Java FX

Denis MADEC
dma@magic.fr

v1.10

Plan

- Présentation de JavaFX
- Premiers pas avec JavaFX
- Composants graphiques
- Gestion d'évènements
- Propriétés et beans Java FX
- Collections JavaFX
- Data binding
- Composants graphiques évolués: TableView,TreeView,TreeTableView
- Cascading Style Sheet (CSS)
- Multi-threading et concurrence
- Interopérabilité avec Swing
- Contenu HTML
- Développer en FXML
- Composants personnalisés
- Création de diagrammes
- Graphismes 2D et 3D
- Animations, effets visuels, médias
- Déploiement d'applications JavaFX

Présentation de JavaFX

Présentation de JavaFX

- qu'est-ce que JavaFX ?
- caractéristiques
- historique
- outils de développement
- principales solutions concurrentes

Qu'est-ce que JavaFX ?

- JavaFX est un jeu d'API qui permet de développer des clients riches et de les déployer sur différentes plateformes
  - le look & feel est personnalisable via CSS
  - le développement peut être réalisé en Java ou en FXML
  - l'outil SceneBuilder d'Oracle permet de générer du FXML par glisser/déposer
- JavaFX est utile pour développer:
  - des applications standalone
  - des RDA (Rich Desktop Applications)
    - via Java Web Start
  - des RIA (Rich Internet Applications)
    - via les applets
- JavaFX était intégré au JDK 1.8 et JRE 1.8

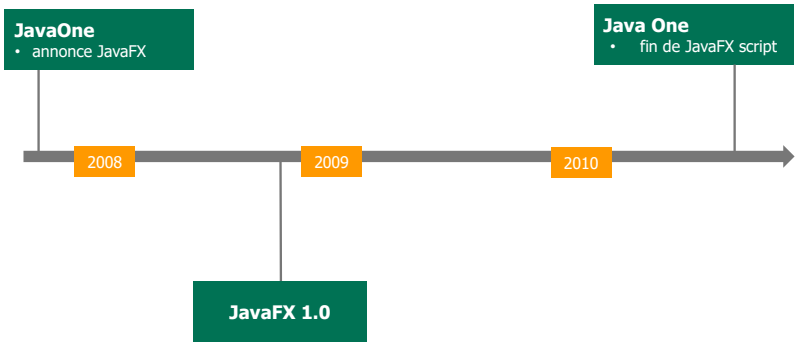
JavaFX 11

- JavaFX est désormais un module séparé du JDK depuis Java 11
  - il convient alors de télécharger le JavaFX SDK
- le composant WebView de JavaFX est désormais capable de visualiser des formules mathématiques écrites en MathML
- Robot est une nouvelle API permettant de simuler les interactions avec l'utilisateur (clavier et souris)

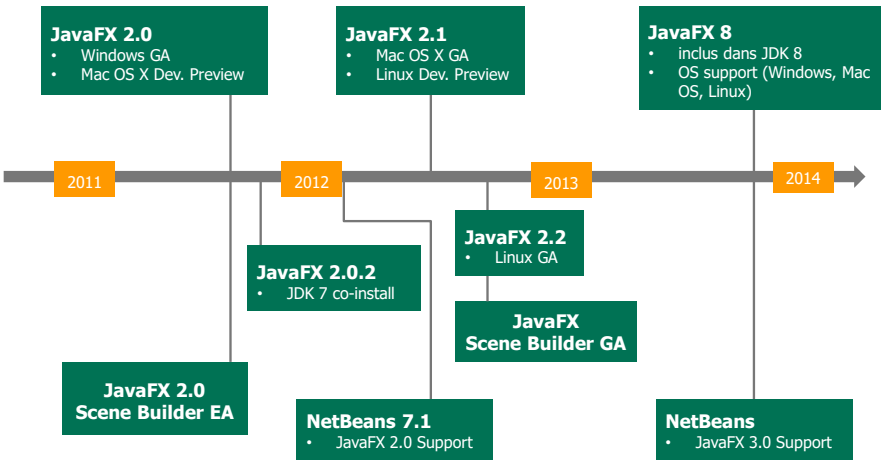
Caractéristiques

- theme Modena par défaut
- FXML et SceneBuilder
- WebView
- interopérabilité avec Swing
- API Canvas
- API d'impression
- fonctionnalités 3D
- support sensitif
- moteur multimedia à haute performance
- rendu graphique accéléré par hardware

Historique



Historique



Outils de développement

- NetBeans
  - IDE Open Source, intègre les menus qui facilitent le développement d'applications JavaFX
  - permet d'intégrer l'outil de développement SceneBuilder d'Oracle
- E(fx)clipse
  - IDE Open Source, il s'agit d'Eclipse avec plug-in pour JavaFX
  - comme NetBeans, comporte des menus qui facilitent le développement d'applications JavaFX et permet d'intégrer SceneBuilder
- Intellij
  - outil commercial, comportant des fonctionnalités proches de NetBeans et Eclipse

Principales solutions concurrentes

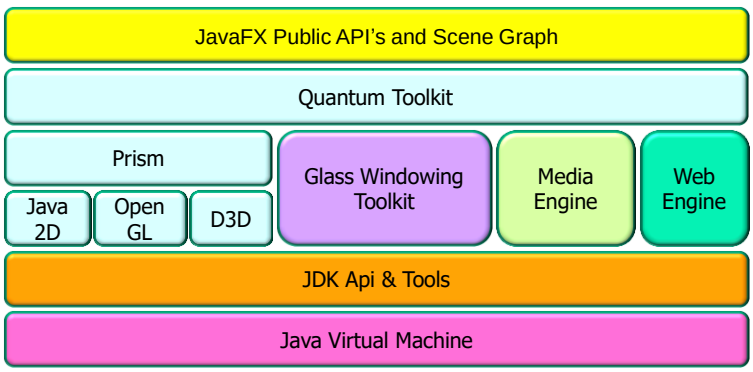
- GWT (Google)
  - les applications sont développées en Java et utilisent des API Google
  - le code est compilé pour générer du code JavaScript
- JSF Java Server Faces (Oracle)
  - framework java EE orienté web
- Angular JS (Google)
  - repose sur JavaScript
  - s'exécute directement dans tout navigateur
- React.js (FaceBook)
- Ember.js

Premiers pas avec JavaFX

Premiers pas avec JavaFX

- architecture
- classe Application
- exemple

Architecture



Architecture

- JavaFX public API
  - permettent l'utilisation d'autres langages comme Groovy, JavaScript
- Scene Graph
  - arborescence de noeuds qui représentent les éléments visuels de l'application
  - chaque noeud possède un style, une taille
- Quantum Toolkit
  - associe Prism avec le Glass Windowing Toolkit
  - gère les règles de multi-threading liées à la gestion d'évènements

Architecture

- Prism
  - effectue le rendu graphique, en s'appuyant sur le hardware ou le software
    - Direct 9 sur Windows XP et Vista
    - Direct 11 sur Windows 7 & 8
    - OpenGL sur Mac, Linux, Java Embedded
- Glass Windowing Toolkit
  - fournit les services de bas niveau de la couche graphique: gestion des fenêtres, timers, file d'attente des évènements

Architecture

- Media engine
  - cet élément assure le support de média audio et vidéo
    - fichiers audio MP3, AIFF, et WAV
    - fichiers vidéo FLV
  - trois composants distincts assurent cette fonctionnalité:
    - Media object: représente un fichier audio ou vidéo
    - MediaPlayer: diffuse le fichier audio ou vidéo
    - MediaView: noeud qui affiche le média
- Web engine
  - cet élément est un contrôle graphique JavaFX, et constitue un navigateur web pour le support de HTML5, CSS, JavaScipt, DOM, SVG

classe Application

- la classe principale d'une application JavaFX hérite de `javafx.application.`**`Application`**
  - sa méthode **`start()`** est le point d'entrée principal de toute application JavaFX
  - après lancement de l'application via la méthode **`launch()`**, l'environnement d'exécution javaFX:
    - crée une instance de la classe courante
    - appelle la méthode **`init()`**
    - apelle la méthode **`start(Stage)`**
    - attends la fin de l'application
    - appelle la méthode **`stop()`**

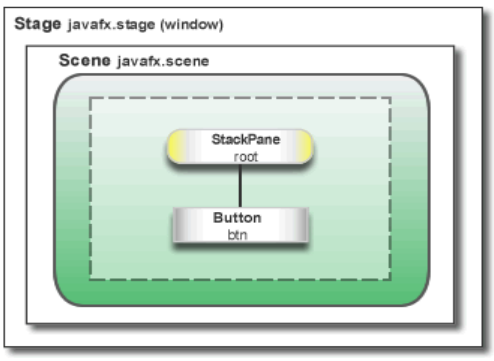
classe Application

- `javafx.application.`**`Application`**

HostServices	getHostServices() Gets the HostServices provider for this application.
Application.Parameters	getParameters() Retrieves the parameters for this Application, including any arguments passed on the command line and any parameters specified in a JNLP file for an applet or WebStart application.
static java.lang.String	getUserAgentStylesheet() Get the user agent stylesheet used by the whole application.
void	init() The application initialization method.
static void	launch(java.lang.Class<? extends Application> appClass, java.lang.String... args) Launch a standalone application.
static void	launch(java.lang.String... args) Launch a standalone application.
void	notifyPreloader(Preloader.PreloaderNotification info) Notifies the preloader with an application-generated notification.
static void	setUserAgentStylesheet(java.lang.String url) Set the user agent stylesheet used by the whole application.
abstract void	start(Stage primaryStage) The main entry point for all JavaFX applications.
void	stop() This method is called when the application should stop, and provides a convenient place to prepare for application exit and destroy resources.

Exemple

- exemple
  - il s'agit de créer une application JavaFX comportant un simple bouton



Exemple

- l'objet `javafx.stage.`**`Stage`** transmis par l'environnement à la méthode **`start()`** représente la fenêtre principale de l'application JavaFX
- l'objet `javafx.scene.`**`Scene`** transmis par l'environnement à la méthode **`start()`** représente le conteneur de l'arborescence des composants graphiques
- l'objet `javafx.scene.layout.`**`StackPane`** est un gestionnaire de mise en forme chargé de la disposition des composants graphiques
- l'objet `javafx.scene.control.`**`Button`** est un bouton

Exemple

- exemple (suite)

```
package dma;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class Bonjour extends Application {
   public static void main(String[] args) {
      launch(args);
   }
```

Exemple

- exemple (suite)

```
    @Override
    public void start(Stage primaryStage) {
        StackPane root = new StackPane();
        Button btn = new Button();
        btn.setText("Bonjour");
        root.getChildren().add(btn);
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("Bonjour!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Composants graphiques

Composants graphiques

- Stage
- Scene
- arborescence graphique (Scene graph)
- Layouts
- objets graphiques

Stage

- l'interface graphique de JavaFX est défini au moyen des objets **`Stage`** et **`Scene`**
  - **`Stage`** est la fenêtre principale de l'application
    - la première instance de **`Stage`** est créée par la plateforme
      - d'autres instances peuvent être créées par l'application

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

      - les instances de **`Stage`** doivent être créées et modifiées dans le thread Application JavaFX

Scene

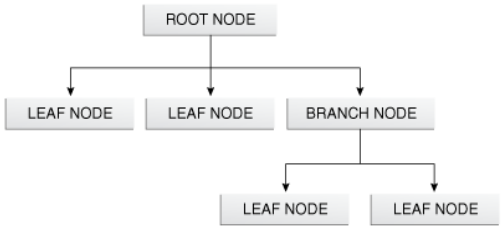
- **`Scene`** contient l'arborescence des objets graphiques

```
@Override
public void start(Stage primaryStage) {
    ...
    StackPane root = new StackPane();
    root.getChildren().add(btn);
    Scene scene = new Scene(root, 300, 250);
    ...
}
```

  - dans l'exemple, le noeud racine (*root node*) de l'arborescence graphique est un objet **`StackPane`** accroché à l'objet **`Scene`** via son constructeur

Arborescence graphique

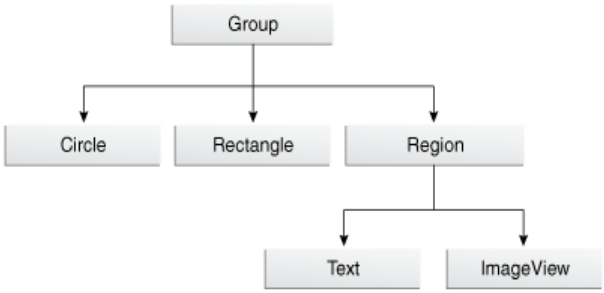
- l'arborescence graphique (*scene graph)* représente tous les objets graphiques de l'application, accrochés à l'objet **`Scene`**



  - cette arborescence peut être créée et modifiée dans tout thread tant qu'elle n'est pas accrochée à l'objet **Scene**
  - sinon, cela doit être effectué dans le thread JavaFX application

Arborescence graphique

- les API JavaFX définissent les classes dont les instances peuvent devenir *root node*, *branch node* ou *leaf node*



Arborescence graphique

- exemple

```
package dma;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
public class Main extends Application {
    @Override public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        primaryStage.setTitle("JavaFX Scene Graph Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

le seul élément de l'arborescence est root

Arborescence graphique

- exemple

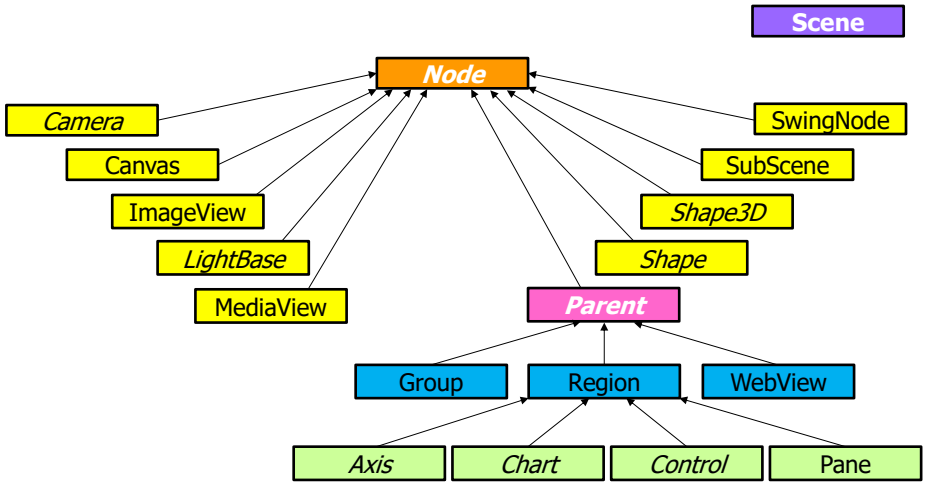
```
package dma;
import javafx.application.Application;
...
public class Main extends Application {
    @Override public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 500, Color.BLACK);
        Rectangle rect = new Rectangle(25,25,250,250);
        rect.setFill(Color.BLUE);
        root.getChildren().add(rect);
        stage.setTitle("JavaFX Scene Graph Demo");
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

le rectangle rect est fils de root

Arborescence graphique

- le package **`javafx.scene`** définit plus d'une douzaine de classes, parmi lesquelles:
  - **`Scene`**
    - conteneur pour tous les éléments de l'arborescence graphique
  - **`Node`**
    - classe abstraite de base de tous les noeuds de l'arborescence graphique
  - **`Parent`**
    - classe abstraite de base de tous les noeuds parents

Arborescence graphique



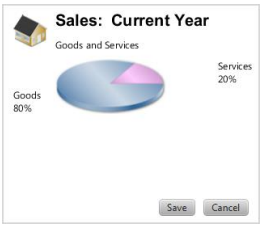
Layouts

- les layouts sont des conteneurs qui gèrent la disposition des objets graphiques qui sont incorporés dans les conteneurs
  - font partie du package **`javafx.scene.layout`**
  - héritent de `javafx.scene.layout.`**`Pane`**
  - **`AnchorPane, BorderPane, FlowPane, GridPane, StackPane, TilePane, HBox, VBox`**

Layouts: AnchorPane

- le layout **`AnchorPane`** permet d'ancrer les noeuds en haut, en bas, à gauche, à droite ou au centre d'un conteneur

```
public AnchorPane addAnchorPane(GridPane grid) {
    AnchorPane anchorpane = new AnchorPane();
    Button buttonSave = new Button("Save");
    Button buttonCancel = new Button("Cancel");
    HBox hb = new HBox();
    hb.setPadding(new Insets(0, 10, 10, 10));
    hb.setSpacing(10);
    hb.getChildren().addAll(buttonSave,
    buttonCancel);
    anchorpane.getChildren().addAll(grid,hb);
    AnchorPane.setBottomAnchor(hb, 8.0);
    AnchorPane.setRightAnchor(hb, 5.0);
    AnchorPane.setTopAnchor(grid, 10.0);
    return anchorpane;
}
```



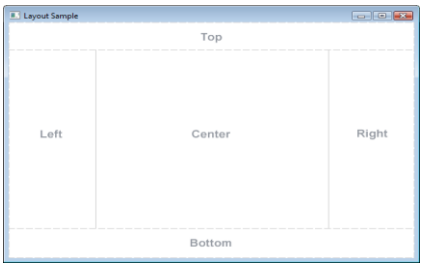
Layouts: AnchorPane

- le redimensionnement du conteneur conserve l'ancrage
- il est possible de préciser plusieurs positions pour ancrer un même nœud
- plusieurs nœuds peuvent être ancrés à une position donnée

Layouts: BorderPane

- le layout **`BorderPane`** fournit 5 zones pour y placer les différents noeuds graphiques:

```
BorderPane border =
        new BorderPane();
HBox hbox = addHBox();
border.setTop(hbox);
border.setLeft(addVBox());
addStackPane(hbox);
border.setCenter(addGridPane());
border.setRight(addFlowPane());
```



  - `Top, Bottom, Left, Right, Center`

Layouts: BorderPane

- si l'application n'utilise pas l'une des régions, aucun espace ne lui est alloué
- si la fenêtre est plus grande que l'espace nécessaire pour visualiser le contenu de chacune des régions, l'espace supplémentaire est attribué par défaut à la région centrale
- si la fenêtre est plus petite que l'espace nécessaire pour visualiser le contenu de chacune des régions, celles-ci peuvent se chevaucher

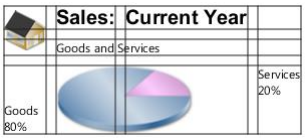
Layouts: FlowPane

- le layout **`FlowPane`** place tous les noeuds en ligne puis en colonne en fonction de la place disponible

```
public FlowPane addFlowPane() {
    FlowPane flow = new FlowPane();
    flow.setPadding(new Insets(5, 0, 5, 0));
    flow.setVgap(4); flow.setHgap(4);
    flow.setPrefWrapLength(170);
    flow.setStyle("-fx-background-color: DAE6F3;");
    ImageView pages[] = new ImageView[8];
    for (int i=0; i<8; i++) {
        pages[i] = new ImageView(new Image(
          LayoutSample.class.getResourceAsStream(
                "graphics/chart_"+(i+1)+".png")));
        flow.getChildren().add(pages[i]);
    }
    return flow;
}
```

Layouts: GridPane

- le layout **`GridPane`** place tous les noeuds en grille




```
public GridPane addGridPane() {
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets(0, 10, 0, 10));
    Text category = new Text("Sales:");
    category.setFont(Font.font("Arial", FontWeight.BOLD, 20));
    grid.add(category, 1, 0);
    Text chartTitle = new Text("Current Year");
    chartTitle.setFont(Font.font("Arial", FontWeight.BOLD, 20));
    grid.add(chartTitle, 2, 0);
    Text chartSubtitle = new Text("Goods and Services");
    grid.add(chartSubtitle, 1, 1, 2, 1);
```

Layouts: GridPane

- (suite)

```
        ImageView imageHouse = new ImageView(new Image(
            LayoutSample.class.getResourceAsStream(
                    "graphics/house.png")));
        grid.add(imageHouse, 0, 0, 1, 2);
        Text goodsPercent = new Text("Goods\n80%");
        Text goodsPercent = new Text("Goods\n80%");
        GridPane.setValignment(goodsPercent, VPos.BOTTOM);
        grid.add(goodsPercent, 0, 2);
        ImageView imageChart = new ImageView(
            new Image(LayoutSample.class.getResourceAsStream(
                    "graphics/piechart.png")));
        grid.add(imageChart, 1, 2, 2, 1);
        servicesPercent = new Text("Services\n20%");
        GridPane.setValignment(servicesPercent, VPos.TOP);
        grid.add(servicesPercent, 3, 2);
        return grid;
    }
```

Layouts: StackPane

- le layout **`StackPane`** place tous les noeuds en pile, chaque noeud cachant le précédent

```
public void addStackPane(HBox hb) {
    StackPane stack = new StackPane();
    Rectangle helpIcon = new Rectangle(30.0, 25.0);
    helpIcon.setFill(
        new LinearGradient(0,0,0,1, true,
    CycleMethod.NO_CYCLE,
        new Stop[]{ new Stop(0,Color.web("#4977A3")),
                new Stop(0.5, Color.web("#B0C6DA")),
                newStop(1,Color.web("#9CB6CF")),}));
    helpIcon.setStroke(Color.web("#D0E6FA"));
    helpIcon.setArcHeight(3.5);
    helpIcon.setArcWidth(3.5);
```

Layouts: StackPane

- (suite)

```
        Text helpText = new Text("?");
        helpText.setFont(Font.font("Verdana", FontWeight.BOLD,
        18));
        helpText.setFill(Color.WHITE);
        helpText.setStroke(Color.web("#7080A0"));
        stack.getChildren().addAll(helpIcon, helpText);
        stack.setAlignment(Pos.CENTER_RIGHT);
        StackPane.setMargin(helpText, new Insets(0,10,0,0));
        hb.getChildren().add(stack);
        HBox.setHgrow(stack, Priority.ALWAYS);
    }
```

Layouts: TilePane

- le layout **`TilePane`** est similaire au layout **`FlowPane`** mais les noeuds sont tous de mêmes dimensions

```
        TilePane tile = new TilePane();
        tile.setPadding(new Insets(5, 0, 5, 0));
        tile.setVgap(4);
        tile.setHgap(4);
        tile.setPrefColumns(2);
        tile.setStyle("-fx-background-color: DAE6F3;");
        ImageView pages[] = new ImageView[8];
        for (int i=0; i<8; i++) {
        pages[i] = new ImageView( new
            Image(LayoutSample.class.getResourceAsStream
        (
            "graphics/chart_"+(i+1)+".png")));
            tile.getChildren().add(pages[i]);
        }
```

Layouts: HBox

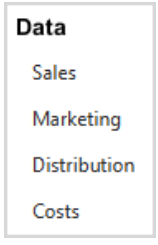
- le layout **`HBox`** permet d'aligner en ligne les différents noeuds graphiques:


