est définie dans la classe ButtonBase
(ButtonBase est la classe mère de Button mais aussi Checkbox …)

associer un handler à un composant

button.setOnAction**(h)**	ActionEvent	click bouton
node. setOnMouseEntered**(h)**	MouseEvent	entrée de la sourie sur un composant
node. setOnMouseExited**(h)**	MouseEvent	sortie de la sourie d'un composant
node.setOnKeyPressed(h)	KeyEvent	appui sur une touche du clavier
window.setOnCloseRequest(h)	WindowEvent	click sur la case de fermeture

Une variante du compteur de click (1)

On souhaite maintenant que le compteur soit incrémenté juste par l'arrivée de la sourie sur le bouton

L'événement est alors un MouseEvent (au lieu de ActionEvent)

La méthode pour affecter le handler est : setOnMouseEntered (au lieu de SetOnAction)

Le reste est inchangé

Une variante du compteur de click (2)

```
public class PanelCompteurSansClick extends BorderPane{

        public PanelCompteurSansClick (Compteur compteur){

                Button btn = new Button("OK");

                Label label =new Label("--> 0");

                btn.setOnMouseEntered(new HandlerSansClick(label,compteur))

                this.setBottom(btn);
                this.setCenter(label);

        }
}
```

Une variante du compteur de click (3)

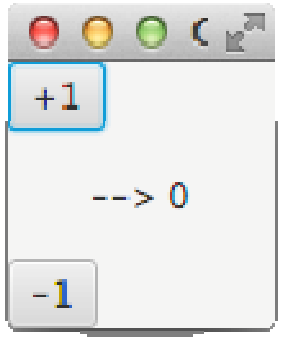
```
class HandlerSansClick implements EventHandler<MouseEvent> {
        private Label label;
        private Compteur compteur;

        public HandlerSansClick (Label label,Compteur compteur){
                this.label=label;
                this.compteur=compteur;
        }
        @Override
        public void handle(MouseEvent event) {
                this.compteur.incrementer(1);
                this.label.setText("--> "+this.compteur.getValue());
        }
```

Sans changement

Handler à l'écoute de plusieurs sources

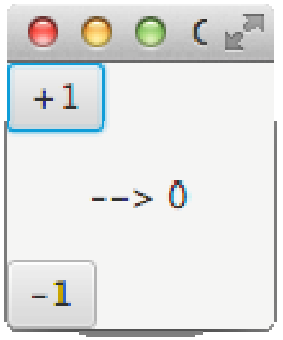
Un même handler peut gérer des événements de même nature provenant de plusieurs sources
exemple : un même handler à l'écoute de plusieurs boutons

exemple traité : PanelClickPlusMoins

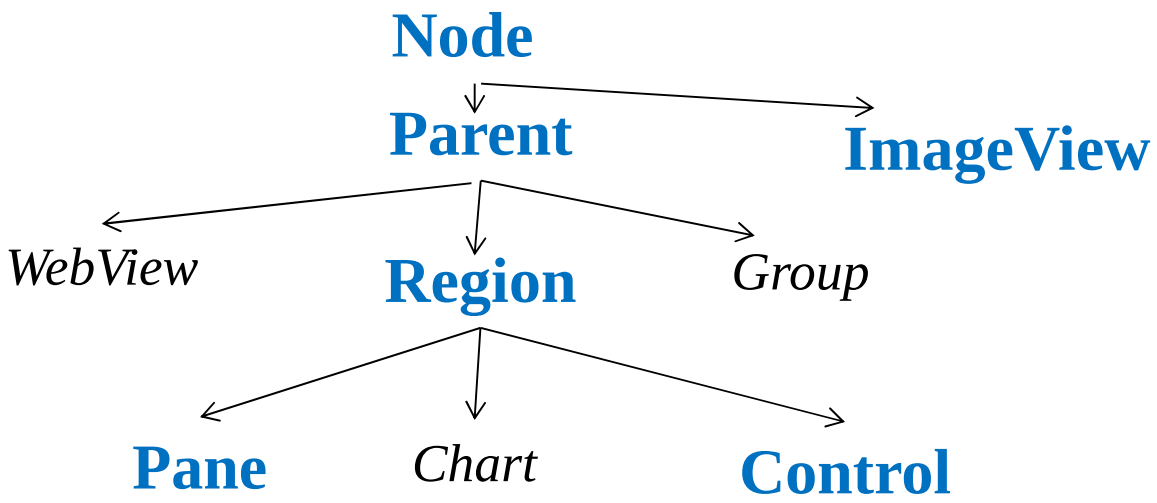


Un handleur à l'écoute de deux boutons

```
public class PanelClickPlusMoins extends BorderPane{
        public PanelClickPlusMoins(Compteur compteur){
                Button buttonPlus = new Button("+1");
                Button buttonMoins = new Button("-1");

                Label label =new Label("--> 0");

                EventHandler<ActionEvent> handler = (event) -> {
                        compteur.incrementer(event.getSource()==buttonPlus?1:-1);
                        label.setText("--> "+compteur.getValue());
                };
                buttonPlus.setOnAction(handler);
                buttonMoins.setOnAction(handler);

                this.setTop(buttonPlus);
                this.setBottom(buttonMoins);
                this.setCenter(label);
        }
}
```



Quelques sous classes de Node



Node		**Une scène permet de visualiser une arborescence**
	Parent	**peut avoir des fils**
	WebView	*pour visualiser une page WEB*
	Region	**on peut associer un style CSS**
	Pane	**la liste des fils est directement modifiable**
	Chart	*des graphiques (camembert, baton, …)*
	Control	**des composants sur lequel on peut agir (boutons …**
	Group	*pas de gestion automatisée du positionnement des fil*
	ImageView	**pour la visualisation des images**

Label Button et CheckBox

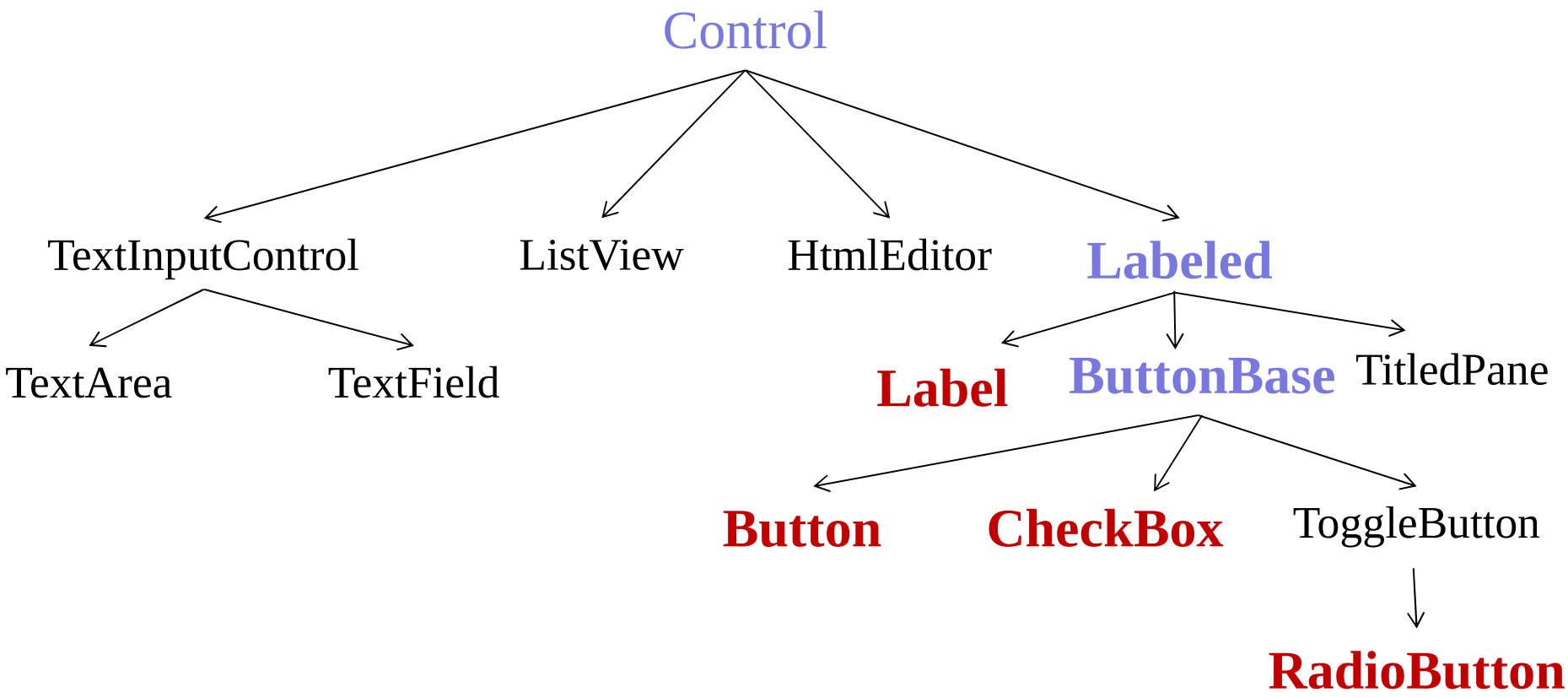
Description de la séquence

Description du contenu : Arborescence des composants, les composants Label Button et Checkbox

Durée de cette vidéo : environ 12 minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs :

Connaissances acquises : connaissances de base sur 3 classes de composants
Compétences acquises : savoir utiliser 3 composants javafx basiques : Label Button et Checkbox
Pré requis :

l'arborescence de derivation de Control



Javafx.scene.control.Label : les 3 constructeurs

```
public  class  Label extends  Labeled{


 /** crée un label avec un texte vide */
 public Label( ){…}

 /** crée un label avec un texte donnée*/
 public Label(String  text){…}

 /** crée un label avec un texte et un Node (le plus souvent un ImageView)
                    @param text le texte du label
                    @param graphics un noeud (le plus souvent une image) associé à ce
label */
 public Label(String text, Node graphics) {…}
}
```

```
Label label1 = new Label( );
Label label2 = new Label("hello");
Label label3 = new Label("hello", new ImageView(new Image("logo.jpg")));
```

Javafx.scene.control.Labeled

```
public  abstract class  Labeled  extends  Control {

        /** rend le texte associé à ce control*/
        public String getText( ) {…}
        /** modifie le texte associé à ce control*/
        public void setText(String texte) {…}

        /** rend l'image associée à ce control*/
        public Node getGraphic( ) {…}
        /** modifie l'image associée à ce control */
        public void setGraphics(Node graphic) {…}


}
```

javafx.scene.control.Button

```
public  class  Button extends ButtonBase {

// Les constructeurs
 public Button( ){…}
 public Button (String  text){…}
 public Button (String text, Node graphics) {…}
}
```

Même type de constructeurs que la classe Label …

ButtonBase

public abstract class ButtonBase extends Labeled{

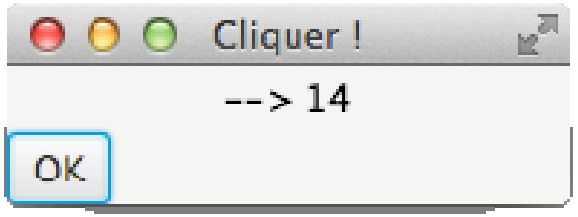
/ Sets the value of the property onAction */**
public final void **setOnAction**(EventHandler<ActionEvent> onAction){…}

/** gets the value of the property onAction*/
public final EventHandler<ActionEvent> getOnAction( ){…}

…
}

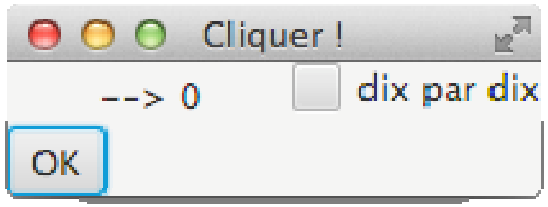
Exemple d'utilisation d'une CheckBox

```
public class PanelClickB extends BorderPane{

        public PanelClickB(Compteur compteur){

                Label label=new Label("--> 0");
                this.setCenter(label);
                Button button = new Button("OK");
                this.setBottom(button);

                button.setOnAction(
                        (event) -> {
                                compteur.incrementer(1);
                                label.setText("-->
"+compteur.getValue( ));
                                }
                        );
        }}
```

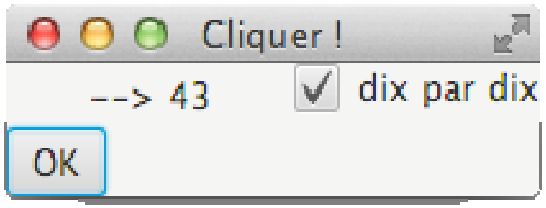


Checkbox

Une Checkbox est une case à cocher à laquelle est associée une étiquette



Checkbox non cochée
Etiquette associée: "dix par dix"



Checkbox cochée

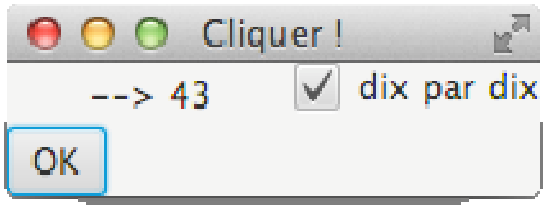
```
// exemple de création d'une CheckBox
CheckBox box10  = new CheckBox("dix par dix");
```

javafx.scene.control.CheckBox

```
/** La classe CheckBox implémente les cases à cocher */

public  class  CheckBox extends ButtonBase {

            /** construit une CheckBox sans texte associé*/
        public Checkbox( ) {…}

            /** construit une CheckBox de texte 'text' */
             public Checkbox(String text) {…}

            /** rend true si la checkbox est sélectionnée, false sinon*/
            public boolean isSelected( )

            /* permet le cochage et le décochage de la checkbox */
            public void setSelected(boolean value)

}
```

Exemple d'utilisation d'une CheckBox

```
public class PanelClickCB extends BorderPane{

    public PanelClickCB(Compteur compteur){

        Label label=new Label("--> 0");
        CheckBox checkBox=new CheckBox("dix par dix");
        Button button = new Button("OK");

        button.setOnAction(
                (event) -> {

compteur.incrementer(checkBox.isSelected( ) ? 10 : 1);
                        label.setText("--> "+compteur.getValue());
                        }
                );
        this.setBottom(button);
        this.setCenter(label);
        this.setRight(checkBox);
    }
```



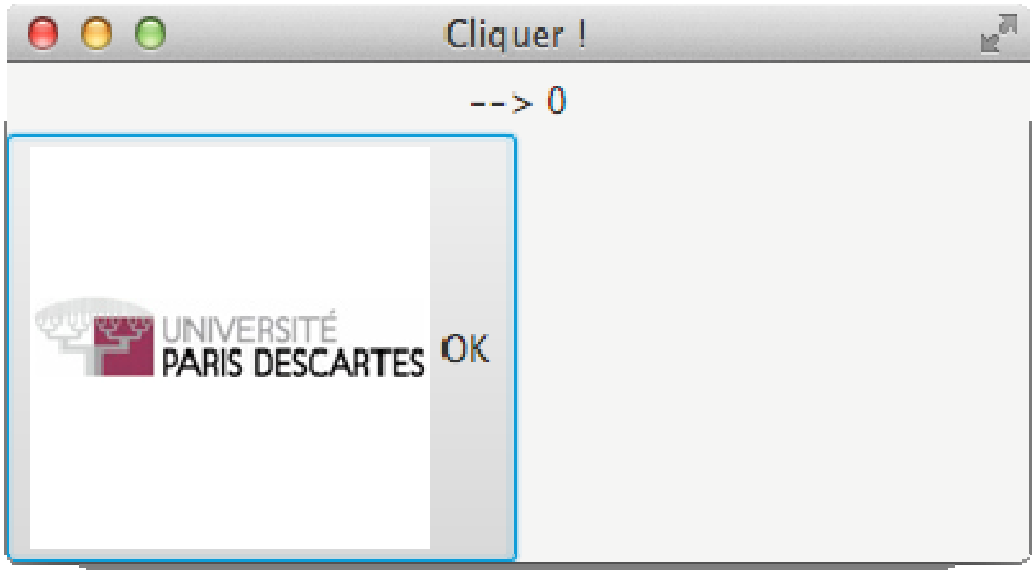
JAVAFX : Label Button et CheckBox

Activités associées

Les activités associées sont sur le site du cours.
Il est nécessaire de faire ces activités pour pouvoir considérer la séquence comme terminée.

exemple de création d'un Button avec image

```
Image image = new Image("file:logo-descartes.jpg",false)));

Button button = new Button("OK",new ImageView(image));
```



Remarque : formats possibles : gif jpeg png mpo

La classe Pane et ses dérivées

Description de la séquence

Description du contenu :

Durée de travail estimée : minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs :

Connaissances acquises :
Compétences acquises :
Pré requis :

La classe Pane et ses dérivées

Un pane (panneau) est un composant graphique conçu pour contenir d'autres composants

Pane — **La classe mère des panneaux**

FlowPane — **Les composants sont mis les uns à la suite des autres avec passage à la ligne quand c'est nécessaire**

BorderPane — **répartit les composants à 5 endroits: Right, Left, Top, Bottom et Center**

GridPane — **Les composants sont placés dans une grille (composée de lignes et de colonnes)**