```
public HBox addHBox() {
    HBox hbox = new HBox();
    hbox.setPadding(new Insets(15, 12, 15, 12));
    hbox.setSpacing(10);
    hbox.setStyle("-fx-background-color: #336699;");
    Button buttonCurrent = new Button("Current");
    buttonCurrent.setPrefSize(100, 20);
    Button buttonProjected = new Button("Projected");
    buttonProjected.setPrefSize(100, 20);
    hbox.getChildren().addAll(buttonCurrent, buttonProjected);
    return hbox;
}
```

Layouts: VBox

- le layout **`VBox`** permet d'aligner en ligne les différents noeuds graphiques:

```
public VBox addVBox() {
     VBox vbox = new VBox();
     vbox.setPadding(new Insets(10));
     vbox.setSpacing(8);
     Text title = new Text("Data");
     title.setFont(Font.font("Arial",FontWeight.BOLD,14));
     vbox.getChildren().add(title);
     Hyperlink options[] = new Hyperlink[] {
     new Hyperlink("Sales"),new Hyperlink("Marketing"),
     new Hyperlink("Distribution"),new Hyperlink("Costs")};
     for (int i=0; i<4; i++) {
        VBox.setMargin(options[i], new Insets(0,0,0,8));
        vbox.getChildren().add(options[i]);
     }
     return vbox;
}
```



Objets graphiques

- Java FX comporte de nombreux objets graphiques héritant la plupart de **`Node`** et classés par packages:
- **`javafx.scene.image`**
  - contient les classes **`Image`** et **`ImageView`** permettant la visualisation d'images dans **`Scene`**
- **`javafx.scene.shape`**
  - contient des classes permettant de dessiner des formes comme des cercles, rectangles, polygones, arcs, etc... La classe de base est **`Shape`**

Objets graphiques

- **`javafx.scene.text`**
  - contient la classe **`Text`** permettant de dessiner du texte en spécifiant la police (classe **`Font`**)
- **`javafx.scene.media`**
  - contient des classes permet de diffuser du son ou des vidéos
- **`javafx.scene.chart`**
  - contient des classes permet d'afficher différents types de graphiques (camembert, histogrammes, courbes, etc...)

Objets graphiques

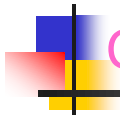
- **`javafx.scene.control`**
  - contient des classes des contrôles UI (boutons, champs de texte, menus, etc...), paramétrables via des feuilles de style CSS
- **`javafx.scene.transform`**
  - permet d'effectuer des transformations sur les noeuds (rotations, translations, changement d'échelle)
- **`javafx.scene.input`**
  - contient des classes comme **`MouseEvent`** ou **`KeyEvent`**

Objets graphiques

- **`javafx.scene.layout`**
  - contient des classes des gestionnaires Layout: **`BorderPane`**, **`FlowPane`**, **`HBox`**, **`VBox`**, etc...
- **`javafx.scene.effect`**
  - contient des effets comme **`Reflection`**, **`Glow`**, **`Shadow`**, **`Lighting`**
- **`javafx.scene.web`**
  - contient notamment les classes **`WebView`** et **`WebEngine`** permettant d'inclure un navigateur dans l'application JavaFX

Objets graphiques

- **`javafx.animation`**
  - contient des classes dédiées aux animations
- **`javafx.embed.swing`**
  - contient des classes permettant d'include du code JavaFX dans une application Swing
- **`javafx.embed.swt`**
  - contient des classes permettant d'include du code JavaFX dans une application SWT



Gestion d'évènements

Gestion d'évènements

- principe
- évènements
- gestion d'évènements
- expressions Lambda

Principe

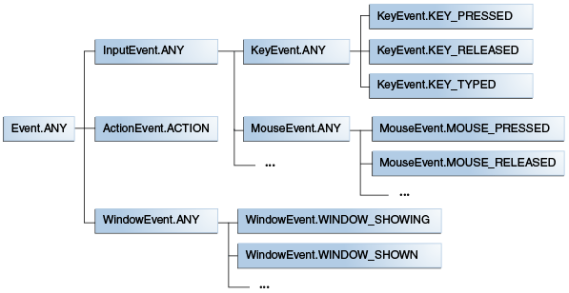
- gérer un évènement fait intervenir trois objets:
  - la source de l'évènement
    - ce peut être un bouton, un champ de texte, ...
  - l'évènement lui-même
    - dépend de la source, instance d'une sous-classe de **`javafx.event.Event`**
  - le gestionnaire de l'évènement
    - lié au traitement à effectuer lorsque l'évènement se produit: à développer selon les besoins

Evènements

- un type d'évènement est une instance de la classe **`EventType`**
- les types d'évènement classent ensuite les évènements d'une classe d'évènements
  - exemple:
    - la classe **`KeyEvent`** contient les types d'évènements suivants:
      - **`KEY_PRESSED`**
      - **`KEY_RELEASED`**
      - **`KEY_TYPED`**

Evènements

- les types d'évènements forment une hiérarchie



  - chaque type d'evenement possede un nom et un super-type:
    - par exemple, le nom de l'évènement qui correspond à une touche appuyée **`KEY_PRESSED`**, et son super-type **`KeyEvent.ANY`**

Gestion d'évènements

- un gestionnaire d'évènements doit implémenter l'interface `javafx.event.`**`EventHandler`**

```
@FunctionalInterface
public interface EventHandler<T extends Event>
              extends java.util.EventListener {
       void handle(T event);
}
```

  - il doit être enregistré auprès de l'objet source

```
public final <T extends Event>
   void addEventHandler(EventType<T> eventType,
      EventHandler<? super T> eventHandler);
```

    - il peut ensuite être enlevé avec **`removeEventHandler`**

Gestion d'évènements

- exemple

```
// enregistrement et définition d'un gestionnaire
node.addEventHandler(DragEvent.DRAG_ENTERED,
       new EventHandler<DragEvent>() {
              public void handle(DragEvent) { ... };
});
// définition d'un handler
EventHandler handler = new EventHandler<InputEvent>() {
        public void handle(InputEvent event) {
            System.out.println("event: "+event.getEventType());
            event.consume();
        }
};
// enregistrement du handler sur deux noeuds distincts
myNode1.addEventHandler(DragEvent.DRAG_EXITED, handler);
myNode2.addEventHandler(DragEvent.DRAG_EXITED, handler);
// enregistrement du handler pour un autre type d'évènement
myNode1.addEventHandler(MouseEvent.MOUSE_DRAGGED, handler);
```

Gestion d'évènements

- JavaFX fournit par commodité des propriétés simplifiant la mise en oeuvre de gestionnaires d'évènements
  - elles permettent de préciser une fonction qui sert de gestionnaire d'évènement pour une action donnée
  - la plupart de ces propriétés sont définies dans la classe **`Node`** (environ 40 !), les autres le sont dans ses sous-classes
    - exemple:

```
public final
ObjectProperty<EventHandler<? super MouseDragEvent>>
    onMouseDragOverProperty
```

Gestion d'évènements

- les méthodes **`set`** associées à ces propriétés ont le format suivant:

```
void setOnEvent-type(
        EventHandler<? super Event-class> value)
```

  - *Event-type* est le type d'évènement que doit gérer le gestionnaire
  - *Event-class* est la classe qui définit le type d'évènement
    - exemple

```
public final void setOnMouseDragOver(
    EventHandler<? super MouseDragEvent> value)
```

Gestion d'évènements

- exemple:

```
final Circle circle = new Circle(radius, Color.RED);
circle.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        ...
    }
});
circle.setOnMouseExited(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        ...
    }
});
circle.setOnMousePressed(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent me) {
        ...
    }
});
```

Gestion d'évènements

- exemple:

```
package login;
import javafx.application.Application;
...
public class Login extends Application {

  @Override public void start(Stage primaryStage) {
    primaryStage.setTitle("Login");

    GridPane grid = new GridPane();
    grid.setAlignment(Pos.CENTER);
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets(25, 25, 25, 25));
    Text scenetitle = new Text("Bienvenue");
    scenetitle.setFont(Font.font("Tahoma",FontWeight.NORMAL,20));
    grid.add(scenetitle, 0, 0, 2, 1);
```

Gestion d'évènements

- exemple (suite):

```
Label userName = new Label("identifiant:");
grid.add(userName, 0, 1);
TextField userTextField = new TextField();
grid.add(userTextField, 1, 1);
Label pw = new Label("mot de passe:");
grid.add(pw, 0, 2);
PasswordField pwBox = new PasswordField();
grid.add(pwBox, 1, 2);

Button btn = new Button("Valider");
HBox hbBtn = new HBox(10);
hbBtn.setAlignment(Pos.BOTTOM_RIGHT);
hbBtn.getChildren().add(btn);
grid.add(hbBtn, 1, 4);

final Text actiontarget = new Text();
grid.add(actiontarget, 1, 6);
```

Gestion d'évènements

- exemple (suite):

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        actiontarget.setFill(Color.FIREBRICK);
        actiontarget.setText("Validation demandée");
    }
});
Scene scene = new Scene(grid, 300, 275);
primaryStage.setScene(scene);
primaryStage.show();
}
public static void main(String[] args) {
  launch(args);
}
}
```

Expressions lambda

- les expressions Lambda sont apparues avec la version Java 1.8
- une expression lambda est une fonction avec ou sans paramètres, une expression ou bloc de code, retournant éventuellement une valeur
  - `e -> anim.playFromStart()`

Expressions lambda

- application à la gestion d'évènements

```
// classe interne anonyme
bouton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        anim.playFromStart();
    }
});
```

une interface qui déclare une seule méthode est appelée interface fonctionnelle: elle peut être remplacée par une expression lambda

Expressions lambda

- application à la gestion d'évènements (suite)

```
// expression lambda
bouton.setOnAction((ActionEvent event) -> {
      anim.playFromStart();
});
```

comme **`EventHandler`** ne comporte qu'une seule méthode **`handle()`** l'expression lambda doit implémenter la méthode **`handle()`**
le type de l'expression lambda est déduit par le compilateur comme étant de type **`EventHandler<ActionEvent>`** car la méthode **`onAction()`** prend en argument un objet de type **`EventHandler<ActionEvent>`**

Expressions lambda

- application à la gestion d'évènements (suite)

```
// expression lambda avec type déduit
bouton.setOnAction((event) -> {
      anim.playFromStart();
});
```

l'argument de l'expression lambda doit être de type **`ActionEvent`**, car c'est le type de l'argument reçu par la méthode **`handle()`** de **`EventHandler`**
l'expression lambda peut donc être simplifiée car le type de l'argument est déduit

Expressions lambda

- application à la gestion d'évènements (suite)

```
// expression lambda avec un seul argument
bouton.setOnAction(event -> {
      anim.playFromStart();
});
```

quand une expression lambda ne reçoit qu'un seul argument et que son type est déduit, les parenthèses deviennent facultatives

```
// expression lambda avec une seule instruction
bouton.setOnAction(event -> anim.playFromStart());
```

comme le bloc de l'expression lambda ne contient qu'une seule instruction, les parenthèses deviennent facultatives

Propriétés et beans JavaFX

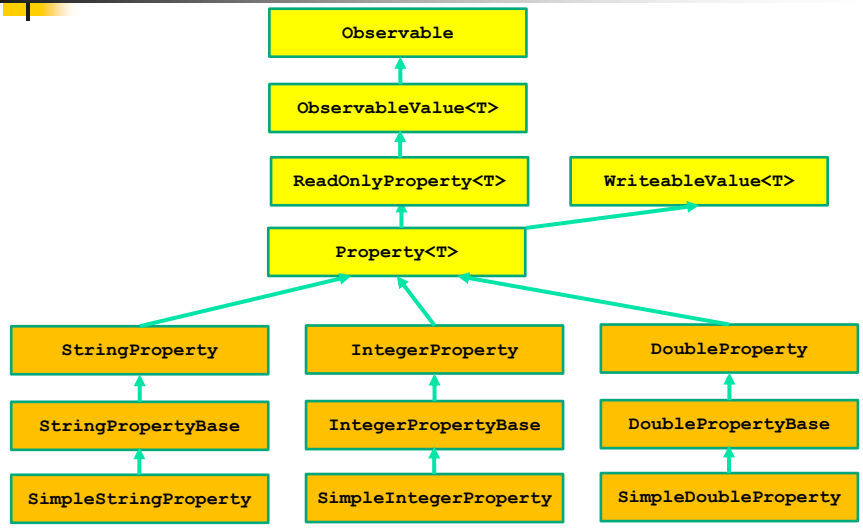
Propriétés et beans JavaFX

- propriétés JavaFX
- beans Java FX

Propriétés JavaFX

- une propriété JavaFX est particulière car elle permet, lorsque sa valeur est modifiée, de notifier un objet de son changement de valeur (binding)
  - cela permet de synchroniser la vue avec les données du modèle
- le package **`javafx.beans.property`** fournit des classes enveloppes pour encapsuler des propriétés de JavaBeans dans des propriétés JavaFX

Propriétés JavaFX



Beans JavaFX

- les beans de Java FX doivent pouvoir exploiter les propriétés décrites précédemment
  - tout en restant compatible avec le standard Java Bean
  - les attributs sont donc des sous-types de **`Property`**
  - les méthodes **`get`** renvoient le type primitif encapsulé dans l'attribut correspondant
  - les méthodes **`set`** encapsulent dans l'attribut le type primitif tansmis en argument
  - des méthodes supplémentaires nommées **`XxxxProperty`** permettent d'obtenir une référence sur l'attribut correspondant

Beans JavaFX

- exemple:

```
public class Personne {
   private final StringProperty nom;
   private final StringProperty prenom;
   private final IntegerProperty age;

   public Personne() {
      this("", "", 0);
   }
   public Personne(String nom, String prenom, int age) {
      this.nom = new SimpleStringProperty(nom);
      this.prenom = new SimpleStringProperty(prenom);
      this.age = new SimpleIntegerProperty(age);
   }
```

Beans JavaFX

- exemple (suite):

```
   public String getNom() { return nom.get(); }
   public void setNom(String nom) { this.nom.set(nom); }
   public StringProperty nomProperty(){return nom;}

   public String getPrenom() { return prenom.get(); }
   public void setPrenom(String prenom) {
          this.prenom.set(prenom); }
   public StringProperty prenomProperty(){return prenom;}

   public String getAge() { return age.get(); }
   public void setAge(int age) { this.age.set(age); }
   public IntegerProperty ageProperty(){return age;}
    ...
}
```

Beans JavaFX

- si l'on souhaite exploiter des java beans existants, il faut utiliser des classes du package `javafx.beans.property.adapter`
  - `JavaBeanBooleanProperty`
  - `JavaBeanIntegerProperty`
  - `JavaBeanLongProperty`
  - `JavaBeanFloatProperty`
  - `JavaBeanDoubleProperty`
  - `JavaBeanStringProperty`
  - `JavaBeanObjectProperty<T>`
  - ces classes fournissent des adaptateurs entre des propriétés standards de Java bean et des propriétés JavaFX

Beans JavaFX

- il faut nécessairement utiliser des builders appropriés pour créer des instances de ces classes

```
Personne personne = new Personne(); // Java Bean
JavaBeanStringProperty personneNom =
    JavaBeanStringPropertyBuilder.create()
    .bean(personne).name("nom").build();
personne.setNom("MARTIN");
System.out.println(personneNom.get()); // MARTIN
personneNom.set("DUPONT");
System.out.println(personne.get()); // DUPONT
```

  - la modificiation d'une propriété du javaBean entraîne celle de la propriété Java FX correspondante et vice-versa

Beans JavaFX

- exemple: Bean Java

```
public class Personne {
    private String nom;
    private String prenom;
    private Integer age;
    private PropertyChangeSupport listenerList =
             new PropertyChangeSupport(this);
    public Personne() {
        this("", "", 0);
    }
    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
    public String getNom() { return nom; }
    public String getPrenom() { return prenom; }
    public int getAge() { return age; }
```

Beans JavaFX

- exemple (suite):

```
    public void setNom(String nom) {
          String oldNom=this.nom;
          this.nom=nom;
          listenerList.firePropertyChange("nom",oldNom,nom);
    }
    public void setPrenom(String prenom) {
          String oldPrenom=this.prenom;
          this.prenom=prenom;
          listenerList.firePropertyChange("prenom",oldPrenom,prenom);
    }
    public void setAge(int age) {
          int oldAge=this.age;
          this.age=age;
          listenerList.firePropertyChange("age",oldAge,age);
    }
```

Beans JavaFX

- exemple (suite):

```
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        listenerList.addPropertyChangeListener(listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener listener){
        listenerList.removePropertyChangeListener(listener);
    }
    public void addPropertyChangeListener(String propertyName,
            PropertyChangeListener listener) {
        listenerList.addPropertyChangeListener(propertyName, listener);
    }
    public void removePropertyChangeListener(String propertyName,
            PropertyChangeListener listener) {
        listenerList.removePropertyChangeListener(propertyName, listener);
    }
}
```

Beans JavaFX

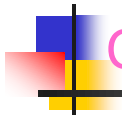
- exemple: Bean JavaFX

```
public class PersonneFX {
    private Personne personne;
    private JavaBeanStringProperty persNom;
    private JavaBeanStringProperty persPrenom;
    private JavaBeanIntegerProperty persAge;

    public PersonneFX(Personne personne) throws NoSuchMethodException{
        try {
            this.personne = personne;
            persNom = JavaBeanStringPropertyBuilder.create().
                    bean(personne).name("nom").build();
            persPrenom = JavaBeanStringPropertyBuilder.create().
                    bean(personne).name("prenom").build();
            persAge = JavaBeanIntegerPropertyBuilder.create().
                    bean(personne).name("age").build();
        }catch(NoSuchMethodException e){throw e;}
    }
```

Beans JavaFX

- exemple (suite):

```
        public String getNom() {return persNom.get();}
        public void setNom(String nom) {persNom.set(nom);}
        public StringProperty nomProperty() {return persNom;}
        public String getPrenom() {return persPrenom.get();}
        public void setPrenom(String prenom) {persPrenom.set(prenom);}
        public StringProperty prenomProperty() {return persPrenom;}
        public int getAge() {return persAge.get();}
        public void setAge(int age) {persAge.set(age);}
        public IntegerProperty ageProperty() {return persAge;}
        public Personne getPersonne() {
        return personne;
    }

    @Override
    public String toString() {
     return getNom() + " " + getPrenom() + " " + getAge();
    }
}
```



Collections Java FX

Collections JavaFX

- les collections JavaFX émettent des notifications lors de changements opérés dans la collection
  - sont définies dans le package **`javafx.collections`**
  - classes:
    - **`FXCollections`**: classe utilitaire avec méthodes *static* identiques à celles de `java.util.Collections`
    - **`ListChangeListener.Change`**: représente un changement effectué dans une `ObservableList`
    - **`SetChangeListener.Change`**: représente un changement effectué dans un `ObservableSet`
    - **`MapChangeListener.Change`**: représente un changement effectué dans une `ObservableMap`

Collections JavaFX

- Interfaces:
  - **`ObservableList`**: une liste permettant aux listeners d'être notifié des changements
  - **`ObservableSet`**: un set permettant aux listeners d'être notifié des changements
  - **`ObservableMap`**: une map qui permet aux listeners d'être notifié des changements
  - **`ListChangeListener`**: une interface qui reçoit des notifications de changements dans une `ObservableList`
  - **`SetChangeListener`**: une interface qui reçoit des notifications de changements dans un `ObservableSet`
  - **`MapChangeListener`**: une interface qui reçoit des notifications de changements dans une `ObservableMap`

Collections JavaFX

- exemple:

```
liste = new ArrayList<String>();
ObservableList<String> obsListe = FXCollections.observableList(liste);
obsList.addListener(new ListChangeListener() {
    @Override public void onChanged(ListChangeListener.Change change) {
            System.out.println("changement détecté! ");
    }
});

obsListe.add("élément un");                    //"changement détecté!"
list.add(" élément deux");
System.out.println("taille: "+obsListe.size()); // 2
```

Collections JavaFX

- exemple:

```
Map<String,String> map = new HashMap<String,String>();
ObservableMap<String,String> obsMap = FXCollections.observableMap(map);
obsMap.addListener(new MapChangeListener() {
    @Override public void onChanged(MapChangeListener.Change change) {
            System.out.println("changement détecté! ");
    }
});

obsMap.put("cle 1","valeur 1");                //"changement détecté!"
System.out.println("taille : " + obsMap.size()); // 1
map.put("cle 2","value 2");
System.out.println("taille: " + obsMap.size());  // 2
```

Data binding

Data binding

- data binding: principe
- data binding et propriétés JavaFX
- data binding avec API de bas niveau
- data binding avec API de haut niveau
- data binding: expressions

Data binding: principe

- la liaison de données (*Data binding*) consiste à associer une propriété JavaFX à un objet
  - cet objet sera notifié dès qu'un changement de valeur de cette propriété aura lieu
- les API de binding définissent un jeu d'interfaces permettant la mise en oeuvre du binding
  - `javafx.beans.`**`Observable`** et `javafx.beans.value.`**`ObservableValue`** notifient les changements
  - `javafx.beans.`**`InvalidationListener`** et `javafx.beans.value.`**`ChangeListener<T>`** sont notifiés des changements

Data binding: principe

- interface **`Observable`**

void	addListener(InvalidationListener listener) Adds an InvalidationListener which will be notified whenever the Observable becomes invalid.
void	removeListener(InvalidationListener listener) Removes the given listener from the list of listeners, that are notified whenever the value of the Observable becomes invalid.

- interface **`InvalidationListener`**

void	invalidated(Observable observable) This method needs to be provided by an implementation of InvalidationListener.

Data binding: principe

- interface **`ObservableValue<T>`** `extends Observable`

void	addListener(ChangeListener<? super T> listener) Adds a ChangeListener which will be notified whenever the value of the ObservableValue changes.
T	getValue() Returns the current value of this ObservableValue
void	removeListener(ChangeListener<? super T> listener) Removes the given listener from the list of listeners, that are notified whenever the value of the ObservableValue changes.

- interface **`ChangeListener<T>`**

void	changed(ObservableValue<? extends T> observable, T oldValue, T newValue) This method needs to be provided by an implementation of ChangeListener.