StackPane — **Les composants sont "empilés" les uns sur les autres**

Hbox — **Les composants sont tous mis sur une ligne**

VBox — **Les composants sont tous mis sur une colonne**

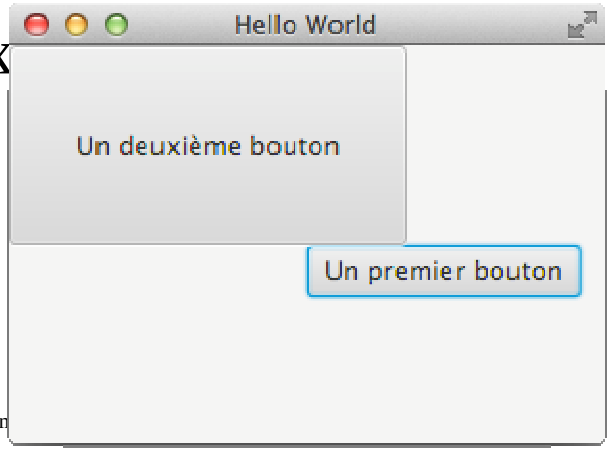
Utilisation de la classe Pane

```
public class Pane extends Region {
        @Override
        public ObservableList<Node> getChildren( )
        …
}
```

```
Pane pane= new Pane();
```

Les instances directes de Pane sont utilisées lorsque l'on souhaite fixer soi même la taille et l'emplacement des composants

```
Button button1 = new Button("Un premier
n");
Button button2 = new Button("Un deux
n");

button1.relocate(150, 100);
button2.setPrefSize(200, 100);
```

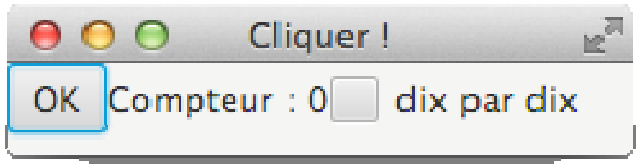


Le container FlowPane

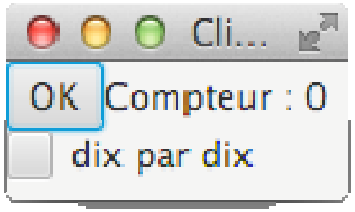
FlowPane

Les composants sont mis les uns à la suite des autres avec passage à la ligne quand c'est nécessaire

```
Pane pane = new FlowPane( );
Button button = new Button("OK");
Label label =new Label("Compteur : 0");
Checkbox checkBox=new CheckBox("dix par dix");

pane.getChildren( ).addAll(button,label,checkBox);
```



Une ligne suffit :
Le panneau est suffisamment large



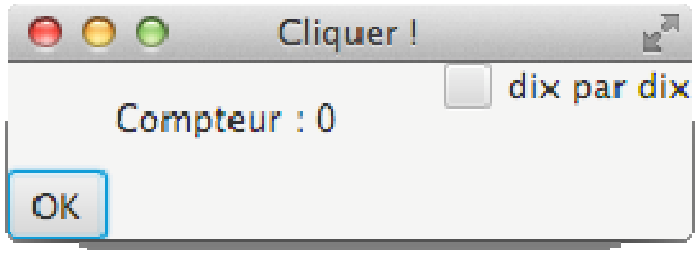
Deux lignes sont nécessaires

Le container BorderPane

BorderPane

5 localisations possibles pour les composants : top bottom center left right

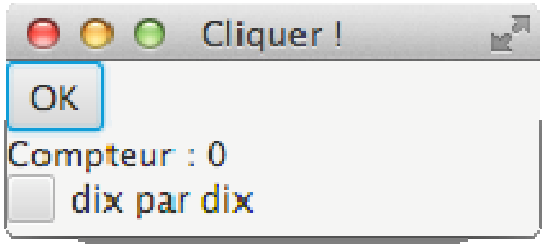
```
BorderPane pane = new BorderPane( );

Button button = new Button("OK");
Label label =new Label("Compteur : 0");
Checkbox checkBox=new CheckBox("dix par dix");

pane.setBottom(button);
pane.setCenter(label);
pane.setRight(this.checkBox);
```



Le container GridPane

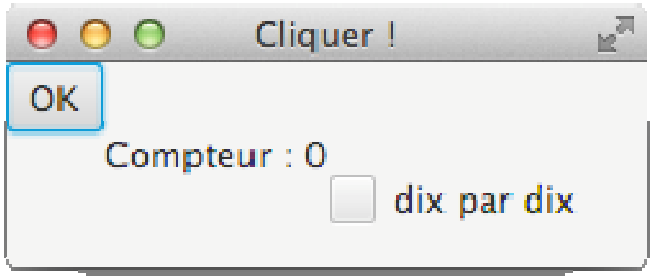
GridPane **Une grille avec des lignes et des colonnes**

```
/** adds a child to the gridpane at the specified column,row position*/
public void add(Node child, int columnIndex, int rowIndex)
```




```
GridPane pane = new GridPane();
```

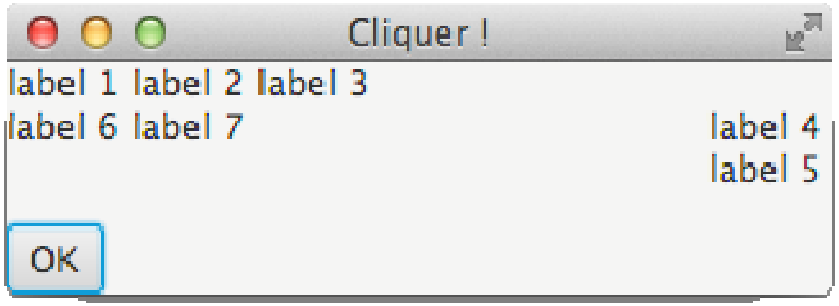
```
pane.add(button, 1, 1);
pane.add(label, 1, 2);
pane.add(checkBox, 1, 3);
```




```
GridPane pane = new GridPane();
pane.add(button, 1, 1);
pane.add(label, 2, 2);
pane.add(checkBox, 3, 3);
```

exemple d'imbrication de panels

```
BorderPane pane = new BorderPane();

Pane pTop= new FlowPane( );
Label labN1 = new Label("label 1 ");
Label labN2 = new Label("label 2 ");
Label labN3 = new Label("label 3 ");
pTop.getChildren( ).addAll(labN1,labN2,labN3);
pane.setTop(pTop);

GridPane pRight = new GridPane( );
pRight.add(new Label("label 4 "),1,1);
pRight.add(new Label("label 5 "),1,2);
pane.setRight(pRight);

GridPane pLeft = new GridPane( );
pLeft .add(new Label("label 6 "),1,1);
pLeft .add(new Label("label 7 "),2,1);
pane.setLeft(pLeft);

Button bOK = new Button("OK");
pane.setBottom(bOK);
```



RadioButton, TextField, TextArea

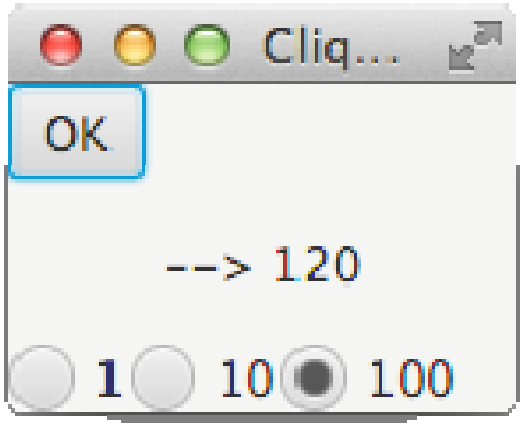
Description de la séquence

Description du contenu :

Durée de travail estimée : minutes
Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs :

Connaissances acquises :
Compétences acquises :
Pré requis :

RadioButton et ToggleGroup



3 RadioButton appartenant au même ToggleGroup

ToggleGroup

Un ToggleGroup est un ensemble deToggle (= bascule)

Au maximum, un seul élément d'un ToggleGroup peut être sélectionné

La sélection d'un élément appartenant à un ToggleGroup entraîne donc la désactivation de l'élément du groupe qui était précédemment actif.

javafx.scene.control.**ToggleGroup**

```
public  class ToggleGroup {

/** construit un ToggleGroup */
public ToggleGroup( ) {…}

/** @return  le composant sélectionné
                                    null si aucun composant n'est sélectio
public Toggle getSelectedToggle ( ){…}

/** modifie le composant sélectionné */
public void selectToggle(Toggle toggle){…}
...
}
```

Les éléments d'un ToggleGroup sont le plus souvent des RadioButton

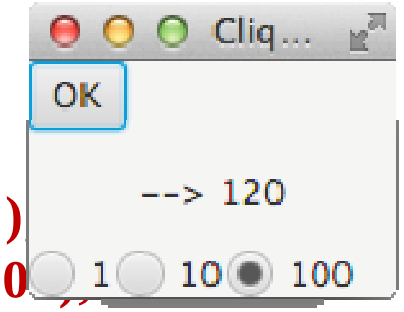
RadioButton

```
public  class  RadioButton extends  ToggleButton implements Toggle{
/** construit une CheckBox de text 'text'*/
 public RadioButton (String text) {…}
…
}
public interface Toggle {
/** Indicates whether this Toggle is selected */
public boolean    isSelected( );
/** Sets this Toggle as selected or unselected */
public void         setSelected(boolean selected);

/** Sets the ToggleGroup to which this Toggle belongs */
public void         setToggleGroup(ToggleGroup toggleGroup);
/** Returns The ToggleGroup to which this Toggle belongs */
public ToggleGroup        getToggleGroup( );
}
```

PanelClickBR

public PanelClickBR(Compteur compteur){

Label label=**new Label("--> 0");;**
this.setCenter(label);



RadioButton rb1=**new RadioButton("1")**
RadioButton rb10=**new RadioButton("10**
RadioButton rb100=**new RadioButton("100");**

ToggleGroup toggleGroup = **new ToggleGroup();**
rb1.setToggleGroup(toggleGroup);
rb1.setSelected(**true);**
rb10.setToggleGroup(toggleGroup);
rb100.setToggleGroup(toggleGroup);

FlowPane pane = **new FlowPane();**
pane.getChildren().addAll(rb1,rb10,rb100);
this.setBottom(pane);

PanelClickBR

```
        Button button = new Button("OK");
        this.setTop(button);

        button.setOnAction(
                (event) -> {
                        RadioButton radio= (RadioButton)toggleGroup.getSelect
                        compteur.incrementer(Integer.parseInt(radio.getText()));
                        label.setText("--> "+compteur.getValue());
                }
        );


}
```

Fenêtre avec TextField

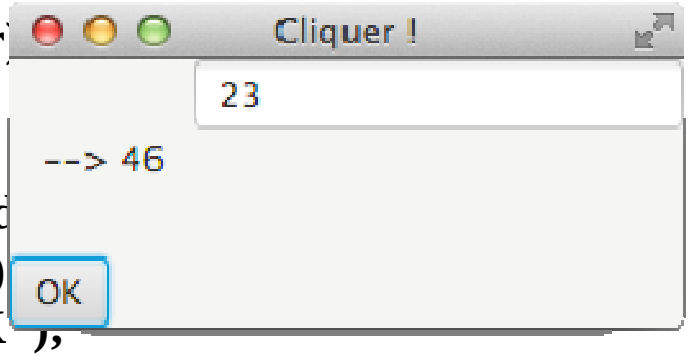
TextField classe des zones de texte sur une ligne pouvant être éditée

TextArea classe des zones de texte sur plusieurs lignes pouvant être éditées

TextInputControl et TextField et TextArea

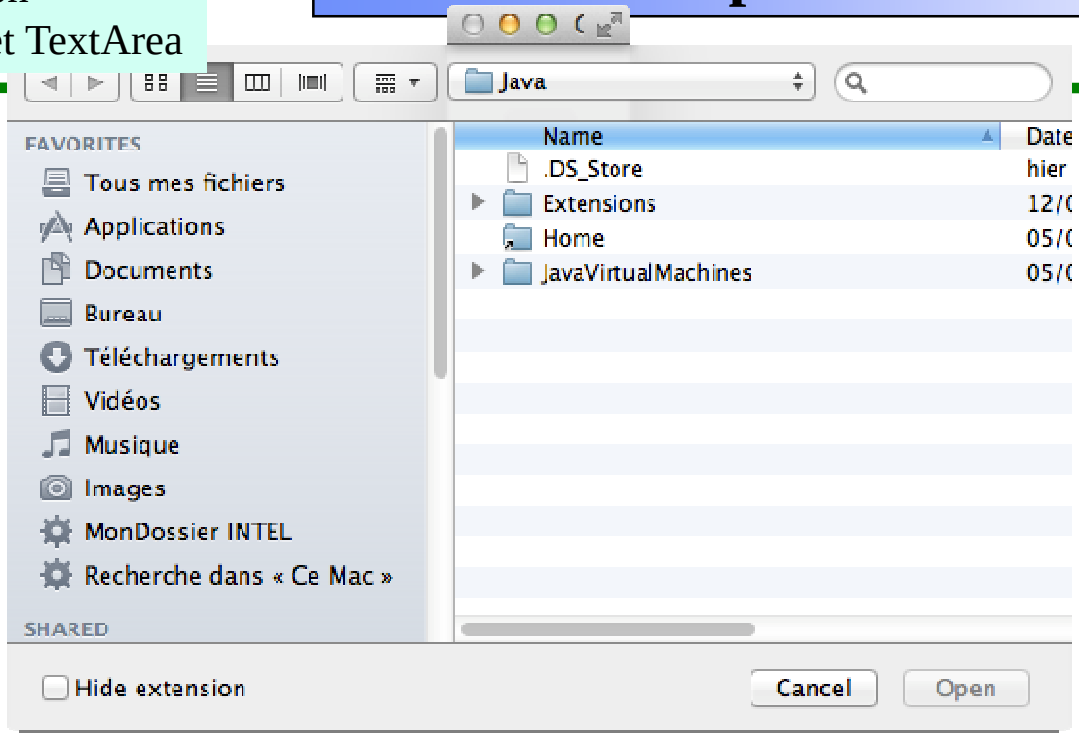
```
public  class  TextInputControl extends Control
{…..
    public String getText( ) {…}
    public void setText(String  text) {…}
    public void appendText(String text) {…}
    public boolean isEditable( ) {…}
    public void setEditable(boolean  editable) {…}
...}
public  class  TextField extends TextInputControl
{   ...
   public TextField(String  text){…}
   public final void setPrefColumnCount(int value) {…}}

public  class  TextArea extends  TextInputControl
{...
public TextArea(String text) {…}
public final void setPrefColumnCount(int value) {…}
public final void setPrefRowCount(int value) {…}}
```

Fenêtre avec TextField




```
public PanelClickTF(Compteur compteur){

        Label label=new Label("--> 0");;
        TextField textField=new TextFiel
        textField.setPrefColumnCount(15);
        Button button = new Button("OK");

        button.setOnAction(
                (event) -> {
                        compteur.incrementer(Integer.parseInt(textField.getText()
                        label.setText("--> "+compteur.getValue());
                }
        );

        this.setBottom(button);
        this.setCenter(label);
        this.setRight(textField);

}
```

FileChooser : pour saisir des adresses de fichier




```
// création d'une instance de JFileChooser
FileChooser hf = new FileChooser( );
// on indique ici le répertoire devant être proposé au départ
hf.setInitialDirectory(new java.io.File("/Library/Java"));
// On demande l'affichage de la fenêtre de dialogue
File file = hf.showOpenDialog(window);// window = stage par exemple
// l'appel précédent est bloquant :
System.out.println(file);
```

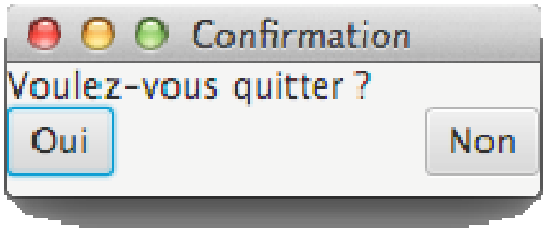
Gestion de la case de fermeture

L'utilisateur clique sur OK

incrémenter le compteur
(GestionnaireClicks)

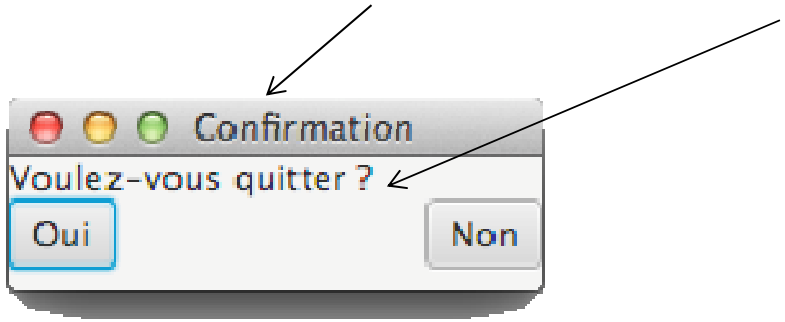
l'utilisateur clique sur la case de fermeture

fermer la fenêtre
après confirmation de l'utilisateur
(GestionnaireFermeture)



Fenêtre de dialogue

new ConfirmStage(stage,"Confirmation","Voulez-vous quitter ?");




```
public class ConfirmStage extends Stage {
        public ConfirmStage(Stage mainStage,String title,String message){
                this.setTitle(title);
                Button bOui=new Button("Oui");
                Button bNon=new Button("Non");
                BorderPane pane = new BorderPane( );
                pane.setTop(new Label(message));
                pane.setLeft(bOui);
                pane.setRight(bNon);
                …
```

Fenêtre de dialogue

```
public class ConfirmStage extends Stage {
        private enum Reponse {INCONNU,OUI,NON;}
        private Reponse reponse = Reponse.INCONNU;
                public ConfirmStage(Stage mainStage,String title,String message){
                        this.setTitle(title);
                        Button bOui=new Button("Oui");
                        Button bNon=new Button("Non");
                        BorderPane pane = new BorderPane();
                        pane.setTop(new Label(message));
                        pane.setLeft(bOui);pane.setRight(bNon);

                        bOui.setOnAction((e) -> {close(
);reponse=Reponse.OUI;});
                        bNon.setOnAction((e)-> {close(
);reponse=Reponse.NON;});
```