Data binding: principe

- un objet de type **`ObservableValue`** enveloppe une donnée et permet d'observer ses changements de valeur
  - une implémentation de cette interface peut supporter une évaluation paresseuse (*lazy evaluation*), qui signifie que la valeur n'est pas recalculée immédiatement après un changement, mais lorsque la valeur sera utilisée
- un objet de type **`ObservableValue`** générère deux types d'évènements
  - change events
  - invalidation events

Data binding: principe

- un *change event* indique que la valeur d'un **`ObservableValue`** a été modifiée
- un *invalidation event* est généré si la valeur courante n'est plus valide
  - si l'**`ObservableValue`** supporte l'évaluation paresseuse, on ne sait pas réellement si la valeur invalide a changé tant qu'elle n'a pas été recalculée
  - les *change events* sont générés pour une évaluation immédiate
  - les *invalidation events* peuvent être générés pour des implémentations immédiates ou paresseuses

Data binding: principe

- exemple:

```
public static void main(String[] args) {
    IntegerProperty alpha = new SimpleIntegerProperty(100);

    ChangeListener changeListener = new ChangeListener(){
       @Override
       public void changed(ObservableValue observable,
           Object oldValue, Object newValue) {
            System.out.println("valeur modifiée:"+
            "ancienne valeur = "+ oldValue +
            ", nouvelle valeur = "+ newValue);
        }
     };
```

Data binding: principe

- exemple (suite):

```
InvalidationListener invalidationListener =
        new InvalidationListener(){
    @Override
    public void invalidated(Observable observable) {
        System.out.println("valeur invalidée:"+
        " nouvelle valeur "+ observable );
    }
};
alpha.addListener(changeListener);
alpha.addListener(invalidationListener);
alpha.set(200); // les deux listeners sont déclenchés
```

  - à l'exécution:

```
valeur invalidée: nouvelle valeur IntegerProperty [value: 200]
valeur modifiée: ancienne valeur = 100, nouvelle valeur = 200
```

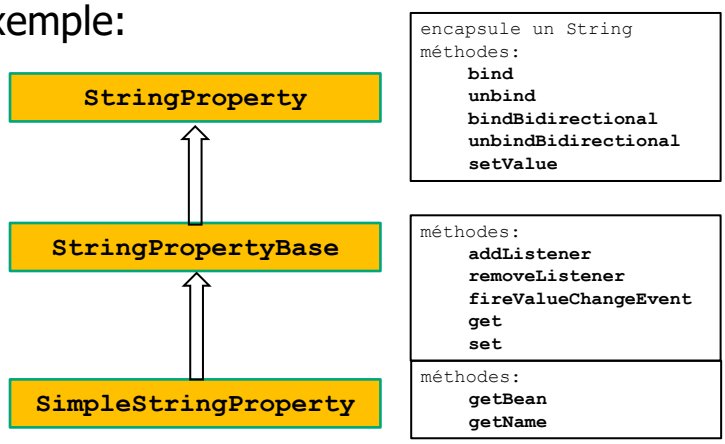
Data binding et propriétés JavaFX

- l'interface `javafx.beans.property.`**`Property<T>`** représente une propriété JavaFX

void	bind(ObservableValue<? extends T> observable) Create a unidirection binding for this Property.
void	bindBidirectional(Property<T> other) Create a bidirectional binding between this Property and another one.
boolean	isBound() Can be used to check, if a Property is bound.
void	unbind() Remove the unidirectional binding for this Property.
void	unbindBidirectional(Property<T> other) Remove a bidirectional binding between this Property and another one.

Data binding et propriétés JavaFX

- exemple:



  - des classes similaires existent pour les types **`Boolean`**, **`Integer`**, **`Long`**, **`Float`**, **`Double`**, **`Object`**

Data binding et propriétés JavaFX

- de nombreux contrôles graphiques possèdent des propriétés JavaFX
  - ceci les rend particulièrement adaptés comme source ou cible d'un binding
- exemple:

```
TextField tf = new TextField();
modele.nomProperty().bindBidirectional(
    tf.textProperty());
```

Data binding: mise en oeuvre

- le *Data binding* peut être effectué par:
  - des API de bas niveau (*low level binding*)
  - des API de haut niveau (*high level binding*)
    - API Fluent
    - classe **`Bindings`**

Data binding avec API bas niveau

- la méthode **`bind()`** permet de créer une liaison unidirectionnelle entre cette propriété et un **`ObservableValue`** (autre propriété par exemple)
  - toute modification de la valeur de l'**`ObservableValue`** est reflètée dans la propriété
  - toute modification de la propriété lance une **`RunTimeException`**
  - un seul **`ObservableValue`** peut être lié à une propriété donnée: un deuxième appel à **`bind()`** sur une propriété supprime la liaison précédente
- la méthode **`unbind()`** permet de supprimer la liaison

Data binding avec API bas niveau

- exemple

```
IntegerProperty alpha = new SimpleIntegerProperty(200);
IntegerProperty beta = new SimpleIntegerProperty(5);
beta.bind(alpha);
System.out.println("beta liée à alpha :" + beta.get());
System.out.println("alpha est modifiée ");
alpha.set(300);
System.out.println("alpha = " + alpha.get()+
       ", beta = " + beta.get());
```

  - à l'exécution:

```
beta liée à alpha = 200
alpha est modifiée
alpha = 300, beta = 300
```

Data binding avec API bas niveau

- la méthode **`bindBidirectional()`** permet de créer une liaison bidirectionnelle entre deux propriétés
  - toute modification de la valeur de l'une est reflètée dans l'autre
  - une même propriété peut être liée à plusieurs autres de façon bidirectionnelle
- la méthode **`unbindBidirectional`** permet de supprimer une liaison

Data binding avec API bas niveau

- exemple

```
IntegerProperty alpha = new SimpleIntegerProperty(200);
IntegerProperty beta = new SimpleIntegerProperty(100);
beta.bindBidirectional(alpha);
System.out.println("beta liée bidirect. à alpha = " +
       beta.get());
System.out.println("alpha est modifiée ");
alpha.set(300);
System.out.println("alpha = " + alpha.get()+", beta = " +
       beta.get());
System.out.println("beta est modifiée ");
beta.set(500);
System.out.println("alpha = " + alpha.get()+", beta = " +
       beta.get());
```

Data binding avec API bas niveau

- à l'exécution:

```
beta liée bidirect. à alpha = 200
alpha est modifiée
alpha = 300, beta = 300
beta est modifiée
alpha = 500, beta = 500
```

Data binding avec API bas niveau

- l'interface `javafx.beans.binding.`**`Binding<T>`** permet d'étendre les liaisons

...	...
void	dispose() Signals to the Binding that it will not be used anymore and any references can be removed.
ObservableList<?>	getDependencies() Returns the dependencies of a binding in an unmodifiable ObservableList.
void	invalidate() Mark a binding as invalid.
boolean	isValid() Checks if a binding is valid.
void	dispose() Signals to the Binding that it will not be used anymore and any references can be removed.
ObservableList<?>	getDependencies() Returns the dependencies of a binding in an unmodifiable ObservableList.
void	invalidate() Mark a binding as invalid.

Data binding avec API bas niveau

- l'interface **`Binding`** est implémentée par les classes abstraites:
  - **`BooleanBinding, DoubleBinding, FloatBinding, IntegerBinding, ListBinding, LongBinding, MapBinding, ObjectBinding, SetBinding, StringBinding`**
- pour mettre en oeuvre ces classes, il faut:
  1. appeler la méthode
     `void bind(Observable... dependencies)`
  2. redéfinir la méthode abstraite **`computeValue`** qui exploite la valeur courante des dépendances
     - elle est appelée lorsque la méthode **`get()`** est appelée lors d'un binding invalide

Data binding avec API bas niveau

- exemple avec **`DoubleBinding`**

```
final DoubleProperty a = new SimpleDoubleProperty(1);
final DoubleProperty b = new SimpleDoubleProperty(2);
final DoubleProperty c = new SimpleDoubleProperty(3);
final DoubleProperty d = new SimpleDoubleProperty(4);
DoubleBinding db = new DoubleBinding() {
   {
      super.bind(a, b, c, d); // lie la variable db à a, b, c, d
   }
   @Override protected double computeValue() {
     return ( a.get() * b.get()) + (c.get() * d.get() );
   }
};
System.out.println(db.get());//14 (entraîne l'appel à computeValue)
b.set(3);
System.out.println(db.get());//15 (entraîne l'appel à computeValue)
```

Data binding avec API haut niveau

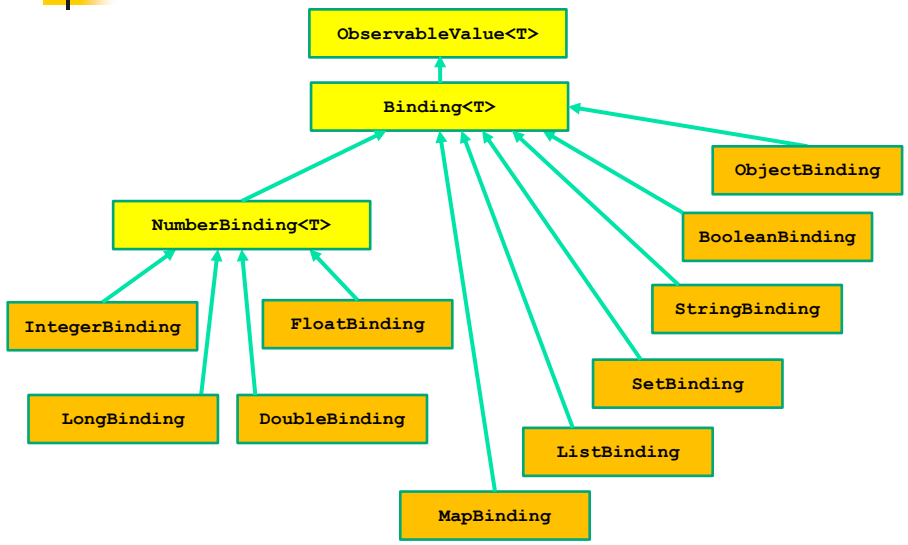
- des API de haut niveau simplifient la mise en oeuvre de liaisons dans des cas simples
  - deux possibilités:
    - API Fluent
    - classe **`Bindings`**
- exemple: l'expression result = a*b + c*d
  - avec API Fluent:

```
DoubleBinding result =
   a.multiply(b).add(c.multiply(d));
```

  - avec classe **`Bindings`**:

```
NumberBinding result = Bindings.add (
   Bindings.multiply(a, b),Bindings.multiply(c,d));
```

Data binding avec API Fluent



Data binding avec API Fluent

- exemple avec l'API Fluent:

```
IntegerProperty num1 = new SimpleIntegerProperty(1);
IntegerProperty num2 = new SimpleIntegerProperty(2);
NumberBinding sum = num1.add(num2);
System.out.println( sum.getValue() );// 3
num1.set(2);
System.out.println( sum.getValue() );// 4
```

  - l'objet `sum` est lié aux propriétés `num1` et `num2`

Data binding avec classe Bindings

- la classe **`Bindings`** contient de nombreuses méthodes utilitaires permettant d'effectuer des opérations (`add, divide, multiply, min, max …`) sur des types primitifs et/ou des objets de type **`ObservableValue`**

static NumberBinding	add(ObservableNumberValue op1, ObservableNumberValue op2) Creates a new NumberBinding that calculates the sum of the values of two instances of ObservableNumberValue.

Data binding avec classe Bindings

- exemple avec classe **`Bindings`**:

```
IntegerProperty num1 = new SimpleIntegerProperty(1);
IntegerProperty num2 = new SimpleIntegerProperty(2);
NumberBinding sum = Bindings.add(num1,num2);
System.out.println( sum.getValue() );// 3
num1.set(2);
System.out.println( sum.getValue() );// 4
```

  - l'objet `sum` est lié aux propriétés `num1` et `num2`

Data binding avec classe Bindings

- l'API Fluent et la classe **`Bindings`** peuvent être utilisées simultanément

```
NumberBinding result = Bindings.add
   (a.multiply(b), c.multiply(d));
```

- différence principale:
  - l'API Fluent nécessite au moins qu'un des opérandes soit une expression
    - son type de retour est plus précis

Data binding: expressions

- il est possible de constituer des expressions faisant intervenir des opérateurs conditionnels
  - exemple:

```
DoubleProperty a = new SimpleDoubleProperty(0);
DoubleProperty b = new SimpleDoubleProperty(0);
DoubleBinding s = a.add(b).divide(2.0D);
final DoubleBinding aBinding = new When(
   a.add(b).greaterThan(b).and(a.add(a).greaterThan(b)))
   .then(s.multiply(s.subtract(a)).multiply(s.subtract(b)))
   .otherwise(0.0D);
a.set(3);
b.set(4);
System.out.println(a.get());
System.out.println(b.get());
System.out.println(aBinding.get());
```

Data binding: expressions

Operations	Examples	Type
Arithmetic Operations	num1.add(num2) Bindings.divide(num1, num2) num1.negate()	Number
Boolean Operations	Bindings.or(bool1, bool2) bool1.not()	Boolean
	obj1.isEqualTo(obj2) Bindings.notEqual(obj1, obj2)	All
Comparisons	num1.greaterThan(num2) num1.lessThanOrEqualTo(num2)	Number, String
	Bindings.equalIgnoreCase(s1, s2) s1.isNotEqualToIgnoreCase(s2)	String
	obj.asString()	All
Conversions	num.asString(format) Bindings.format(format, val...)	Number, Object
Null Check	obj.isNull() Bindings.isNotNull(obj)	Object, String
String Operations	Bindings.concat(s1, s2) s.length()	String
Min / Max	Bindings.min(num1, num2) Bindings.max(num1, num2)	Number
Collections	Bindings.valueAt(list, index) Bindings.size(collection)	Collection
Select Binding	Bindings.select(root, properties...)	
Ternary Expression	Bindings.when(cond).then(val1).otherwise(val2)	

Composants graphiques évolués:
TableView, TreeView, TreeTableView

présentation

- la présentation de certaines informations nécessite des composants graphiques évolués
- JavaFX propose entre autres:
  - TableView
  - TreeView
  - TreeTableView

TableView

- la classe **`TableView<S>`** permet d'afficher des données sous forme tabulaire, avec un nombre illimité de lignes
  - elle s'utilise avec les classes **`TableColumn<S,T>`** et **`TableCell<S,T>`**
  - principe
    - l'instance de **`TableView`** précise le type des données qui seront affichées

```
TableView<UserAccount> table = new TableView<>();
```

    - l'instance de **`TableView`** est alimentée par la liste d'objets à visualiser

```
ObservableList<UserAccount> list = getUserList();
table.setItems(list);
```

TableView

- les instances de **`TableColumn`** précisent, pour chaque colonne, son titre ainsi que le type des données qui seront affichées, en principe extraites du type principal

```
TableColumn<UserAccount, String> userNameCol =
    new TableColumn<>("User Name");
TableColumn<UserAccount, String> emailCol =
    new TableColumn<>("Email");
```

- les instances de **`TableColumn`** sont ajoutées à l'instance de **`TableView`**

```
table.getColumns().addAll(userNameCol, emailCol);
```

TableView

- les données affichées dans chaque colonne doivent être précisées au moyen de la méthode **`setCellValueFactory`** de **`TableColumn`**

```
public final void setCellValueFactory(
    Callback<TableColumn.CellDataFeatures<S,T>,
    ObservableValue<T>> value)
```

```
emailCol.setCellValueFactory(
new Callback<CellDataFeatures<UserAccount,String>,
    ObservableValue<String>>() {
       public ObservableValue<String> call(
        CellDataFeatures<UserAccount, String> u) {
           return u.getValue().emailProperty();
        }
    }
);
```

TableView

- si le JavaBean ne possède pas de propriété JavaFX mais des propriétés d'un JavaBean ordinaire, celles-ci peuvent être enveloppées dans un `ReadOnlyObjectWrapper`

```
emailCol.setCellValueFactory(
  new Callback<CellDataFeatures<UserAccount, String>,
    ObservableValue<String>>() {
        public ObservableValue<String> call(
           CellDataFeatures<UserAccount, String> u) {
             return new ReadOnlyObjectWrapper(
                  u.getValue().getEmail());
             }
    }
);
```

TableView

- en pratique, il est plus simple de faire appel à une instance de **`PropertyValueFactory`** qui traite les deux cas précédent

```
emailCol.setCellValueFactory(
    new PropertyValueFactory<UserAccount,String>
        ("email"));
```

  - s'il existe une propriété JavaFX `email` dans la classe `UserAccount`, la méthode `call` retourne cette propriété
  - si la propriété `email` est une propriété ordinaire de JavaBean dans la classe `UserAccount`, la méthode `call` retourne un **`ReadOnlyObjectWrapper`**

TableView

- exemple:

```
public class TableViewDemo extends Application {
  @Override public void start(Stage stage) {
      TableView<UserAccount> table = new TableView<UserAccount>();
      // Create column UserName (Data type of String).
      TableColumn<UserAccount, String> userNameCol //
              = new TableColumn<UserAccount, String>("User Name");
      // Create column Email (Data type of String).
      TableColumn<UserAccount, String> emailCol//
              = new TableColumn<UserAccount, String>("Email");
      // Create column FullName (Data type of String).
      TableColumn<UserAccount, String> fullNameCol//
              = new TableColumn<UserAccount, String>("Full Name");
      // Create 2 sub column for FullName.
      TableColumn<UserAccount, String> firstNameCol//
              = new TableColumn<UserAccount, String>("First Name");
      TableColumn<UserAccount, String> lastNameCol //
              = new TableColumn<UserAccount, String>("Last Name");
```

TableView

- exemple (suite):

```
      // Add sub columns to the FullName
      fullNameCol.getColumns().addAll(firstNameCol, lastNameCol);
      // Active Column
      TableColumn<UserAccount, Boolean> activeCol//
              = new TableColumn<UserAccount, Boolean>("Active");

      // Defines how to fill data for each cell.
      // Get value from property of UserAccount. .
      userNameCol.setCellValueFactory(
             new PropertyValueFactory<>("userName"));
      emailCol.setCellValueFactory(
             new PropertyValueFactory<>("email"));
      firstNameCol.setCellValueFactory(
             new PropertyValueFactory<>("firstName"));
      lastNameCol.setCellValueFactory(
             new PropertyValueFactory<>("lastName"));
      activeCol.setCellValueFactory(
             new PropertyValueFactory<>("active"));
```

TableView

- exemple (suite):

```
        // Set Sort type for userName column
        userNameCol.setSortType(TableColumn.SortType.DESCENDING);
        lastNameCol.setSortable(false);

        // Display row data
        ObservableList<UserAccount> list = getUserList();
        table.setItems(list);
        table.getColumns().addAll(
               userNameCol, emailCol, fullNameCol, activeCol);
        StackPane root = new StackPane();
        root.setPadding(new Insets(5));
        root.getChildren().add(table);

        stage.setTitle("TableView (demo)");
        Scene scene = new Scene(root, 450, 300);
        stage.setScene(scene);
        stage.show();
    }
```

TableView

- exemple (suite):

```
public class UserAccount {
   private Long id;
   private String userName;
   private String email;
   private String firstName;
   private String lastName;
   private boolean active;

   public UserAccount(Long id, String userName, String email,
           String firstName, String lastName, boolean active) {
       this.id = id;
       this.userName = userName;
       this.email = email;
       this.firstName = firstName;
       this.lastName = lastName;
       this.active = active;
   }
    // get and set methods
}
```

TableView

- exemple (suite):

```
    private ObservableList<UserAccount> getUserList() {
        UserAccount user1 = new UserAccount(
               1L, "smith", "smith@gmail.com","Susan", "Smith", true);
        UserAccount user2 = new UserAccount(
               2L, "mcneil","mcneil@gmail.com","Anne", "McNeil", true);
        UserAccount user3 = new UserAccount(
               3L, "white", "white@gmail.com", "Kenvin", "White", false);

        ObservableList<UserAccount> list =
                        FXCollections.observableArrayList(user1, user2, user3);
        return list;
    }

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

TableView

- si les données doivent être affichées d'une manière particulière ou doivent être éditées, il faut définir les cellules au moyen de la méthode **`setCellFactory`** de **`TableColumn`**

`public final void setCellFactory(`
`Callback<TableColumn<S,T>,TableCell<S,T>> value)`

```
colConges.setCellFactory(
   new Callback<TableColumn<Personne,Boolean>,
                TableCell<Personne,Boolean>>() {
      @Override public TableCell<Personne, Boolean>
         call(TableColumn<Personne, Boolean> p) {
            CheckBoxTableCell<Personne, Boolean> cell =
                        new CheckBoxTableCell<Personne, Boolean>();
            cell.setAlignment(Pos.CENTER);
            return cell;
         }
    });
```

TableView

- de façon à faciliter la personnalisation des cellules, JavaFX contient des fabriques de cellules prédéfinies, dans les sous-classes de **`TableCell<S,T>`** :
  - **`CheckBoxTableCell<S,T>`**
  - **`ChoiceBoxTableCell<S,T>`**
  - **`ComboBoxTableCell<S,T>`**
  - **`ProgressBarTableCell<S>`**
  - **`TextFieldTableCell<S,T>`**

  - exemple:

```
colNom.setCellFactory(
        TextFieldTableCell.<Personne>forTableColumn());
```

TableView

- dans **`TableColumn`**, des méthodes permettent de gérer la modification d'une cellule:

void	setOnEditCancel(EventHandler<TableColumn.CellEditEvent<S,T>> value) Sets the value of the property onEditCancel.
void	setOnEditCommit(EventHandler<TableColumn.CellEditEvent<S,T>> value) Sets the value of the property onEditCommit.
void	setOnEditStart(EventHandler<TableColumn.CellEditEvent<S,T>> value) Sets the value of the property onEditStart.