Fenêtre de dialogue

```
public class ConfirmStage extends Stage {
        private enum Reponse {INCONNU,OUI,NON;}
        private Reponse reponse = Reponse.INCONNU;
                public ConfirmStage(Stage mainStage,String title,String message){
                        this.setTitle(title);
                        …… (voir pages précédentes)
                        // indiquer que ca peut être une fenetre bloquante

        this.initModality(Modality.APPLICATION_MODAL);
                        // la fenetre dont dépend cette fenetre
                        this.initOwner(mainStage);

                        Scene scene = new Scene(pane, 200, 50);
                        this.setScene(scene);

                        this.showAndWait();
                }
                public boolean isReponsePositive(){return
reponse==Reponse.OUI;} }
```

Utilisation du gestionnaire de fermeture

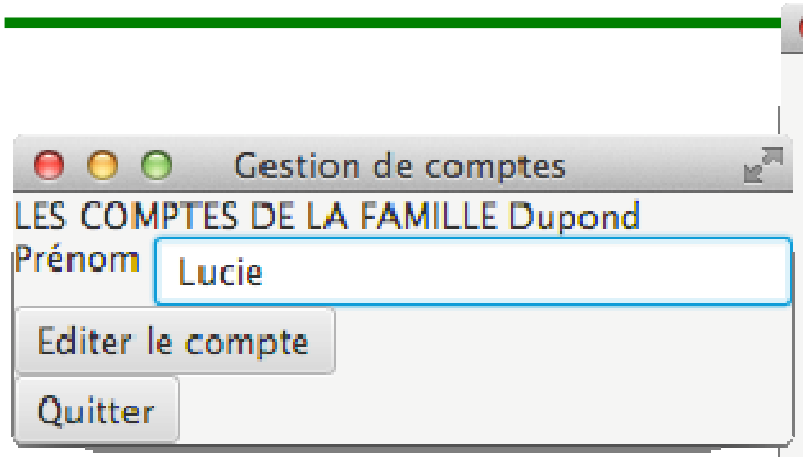
```
public void start(Stage stage) {

        stage.setTitle("Cliquer !");

        Compteur compteur = new Compteur( );
        Scene scene = new Scene(new PanelClickCB(compteur), 200, 50);
        stage.setScene(scene);

        stage.setOnCloseRequest( (e)->{
                ConfirmStage confirm = new ConfirmStage(stage,"Confirmation","Voulez-v
                if (!confirm.isReponsePositive( ))
                        e.consume();
                });
        stage.show();
        }
```

Avec utilisation de Alert (javafx 8.0.40)

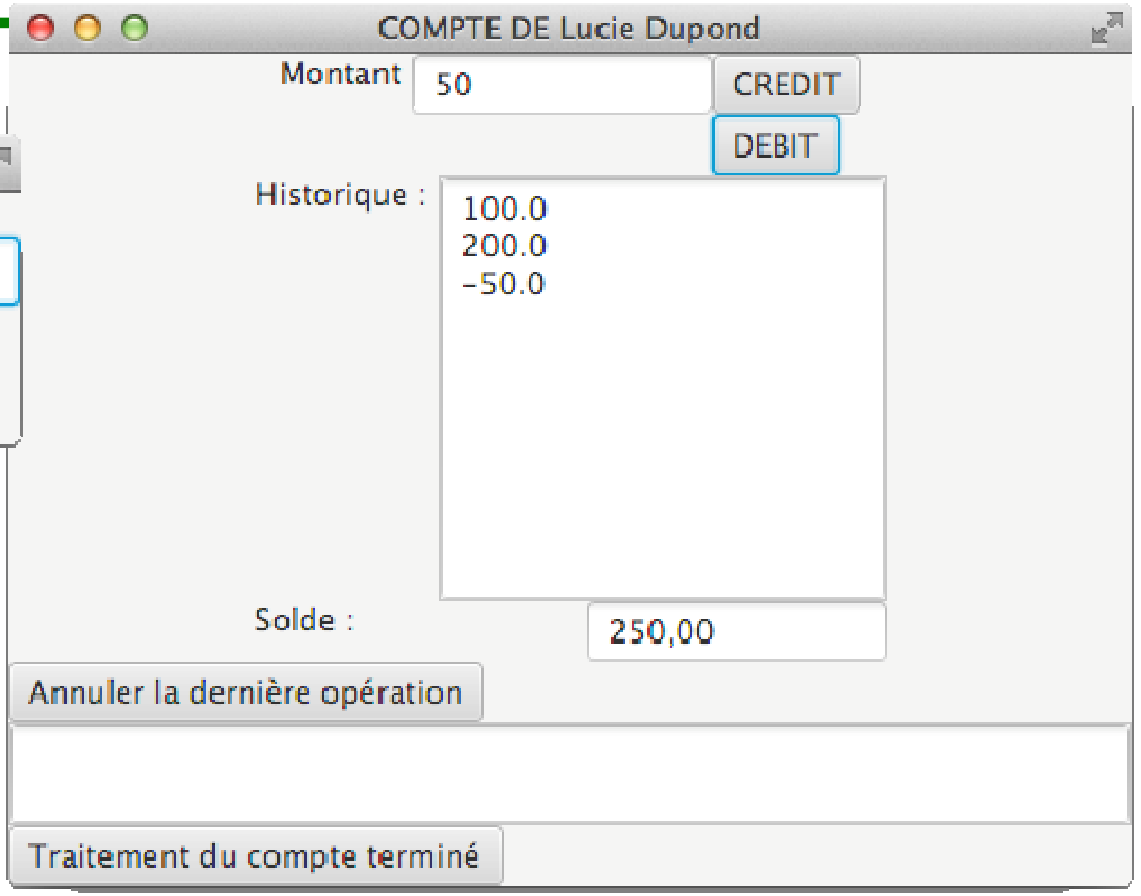
```
public void start(Stage stage) {

        stage.setTitle("Cliquer !");

        Compteur compteur = new Compteur( );
        Scene scene = new Scene(new PanelClickCB(compteur), 200, 50);
        stage.setScene(scene);

        stage.setOnCloseRequest( (e)->{
                Alert alert = new Alert(AlertType.CON
                alert.setTitle("Confirmation");
                alert.setContentText("Voulez-vous quitter ?");

                Optional<ButtonType> result = alert.showAndWait( );
                if (result.get( ) != ButtonType.OK)
                                        e.consume( );
                                    });
        stage.show();
        }
```

Nouveauté javafx 8.0.40

Un exemple complet



La fenêtre d'accueil
(PanelFamille)



La fenêtre de gestion du compte
(PanelCompte)

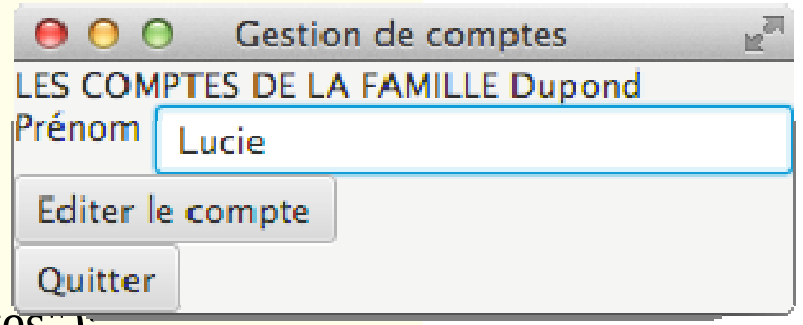
Les deux fenêtres de l'exemple

La classe CompteE

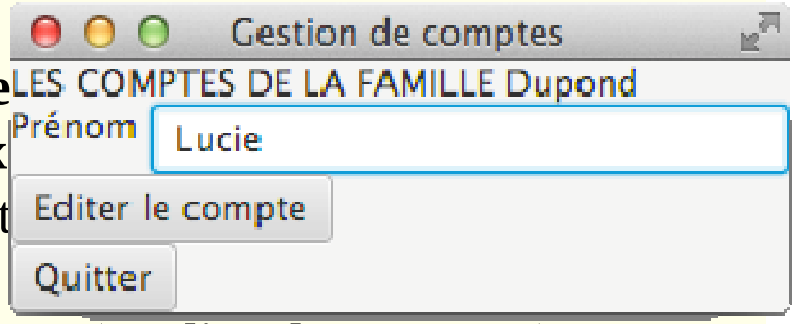
```
package up5.mi.pary.jc.compte;
public class CompteE {

public CompteE(String nom) throws CompteException{…}
public CompteE(String nom,double versementInitial,double decouvertAutorise)

public void addOperation(double montant) throws CompteException {…}

public double getSolde( ) {…}

public double getDecouvertAutorise ( ) {…}
public void setDecouvertAutorise(double decouvertAutorise) {…}
public String getNomTitulaire( ) {…}
public String getHistorique( ) {…}
public void annulerDerniereOperation( ) throws CompteVideException {…}

public static CompteE recuperer(String nom) {…}
public void sauvegarder( ) {…}
}
```