```
colNom.setOnEditCommit((CellEditEvent<Personne, String> event) -> {
    TablePosition<Personne, String> pos = event.getTablePosition();
    String nouveauNom = event.getNewValue();
    int row = pos.getRow();
    Personne personne = event.getTableView().getItems().get(row);
    personne.setNom(nouveauNom);
});
```

- Note: l'instance de **`TableView`** doit être éditable

```
table.setEditable(true);
```

TableView

- exemple:

```
public class Main extends Application {
  @SuppressWarnings("unchecked")
  @Override
   public void start(Stage stage) {

   TableView<Personne> table = new TableView<Personne>();
   ObservableList<Personne> list = getPersonnel();
   table.setItems(list);
   table.setEditable(true);
   TableColumn<Personne, String> colNom = new TableColumn<Personne, String>("Nom");
   TableColumn<Personne, Sexe> colSexe = new TableColumn<Personne, Sexe>("Sexe");
   TableColumn<Personne, Boolean> colConges = new TableColumn<Personne, Boolean>("En
   congés ?");

   // ajout des colonnes à la table
   table.getColumns().addAll(colNom, colSexe, colConges);

   // colonne Nom
   colNom.setCellFactory(TextFieldTableCell.<Personne>forTableColumn());
   colNom.setCellValueFactory(new PropertyValueFactory<>("nom"));
   colNom.setMinWidth(200);
```

TableView

- exemple (suite):

```
colNom.setOnEditCommit((CellEditEvent<Personne, String> event) -> {
     TablePosition<Personne, String> pos = event.getTablePosition();
     String nouveauNom = event.getNewValue();
     int row = pos.getRow();
     Personne personne = event.getTableView().getItems().get(row);
     personne.setNom(nouveauNom);
});
// colonne Sexe
ObservableList<Sexe> listeSexe = FXCollections.observableArrayList(Sexe.values());
colSexe.setCellFactory(ComboBoxTableCell.forTableColumn(listeSexe));
colSexe.setMinWidth(120);
colSexe.setCellValueFactory(
    new Callback<CellDataFeatures<Personne, Sexe>, ObservableValue<Sexe>>() {
       @Override
     public ObservableValue<Sexe> call(CellDataFeatures<Personne, Sexe> param) {
          Personne personne = param.getValue();
          String codeSexe = personne.getSexe();
          Sexe sexe = Sexe.getByCode(codeSexe);
          return new SimpleObjectProperty<Sexe>(sexe);
     }
});
```

TableView

- exemple (suite):

```
colSexe.setOnEditCommit((CellEditEvent<Personne, Sexe> event) -> {
     TablePosition<Personne, Sexe> pos = event.getTablePosition();
     Sexe nouveauSexe = event.getNewValue();
     int row = pos.getRow();
     Personne personne = event.getTableView().getItems().get(row);
     personne.setSexe(nouveauSexe.getCode());
});

// colonne Conges
colConges.setCellFactory(
   new Callback<TableColumn<Personne, Boolean>, TableCell<Personne, Boolean>>() {
      @Override
      public TableCell<Personne, Boolean> call(TableColumn<Personne, Boolean> p) {
          CheckBoxTableCell<Personne, Boolean> cell =
                    new CheckBoxTableCell<Personne, Boolean>();
          cell.setAlignment(Pos.CENTER);
          return cell;
      }
});
```

TableView

- exemple (suite):

```
colConges.setCellValueFactory(
    new Callback<CellDataFeatures<Personne, Boolean>, ObservableValue<Boolean>>() {
     @Override
     public ObservableValue<Boolean> call(CellDataFeatures<Personne, Boolean> param){
          Personne personne = param.getValue();
          SimpleBooleanProperty booleanProp = new
          SimpleBooleanProperty(personne.isEnConges());
          return booleanProp;
     }
});

colConges.setOnEditCommit((CellEditEvent<Personne, Boolean> event) -> {
     TablePosition<Personne, Boolean> pos = event.getTablePosition();
     Boolean enConges = event.getNewValue();
     int row = pos.getRow();
     Personne personne = event.getTableView().getItems().get(row);
     personne.setEnConges(enConges);
});
```

TableView

- exemple (suite):

```
        StackPane root = new StackPane();
        root.setPadding(new Insets(5));
        root.getChildren().add(table);
        Scene scene = new Scene(root, 450, 300);
        stage.setScene(scene);
        stage.setTitle("PERSONNEL");
        stage.show();
    }
    private ObservableList<Personne> getPersonnel() {
        Personne personne1 =
            new Personne("LEMAIRE Christine", Sexe.FEMININ.getCode(), true);
        Personne personne2 =
            new Personne("LEGOFF Anne", Sexe.FEMININ.getCode(), false);
        Personne personne3 =
            new Personne("MARTIN Nicolas", Sexe.MASCULIN.getCode(), false);
        Personne personne4 = new Personne("DUPONT Jean", Sexe.MASCULIN.getCode(), true);
        ObservableList<Personne> list =
          FXCollections.observableArrayList(personne1, personne2, personne3, personne4);
        return list;
    }
    public static void main(String[] args) {launch(args);}
}
```

TableView

- exemple (suite):

```
public enum Sexe {
   FEMININ("F", "Femme"), MASCULIN("M", "Homme");
   private String code;
   private String text;

   private Sexe(String code, String text) {
       this.code = code;
       this.text = text;
   }
   public String getCode() {return code;}
   public String getText() {return text;}
   public static Sexe getByCode(String codeSexe) {
       for (Sexe g : Sexe.values()) {
           if (g.code.equals(codeSexe)) {
               return g;
           }
       }
       return null;
   }
   @Override public String toString() {return this.text;}
}
```

TableView

- exemple (suite):

```
public class Personne {
    private String nom;
    private String sexe;
    private boolean enConges;

    public Personne(String nom, String sexe, boolean enConges) {
        this.nom = nom;
        this.sexe = sexe;
        this.enConges = enConges;
    }

    // getters et setters

}
```

TreeView

- la classe **`TreeView`** permet d'afficher des données sous forme d'un arbre
  - elle s'utilise avec la classe **`TreeItem<T>`**
  - principe
    - l'instance de **`TreeView`** précise le type des données qui seront affichées

```
TreeView<BookCategory> tree = new TreeView<>();
```

    - la racine de l'arbre est représentée par une instance de **`TreeItem`**

```
BookCategory catRoot =new BookCategory("ROOT","Root");
TreeItem<BookCategory> rootItem =
    new TreeItem<>(catRoot);
rootItem.setExpanded(true);
```

TreeView

- les autres noeuds de l'arbre sont représentés par des instances de **`TreeItem`**

```
TreeItem<BookCategory> itemJava = new TreeItem<>(catJava);
TreeItem<BookCategory> itemJSP = new TreeItem<>(catJSP);
TreeItem<BookCategory> itemSpring = new TreeItem<>(catSpring);
TreeItem<BookCategory> itemCSharp = new TreeItem<>(catCSharp);
TreeItem<BookCategory> itemWinForm = new TreeItem<>(catWinForm);
```

- les noeuds sont ajoutés les uns aux autres pour constituer l'arborescence

```
itemJava.getChildren().addAll(itemJSP, itemSpring);
itemCSharp.getChildren().addAll(itemWinForm);
rootItem.getChildren().addAll(itemJava,itemCSharp);
```

- l'élément racine est accroché au **`TreeView`**

```
tree.setRoot(rootItem);
```

TreeView

- exemple:

```
public class TreeViewMultiRootDemo extends Application {
   @Override
   public void start(Stage primaryStage) {

       BookCategory catRoot = new BookCategory("ROOT", "Root");
       BookCategory catJava = new BookCategory("JAVA-00", "Java");
       BookCategory catJSP = new BookCategory("JAVA-01", "Jsp");
       BookCategory catSpring = new BookCategory("JAVA-02", "Spring");
       BookCategory catCSharp = new BookCategory("C#-00", "CSharp");
       BookCategory catWinForm = new BookCategory("C#-01", "Win Form");

       // Root Item
       TreeItem<BookCategory> rootItem = new TreeItem<BookCategory>(catRoot);
       rootItem.setExpanded(true);
```

TreeView

- exemple (suite):

```
TreeItem<BookCategory> itemJava = new TreeItem<BookCategory>(catJava);
TreeItem<BookCategory> itemJSP = new TreeItem<BookCategory>(catJSP);
TreeItem<BookCategory> itemSpring =
      new TreeItem<BookCategory>(catSpring);
itemJava.getChildren().addAll(itemJSP, itemSpring);

TreeItem<BookCategory> itemCSharp =
      new TreeItem<BookCategory>(catCSharp);
TreeItem<BookCategory> itemWinForm =
      new TreeItem<BookCategory>(catWinForm);
itemCSharp.getChildren().addAll(itemWinForm);

// Add to Root
rootItem.getChildren().addAll(itemJava,itemCSharp);
TreeView<BookCategory> tree = new TreeView<BookCategory>(rootItem);
// Hide the root Item.
tree.setShowRoot(false);
```

TreeView

- exemple (suite):

```
        StackPane root = new StackPane();
        root.setPadding(new Insets(5));
        root.getChildren().add(tree);

        primaryStage.setTitle("JavaFX TreeView (demo)");
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }

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

TreeView

- si les données doivent être affichées d'une manière particulière ou doivent être éditées, il faut définir les cellules au moyen de la méthode **`setCellFactory`** de **`TreeView`**

```
public final void setCellFactory(
Callback<TreeView<T>,TreeCell<T>> value)
```

```
treeView.setCellFactory(
   new Callback<TreeView<Layer>, TreeCell<Layer>>() {
        @Override
        public TreeCell<Layer> call(TreeView<Layer> param){
             return new CheckBoxTreeCell<Layer>();
         }
});
```

TreeView

- de façon à faciliter la personnalisation des noeuds, JavaFX contient des fabriques de noeuds prédéfinies, dans les sous-classes de **`TreeCell<T>`** :
  - **`CheckBoxTreeCell<T>`**
  - **`ChoiceBoxTreeCell<T>`**
  - **`ComboBoxTreeCell<T>`**
  - **`TextFieldTreeCell<T>`**

- exemple:

```
treeView.setCellFactory(TextFieldTreeCell.forTreeView());
```

TreeView

- dans **`TreeView`**, des méthodes permettent de gérer la modification d'un noeud:

void	setOnEditCancel(EventHandler<TreeView.EditEvent<T>> value) Sets the EventHandler that will be called when the user cancels an edit.
void	setOnEditCommit(EventHandler<TreeView.EditEvent<T>> value) Sets the EventHandler that will be called when the user commits an edit.
void	setOnEditStart(EventHandler<TreeView.EditEvent<T>> value) Sets the EventHandler that will be called when the user begins an edit.

```
treeView.setOnEditCommit(event -> commitEdition(event));
```

TreeView

- la gestion d'évènements sur les nœuds peut se simplifier par une gestion d'évènement sur le seul nœud racine (**`TreeItem`**)

```
elementRacine.addEventHandler(
    TreeItem.<String>branchExpandedEvent(),
    new EventHandler<TreeModificationEvent<String>>() {
      @Override
      public void handle(TreeModificationEvent<String> evt) {
           branchExpanded(event);
      }
});
```

TreeView

- différents types d'évènements (**`TreeItem`**):

`TreeItem.treeNotificationEvent()`
`TreeItem.valueChangedEvent()`
`TreeItem.graphicChangedEvent()`
`TreeItem.treeItemCountChangeEvent()`
`TreeItem.branchExpandedEvent()`
`TreeItem.branchCollapsedEvent()`
`TreeItem.childrenModificationEvent()`

TreeView

- exemple:

```
public class TreeViewCheckBoxDemo extends Application {
    @Override public void start(Stage primaryStage) {
        Layer layer0 = new Layer("Layers", false);
        Layer layer1 = new Layer("Wet Area", false);
        Layer layer2 = new Layer("Titles", true);
        Layer layer3 = new Layer("Logos", true);
        // Root Item
        TreeItem<Layer> rootItem = new TreeItem<Layer>(layer0);
        rootItem.setExpanded(true);
        // Item 1
        CheckBoxTreeItem<Layer> item1 = new CheckBoxTreeItem<Layer>(layer1);
        item1.setSelected(layer1.isSelected());
        // Item 2
        CheckBoxTreeItem<Layer> item2 = new CheckBoxTreeItem<Layer>(layer2);
        item1.setSelected(layer2.isSelected());
        // Item 3
        CheckBoxTreeItem<Layer> item3 = new CheckBoxTreeItem<Layer>(layer3);
        item1.setSelected(layer3.isSelected());
```

TreeView

- exemple (suite):

```
        // Add to Root
        rootItem.getChildren().addAll(item1, item2, item3);
        TreeView<Layer> treeView = new TreeView<Layer>(rootItem);
        // Set Cell Factory.
        treeView.setCellFactory(
             new Callback<TreeView<Layer>, TreeCell<Layer>>() {
            @Override public TreeCell<Layer> call(TreeView<Layer> param) {
                return new CheckBoxTreeCell<Layer>();
            }
        });
        StackPane root = new StackPane();
        root.setPadding(new Insets(5));
        root.getChildren().add(treeView);
        primaryStage.setTitle("JavaFX TreeView");
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
    public static void main(String[] args) {launch(args);}
}
```

TreeView

- exemple (suite):

```
public class Layer {
    private String layerName;
    private boolean selected;

    public Layer() {

    }
    public Layer(String layerName, boolean selected) {
        this.layerName = layerName;
        this.selected = selected;
    }
    // accessors
    // ….
    @Override
    public String toString() {
        return this.layerName;
    }

}
```

TreeTableView

- la classe **`TreeTableView`** permet d'afficher des données sous forme tabulaire et arborescente simultanément
  - elle fonctionne avec les classes **`TreeItem`**, **`TreeTableColumn`** et **`TreeTableCell`**
  - principe:
    - créer un **`TreeTableView`**:
    - ajouter un **`TreeItem`** racine au **`TreeTableView`**
    - ajouter d'autres **`TreeItem`** au noeud racine
    - définir les colonnes de la table via **`TreeTableColumn`**
    - faire éventuellement appel à **`TreeTableCell`** pour personnaliser certaines cellules

TreeTableView

- exemple:

```
public class TreeTableViewEditDemo extends Application {
    @Override public void start(Stage stage) {
        TreeTableView<Employee> treeTableView =
             new TreeTableView<Employee>();
        treeTableView.setEditable(true);
        // Create column EmpNo (Data type of String).
        TreeTableColumn<Employee, String> empNoCol //
                = new TreeTableColumn<Employee, String>("Emp No");

        // Create column FullName (Data type of String).
        TreeTableColumn<Employee, String> fullNameCol//
                = new TreeTableColumn<Employee, String>("Full Name");

        // Create 2 sub column for FullName.
        TreeTableColumn<Employee, String> firstNameCol //
                = new TreeTableColumn<Employee, String>("First Name");

        TreeTableColumn<Employee, String> lastNameCol //
                = new TreeTableColumn<Employee, String>("Last Name");
```

TreeTableView

- exemple (suite):

```
// Add sub columns to the FullName
fullNameCol.getColumns().addAll(firstNameCol, lastNameCol);

// Gender Column
TreeTableColumn<Employee, Gender> genderCol //
        = new TreeTableColumn<Employee, Gender>("Gender");
genderCol.setMinWidth(90);

// Position Column
TreeTableColumn<Employee, String> positionCol //
        = new TreeTableColumn<Employee, String>("Position");

// Single? Column
TreeTableColumn<Employee, Boolean> singleCol//
        = new TreeTableColumn<Employee, Boolean>("Single?");

// Add columns to TreeTable.
treeTableView.getColumns().addAll(empNoCol, fullNameCol, positionCol,
genderCol, singleCol);
```

TreeTableView

- exemple (suite):

```
// Data
Employee empBoss = new Employee("E00", "Abc@gmail.com", //
        "Boss", "Boss", "Manager", "M", false);

Employee empSmith = new Employee("E01", "Smith@gmail.com", //
        "Susan", "Smith", "Salesman", "F", true);

Employee empMcNeil = new Employee("E02", "McNeil@gmail.com", //
        "Anne", "McNeil", "Cleck", "M", false);

// Root Item
TreeItem<Employee> itemRoot = new TreeItem<Employee>(empBoss);
TreeItem<Employee> itemSmith = new TreeItem<Employee>(empSmith);
TreeItem<Employee> itemMcNeil = new TreeItem<Employee>(empMcNeil);

itemRoot.getChildren().addAll(itemSmith, itemMcNeil);
treeTableView.setRoot(itemRoot);
```

TreeTableView

- exemple (suite):

```
// Defines how to fill data for each cell.
// Get value from property of Employee.
empNoCol.setCellValueFactory(
     new TreeItemPropertyValueFactory<Employee, String>("empNo"));
firstNameCol.setCellValueFactory(
     new TreeItemPropertyValueFactory<Employee, String>("firstName"));
lastNameCol.setCellValueFactory(
     new TreeItemPropertyValueFactory<Employee, String>("lastName"));
positionCol.setCellValueFactory(
     new TreeItemPropertyValueFactory<Employee, String>("position"));
```

TreeTableView

- exemple (suite):

```
        // GENDER (COMBO BOX).
        genderCol.setCellValueFactory(new
Callback<TreeTableColumn.CellDataFeatures<Employee, Gender>, //
        ObservableValue<Gender>>() {
            @Override public ObservableValue<Gender>
call(TreeTableColumn.CellDataFeatures<Employee, Gender> param) {
                TreeItem<Employee> treeItem = param.getValue();
                Employee emp = treeItem.getValue();
                // F,M
                String genderCode = emp.getGender();
                Gender gender = Gender.getByCode(genderCode);
                return new SimpleObjectProperty<Gender>(gender);
            }
        });
        ObservableList<Gender> genderList =
FXCollections.observableArrayList(Gender.values());
        genderCol.setCellFactory(
             ComboBoxTreeTableCell.forTreeTableColumn(genderList));
```

TreeTableView

- exemple (suite):

```
        // After user edit on cell, update to Model.
        genderCol.setOnEditCommit(new
EventHandler<TreeTableColumn.CellEditEvent<Employee, Gender>>() {
            @Override public void handle(
             TreeTableColumn.CellEditEvent<Employee, Gender> event) {
                TreeItem<Employee> item = event.getRowValue();
                Employee emp = item.getValue();
                Gender newGender = event.getNewValue();
                emp.setGender(newGender.getCode());
                System.out.println("Single column commit. new gender:"
+newGender);
                System.out.println("EMP:"+emp.isSingle());
            }
        });
```

TreeTableView

- exemple (suite):

```
        // ==== SINGLE? (CHECH BOX) ===
        singleCol.setCellValueFactory(new
Callback<TreeTableColumn.CellDataFeatures<Employee, Boolean>,
        ObservableValue<Boolean>>() {
            @Override
             public ObservableValue<Boolean>
call(TreeTableColumn.CellDataFeatures<Employee, Boolean> param) {
                TreeItem<Employee> treeItem = param.getValue();
                Employee emp = treeItem.getValue();
                SimpleBooleanProperty booleanProp=
                      new SimpleBooleanProperty(emp.isSingle());

                // Note: singleCol.setOnEditCommit(): Not work for
                // CheckBoxTreeTableCell.
                // When "Single?" column change.
```

TreeTableView

- exemple (suite):

```
booleanProp.addListener(new ChangeListener<Boolean>() {
   @Override public void changed(
      ObservableValue<? extends Boolean> observable, Boolean oldValue,
                    Boolean newValue) {
                emp.setSingle(newValue);
            }
        });
        return booleanProp;
    }
})
```

TreeTableView

- exemple (suite):

```
      singleCol.setCellFactory(new
Callback<TreeTableColumn<Employee,Boolean>,TreeTableCell<Employee,Boolean>>()
{  @Override public TreeTableCell<Employee,Boolean> call(
TreeTableColumn<Employee,Boolean> p ) {
                CheckBoxTreeTableCell<Employee,Boolean> cell =
                     new CheckBoxTreeTableCell<Employee,Boolean>();
                cell.setAlignment(Pos.CENTER);
                return cell;
            }
        });
        StackPane root = new StackPane();
        root.setPadding(new Insets(5));
        root.getChildren().add(treeTableView);
        stage.setTitle("TreeTableView (o7planning.org)");
        Scene scene = new Scene(root, 450, 300);
        stage.setScene(scene);stage.show();
    }
    public static void main(String[] args) {launch(args);}
}
```

Cascading Style Sheet (CSS)

Cascading Style Sheet (CSS)

- objectifs
- JavaFX CSS
- mise-en-oeuvre
- Syntaxe
- CSS pour objets Layout

Objectifs

- les feuilles de style CSS facilitent la présentation des composants graphiques de JavaFX
  - elles sont déjà largement utilisés pour la présentation de pages HTML ou XML
- l'objectif majeur est de permettre la mise en forme de composants graphiques séparément de leur construction
  - couleurs, polices de caractères, styles, positionnement,..
  - les feuilles de style CSS se trouvent dans des fichiers distincts d'extension .css

JavaFX CSS

- JavaFX Cascading Style Sheet (CSS) s'appuie sur CSS 2.1 et CSS 3
  - comporte des extensions spécifiques à JavaFX
- la feuille de style par défaut des applications JavaFX est **`modena.css`**
  - se trouvant dans le fichier **`jfxrt.jar`** de JavaFX
- guide de référence complet à l'URL:
  http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Mise-en-oeuvre

- les styles CSS s'appliquent aux noeuds de l'arborescence graphique
  - ils s'appliquent en premier aux noeuds parents, puis aux noeuds enfants
  - un noeud ne reçoit le style CSS qu'après son ajout à l'arborescence graphique
  - le style CSS est ré-appliqué en cas de modification du style CSS

Mise-en-oeuvre

- les styles CSS sont appliqués au choix:
  - via la propriété **`style`** sur le composant (inline)

```
TilePane tile = new TilePane();
tile.setStyle("-fx-background-color: DAE6F3;");
```

  - via l'objet **`Scene`** par appel à sa méthode **`getStylesheets`**

```
scene.getStylesheets().add("vue/monstyle.css");
```

    - si le fichier CSS se trouve dans le fichier .jar :

```
String css = getClass().getResource(
   "/monstyle.css").toExternalForm();
scene.getStylesheets().addAll(css);
```

Syntaxe

- définition d'un style CSS
  - définir un style CSS consiste:
    - à lui donner un nom, appelé sélecteur (*selector*)
    - à lui associer un ensemble de règles placées entre accolades et appelées propriétés du style
  - exemple:

```
.custom-button {
    -fx-font: 16px "Serif";
    -fx-padding: 10;
    -fx-background-color: #CCFF99;
}
```

Syntaxe

- plusieurs types de sélecteurs peuvent être définis
  - sélecteurs de classe (*style classes selectors*)
    - le nom du sélecteur est prédéfini et commence par un point
    - son nom correspond à celui d'une classe graphique JavaFX en minuscules
      - il comporte éventuellement des tirets pour chaque mot du nom de la classe correspondante
  - exemple:

```
.button { …}
.check-box { …}
.scroll-bar { …}
```

Syntaxe

- sélecteurs d'identifiants (*ID style selectors*)
  - le nom du sélecteur commence par un #
  - son nom correspond à celui d'un identifiant **`id`** d'un objet graphique JavaFX
- exemple:
  - dans le code:

```
Button bok = new Button("OK");
bok.setId("bouton-ok");
```