La fenêtre principale

```
public class AppliCompte extends Application{
        @Override
        public void start(Stage stage) {

                stage.setTitle("Gestion de comptes");

                Scene scene = new Scene(new PanelFamille("Dupond"));

                stage.setScene(scene);
                stage.show();
        }
        public static void main(String[] args) {
                launch(args);
        }

}
```



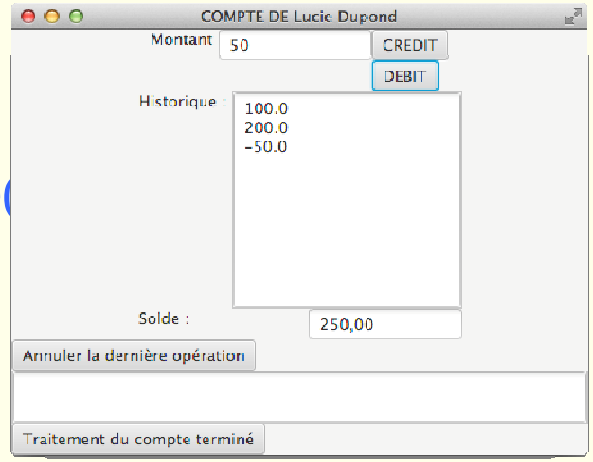
Le panel de la fenêtre principale




```
public class PanelFamille  extends GridPane {

        public PanelFamille(String nomFamille
                TextField textPrenom = new Tex
                textPrenom.setPrefColumnCount

                Button buttonCompte = new Button("Editer le compte");

                Button buttonQuitter = new Button("Quitter");

                BorderPane panelPrenom = new BorderPane( );
                panelPrenom.setLeft(new Label("Pr\u00E9nom "));
                panelPrenom.setRight(textPrenom);

                this.add(new Label(
                                        "LES COMPTES DE LA FAMILLE
"+nomFamille+"  "),
                                1,1);
                this.add(panelPrenom,1,2);
                this.add(buttonCompte,1,3);
                this.add(buttonQuitter,1,4);
```

Handler

```
buttonQuitter.setOnAction(new GestionnaireExit( ));

EventHandler<ActionEvent> editListener=
  (e)->{
        try{
              String nomTitulaire=textPrenom.getText( )+" "+ nomFamille;
              CompteE compte= getCompte(nomTitulaire);
              final Stage stage = new Stage();
              stage.setTitle("COMPTE DE "+compte.getNomTitulaire( ));
              Scene scene = new Scene(new PanelCompte(compte));
              stage.setScene(scene);
              stage.show();
              }
        catch (CompteException exp){exp.printStackTrace(
        };
buttonCompte.setOnAction(editListener);
textPrenom.setOnAction(editListener);
}
```



GestionnaireExit

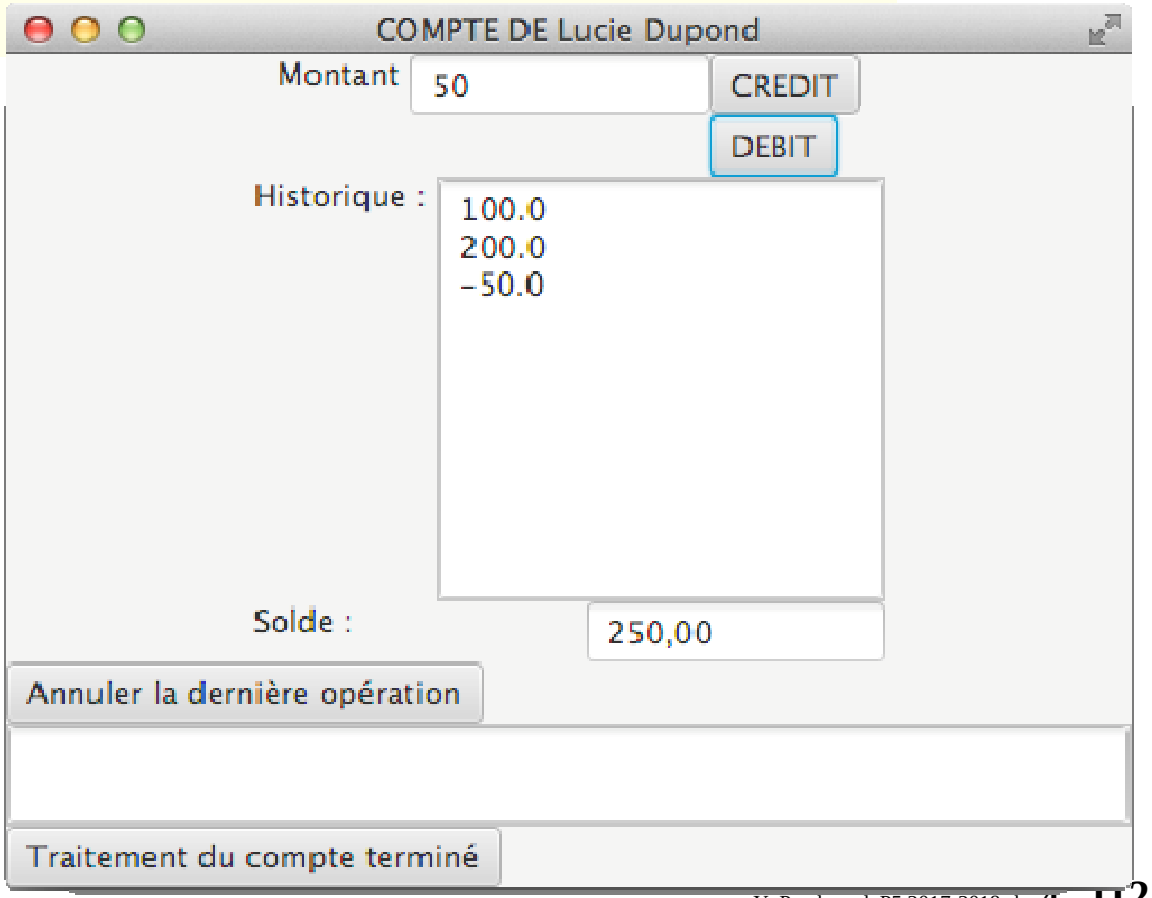
```
public class GestionnaireExit implements EventHandler<ActionEvent> {

	@Override
	public void handle(ActionEvent event) {
		System.exit(0);
	}

}
```

La méthode getCompte

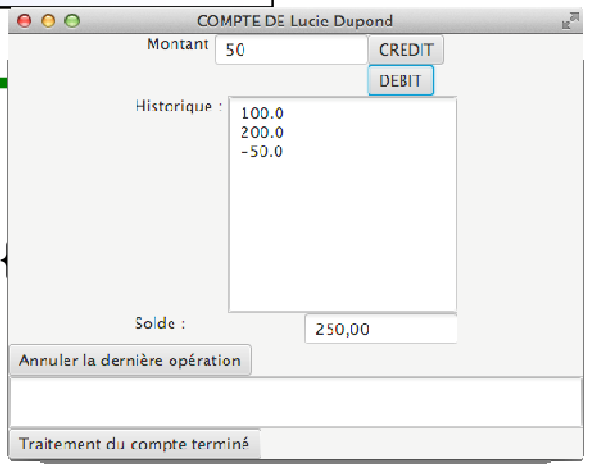
```
/**
 * rend le compte stocké sur fichier connaissant le nom de son titulaire
 * @param nomTitulaire
 * @return le compte correspondant à ce nom
 * @throws CompteException si une erreur empeche la creation du compte
 */
private CompteE getCompte(String nomTitulaire) throws CompteException{
                    CompteE compte;
                    try {compte= CompteE.recuperer(nomTitulaire);}
                    catch (Exception exp){
                              compte=new CompteE(nomTitulaire);
                    }
                    return compte;

          }
```

L'interface de la classe PanelCompte

```
public class PanelCompte extendsBorderPane {
        public PanelCompte(CompteE compte){…}
…
}
```