  - dans css:

```
#bouton-ok {
    -fx-background-color: blue;
}
```

Syntaxe

- sélecteurs d'éléments fils

```
.hbox > .button {
    -fx-text-fill: black;
}
```

  - le sélecteur opère sur des boutons fils directs d'une HBox
- sélecteurs d'éléments descendants

```
.hbox .button {
    -fx-text-fill: black;
}
```

  - le sélecteur opère sur des boutons descendants d'une HBox

Syntaxe

- sélecteurs de pseudo-classes (*pseudo-classes selectors*)
  - le nom du sélecteur commence par un point
  - son nom correspond à celui d'une classe graphique JavaFX en minuscules
  - l'état du noeud suit le nom, précédé d'un :
  - exemple:

```
.radio-button:focused
.radio-button:hover
.scroll-bar:vertical
```

Syntaxe

- règles et propriétés
  - le nom des propriétés CSS commence par **`-fx-`** suivi du nom d'une propriété Java du composant
    - exemple:

```
-fx-background-color: #333333;
-fx-text-fill: white;
-fx-alignment: CENTER;
```

Syntaxe

- le style CSS nommé **`.root`** s'applique au noeud racine de l'instance de **`Scene`**
  - tous les noeuds de l'arborescence graphique en bénéficient
  - exemple:

```
.root{
   -fx-font-size: 16pt;
   -fx-font-family: "Courier New";
   -fx-base: rgb(132, 145, 47);
   -fx-background: rgb(225, 228, 203);
}
```

Syntaxe

- une propriété du style **`.root`** peut être utilisée dans le style d'un autre composant

```
.button {
   -fx-background-color: -fx-background;
}
```

Syntaxe

- redéfinir un style pour la classe **`Button`**
  - exemple:

```
.button{
    -fx-text-fill: rgb(49, 89, 23);
    -fx-border-color: rgb(49, 89, 23);
    -fx-border-radius: 5;
    -fx-padding: 3 6 6 6;
}
```

Syntaxe

- définir un nouveau style de classe applicable à la demande
  - exemple:

```
.button1{
    -fx-text-fill: #006464;
    -fx-background-color: #DFB951;
    -fx-border-radius: 20;
    -fx-background-radius: 20;
    -fx-padding: 5;
}
```

  - pour associer ce nouveau style à un bouton:

```
Button buttonAccept = new Button("OK");
buttonAccept.getStyleClass().add("button1");
```

Syntaxe

- définir un nouveau style ID applicable à la demande
  - exemple:

```
#font-button {
    -fx-font: bold italic 20pt "Arial";
    -fx-effect: dropshadow( one-pass-box,black,8,0.0,2,0);
}
```

  - pour associer ce nouveau style à un bouton:

```
Button buttonFont = new Button("Font");
buttonFont.setId("font-button");
```

Syntaxe

- le style peut être associé à un composant de façon programmée
  - exemple:

```
Button buttonColor = new Button("Color");
buttonColor.setStyle(
 "-fx-background-color: slateblue; -fx-text-fill: white;"
);
```

  - le style associé de cette manière est prioritaire sur celui fourni par une CSS

Syntaxe

- le même style peut être appliqué à plusieurs sélecteurs

```
.label,
.text {
    -fx-font-size: 20px;
}
```

CSS pour objets layout

- les objets layout ne comportent pas de style par défaut
  - contrairement aux autres composants graphiques
  - il faut donc créer un style et l'associer au layout
- exemple:

```
.hbox {
    -fx-background-color: #2f4f4f;
    -fx-padding: 15;
    -fx-spacing: 10;
}
```

  - dans le code de l'application:

```
HBox hbox = new HBox();
hbox.getStyleClass().add("hbox");
```

CSS pour objets layout

- plusieurs styles peuvent être associés à un layout
  - pour cumuler des styles communs à tous les layouts et des styles propres au layout

```
.pane {
      -fx-background-color: #8fbc8f;
}
```

  - dans le code de l'application:

```
HBox hbox = new HBox();
hbox.getStyleClass().addAll("pane", "hbox");
```

CSS pour objets layout

- certains styles peuvent être redéfinis

```
#hbox-custom {
   -fx-background-radius: 5.0;
   -fx-padding: 8;
}
```

  - dans le code de l'application:

```
HBox hb = new HBox();
hb.getStyleClass().add("hbox");
hb.setId("hbox-custom");
```

CSS pour objets layout

- exemple:
  - pour **`GridPane`**

```
.grid {
    -fx-background-color: white;
    -fx-background-radius: 5.0;
    -fx-background-insets: 0.0 5.0 0.0 5.0;
    -fx-padding: 10;
    -fx-hgap: 10;
    -fx-vgap: 10;
}
```

CSS pour objets layout

- exemple:
  - pour **`FlowPane`** ou **`TilePane`**

```
.pane {
    -fx-background-color: #8fbc8f;
}
.grid {
    -fx-background-color: white;
    -fx-background-radius: 5.0;
    -fx-background-insets: 0.0 5.0 0.0 5.0;
    -fx-padding: 10;
    -fx-hgap: 10;
    -fx-vgap: 10;
}
```

CSS pour objets layout

- exemple:
  - pour **Vbox** ou **HBox**

```
.vbox {
   -fx-background-image:
   url("graphics/arrow_t_up_right.png");
   -fx-background-size: 96, 205;
   -fx-background-repeat: no-repeat;
   -fx-border-color: #2e8b57;
   -fx-border-width: 2px;
   -fx-padding: 10;
   -fx-spacing: 8;
}
```

CSS pour objets layout

- exemple:
  - pour **Vbox** ou **HBox**

```
#hbox-custom {
   -fx-background-image: url("graphics/cloud.png");
   -fx-background-size: 64,64;
   -fx-padding: 18 4 18 6;
   -fx-spacing: 25;
   -fx-background-radius: 5.0;
}
```

Multi-threading et concurrence

Multi-threading et concurrence

- threads JavaFX
- méthodes de la classe Application
- package javafx.concurrent
- interface Worker
- classe Task
- classe Service
- classe ScheduledService
- classe WorkerStateEvent

Threads JavaFX

- au moins deux des threads suivants s'exécutent à un instant donné dans une application JavaFX:
  - JavaFX application thread
  - Prism render thread
  - Media thread

JavaFX application thread

- le *javaFX application thread* est le thread principalement utilisé lors du développement
  - toute scène active, c'est-à-dire appartenant à une fenêtre doit manipulée dans ce thread
- une arborescence graphique (scene graph) peut être créée et manipulée dans un autre thread en tâche de fond, mais quand son noeud racine est attaché à un object actif dans la scène, tout noeud de l'arborescence doit être manipulé dans le thread d'application JavaFX

JavaFX application thread

- pour exécuter dans le *javaFX application thread* depuis un autre thread

```
Platform.runLater(new Runnable() {
    @Override public void run() {
        // cette méthode est exécutée dans le JavaFX AT
        initFX(fxPanel);
    } });
```

- le thread d'application JavaFX est différent du thread EDT de Swing
  - des précautions doivent être prises lorsque du code JavaFX est intégré dans une application Swing

Prism render thread

- le *Prism render thread* gère le rendu graphique séparément de l'*event dispatcher*
  - il assure le rendu de la trame N pendant que la trame N+1 est en cours de traitement
  - cela est particulièrement efficace sur les machines multi-processeurs
- le *Prism render thread* peut aussi comporter d'autres threads de rasterization s'exécutant en tâche de fond

Media thread

- le *media thread* s'exécute en tâche de fond et synchronise la dernière trame avec l'arborescence graphique en faisant appel au *javaFX application thread*

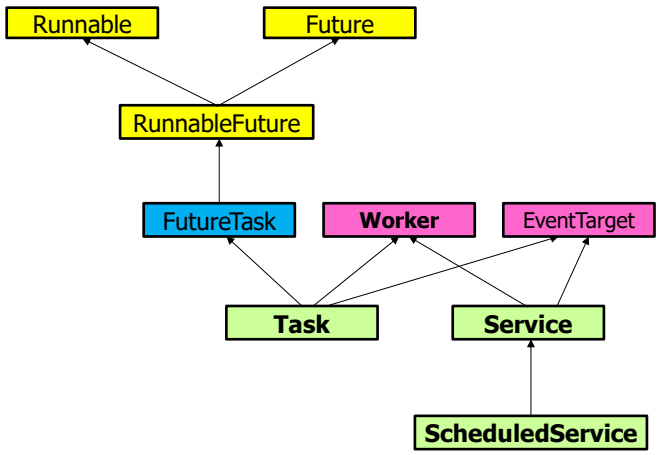
Méthodes de la classe Application

- méthodes de **`Application`**:
  - **`init()`**
    - méthode appelée depuis le thread de lancement, qui n'est pas le thread application JavaFX
    - cette méthode ne doit pas créer une instance de **`Scene`** ou de **`Stage`**
  - **`start(Stage)`**
    - méthode appelée depuis le thread application JavaFX
  - **`stop()`**
    - méthode appelée depuis le thread application JavaFX

Package javafx.concurrent

- afin de conserver la réactivité d'une application JavaFX, aucune tâche de longue durée ne doit s'exécuter dans le *JavaFX Application Thread*
- le package **`javafx.concurrent`** comporte:
  - l'interface **`Worker`**
  - la classe **`Task`**
    - implémente **`Worker`**
  - la classe **`Service`**
    - implémente **`Worker`**
  - la classe **`WorkerStateEvent`**

Package javafx.concurrent



Interface Worker

- l'interface **`Worker`** représente un objet qui accomplit un travail en tâche de fond dans un ou plusieurs threads
  - l'état du **`Worker`** est visible du JavaFX AT
  - cycle de vie:
    - à tout instant, ce thread peut être interrompu par sa méthode **`cancel`**
  - l'état d'avancement du **`Worker`** peut être obtenu par trois propriétés:
    - **`totalWork`**, **`workDone`**, et **`progress`**

Classe Task

- la classe **`Task`** est une implémentation d'un **`Worker`**
  - toute sous-classe de **`Task`** devra redéfinir la méthode **`call`**
    - cette méthode **`call`** ne devra pas accéder à l'objet **`Scene`**
  - la classe **`Task`** définit un objet jetable qui ne doit pas être réutilisé
    - dans le cas contraire, utiliser la classe **`Service`**

Classe Task

- une **`Task`** peut être démarrée au choix:
  - en démarrant un thread ayant cette tâche en argument:

```
Thread th = new Thread(task);
th.setDaemon(true);
th.start();
```

  - en faisant appel à un **`ExecutorService`**:

```
ExecutorService.submit(task);
```

Classe Task

- afin qu'une tâche puisse être arrêtée par appel à sa méthode **`cancel()`**, elle doit périodiquement faire appel à sa méthode **`isCancelled()`** et terminer son traitement s'il y a lieu

```
Task<Integer> task = new Task<Integer>() {
   @Override protected Integer call() throws Exception {
      int iter;
      for (iter = 0; iter < 100000; iter++) {
         if (isCancelled()) { break;}
         System.out.println("Iterations " + iter);
      }
      return iter;
   }
};
```

Classe Task

- exemple: affichage d'une barre de progression

```
Task task = new Task<Void>() {
   @Override public Void call() {
       static final int max = 1000000;
       for (int i=1; i<=max; i++) {
           if (isCancelled()) {break;}
           updateProgress(i, max);
       }
       return null;
   }
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

Classe Service

- la classe **`Service`** permet d'exécuter un objet **`Task`** dans ou plusieurs threads en tâche de fond
  - l'accès aux méthodes de la classe **`Service`** doit être effectué depuis le thread JavaFX application exclusivement
    - l'objectif de cette classe est d'aider le développeur à mettre en oeuvre l'interaction entre les threads en tâche de fond et le thread JavaFX application
  - la classe **`Service`** définit un objet réutilisable, qui peut être ré-initialisé et redémarré

Classe Service

- une instance de **`Service`** peut être exécutée au choix:
  - par un objet **`Executor`**
  - par un objet **`Executor`** spécialisé comme **`ThreadPoolExecutor`**
  - par un thread démon, si aucun **`Executor`** n'est précisé

Classe Service

- exemple:

```
private static class FirstLineService
    extends Service<String> {
    private StringProperty url =
                  new SimpleStringProperty();
    public final void setUrl(String value) {
           url.set(value);
    }
    public final String getUrl() {
           return url.get();
    }
    public final StringProperty urlProperty() {
       return url;
    }
```

Classe Service

- exemple (suite):

```
@Override protected Task<String> createTask() {
    final String _url = getUrl();
    return new Task<String>() {
        protected String call() throws Exception {
            URL u = new URL(_url);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(u.openStream()));
            String result = in.readLine();
            in.close();
            return result;
        }
    };
}
}
```

Classe ScheduledService

- la classe **`ScheduledService`** représente un **`Service`** qui redémarre automatiquement après exécution réussie
  - dans certains cas, il peut également redémarrer en cas d'éxécution avec échec
  - un exemple d'utilisation concerne la scrutation périodique de données

Classe ScheduledService

- cycle de vie:
  - après création, un objet **`ScheduledService`** est dans l'état **`READY`**
  - après appel à sa méthode **`start()`** ou **`restart()`**, l'objet **`ScheduledService`** passe dans l'état **`SCHEDULED`** pour la durée précisée dans la propriété **`delay`**
  - dans l'état **`RUNNING`**, l'objet **`ScheduledService`** exécute la tâche

Classe ScheduledService

- cycle de vie (suite):
  - si la tâche s'exécute avec succès:
    - l'objet **`ScheduledService`** passe dans l'état **`SUCCEEDED`**, puis dans l'état **`READY`**, puis à nouveau dans l'état **`SCHEDULED`**
      - la durée pendant laquelle il reste dans l'état **`SCHEDULED`** dépends de l'instant auquel il est passé dernièrement dans l'état **`RUNNING`**, et de la valeur de la propriété **`period`**, qui définit la durée minimum entre deux exécutions successives
    - si l'exécution se termine avant la fin de la période, l'objet **`ScheduledService`** reste dans l'état **`SCHEDULED`** jusqu'à l'expiration de la période, sinon, il passe dans l'état **`RUNNING`**

Classe ScheduledService

- cycle de vie (suite):
  - si la tâche s'exécute avec échec:
    - l'objet **`ScheduledService`** passe dans l'état **`FAILED`**, puis redémarre ou non, selon la valeur des propriétés:
      - **`restartOnFailure`**, **`backoffStrategy`**, **`maximumFailureCount`**
    - **`restartOnFailure==false`**
      - l'objet **`ScheduledService`** passe dans l'état **`FAILED`** et s'arrête. Il peut être relancé manuellement
    - **`restartOnFailure==true`**
      - l'objet **`ScheduledService`** passe dans l'état **`SCHEDULED`** et y reste pendant la durée indiquée par la propriété **`cumulativePeriod`**

Classe ScheduledService

- cycle de vie (suite):
  - l'objet **`ScheduledService`** passe dans l'état **`SCHEDULED`** et y reste pendant la durée indiquée par la propriété **`cumulativePeriod`** (obtenue par appel à la méthode **`backoffStrategy()`**)
  - via la propriété **`cumulativePeriod`** on peut forcer l'objet **`ScheduledService`** à attendre plus longtemps avant le prochain redémarrage
  - après exécution réussie, la propriété **`cumulativePeriod`** est réinitialisée la celle de la période
  - lorsque le nombre d'échec successifs atteint la valeur indiquée dans la propriété **`maximumFailureCount`**, l'objet **`ScheduledService`** passe dans l'état **`FAILED`** et se termine

Classe WorkerStateEvent

- la classe **`WorkerStateEvent`** représente un évènement qui se produit losque l'état d'une implémentation de **`Worker`** change
  - les classes **`Task`** et **`Service`** implémentent l'interface **`EventTarget`** et supportent par conséquent la réception d'évènements **`WorkerStateEvent`**

Classe WorkerStateEvent

- chaque fois que l'état d'un **`Worker`** change, une instance de **`WorkerStateEvent`** est créée
  - exemple:
    - si un objet **`Task`** passe vers l'état **`SUCCEEDED`**, l'évènement **`WORKER_STATE_SUCCEEDED`** est créé, le handler **`onSucceeded()`** est déclenché, entraînant l'appel à la méthode **`succeeded()`** dans le thread JavaFX application
  - par commodité, un jeu de méthodes (à redéfinir) permet de détecter les transitions d'un **`Worker`**
    - **`succeeded`**, **`cancelled`**, **`failed`**, **`running`**, **`scheduled`**

Classe WorkerStateEvent

- exemple:

```
Task<Integer> task = new Task<Integer>() {
   @Override protected Integer call() throws Exception {
      int iterations = 0;
      for (iterations=0; iterations<100000; iterations++) {
         if (isCancelled()) {
                break;
         }
         System.out.println("Iteration " + iterations);
      }
      return iterations;
   }
```

Classe WorkerStateEvent

- exemple (suite):

```
   @Override protected void succeeded() {
      super.succeeded();
      updateMessage("Done!");
   }
   @Override protected void cancelled() {
      super.cancelled();
      updateMessage("Cancelled!");
   }
   @Override protected void failed() {
      super.failed();
      updateMessage("Failed!");
   }
};
```

Interopérabilité avec Swing

Interopérabilité avec Swing

- intégrer JavaFX dans une application Swing
- intégrer Swing dans une application JavaFX

Intégrer JavaFX dans Swing

- bénéfices procurés par l'intégration de JavaFX dans une application Swing:
  - FXML
  - SceneBuilder
  - support de CSS
  - support de JavaFX Media
  - animations
  - contenu HTML

Intégrer JavaFX dans Swing

- JavaFX fournit la classe **`JFXPanel`** du package **`javafx.embed.swing`** qui permet d'inclure du contenu JavaFX dans une application Swing
- dans toute application Swing, l'interface graphique est créé dans le thread EDT (*event Dispatch Thread*)
- dans JavaFX, l'instance de **`Scene`** contenant l'arborescence graphique doit être créée dans le thread JavaFX AT (*JavaFx Application Thread*)

Intégrer JavaFX dans Swing

- exemple:

```
public class JavaFX_inside_Swing {
   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
         @Override public void run() {
            // cette méthode est exécutée
            // dans le thread EDT
            initAndShowGUI();
         }
      });
   }
```

Intégrer JavaFX dans Swing

- exemple:

```
// cette méthode est exécutée dans le thread EDT
private static void initAndShowGUI() {
   JFrame frame = new JFrame("Swing and JavaFX");
   final JFXPanel fxPanel = new JFXPanel();
   frame.add(fxPanel);
   frame.setSize(300, 200);
   frame.setVisible(true);
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   Platform.runLater(new Runnable() {
      @Override public void run() {
         // cette méthode est exécutée dans le JavaFX AT
         initFX(fxPanel);
      } });
}
```

Intégrer JavaFX dans Swing

- exemple (suite):

```
// cette méthode est exécutée dans le JavaFX AT
private static void initFX(JFXPanel fxPanel) {
    Scene scene = createScene();
    fxPanel.setScene(scene);
}
private static Scene createScene() {
    Group root = new Group();
    Scene scene = new Scene(root, Color.ALICEBLUE);
    Text text = new Text();
    text.setX(40); text.setY(100);
    text.setFont(new Font(25));
    text.setText("bienvenue JavaFX!");
    root.getChildren().add(text);
    return (scene);
}
```

Intégrer JavaFX dans Swing

- la coexistence de code JavaFX et de code Swing dans une même application peut entraîner deux situations:
  - un changement de valeur d'une donnée JavaFX est déclenchée par un changement de valeur d'une donnée Swing
  - un changement de valeur d'une donnée Swing est déclenchée par un changement de valeur d'une donnée JavaFX

Intégrer JavaFX dans Swing

- modification d'une donnée JavaFX en réponse à un changement de valeur d'une donnée Swing:
  - les données JavaFX ne doivent être accédées que dans le thread JavaFX AT
    - chaque fois qu'une donnée JavaFX doit être modifiée, placer le code dans un objet **`Runnable`** transmis à la méthode **`Platform.runLater()`**

Intégrer JavaFX dans Swing

- exemple:

```
jbutton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      Platform.runLater(new Runnable() {
         @Override public void run() {
            fxlabel.setText("bouton Swing cliqué!");
         }
      });
   }
});
```

Intégrer JavaFX dans Swing

- modification d'une donnée Swing en réponse à un changement de valeur d'une donnée JavaFX:
  - les données Swing ne doivent être modifiées que dans l'EDT
    - chaque fois qu'une donnée Swing doit être modifiée, placer le code dans un objet **`Runnable`** transmis à la méthode **`SwingUtilities.invokeLater()`**

```
SwingUtilities.invokeLater(new Runnable() {
   @Override public void run() {
      //Code modifiant une donnée Swing
}});
```

Intégrer Swing dans JavaFX

- JavaFX fournit la classe **`SwingNode`** du package **`javafx.embed.swing`** qui permet d'inclure du contenu Swing dans une application JavaFX
  - le contenu de l'objet **`SwingNode`** est transmis à sa méthode **`setContent`** sous la forme d'un objet de la classe **`javax.swing.JComponent`** class
  - la méthode **`setContent`** peut être appelée soit du thread EDT soit du thread JavaFX AT
    - mais l'accès aux composants Swing doit toujours être effectué dans le thread EDT

Intégrer Swing dans JavaFX

- exemple:

```
public class SwingFx extends Application {
   @Override public void start (Stage stage) {
      // exécution dans le thread JavaFX AT
      final SwingNode swingNode = new SwingNode();
      createSwingContent(swingNode);
      StackPane pane = new StackPane();
      pane.getChildren().add(swingNode);
      stage.setTitle("Swing in JavaFX");
      stage.setScene(new Scene(pane, 250, 150));
      stage.show();
   }
```

Intégrer Swing dans JavaFX

- exemple (suite):

```
   private void createSwingContent(
         final SwingNode swingNode) {
      SwingUtilities.invokeLater(new Runnable() {
         @Override public void run() {
            // exécution dans le thread EDT
            swingNode.setContent(new JButton("OK!"));
         }
      });
   }
}
```

Intégrer Swing dans JavaFX

- la gestion d'évènements Swing peut faire appel à des composants JavaFX
- exemple:

```
// exécution de actionPerformed dans le thread EDT
@Override public void actionPerformed(ActionEvent e) {
   if ("disable".equals(e.getActionCommand())) {
      Platform.runLater(new Runnable() {
         // exécution de run dans le thread JavaFx AT
         @Override public void run() {
                EnableFXButton.fxbutton.setDisable(true);
         }
       });
   }
}
```

Intégrer Swing dans JavaFX

- la gestion d'évènements JavaFX peut faire appel à des composants Swing
- exemple:

```
// exécution de setOnAction dans le thread JavaFX AT
fxbutton.setOnAction(
    new javafx.event.EventHandler<ActionEvent>() {
       @Override public void handle(ActionEvent e) {
          SwingUtilities.invokeLater(new Runnable() {
             // exécution de run dans le thread EDT
             @Override public void run() {
                ButtonHtml.b1.setEnabled(true); }
          });
       fxbutton.setDisable(true);
    } });
```

Contenu HTML

Contenu HTML

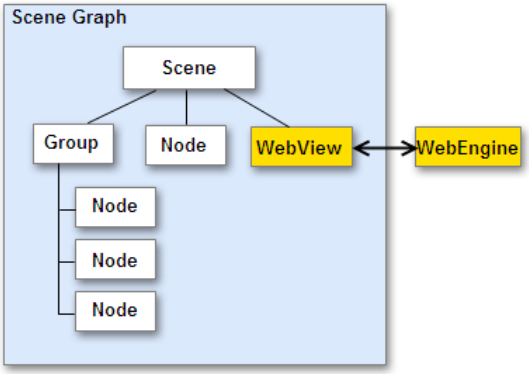
- présentation
- architecture
- mise-en-oeuvre
- exécution de code JavaScript

Présentation

- une application JavaFX peut embarquer un navigateur web afin de:
  - visualiser des pages HTML locales ou distantes
  - exécuter des commandes JavaScript
  - obtenir l'historique de navigation
  - gérer les fenêtres pop-up
  - appliquer des effets sur le navigateurs
- le navigateur embarqué hérite de la classe **`Node`**, comme tout contrôle UI

Architecture

- **`WebView`** et **`WebEngine`**



Architecture

- les deux classes utilisées sont:
  - **`WebEngine`**
    - assure l'interaction avec l'utilisateur via les liens hypertextes et les formulaires
    - assure les fonctionnalités essentielles liées au décodage HTML et l'accès à DOM, ainsi que l'exécution des commandes JavaScript
    - ne gère qu'une page à la fois
  - **`WebView`**
    - encapsule un objet **`WebEngine`**
    - implémente l'interface **`Node`**

Mise-en-oeuvre

- exemple:

```
public class WebViewSample extends Application {
   private Scene scene;
   @Override public void start(Stage stage) {
       // create the scene stage.setTitle("Web View");
       scene = new Scene(new Browser(),900,600,
           Color.web("#666970"));
       stage.setScene(scene);
       scene.getStylesheets().add("webview/Toolbar.css");
       stage.show();
   }
   public static void main(String[] args){
       launch(args);
   }
}
```

Mise-en-oeuvre

- exemple (suite)

```
class Browser extends Region {
    final WebView browser = new WebView();
    final WebEngine webEngine = browser.getEngine();

    public Browser() {
        getStyleClass().add("browser");
        webEngine.load("http://www.alpha.com/beta/index.htm");
        getChildren().add(browser);
    }
    private Node createSpacer() {
        Region spacer = new Region();
        HBox.setHgrow(spacer, Priority.ALWAYS);
        return spacer;
    }
```

Mise-en-oeuvre

- exemple (suite):

```
    @Override protected void layoutChildren() {
        double w = getWidth();
        double h = getHeight();
        layoutInArea(browser,0,0,w,h,0, HPos.CENTER,
            VPos.CENTER);
    }
    @Override
    protected double computePrefWidth(double height) {
        return 900;
    }
    @Override
    protected double computePrefHeight(double width) {
        return 600;
    }
}
```

Exécution de code JavaScript

- la méthode **`executeScript()`** de la classe **`WebEngine`** permet d'exécuter toute commande JavaScript déclarée dans la page HTML chargée

```
public Object executeScript(String script);
```

  - exemple:

```
showPrevDoc.setOnAction(new EventHandler() {
    @Override public void handle(Event t) {
        webEngine.executeScript(
                "toggleDisplay('PrevRel')«  );
    }
});
```

Développer en FXML

Développer en FXML

- présentation
- chargement du document FXML
- développer avec SceneBuilder
- gestion d'évènements
- syntaxe
- propriétés
- data binding
- composants personnalisés

Présentation

- le langage FXML permet de représenter en XML un graphe d'objets graphiques
  - il permet de séparer la présentation de la logique applicative
    - FXML permet également de créer des objets de service ou de domaine
  - en architecture MVC (Model View Controller) :
    - la vue est représentée par un ou plusieurs document FXML
    - le contrôleur est une classe Java
    - le modèle est constitué d'objets du domaine
  - FXML ne comporte pas de schéma

Présentation

- FXML fait correspondre des éléments XML à des instructions Java
- exemple:

FXML

```
<BorderPane>
   <top><Label text="Page Title"/></top>
   <center><Label text="Some data here"/></center>
</BorderPane>
```

```
BorderPane border = new BorderPane();
Label toppanetext = new Label("Page Title");
border.setTop(toppanetext);
Label centerpanetext = new Label ("Some data here");
border.setCenter(centerpanetext);
```

Présentation

- l'élément racine d'un document FXML est une balise qui correspond à un *layout*, au choix du développeur
- l'espace de noms des balises FXML est:
  `http://javafx.com/javafx/8`
- le préfixe **`fx`** fréquemment rencontré dans les attributs correspond à l'espace de noms:
  `http://javafx.com/fxml/1`

Présentation

- exemple:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>
<BorderPane prefHeight="400.0" prefWidth="600.0"
     xmlns:fx="http://javafx.com/fxml/1"
     xmlns="http://javafx.com/javafx/8">
   <top>
      <MenuBar BorderPane.alignment="CENTER">
        <menus>
          <Menu mnemonicParsing="false" text="File">
            <items>
              <MenuItem mnemonicParsing="false" text="Close" />
            </items>
          </Menu>
          …
      </MenuBar>
   </top>
</BorderPane>
```

Chargement du document FXML

- le chargement d'un document FXML fait appel à la classe `javafx.fxml.FXMLLoader`

```
Parent root = FXMLLoader.load(
       getClass().getResource("Login.fxml"));
```

ou

```
FXMLLoader loader = new FXMLLoader(
       getClass().getResource("Login.fxml"));
Parent root = (Parent) loader.load();
```

Développer avec SceneBuilder

- SceneBuilder est un éditeur WYSIWYG de développement d'applications JavaFX en FXML
  - les modifications du fichier FXML apparaissent dans l'éditeur graphique de SceneBuilder
  - les modifications dans l'éditeur graphique entraînent des modifications du fichier FXML
- des composants graphiques personnalisés peuvent être ajoutés à la palette standard

Développer avec SceneBuilder

- SceneBuilder peut fonctionner:
  - en mode standalone
  - intégré à un IDE comme NetBeans ou E(fx)clipse
- SceneBuilder est multi-plateformes
  - Windows
  - Linux
  - Mac OS
- SceneBuilder est téléchargeable depuis:
  https://gluonhq.com/products/scene-builder/

Syntaxe

- dans FXML, un élément représente:
  - une instance de classe
  - une propriété
  - une propriété static
  - un bloc *define*
  - un bloc d'instructions en script

Instances

- instance de classe
  - le nom de l'élément doit commencer par une majuscule, et représente la classe de l'objet

```
<javafx.scene.control.Label text="Bonjour !" />
```

  - en pratique, on importe préalablement la classe en début de fichier

```
<?import javafx.scene.control.Label?>
…
        <Label text="Bonjour !"/>
```

Instances

- les classes peuvent également être instanciées en faisant appel à un *builder*
  - utile pour les classes *Immutable*
  - FXML fournit deux interfaces:

```
public interface Builder<T> {
   public T build();
}
```

  - et

```
public interface BuilderFactory {
   public Builder<?> getBuilder(Class<?> type);
}
```

Instances

- une fabrique de builder par défaut est fournie dans JavaFX:

```
public final class JavaFXBuilderFactory
         implements BuilderFactory {
   JavaFXBuilderFactory()
   JavaFXBuilderFactory(ClassLoader classLoader);
   Builder<?> getBuilder(java.lang.Class<?> type);
}
```

  - cette fabrique permet de créer et configurer la plupart des types JavaFX *immutable*
  - exemple:

```
<Color red="1.0" green="0.0" blue="0.0"/>
```

fx:controller

- l'attribut **`fx:controller`** dans l'élément racine permet d'indiquer la classe du contrôleur associé à la vue
- exemple:

```
<BorderPane fx:id="mainBorderPane"
prefHeight="400.0" prefWidth="700.0"
xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.dma.vue.MainControleur" >
```

fx:id

- l'attribut **`fx:id`** permet de donner à un objet un identifiant unique
  - cet identifiant permet au contrôleur d'interagir avec le composant graphique
- exemple:

```
<Rectangle fx:id="rectangle" x="10" y="10" width="320"
height="240" fill="#ff0000"/>
```

Gestion d'évènements

- FXML permet de spécifier deux sortes de gestionnaire d'évènements:
  - gestionnaire d'évènement par script
  - gestionnaire d'évènement par contrôleur

Gestion d'évènements par script

- un attribut correspondant à l'évènement à gérer définit le code à exécuter
  - syntaxe: **`onXxxx`**
    - exemple:

```
<?language javascript?>
 ...
<VBox>
    <children>
       <Button text="OK" onAction=
       "java.lang.System.out.println('cliqué!');" />
    </children>
</VBox>
```

Gestion d'évènements par script

- le code à exécuter peut être précisé dans un élement **`fx:script`**
  - exemple:

```
<?language javascript?>
 ...
<fx:script>
    function f1() {
           java.lang.System.out.println("fermeture");
    };
</fx:script>
<VBox>
    <children>
       <Button text="Close" onAction= "f1();" />
    </children>
</VBox>
```

Gestion d'évènements par contrôleur

- le contrôleur doit préalablement être déclaré au moyen de l'attribut **`fx:controller`** dans l'élément racine du document FXML
  - exemple:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
       fx:controller="login.LoginController"
       alignment="center" hgap="10" vgap="10" >
```

  - il peut également être chargé directement via le loader:

```
loader.setController(new login.LoginController());
```

- il est ensuite accessible via **`getController()`** :

```
LoginController controller = (LoginController)
       loader.getController();
```

Gestion d'évènements par contrôleur

- un contrôleur doit respecter quelques règles:
  - le contrôleur est instancié au chargement du document FXML
    - il doit comporter un constructeur public sans argument (ou aucun constructeur)
  - le chargeur FXML recherche automatiquement si le contrôleur comporte des attributs accessibles dont les noms correspondent à des identifiants **`fx:id`** d'objets graphiques déclarés en FXML
    - ces attributs deviendront alors des références sur ces objets graphiques

Gestion d'évènements par contrôleur

- exemple:

Document FXML

```
<Text fx:id="actiontarget"
GridPane.columnIndex="0"GridPane.columnSpan="2"
GridPane.halignment="RIGHT"  GridPane.rowIndex="6"/>
…
```

correspondance (nom et type): le contrôleur possède alors automatiquement une référence sur l'objet graphique de type Text

Contrôleur

```
public class LoginController {
    public Text actiontarget;
    …
```

Gestion d'évènements par contrôleur

- l'annotation **`@FXML`** rend un membre en accès **`private`** ou **`protected`** dans le contrôleur accessible depuis un document FXML
  - inutile si le membre est en accès **`public`**
  - exemple:

```
public class LoginController {
    @FXML private Text actiontarget;
    @FXML protected void handleSubmitButtonAction(
           ActionEvent event) {
        actiontarget.setText("Sign in button pressed");
    }
}
```

Gestion d'évènements par contrôleur

- le gestionnaire d'évènement d'un contrôleur peut être associé à un objet graphique de deux façons:
  1. en étant précisé dans le document FXML au moyen d'un attribut **`onXxxx`** sur l'objet graphique (comme pour les scripts)
  2. en étant associé dans le contrôleur à l'objet graphique

Gestion d'évènements par contrôleur

1. le gestionnaire d'évènements du contrôleur est précisée dans le document FXML précédée de #
   - exemple:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
        fx:controller="login.LoginController"
        alignment="center" hgap="10" vgap="10" >
        ...
        <Text fx:id="actiontarget" text="Sign In"
              onAction="#handleSubmitAction" />
```

   - **handleSubmitAction** désigne une méthode du contrôleur **LoginController**

Gestion d'évènements par contrôleur

- exemple (suite):

```
package login;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class LoginController {
    @FXML private Text actiontarget;

    @FXML private void handleSubmitAction(ActionEvent event){
        actiontarget.setText("Sign in button pressed");
    }
}
```

Gestion d'évènements par contrôleur

2. le gestionnaire d'évènements du contrôleur est associé à l'objet graphique dans la méthode **`initialize()`**
   - appelée une seule fois lorsque le document FXML est chargé pour compléter les traitements déjà effectués
   - exemple:

```
public class LoginController {
    @FXML private Button bouton;
    @FXML private void initialize(){
        bouton.setOnAction(
           evt->System.out.println("cliqué !");
    }
}
```

fx:define

- l'élément **`fx:define`** permet de créer des objets en dehors de la hiérarchie d'objet
  - ils possèdent en général un identifiant afin d'être référencé ailleurs dans le document FXML
- exemple:

```
<VBox>
   <fx:define><ToggleGroup fx:id="TG"/></fx:define>
   <children>
      <RadioButton text="A" toggleGroup="$TG"/>
      <RadioButton text="B" toggleGroup="$TG"/>
   </children>
</VBox>
```

fx:value

- l'attribut **`fx:value`** permet de créer une instance de classe:
  - dépourvue de constructeur sans argument
  - pourvue d'une méthode *static* **`valueOf()`**
- exemples:

```
<String fx:value = "Bonjour !"/>
<Double fx:value = "1.0"/>
<Boolean fx:value = "false"/>
```

fx:factory

- l'attribut **`fx:factory`** permet de créer une instance de classe:
  - dépourvue de constructeur sans argument
  - pourvue d'une méthode *static* sans argument retournant l'instance
- exemple:

```
<FXCollections fx:factory = "observableArrayList">
   <String fx:value="A"/>
   <String fx:value="B"/>
   <String fx:value="C"/>
</FXCollections>
```

fx:include

- l'élément **`fx:include`** permet de créer une ou plusieurs instances de classe par inclusion d'un document FXML
- exemple:

```
<VBox xmlns:fx="http://javafx.com/fxml">
   <children>
      <fx:include source="bouton1.fxml"/>
   </children>
</VBox>
```

  - avec `bouton1.fxml` :

```
<Button text="OK"/>
```

fx:constant

- l'élément **`fx:constant`** crée une référence vers une constante de classe
  - exemple:

```
<Button>
   <minHeight>
      <Double fx:constant="NEGATIVE_INFINITY"/>
   </minHeight>
</Button>
```

fx:reference

- l'élément **`fx:reference`** permet de faire référence à un objet existant, défini ailleurs
  - son attribut **`source`** indique le nom de l'objet cible et doit correspondre à un attribut **`fx:id`** ou à une variable de script
  - exemple:

```
<ArrayList>
   <fx:reference source="element1"/>
   <fx:reference source="element2"/>
   <fx:reference source="element3"/>
</ArrayList>
```

fx:copy, fx:root

- l'élément **`fx:copy`** permet de copier un objet existant, défini ailleurs
  - son attribut **`source`** indique le nom de l'objet cible et doit correspondre à un attribut **`fx:id`** ou à une varaible de script
  - le type de la cible doit comporter un constructeur de copie
- l'élément **`fx:root`** permet de créer une référence vers un objet racine

fx:script

- l'élément **`fx:script`** permet d'insérer ou d'appeler un script dans un document FXML
  - des langages comme JavaScript, Groovy, Clojure sont couramment utilisés

fx:script

- exemple: appel d'un script

```
...
<VBox xmlns:fx="http://javafx.com/fxml">
    <fx:script source="exemple.js" />
    <children>
        <Button text="OK"
           onAction="handleButtonAction(event);"/>
    </children>
</VBox>
```

- script **`exemple.js`**:

```
importClass(java.lang.System);
function handleButtonAction(event) {
    System.out.println('cliqué!');
}
```

fx:script

- exemple: inclusion d'un script

```
<?language javascript?>
...
<VBox xmlns:fx="http://javafx.com/fxml">
   <fx:script>
      importClass(java.lang.System);
      function handleButtonAction(event) {
         System.out.println('cliqué!');
      }
   </fx:script>
   <children>
   <Button text="OK"
          onAction="handleButtonAction(event);"/>
   </children>
</VBox>
```

Propriétés

- les propriétés d'une instance sont initialisées au choix via:
  - un élément:

```
<Label>
   <text> Bonjour ! </text>
</Label>
```

  - un attribut :

```
<Label text = "Bonjour !"/>
```

  - les conversions vers les types primitifs sont réalisées automatiquement
    - la méthode (static) **`valueOf()`** permet d'étendre ce mécanisme

Propriétés

- propriétés de type List
  - une propriété de type **`List`** en lecture seule ne possède qu'une méthode *getter*

```
<Group xmlns:fx="http://javafx.com/fxml">
   <children>
      <Rectangle fx:id="rectangle" x="10" y="10"
      width="320" height="240" fill="#ff0000"/>
      ...
   </children>
</Group>
```

    - la propriété **`children`** de **`javafx.scene.Group`** correspond à la méthode *getter*:

```
public ObservableList<Node> getChildren()
```

Propriétés

- propriétés de type Map
  - une propriété de type **`Map`** en lecture seule ne possède qu'une méthode *getter*

```
<Button>
      <properties alpha="123" beta="456"/>
</Button>
```

    - la propriété **`properties`** de **`javafx.scene.Node`** dont hérite **`Button`** correspond à la méthode *getter*:

```
public final ObservableMap<Object,Object>
   getProperties()
```

Propriétés

- propriétés par défaut
  - une propriété par défaut est définie dans la classe au moyen de l'annotation **`@DefaultProperty`**

```
@DefaultProperty(value="children")
public class Pane extends Region {
    ….
}
```

```
<VBox xmlns:fx="http://javafx.com/fxml">
      <Button text="Click Me!"/> ...
</VBox>
```

    - l'élément **`<children>`** est ici facultatif car **`children`** est une propriété par défaut dans la classe **`Pane`** dont hérite **`VBox`**

Propriétés

- propriétés **`static`**
  - une telle propriété est liée à la classe et correspond à des méthodes *setter* et *getter* **`static`**

```
<Label text="My Label">
    <GridPane.rowIndex>0</GridPane.rowIndex>
    <GridPane.columnIndex>0</GridPane.columnIndex>
</Label>
```

    - est traduit en Java:

```
GridPane gridPane = new GridPane();
Label label = new Label();
label.setText("My Label");
GridPane.setRowIndex(label, 0);
GridPane.setColumnIndex(label, 0);
gridPane.getChildren().add(label);
```

Propriétés

- une propriété peut représenter une URL relative au document FXML courant pour accéder à une ressource
  - exemple:

```
<ImageView image="@mon_image.png"/>
```

    - les caractères *espace* doivent être encodés

```
<ImageView image="@mon%20image.png"/>
```

- une propriété peut prendre différentes valeurs en fonction de la langue utilisée (internationalisation)
  - exemple:

```
<Label text="%validate"/>
```

Propriétés

- une propriété peut prendre la valeur d'une variable
  - exemple:

```
<fx:define>
   <ToggleGroup fx:id="sexeGroup"/>
</fx:define> ...
<RadioButton text="M" toggleGroup="$sexeGroup"/>
<RadioButton text="F" toggleGroup="$sexeGroup"/>
```

- une propriété peut prendre une valeur qui commence par un caractère spécial
  - exemple:

```
<Label text="\$10.00"/>
```

Exemple

- exemple

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.text.Text?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.PasswordField?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.control.Button?>
<?import java.net.URL?>

<GridPane xmlns:fx="http://javafx.com/fxml"
    fx:controller="login.LoginController"
    alignment="center" hgap="10" vgap="10"
    styleClass="root">
   <padding>
      <Insets top="25" right="25" bottom="10" left="25" />
   </padding>
```

Exemple

- exemple (suite):

```
    <Text id="welcome-text" text="Welcome" GridPane.columnIndex="0"
    GridPane.rowIndex="0" GridPane.columnSpan="2" />
    <Label text="User Name:" GridPane.columnIndex="0" GridPane.rowIndex="1" />
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="1" />
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="2" />
    <PasswordField fx:id="passwordField" GridPane.columnIndex="1"
    GridPane.rowIndex="2" />
    <HBox spacing="10" alignment="bottom_right" GridPane.columnIndex="1"
    GridPane.rowIndex="4">
            <Button text="Sign In" onAction="#handleSubmitButtonAction"/>
    </HBox>
    <Text fx:id="actiontarget" GridPane.columnIndex="0"
      GridPane.columnSpan="2" GridPane.halignment="RIGHT"
      GridPane.rowIndex="6"/>
    <stylesheets>
            <URL value="@Login.css" />
    </stylesheets>
</GridPane>
```

Expression binding

- l'*expression binding* permet de lier (*bind*) la valeur d'une propriété à une variable ou à une expression
  - tout changement de valeur de la variable ou de l'expression est alors propagé dans la propriété
  - exemple:

```
<TextField fx:id="nom"/>
<Label text="${nom.text}"/>
```

    - le libellé est automatiquement mis-à-jour lorsque l'utilisateur saisit un nom dans le champ de texte

Expression binding

- les expressions peuvent faire intervenir des opérateurs:

"string" 'string'	A string constant
true false	A boolean constant
null	A constant representing the null value
50.0 3e5 42	A numerical constant
- (unary operator)	Unary minus operator, applied on a number
! (unary operator)	Unary negation of a boolean
+ - * / %	Numerical binary operators
&& ||	Boolean binary operators
> >= < <= == !=	Binary operators of comparison. Both arguments must be of type Comparable

Internationalisation

ResourceBundle

- l'internationalisation d'applications Java FX s'appuie sur les **`ResourceBundle`** de Java

```
Locale locale = new Locale("fr", "FR");
ResourceBundle bundle =
    ResourceBundle.getBundle("messages", locale);
```

  - les fichiers contenant les messages à afficher doivent se nommer :

`messages_fr_FR.properties` pour le français
`messages_en_EN.properties` pour l'anglais
`messages_es_ES.properties` pour l'espagnol

ResourceBundle

- contenu des fichiers contenant les messages à afficher:

fichier `messages-fr-FR.properties`

```
ui.button.text=Valider
...
```

fichier `messages-en_EN.properties`

```
ui.button.text=Validate
...
```

Chargement du document FXML

- **`FXMLLoader`** comporte des constructeurs et des méthodes **`load`** permettant de préciser le **`ResourceBundle`** utilisé