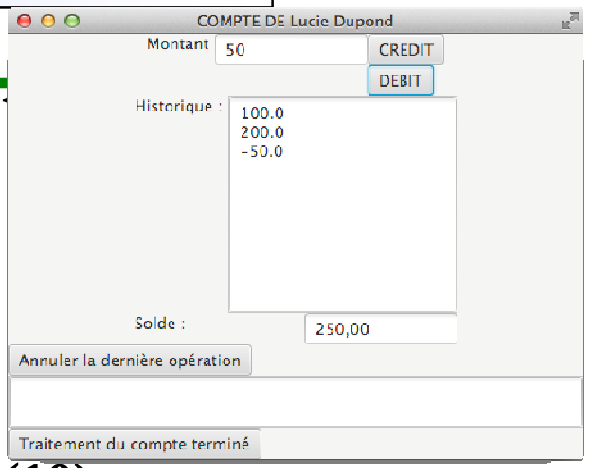
PanelCompte: les attributs



public class PanelCompte extends BorderPane {

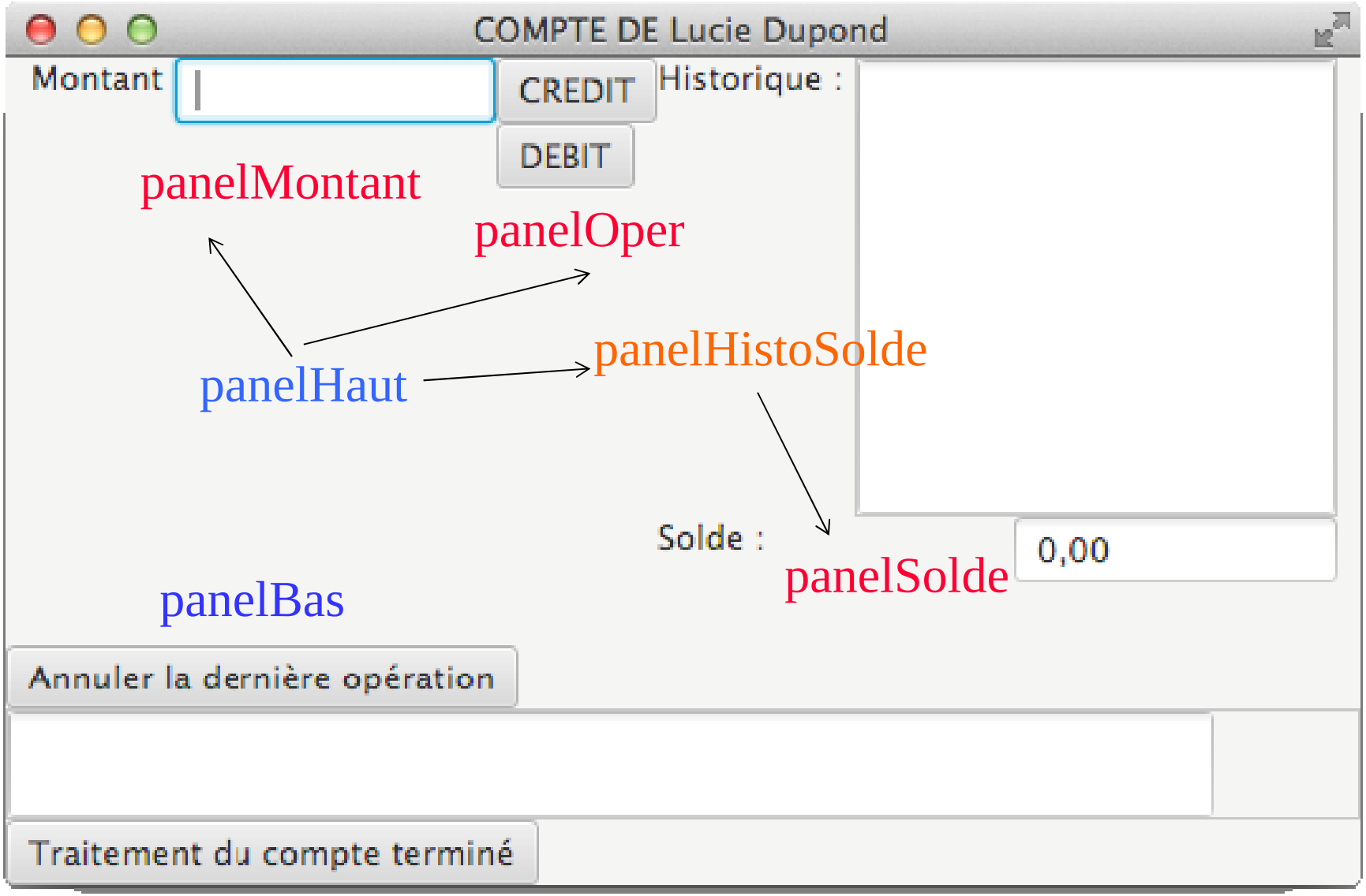
/** le compte affiché par ce panel*/
private CompteE compte;
/** la zone de texte contenant le montant des opérations*/
private TextField textMontant = new TextField();
/** la zone de texte contenant le solde du compte*/
private TextField textSolde = new TextField();
/** la zone de texte contenant l'historique des opérations*/
private TextArea textareaHisto = new TextArea();
/** la zone de texte contenant les messages d'erreurs*/
private TextArea messageErreur = new TextArea();

PanelCompte: les composants (1)

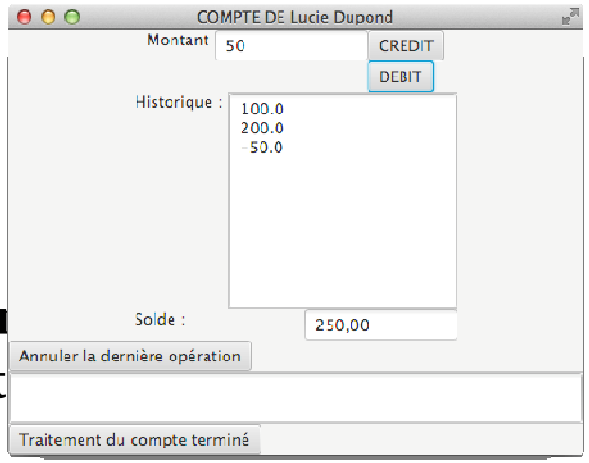



```
public class PanelCompte extends BorderPane {

……
public PanelCompte(CompteE compte){
        this.compte=compte;

        this.textMontant.setPrefColumnCount(10);
        this.textSolde.setPrefColumnCount(10);
        this.textareaHisto.setPrefColumnCount(14);
        this.textareaHisto.setPrefRowCount(10);
        this.messageErreur.setPrefRowCount(1);
        this.messageErreur.setEditable(false);

        Button buttonCredit = new Button("CREDIT");
        Button buttonDebit  = new Button("DEBIT");

        Button buttonAnnuler = new Button("Annuler la dernière opération

        Button buttonQuitter = new Button("Traitement du compte terminé");
```

Hierarchie des Panels

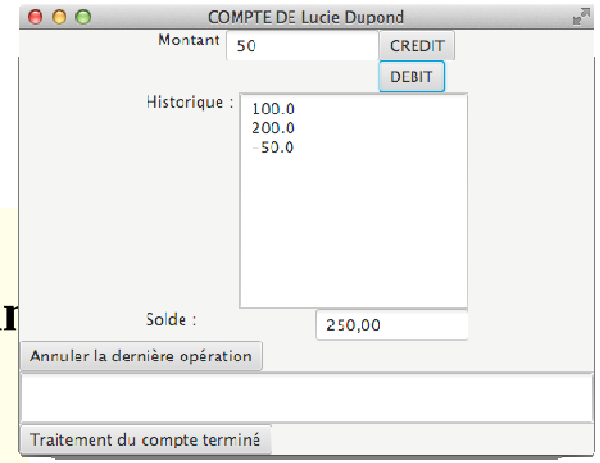


PanelCompte : panelMontant et panelOper



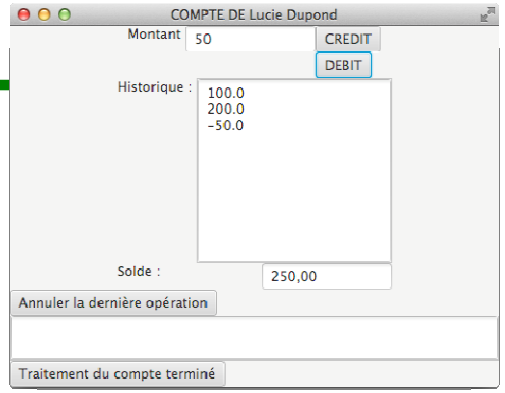

```
FlowPane panelHaut=new FlowPane();

panelHaut.setAlignment(Pos.CENTER);
BorderPane panelMontant = new BorderPa
panelMontant.setLeft(new Label("Montant
panelMontant.setRight(this.textMontant);

panelHaut.getChildren().add(panelMontant);


GridPane panelOper = new GridPane();
panelOper.add(buttonCredit,1,1);
panelOper.add(buttonDebit,1,2);

panelHaut.getChildren().add(panelOper);
```

PanelCompte : panelHistoSolde



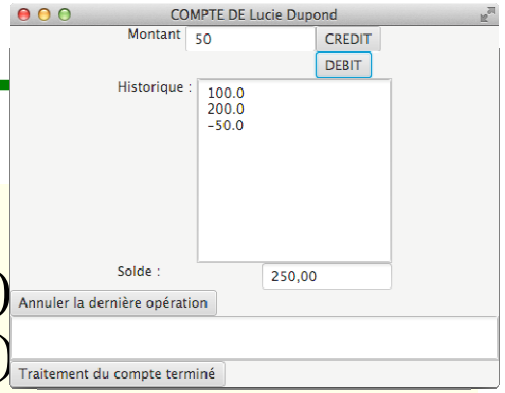

```
BorderPane panelHistoSolde = new BorderPar
BorderPane panelSolde = new BorderPane();
panelSolde.setLeft(new Label("Solde : "));
panelSolde.setRight(this.textSolde);
panelHistoSolde.setLeft(new Label("Historique : ") );
this.textareaHisto.setEditable(false);
ScrollPane jsp=new ScrollPane(this.textareaHisto);
panelHistoSolde.setRight(jsp);
panelHistoSolde.setBottom(panelSolde);

panelHaut.getChildren().add(panelHistoSolde);
```