```
Locale locale = new Locale("fr", "FR");
ResourceBundle bundle =
ResourceBundle.getBundle("messages", locale);
//FXMLLoader loader = new FXMLLoader(
//getClass().getResource("Login.fxml"),resources);
FXMLLoader loader = new FXMLLoader();
loader.setResources(bundle);
loader.setLocation(getClass().
       getResource("Login.fxml"));
Parent root = (Parent) loader.load();
```

Accès au ResourceBundle

- depuis la vue
  - il suffit de préciser la propriété à afficher précédée de %, comme dans:

```
<Button text="%ui.button.validate"/>
```

- depuis le contrôleur
  - le **`ResourceBundle`** est directement injecté dans le contrôleur:

```
@FXML
private ResourceBundle resources;
```

    - cela permet au contrôleur d'obtenir les messages internationalisés:

```
String bienvenue = resources.getString("welcome");
```

Composants personnalisés

Composants personnalisés

- Présentation
- Composant personnalisé tout Java
- Composant personnalisé en FXML et Java
- Composant personnalisé par héritage

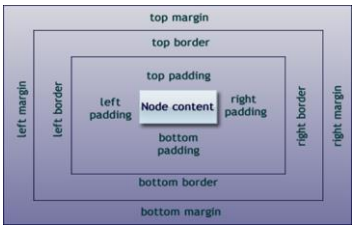
Présentation

- la création d'un composant personnalisé peut être effectuée au choix:
  - à partir de zéro en tout Java
  - par héritage en FXML et java
  - par héritage d'un composant existant

Composant personnalisé tout java

- tout composant hérite de la classe **`Control`**
  - qui hérite elle-même de **`Region`**

```
java.lang.Object
   javafx.scene.Node
      javafx.scene.Parent
         javafx.scene.layout.Region
            javafx.scene.control.Control
```



Composant personnalisé tout java

- propriétés essentielles de la classe **`Region`**

name	access	type	description
width	read-only	Double	the width of the node set by the parent container
height	read-only	Double	the height of the node set by the parent container
minWidth	read-write	Double	
minHeight	read-write	Double	
prefWidth	read-write	Double	the overriden min max, preferred min, preffered and max for width and height of the Region
prefHeight	read-write	Double	
maxWidth	read-write	Double	
maxHeight	read-write	Double	
padding	read-write	Insets	padding of the region

Composant personnalisé tout java

- **`prefWidth`** et **`prefHeight`** sont à zéro par défaut
  - il faut donc leur donner une valeur appropriée
- **`padding`** est l'espace entre le contenu et sa bordure
- **`border`** est la ligne qui entoure chaque côté du carré
- **`margin`** est l'espace qui sépare un noeud d'un autre
- **`Inset`** est la somme de **`padding`** et de **`border`**

- méthodes essentielles de la classe **`Region`**

Name	Description
computeMaxHeight(double width)	Computes the maximum height of this region.
computeMaxWidth(double width)	Computes the maximum width of this region.
computeMinHeight(double width)	Computes the minimum height of this region.
computeMinWidth(double width)	Computes the minimum width of this region.

Composant personnalisé tout java

- la création d'un composant personnalisé en tout Java fait appel aux API:
  - **`javafx.scene.control.Control`**
    - elle représente le modèle du composant graphique

      protected Control() Create a new Control.
  - **`javafx.scene.control.SkinBase`**
    - elle représente la vue du composant graphique

      protected SkinBase(C control) Constructor for all SkinBase instances.
- une feuille de style CSS permet d'apporter des caractéristiques visuelles au composant

Composant personnalisé tout java

- il faut créer une sous-classe de **`SkinBase`**

```
java.lang.Object
    javafx.scene.control.SkinBase<C>
```

  - son constructeur reçoit le contrôleur en argument
    - exemple :

```
public MonComposantView extends SkinBase{
    // contient la visualisation des données
    public MonComposantView(MonComposant control){
            super(control);
            //…
    }
}
```

Composant personnalisé tout java

- il faut créer une sous-classe de **`Control`**
  - sa méthode **`createDefaultSkin`** instancie la vue
    - exemple :

```
public MonComposant extends Control{
    //contient les données à visualiser et la gestion des évènements
    public MonComposant(){
            //…
    }
   @Override protected Skin<?> createDefaultSkin() {
            return new MonComposantView (this);
    }
}
```

Composant personnalisé en FXML et java

- cette solution permet de créer des composants par composition de composants existants
- il suffit de créer un fichier FXML ayant une racine **`fx:root`** dont le type est celui du composite
  - ayant comme fils les composants qui composent le nouveau composant
- il faut également créer une sous-classe du type de **`fx:root`**
  - son constructeur devra charger le fichier fxml

Composant personnalisé en FXML et java

- exemple: fichier **`custom_control.fxml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.TextField ?>
<?import javafx.scene.control.Button ?>

<fx:root type="javafx.scene.layout.VBox"
xmlns:fx="http://javafx.com/fxml">
   <children>
         <TextField fx:id="textField" />
         <Button text="Click Me"
               onAction="#doSomething" />
   </children>
</fx:root>
```

Composant personnalisé en FXML et java

- exemple (suite):

```
public class CustomControl extends VBox {
    @FXML private TextField textField;
    public CustomControl() {
        FXMLLoader fxmlLoader = new FXMLLoader(
            getClass().getResource("custom_control.fxml"));
        fxmlLoader.setRoot(this);
        fxmlLoader.setController(this);
        try {
            fxmlLoader.load();
        } catch (IOException exception) {
            throw new RuntimeException(exception);
        }
    }
```

Composant personnalisé en FXML et java

- exemple (suite):

```
    public String getText() {
        return textProperty().get();
    }
    public void setText(String value) {
        textProperty().set(value);
    }
    public StringProperty textProperty() {
        return textField.textProperty();
    }
    @FXML protected void doSomething() {
        System.out.println("The button was clicked!");
    }
}
```

Composant personnalisé en FXML et java

- on peut ensuite utiliser ce contrôle UI en Java ou en FXML
  - Java

```
HBox hbox = new HBox();
CustomControl customControl = new CustomControl();
customControl.setText("Bonjour!");
hbox.getChildren().add(customControl);
```

  - FXML

```
<HBox>
   <CustomControl text="Bonjour!"/>
</HBox>
```

Composant personnalisé par héritage

- cette solution simple permet de spécialiser un composant existant en créant une sous-classe
  - il suffit ensuite de redéfinir certaines méthodes en fonction des besoins

Composant personnalisé par héritage

- exemple:

```
public class DoubleTextField extends TextField {
    public DoubleTextField() { super(); }
    //http://utilitymill.com/utility/Regex_For_Range
    String numberRegEx =
    "\\b([0-9]{1,2}|[1-6][0-9]{2}|7[0-3][0-9]|74[0-4])\\b";
    @Override public void replaceText(int start,int end,String text)
    {
        String oldValue = getText();
        if ((validate(text))) {
            super.replaceText(start, end, text);
            String newText = super.getText();
            if (!validate(newText)) {
                  super.setText(oldValue);
            }
        }
    }
```

Composant personnalisé par héritage

- exemple (suite):

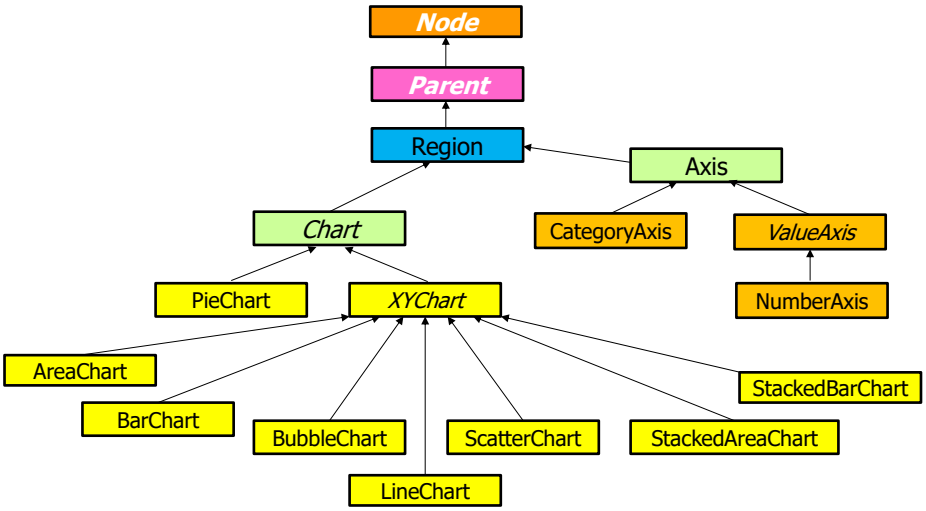
```
    @Override public void replaceSelection(String text) {
        String oldValue = getText();
        if (validate(text)) {
            super.replaceSelection(text);
            String newText = super.getText();
            if (!validate(newText)) {
                super.setText(oldValue);
            }
        }
    }

    private boolean validate(String text) {
         return ("".equals(text) || text.matches(numberRegEx));
    }
}
```

Création de diagrammes

Présentation

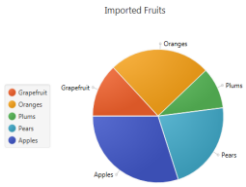
- Java FX 8 offre de nombreuses classes qui facilitent la création de diagrammes (charts)
  - toutes héritent de la classe abstraite `javafx.scene.chart.`**`Chart`**
  - exception faite de **`PieChart`**, toutes sont équipées des axes X et Y et héritent de **`XYChart`**
  - les feuilles de style CSS s'appliquent aux diagrammes

Classes pour diagrammes



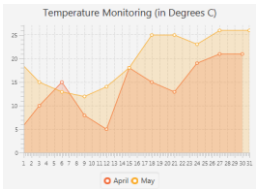
Exemples

PieChart

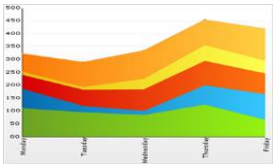


LineChart

AreaChart

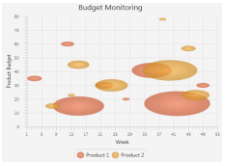


StackedAreaChart



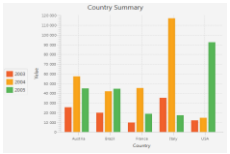
Exemples

BubbleChart

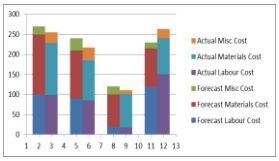


ScatterChart

BarChart



StackedBarChart



Propriétés

- propriétés communes à toutes les classes
  - `StringProperty title`
  - `StringProperty<Side> titleSide`
  - `ObjectProperty<Node> Legend`
  - `ObjectProperty<Side> legendSide`
  - `BooleanProperty legendVisible`
  - `BooleanProperty animated`

Diagramme circulaire

- ce type de diagramme est une instance de **`PieChart`**

```
public PieChart();
public PieChart(ObservableList<PieChart.Data> data);
```

- il faut lui associer au moins une instance de **`PieChart.Data`**
  - soit lors de sa construction
  - soit au moyen de **`setData`**

```
void setData(ObservableList<PieChart.Data> value);
```

Diagramme circulaire

- une instance de **`PieChart.Data`** représente un secteur du diagramme, créé au moyen du constructeur:

```
public Data(String name, double value);
```

  - **`Name`**
    - apparaîtra sur le diagramme, à côté du secteur concerné
  - **`value`**
    - déterminera la largeur du secteur, en fonction du nombre de secteurs associés au diagramme

Diagramme circulaire

- ce type de diagramme est une instance de **`PieChart`** auquel doit être associée au moins une instance de **`PieChart.Data`**
  - une instance de **`PieChart.Data`** représente un secteur du diagramme, créé au moyen du constructeur:

    `public Data(java.lang.String name, double value);`
    - **`value`** déterminera la largeur du secteur, en fonction du nombre de secteurs associés au diagramme

Exemple: diagramme circulaire

- exemple de diagramme circulaire (pie chart)

```
public class PieChartSample extends Application {
    @Override public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Imported Fruits");
        stage.setWidth(500);
        stage.setHeight(500);
        ObservableList<PieChart.Data> pieChartData =
                FXCollections.observableArrayList(
                new PieChart.Data("Grapefruit", 13),
                new PieChart.Data("Oranges", 25),
                new PieChart.Data("Plums", 10),
                new PieChart.Data("Pears", 22),
                new PieChart.Data("Apples", 30));
        final PieChart chart = new PieChart(pieChartData);
```

Exemple: diagramme circulaire

- exemple de diagramme circulaire (suite)

```
chart.setTitle("Imported Fruits");
final Label caption = new Label("");
caption.setTextFill(Color.DARKORANGE);
caption.setStyle("-fx-font: 24 arial;");
chart.getData().stream().forEach((data) ->{
   data.getNode().addEventHandler(
          MouseEvent.MOUSE_PRESSED,
          (MouseEvent e) -> {
                caption.setTranslateX(e.getSceneX());
                caption.setTranslateY(e.getSceneY());
                caption.setText(String.valueOf(
                      data.getPieValue())+"%");
          });
   });
```

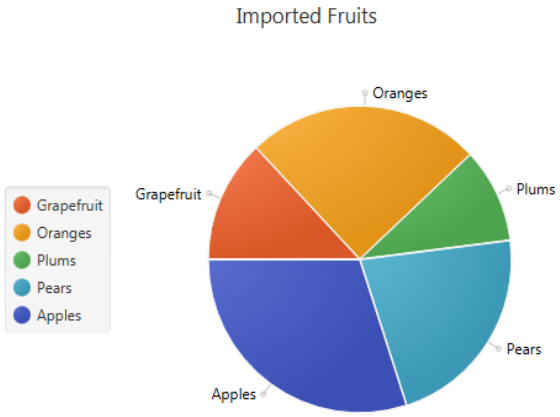
Exemple: diagramme circulaire

- exemple de diagramme circulaire (suite)

```
((Group) scene.getRoot()).getChildren().addAll(
          chart, caption);
stage.setScene(scene);
scene.getStylesheets().add(
          "piechartsample/Chart.css");
stage.show();
}

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

Exemple: diagramme circulaire

- résultat:



Diagrammes XY

- ces diagrammes ont besoin que l'on précise les axes X et Y lors de leur construction
  - au moyen des classes **`CategoryAxis`** et **`NumberAxis`**
    - **`CategoryAxis`** permet de créer une échelle graduée avec des chaînes de caractères (**`String`**)
    - **`NumberAxis`** est utilisée pour les échelles graduées avec des valeurs numériques (**`Number`**)

Diagrammes XY

- Il faut également leur fournir la (les) liste(s) des points constituant la (les) ligne(s) brisée(s)
  - au moyen d'instances de **`XYChart.Series<X,Y>`**

```
public Series();
public Series(
      ObservableList<XYChart.Data<X,Y>> data);
public Series(java.lang.String name,
      ObservableList<XYChart.Data<X,Y>> data);
```

Diagramme en ligne brisée

- ce type de diagramme est une instance de **`LineChart`**

```
public LineChart(Axis<X> xAxis, Axis<Y> yAxis)
public LineChart(Axis<X> xAxis, Axis<Y> yAxis,
ObservableList<XYChart.Series<XY>> data)
```

- une instance de **`XYChart.Series`** représente une ligne brisée du diagramme

```
XYChart.Series series1 = new XYChart.Series();
```

  - les données lui sont ensuite ajoutées

```
series1.setName("Portfolio 1");
series1.getData().add(new XYChart.Data("Jan", 23));
series1.getData().add(new XYChart.Data("Feb", 14));
```

Exemple: diagramme en ligne brisée

- exemple de diagramme en ligne brisée

```
public class LineChartSample extends Application {

    @Override
    public void start(Stage stage) {
        stage.setTitle("Line Chart Sample");
        final CategoryAxis xAxis = new CategoryAxis();
        final NumberAxis yAxis = new NumberAxis();
        xAxis.setLabel("Month");
        final LineChart<String,Number> lineChart =
                new LineChart<>(xAxis,yAxis);
        lineChart.setTitle("Stock Monitoring, 2010");
        lineChart.setCreateSymbols(false);
        lineChart.setAlternativeRowFillVisible(false);
```

Exemple: diagramme en ligne brisée

- exemple de diagramme en ligne brisée (suite)

```
        XYChart.Series series1 = new XYChart.Series();
        series1.setName("Portfolio 1");
        series1.getData().add(new XYChart.Data("Jan", 23));
        series1.getData().add(new XYChart.Data("Feb", 14));
        //…
        XYChart.Series series2 = new XYChart.Series();
        series2.setName("Portfolio 2");
        series2.getData().add(new XYChart.Data("Jan", 33));
        series2.getData().add(new XYChart.Data("Feb", 34));
        //…
        XYChart.Series series3 = new XYChart.Series();
        series3.setName("Portfolio 3");
        series3.getData().add(new XYChart.Data("Jan", 44));
        //…
```

Exemple: diagramme en ligne brisée

- exemple de diagramme en ligne brisée (suite)

```
        Scene scene  = new Scene(lineChart);
        lineChart.getData().addAll(series1, series2,
                series3);
        scene.getStylesheets().add(
          "linechartsample/Chart.css");
        stage.setScene(scene);
        stage.show();
    }

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

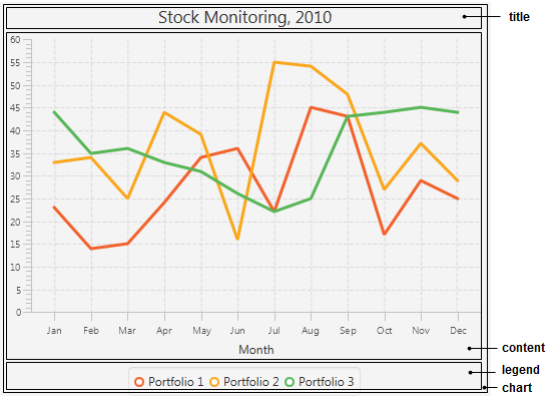
Exemple: diagramme en ligne brisée

- résultat:

CSS pour les diagrammes

- tous les diagrammes JavaFX possèdent un jeu de propriétés communes:

```
.chart
.chart-content
.chart-title
.chart-legend
```



CSS pour les diagrammes

- valeurs par défaut présente dans **`modena.css`**:

```
.chart {
    -fx-padding: 5px;
}
.chart-plot-background {
    -fx-background-color: -fx-background;
}
.chart-content {
    -fx-padding: 10px;
}
.chart-title {
    -fx-font-size: 1.4em;
}
.chart-legend {
   -fx-background-color: -fx-shadow-highlight-color,linear-gradient(
        to bottom, derive(-fx-background, -10%), derive(-fx-background, -5%)),
        linear-gradient(from 0px 0px to 0px 4px, derive(-fx-background, -4%),
        derive(-fx-background, 10%));
   -fx-background-insets: 0 0 -1 0, 0,1;
   -fx-background-radius: 4,4,3;
   -fx-padding: 6px;
}
```

CSS pour les diagrammes

- **`modena.css`** (suite)

```
.axis {
    AXIS_COLOR: derive(-fx-background,-20%);
    -fx-tick-label-font-size: 0.833333em;
    -fx-tick-label-fill: derive(-fx-text-background-color, 30%);
}
.axis:top {
    -fx-border-color: transparent transparent AXIS_COLOR transparent;
}
.axis:right {
    -fx-border-color: transparent transparent transparent AXIS_COLOR;
}
.axis:bottom {
    -fx-border-color: AXIS_COLOR transparent transparent transparent;
}
.axis:left {
    -fx-border-color: transparent AXIS_COLOR transparent transparent;
}
.axis:top > .axis-label,
.axis:left > .axis-label {
    -fx-padding: 0 0 4px 0;
}
```

CSS pour les diagrammes

- **`modena.css`** (suite)

```
.axis:bottom > .axis-label,
.axis:right > .axis-label {
    -fx-padding: 4px 0 0 0;
}
.axis-tick-mark,
.axis-minor-tick-mark {
    -fx-fill: null;
    -fx-stroke: AXIS_COLOR;
}
```

  - plus d'informations sur:

http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/css-styles.htm

Graphismes 2D et 3D

Graphismes 2D et 3D

- Dessins avec l'API Canvas
- Gestion d'images avec l'API ImageOps
- Shape3D
- Camera
- SubScene
- Lumière
- Texture
- Transformations 2D et 3D

Dessins avec l'API Canvas

- un **`Canvas`** désigne un nœud permettant des dessins
  - il fournit un **`GraphicsContext`** comportant de nombreuses méthodes de dessin pour:
    - textes
    - images
    - lignes
    - arcs
    - ovales
    - rectangles
    - polygones
    - ...