PanelCompte : PanelBas




```
BorderPane panelBas = new BorderPane();
panelBas.setTop(buttonAnnuler);
panelBas.setCenter(new ScrollPane(this.messageErreur));
panelBas.setBottom(buttonQuitter);

this.setTop(panelHaut);
this.setBottom(panelBas);
```

PanelCompte : les handlers




```
buttonCredit.setOnAction((e)->addOperation(true)
buttonDebit.setOnAction((e)->addOperation(false)
buttonAnnuler.setOnAction((e)->annulerDerniereOperation());

buttonQuitter.setOnAction((event)->{
        try {
                compte.sauvegarder( );
                PanelCompte.this.getScene().getWindow().hide();
        }
        catch (Exception exp){
                messageErreur.setText("Problème de sauvegarde :"+exp.g
                exp.printStackTrace();}});


miseAJourInfo( );
```

PanelCompte : fonctions appelées par les handlers

```
private void addOperation(boolean credit){
        try {
                double m = Double.valueOf(this.textMontant.getText( )).d
                this.compte.addOperation(credit ? m : -m);
                this.miseAJourInfo( );
        }
        catch (DecouvertException e) {
                this.messageErreur.setText("Operation ignoree "+e);}
        catch (MontantNulException e){this.messageErreur.setText(e.getMessa
        catch (CompteException e){this.messageErreur.setText(""+e);}
}
private void annulerDerniereOperation(){
        try {
                this.compte.annulerDerniereOperation( );
                this.miseAJourInfo( );
        }
        catch (CompteException e) {this.messageErreur.setText(""+e);}
}
```

PanelCompte : miseAJourInfo

```
private void miseAJourInfo( ){

                this.textareaHisto.setText(this.compte.getHistorique( ));

                java.text.DecimalFormat df = new java.text.DecimalFormat("0.00");
                this.textSolde.setText(df.format(this.compte.getSolde( )));

                this.messageErreur.setText("");
        }

}
```

FXML

Description de la séquence

Description du contenu : FXML et JavaFX

Durée de travail estimée : 60 minutes

Auteur : Yannick.Parchemal@parisdescartes.fr
Objectifs : comprendre le principe d'utilisation de FXML pour les applications JavaFX

Connaissances acquises : structuration d'un fichier FXML basique. Notion de vue et de contrôleur
Compétences acquises : savoir définir des interfaces graphiques avec fxml et avec ou sans utilisation de SceneBuilder
Pré requis :

JavaFX et FXML

Décrire la vue en XML (FXML)

<bottom> <Button *text="Ok"*/></*bottom*>

Relier application et vue dans la méthode start

FXMLLoader loader = **new FXMLLoader(** *lien vers le fichier xml***);**
BorderPane pane = loader.load( );

Regrouper les handlers dans une classe (controleur)

```
button.setOnAction((event)->{
                        compteur.incrementer();
                        label.setText("--> "+compteur.getValue( ));
        });
```

FXML : la vue

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

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

<BorderPane xmlns="http://javafx.com/javafx/8.0.40" xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="application.ClickController">

    <bottom> <Button fx:id="button" text="Ok"/></bottom>

    <center> <Label fx:id="label" text="-->0"/> </center>

</BorderPane>
```

Lien entre application javafx et fichier fxml

```
public void start(Stage primaryStage) throws IOException{

FXMLLoader loader = new FXMLLoader(getClass( ).getResource("../compteur.fxml"));

BorderPane pane = loader.load( );

Scene scene = new Scene(pane,350,70);

primaryStage.setTitle("Compteur de click");
primaryStage.setScene(scene);
primaryStage.show( );

}
```

FXML : la vue

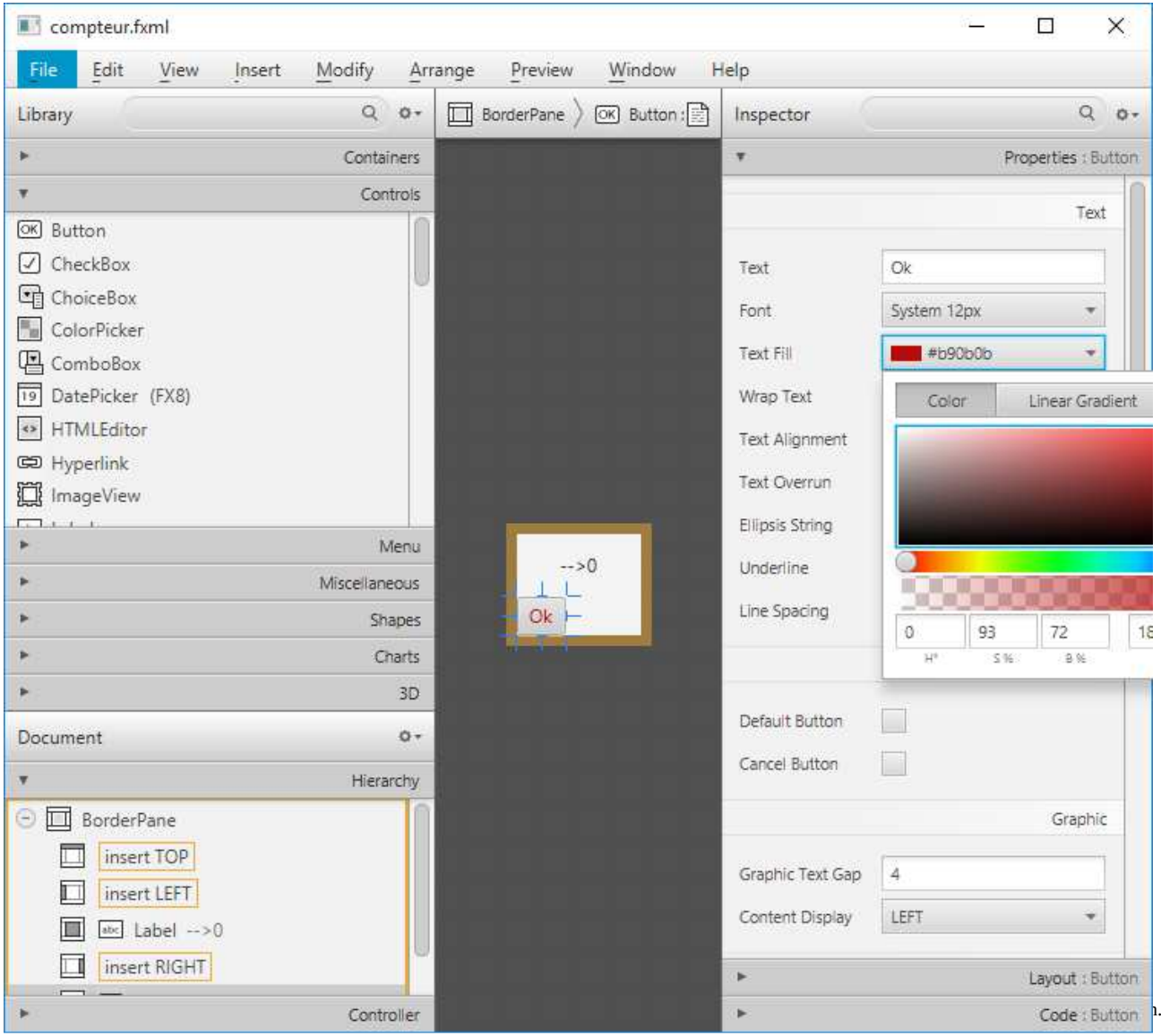
```
<?xml version="1.0" encoding="UTF-8"?>

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

<BorderPane xmlns="http://javafx.com/javafx/8.0.40" xmlns:fx="http://javafx.com/fxml/1"
            fx:controller="application.ClickController">

    <bottom> <Button fx:id="button" text="Ok"/></bottom>

    <center> <Label fx:id="label" text="-->0"/> </center>

</BorderPane>
```

Le contrôleur

```
public class ClickController implements Initializable{

@FXML
private Button button;
@FXML
private Label label;

private Compteur compteur = new Compteur();

@Override
public void initialize(URL location, ResourceBundle resources) {
                    button.setOnAction((event)->{
                                compteur.incrementer();
                                label.setText("--> "+compteur.getValue( ));
            });
   }
}
```

SceneBuilder : pour générer du FXML

http://gluonhq.com/labs/scene-builder/



SceneBuilder et efxclipse

Dernière version de SceneBuilder (9.0.1 10/2017)

http://gluonhq.com/labs/scene-builder/

Plugin e(fx)clipse pour eclipse

http://www.eclipse.org/efxclipse/install.html