Dessins avec l'API Canvas

- exemple:

```
Group root = new Group();
Scene s = new Scene(root, 300, 300, Color.BLACK);
final Canvas canvas = new Canvas(250,250);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setFill(Color.BLUE);
gc.fillRect(75,75,100,100);
root.getChildren().add(canvas);
```

API Image Ops

- L'API Image Ops permet la manipulation directe de pixels composant les images
- le package **`javafx.scene.image`** comporte:
  - la classe **`Image`**
    - représente une image et fournit un **`PixelReader`**
    - formats supportés: bmp, gif, jpeg, png
    - une instance d'**`Image`** peut être visualisée avec **`ImageView`**
  - la classe **`WritableImage`**
    - sous-classe d'**`Image`**, fournit un **`PixelWriter`**

API Image Ops

- exemple:

```
Image image = new Image(
   "http://www.mesvacances.com/images/plage.png");
ImageView imageView = new ImageView();
imageView.setImage(image);
StackPane root = new StackPane();
root.getChildren().add(imageView);
Scene scene = new Scene(root, 300, 250);
primaryStage.setTitle("Image Read Test");
primaryStage.setScene(scene);
primaryStage.show();
```

API Image Ops

- exemple:

```
@Override
public void start(Stage primaryStage) {
    root = new Group();
    canvas = new Canvas(200, 200);
    canvas.setTranslateX(100);
    canvas.setTranslateY(100);
    gc = canvas.getGraphicsContext2D();
    Byte[] imageData = createImageData();
    drawImageData(gc, imageData);
    primaryStage.setScene(new Scene(root, 400, 400));
    primaryStage.show();
}
```

API Image Ops

- exemple (suite):

```
private byte[] createImageData() {
    int i = 0;
    byte imageData[]=new byte[WIDTH * HEIGHT * 3];
    for (int y = 0; y < HEIGHT; y++) {
        int r = y * 255 / HEIGHT;
        for (int x = 0; x < WIDTH; x++) {
            int g = x * 255 / WIDTH;
            imageData[i] = (byte) r;
            imageData[i + 1] = (byte) g;
            i += 3;
        }
    }
    return imageData;
}
```

API Image Ops

- exemple (suite):

```
private void drawImageData(GraphicsContext gc, byte[] imageData) {
    boolean on = true;
    PixelWriter pixelWriter = gc.getPixelWriter();
    PixelFormat<ByteBuffer> pixelFormat =
            PixelFormat.getByteRgbInstance();
    for (int y = 50; y < 150; y += HEIGHT) {
        for (int x = 50; x < 150; x += WIDTH) {
            if (on) {
                pixelWriter.setPixels(x, y, WIDTH, HEIGHT,
                pixelFormat, imageData, 0, WIDTH * 3);
            } on = !on;
        } on = !on;
    }
    gc.applyEffect(new DropShadow(20, 20, 20, Color.GRAY));
    root.getChildren().add(canvas);
}
```

API Image Ops

- l'interface **`PixelReader`**
  - fournit des méthodes pour récupérer les pixels d'une **`Image`** ou autre source
- l'interface **`PixelWriter`**
  - fournit des méthodes pour écrire des pixels dans une **`WritableImage`** ou autre destination
- la classe abstraite **`PixelFormat`**
  - définit la disposition des données pour un pixel dans un format donné
- la classe abstraite **`WritablePixelFormat`**
  - sous-classe de **`PixelFormat`**, peut être utilisée comme format destination pour l'écriture de pixels

Shape3D

- il existe 2 sortes de formes 3D:
  - prédéfinies
  - définies par l'utilisateur
- les formes 3D prédéfinies héritent de **`Shape3D`**

```
java.lang.Object
   javafx.scene.Node
         javafx.scene.shape.Shape3D
               javafx.scene.shape.Box
               javafx.scene.shape.Cylinder
               javafx.scene.shape.Sphere
               javafx.scene.shape.MeshView
```

Shape3D

- pour créer un objet **`Box`**, préciser la largeur, hauteur et profondeur

```
Box myBox = new Box(width, height, depth);
```

- pour créer un objet **`Cylinder`**, préciser le rayon et la hauteur

```
Cylinder myCylinder = new Cylinder(radius, height);
Cylinder myCylinder2 = new Cylinder(radius, height,
divisions);
```

- pour créer un objet **`Sphere`**, préciser le rayon

```
Sphere mySphere = new Sphere(radius);
Sphere mySphere2 = new Sphere(radius, divisions);
```

Shape3D

- objet **`MeshView`** représente la vue d'une surface de type **`Mesh`**, classe abstraite

```
MeshView meshView = new MeshView(mesh);
```

- en pratique, **`TriangleMesh`** est le type le plus utilisé

```
java.lang.Object
    javafx.scene.shape.Mesh
        javafx.scene.shape.TriangleMesh
```

Shape3D

- exemple

```
TriangleMesh mesh = new TriangleMesh();
// définition des côtés de la maille
float points[] = { … };
mesh.getPoints().addAll(points);
// définition des coordonnées de chaque vertex
float texCoords[] = { … };
mesh.getTexCoords().addAll(texCoords);
// construction des faces
int faces[] = { … };
mesh.getFaces().addAll(faces);
// définition du groupe auquel appartient chaque face
int smoothingGroups[] = { … };
mesh.getFaceSmoothingGroups().addAll(smoothingGroups);
```

Camera

- un objet **`Camera`** représente la projection 2D d'une scène 3D
  - il définit la correspondance entre les coordonnées 3D d'une scène et celles de la fenêtre dans laquelle cette scène est visualisée

```
java.lang.Object
    javafx.scene.Node
        javafx.scene.Camera
            javafx.scene.PerspectiveCamera
            javafx.scene.ParallelCamera
```

  - **`PerspectiveCamera`** effectue une projection en perspective
  - **`ParallelCamera`** effectue une projection parallèle

Camera

- le système de coordonnées de **`PerspectiveCamera`** a pour origine le coin du panneau situé en haut à gauche
  - axe X orienté vers la droite
  - axe Y orienté vers le bas
  - axe Z orienté vers la scène
- exemple:

```
Camera camera = new PerspectiveCamera(true);
scene.setCamera(camera);
Group cameraGroup = new Group();
cameraGroup.getChildren().add(camera);
root.getChildren().add(cameraGroup);
camera.rotate(45);
```

SubScene

- **`SubScene`** permet de visualiser une arborescence graphique avec un objet **`Camera`**

```
public SubScene(Parent root, double width,
    double height)
public SubScene(Parent root, double width,
    double height, boolean depthBuffer,
    SceneAntialiasing antiAliasing)
```

- exemple:

```
PerspectiveCamera camera =
          new PerspectiveCamera(true);
…
SubScene subscene = new SubScene(root, 800, 600);
subscene.setCamera(camera);
```

Lumière

- les instances de **`LightBase`** interagissent avec celles des objets **`Shape3D`** et ses textures pour produire la vue

```
java.lang.Object
    javafx.scene.Node
        javafx.scene.LightBase
            javafx.scene.AmbientLight
            javafx.scene.PointLight
```

  - **`AmbientLight`** désigne une source de lumière qui rayonne dans toutes les directions
  - **`PointLight`** désigne une source de lumière de coordonnées précises qui rayonne dans toutes les directions

Lumière

- exemple:

```
PointLight light = new PointLight(Color.WHITE);
light.setTranslateX(50);
light.setTranslateY(-300);
light.setTranslateZ(-400);
Group lightGroup = new Group();
lightGroup.getChildren().add(light);
root.getChildren().add(lightGroup);
light.rotate(45);
lightGroup.setTranslateZ(-75);
```

Texture

- les instances de **`Material`** définissent le type de surface et son interaction avec la lumière

```
java.lang.Object
    javafx.scene.paint.Material
        javafx.scene.paint.PhongMaterial
```

  - **`PhongMaterial`** réféchit la lumière selon 4 composantes:
    - caractéristique de diffusion (diffuse)
    - caracéristique de réflexion (specular)
    - intensité de la lumière ambiante (ambient)
    - caractéristique d'auto-illumination (self illumination)

Texture

- exemple:

```
//Create Material Material mat = new PhongMaterial();
Image diffuseMap = new Image("diffuseMap.png");
Image normalMap = new Image("normalMap.png");
mat.setDiffuseMap(diffuseMap);
mat.setBumpMap(normalMap);
mat.setSpecularColor(Color.WHITE);
shape3d.setMaterial(mat);
```

Transformations 2D et 3D

- un nœud seul ou un groupe de nœuds peut subir des transformations:
  - translation
  - rotation
  - changement d'échelle
  - cisaillement
- ces transformations peuvent être effectuées via des méthodes de la classe **`Node`** ou au moyen d'instances de sous-classes de **`Transform`**

Transformations 2D et 3D

- méthodes de la classe **`Node`**

```
public final void setRotate(double value)
public final void setRotationAxis(Point3D value)
public final void setScaleX(double value)
public final void setScaleY(double value)
public final void setScaleZ(double value)
public final void setTranslateX(double value)
public final void setTranslateY(double value)
public final void setTranslateZ(double value)
public final ObservableList<Transform> getTransforms()
```

Transformations 2D et 3D

- les transformations font partie du package **`javafx.scene.transform`**

```
java.lang.Object
   javafx.scene.transform.Transform
      javafx.scene.transform.Translate
      javafx.scene.transform.Affine
      javafx.scene.transform.Rotate
      javafx.scene.transform.Scale
      javafx.scene.transform.Shear
```

- exemple:

```
Rectangle rect = new Rectangle(50,50, Color.RED);
rect.getTransforms().add(new Rotate(45,0,0));
```

Transformations 2D et 3D

- exemple:

```
private void init(Stage primaryStage) {
    Group root = new Group();
    primaryStage.setTitle("JavaFX 3D");
    primaryStage.setResizable(false);
    Scene scene = new Scene(root, WIDTH, HEIGHT, true);
    scene.setFill(Color.BLACK);
    primaryStage.setScene(scene);
    PerspectiveCamera camera = new PerspectiveCamera();
    Translate translate = new Translate(WIDTH/2, HEIGHT/2);
    Rotate rotate = new Rotate(180, Rotate.Y_AXIS);
    primaryStage.getScene().setCamera(camera);
    root.getTransforms().addAll(translate, rotate);
    Node node = create3dContent();
    root.getChildren().add(node);
}
```

Animations, effets visuels, média

Animations, effets visuels, média

- Animations
- Effets visuels
- Média audio et vidéo

Animations

- les API d'animation font partie du package **`javafx.animation`**

```
java.lang.Object
    javafx.animation.Animation
        javafx.animation.Transition
        javafx.animation.Timeline
```

  - les instances des sous-classes de **`Transition`** fournissent un moyen d'incorporer des animations suivant une chronologie interne
  - les instances de **`Timeline`** permettent l'évolution de propriétés d'une animation au cours de son déroulement

Animations

- propriétés de la classe **`Animation`**

BooleanProperty	autoReverse Defines whether this Animation reverses direction on alternating cycles.
ReadOnlyDoubleProperty	currentRate Read-only variable to indicate current direction/speed at which the Animation is being played.
ReadOnlyObjectProperty<Duration>	currentTime Defines the Animation's play head position.
IntegerProperty	cycleCount Defines the number of cycles in this animation.
ReadOnlyObjectProperty<Duration>	cycleDuration Read-only variable to indicate the duration of one cycle of this Animation: the time it takes to play from time 0 to the end of the Animation (at the default rate of 1.0).
ObjectProperty<Duration>	delay Delays the start of an animation.
ObjectProperty<EventHandler<ActionEvent>>	onFinished The action to be executed at the conclusion of this Animation.
DoubleProperty	rate Defines the direction/speed at which the Animation is expected to be played.
ReadOnlyObjectProperty<Animation.Status>	status The status of the Animation.
ReadOnlyObjectProperty<Duration>	totalDuration Read-only variable to indicate the total duration of this Animation, including repeats.

Animations

- méthodes principales de la classe **`Animation`**

void	jumpTo(Duration time) Jumps to a given position in this Animation.
void	jumpTo(String cuePoint) Jumps to a predefined position in this Animation.
void	pause() Pauses the animation.
void	play() Plays Animation from current position in the direction indicated by rate.
void	playFrom(Duration time) A convenience method to play this Animation from a specific position.
void	playFrom(String cuePoint) A convenience method to play this Animation from a predefined position.
void	playFromStart() Plays an Animation from initial position in forward direction.
void	stop() Stops the animation and resets the play head to its initial position.

Animations

- sous-classes de **`Transition`**:

*javafx.animation.**Transition***

- javafx.animation.**SequentialTransition**
- javafx.animation.**ParallelTransition**
- javafx.animation.**FadeTransition**
- javafx.animation.**FillTransition**
- javafx.animation.**PathTransition**
- javafx.animation.**PauseTransition**
- javafx.animation.**RotateTransition**
- javafx.animation.**TranslateTransition**
- javafx.animation.**ScaleTransition**
- javafx.animation.**StrokeTransition**

Animations

- une liste d'animations peut être jouée:
  - en séquence avec **`SequentialTransition`**

SequentialTransition() The constructor of SequentialTransition.
SequentialTransition(Animation... children) The constructor of SequentialTransition.
SequentialTransition(Node node) The constructor of SequentialTransition.
SequentialTransition(Node node, Animation... children) The constructor of SequentialTransition.

  - en parallèle avec **`ParallelTransition`**

ParallelTransition() The constructor of ParallelTransition.
ParallelTransition(Animation... children) The constructor of ParallelTransition.
ParallelTransition(Node node) The constructor of ParallelTransition.
ParallelTransition(Node node, Animation... children) The constructor of ParallelTransition.

Animations

- **`FadeTransition`** permet de réaliser des fondus par variation de l'opacité d'un noeud

FadeTransition() The constructor of FadeTransition
FadeTransition(Duration duration) The constructor of FadeTransition
FadeTransition(Duration duration, Node node) The constructor of FadeTransition

- **`FillTransition`** permet de réaliser des variations progressives de couleur d'un noeud

FillTransition() The constructor of FillTransition
FillTransition(Duration duration) The constructor of FillTransition
FillTransition(Duration duration, Color fromValue, Color toValue) The constructor of FillTransition
FillTransition(Duration duration, Shape shape) The constructor of FillTransition
FillTransition(Duration duration, Shape shape, Color fromValue, Color toValue) The constructor of FillTransition

Animations

- **`PathTransition`** permet de déplacer un nœud le long d'un chemin

PathTransition() The constructor of PathTransition.
PathTransition(Duration duration, Shape path) The constructor of PathTransition.
PathTransition(Duration duration, Shape path, Node node) The constructor of PathTransition.

- **`PauseTransition`** permet de réaliser des pause dans le déroulement d'une animation

PauseTransition() The constructor of PauseTransition
PauseTransition(Duration duration) The constructor of PauseTransition.

Animations

- **`RotateTransition`** permet d'effectuer la rotation d'un nœud, en précisant l'axe de rotation et l'angle de rotation (en degrés)

ObjectProperty<Point3D>	axis Specifies the axis of rotation for this RotateTransition.
DoubleProperty	byAngle Specifies the incremented stop angle value, from the start, of this RotateTransition.
DoubleProperty	fromAngle Specifies the start angle value for this RotateTransition.
DoubleProperty	toAngle Specifies the stop angle value for this RotateTransition.

Animations

- **`TranslateTransition`** permet de déplacer linéairement un nœud, en précisant les points origine et destination

DoubleProperty	byX Specifies the incremented stop X coordinate value, from the start, of this TranslateTransition.
DoubleProperty	byY Specifies the incremented stop Y coordinate value, from the start, of this TranslateTransition.
DoubleProperty	byZ Specifies the incremented stop Z coordinate value, from the start, of this TranslateTransition.
DoubleProperty	fromX Specifies the start X coordinate value of this TranslateTransition.
DoubleProperty	fromY Specifies the start Y coordinate value of this TranslateTransition.
DoubleProperty	fromZ Specifies the start Z coordinate value of this TranslateTransition.
DoubleProperty	toX Specifies the stop X coordinate value of this TranslateTransition.
DoubleProperty	toY Specifies the stop Y coordinate value of this TranslateTransition.
DoubleProperty	toZ Specifies the stop Z coordinate value of this TranslateTransition

Animations

- **`ScaleTransition`** permet de modifier progressivement la taille d'un nœud en précisant le changement d'échelle sur chacun des axes

DoubleProperty	byX Specifies the incremented stop X scale value, from the start, of this Transition.
DoubleProperty	byY Specifies the incremented stop Y scale value, from the start, of this ScaleTransition.
DoubleProperty	byZ Specifies the incremented stop Z scale value, from the start, of this ScaleTransition.
DoubleProperty	fromX Specifies the start X scale value of this ScaleTransition.
DoubleProperty	fromY Specifies the start Y scale value of this ScaleTransition.
DoubleProperty	fromZ Specifies the start Z scale value of this ScaleTransition.
DoubleProperty	toX Specifies the stop X scale value of this ScaleTransition.
DoubleProperty	toY The stop Y scale value of this ScaleTransition.
DoubleProperty	toZ The stop Z scale value of this ScaleTransition.

Animations

- **`StrokeTransition`** permet de modifier progressivement la couleur du contour d'une forme

StrokeTransition() The constructor of StrokeTransition
StrokeTransition(Duration duration) The constructor of StrokeTransition
StrokeTransition(Duration duration, Color fromValue, Color toValue) The constructor of StrokeTransition
StrokeTransition(Duration duration, Shape shape) The constructor of StrokeTransition
StrokeTransition(Duration duration, Shape shape, Color fromValue, Color toValue) The constructor of StrokeTransition

Animations

- exemple

```
Rectangle rect = new Rectangle (100, 40, 100, 100);
rect.setArcHeight(50);
rect.setArcWidth(50);
rect.setFill(Color.VIOLET);
FadeTransition ft = new
FadeTransition(Duration.millis(3000), rect);
ft.setFromValue(1.0);
ft.setToValue(0.3);
ft.setCycleCount(4);
ft.setAutoReverse(true);
ft.play();
```

Animations

- exemple:

```
Rectangle rect = new Rectangle (100, 40, 100, 100);
rect.setArcHeight(50);
rect.setArcWidth(50);
rect.setFill(Color.VIOLET);
final Duration SEC_2 = Duration.millis(2000);
final Duration SEC_3 = Duration.millis(3000);
FadeTransition ft = new FadeTransition(SEC_3);
ft.setFromValue(1.0f);
ft.setToValue(0.3f); ft.setCycleCount(2f);
ft.setAutoReverse(true);
TranslateTransition tt =
   new TranslateTransition(SEC_2);
tt.setFromX(-100f);
```

Animations

- exemple (suite):

```
tt.setToX(100f);
tt.setCycleCount(2f);
tt.setAutoReverse(true);
RotateTransition rt = new RotateTransition(SEC_3);
rt.setByAngle(180f); rt.setCycleCount(4f);
rt.setAutoReverse(true);
ScaleTransition st = new ScaleTransition(SEC_2);
st.setByX(1.5f); st.setByY(1.5f);
st.setCycleCount(2f);
st.setAutoReverse(true);
ParallelTransition pt =
   new ParallelTransition(rect, ft, tt, rt, st);
pt.play();
```

Animations

- une instance de **`TimeLine`** permet de créer tout type d'animation par variation progressive d'une ou plusieurs propriétés d'un nœud

Timeline() The constructor of Timeline.
Timeline(double targetFramerate) The constructor of Timeline.
Timeline(double targetFramerate, KeyFrame... keyFrames) The constructor of Timeline.
Timeline(KeyFrame... keyFrames) The constructor of Timeline.

  - elle utilise une ou plusieurs instances de:
    - **`KeyValue`**: représente la valeur cible d'une propriété
    - **`KeyFrame`**: représente les valeurs cibles à atteindre à des instants précis

Animations

- **KeyValue**

  KeyValue(WritableValue<T> target, T endValue) Creates a KeyValue that uses Interpolator.LINEAR.

  KeyValue(WritableValue<T> target, T endValue, Interpolator interpolator) Creates a KeyValue.

- **KeyFrame**

  KeyFrame(Duration time, EventHandler<ActionEvent> onFinished, KeyValue... values) Constructor of KeyFrame

  KeyFrame(Duration time, KeyValue... values) Constructor of KeyFrame

  KeyFrame(Duration time, String name, EventHandler<ActionEvent> onFinished, Collection<KeyValue> values) Constructor of KeyFrame

  KeyFrame(Duration time, String name, EventHandler<ActionEvent> onFinished, KeyValue... values) Constructor of KeyFrame

  KeyFrame(Duration time, String name, KeyValue... values) Constructor of KeyFrame

Animations

- exemple:

```
final Rectangle rect=new Rectangle(100,50,100, 50);
rect.setFill(Color.RED);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv=new KeyValue(rect.xProperty(), 300);
final KeyFrame kf=new KeyFrame(Duration.millis(500),
kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

Effets visuels

- un effet est un algorithme qui produit une image, à partir d'une ou plusieurs source:
  - modification des couleurs
  - combinaison d'images
  - déplacement de pixels
- ils peuvent être combinés entre eux
- un effet est appliqué à un nœud via la méthode **`setEffect`** de **`Node`**
- les effets visuels font partie du package **`javafx.scene.effect`**

Effets visuels

```
java.lang.Object
    javafx.scene.effect.Effect
        javafx.scene.effect.Blend
        javafx.scene.effect.Bloom
        javafx.scene.effect.BoxBlur
        javafx.scene.effect.ColorAdjust
        javafx.scene.effect.ColorInput
        javafx.scene.effect.DisplacementMap
        javafx.scene.effect.DropShadow
        javafx.scene.effect.GaussianBlur
        javafx.scene.effect.Glow
        javafx.scene.effect.ImageInput
        javafx.scene.effect.InnerShadow
        javafx.scene.effect.Lighting
        javafx.scene.effect.MotionBlur
        javafx.scene.effect.PerspectiveTransform
        javafx.scene.effect.Reflection
        javafx.scene.effect.SepiaTone
        javafx.scene.effect.Shadow
```

Effets visuels

- exemple:

```
ColorAdjust colorAdjust = new ColorAdjust();
colorAdjust.setContrast(0.1);
colorAdjust.setHue(-0.05);
colorAdjust.setBrightness(0.1);
colorAdjust.setSaturation(0.2);
Image image = new Image("boat.jpg");
ImageView imageView = new ImageView(image);
imageView.setFitWidth(200);
imageView.setPreserveRatio(true);
imageView.setEffect(colorAdjust);
```

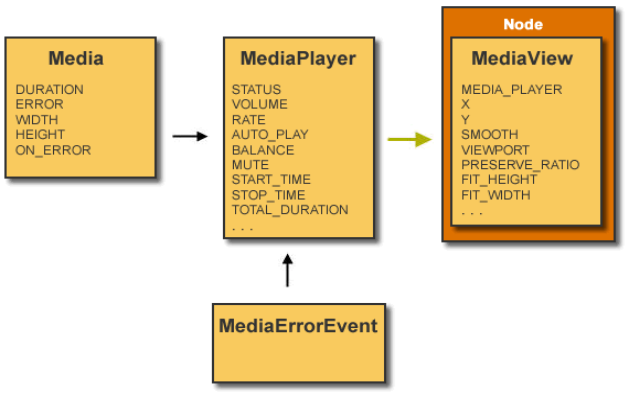
Média audio et vidéo

- formats actuellement supportés:
  - audio:
    - MP3
    - AIFF (PCM non compressé)
    - WAV (PCM non compressé)
    - MPEG-4 (Advanced Audio Coding (AAC))
  - video:
    - FLV (contenant des vidéo VP6 et de l'audio MP3)
    - MPEG-4 (avec H.264/AVC compression video)

API Media

- les API Media font partie du package **`javafx.scene.media`**
  - **`Media`**
    - ressource qui contient de l'information sur le média, comme sa localisation, sa résolution et d'autres méta-données
  - **`MediaPlayer`**
    - composant clé pour diffuser le média
  - **`MediaView`**
    - objet pour animations et d'autres effets

API Media



API Media: mise en oeuvre

- exemple:

```
private static final String MEDIA_URL =
"http://dma.com/javafx/alpha.flv";

// création du media player
Media media = new Media(MEDIA_URL);
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.setAutoPlay(true);

// création du MediaView et ajout du media Player
MediaView mediaView = new MediaView(mediaPlayer);

((Group)scene.getRoot()).getChildren().add(mediaView);
```

Déploiement d'applications JavaFX

Déploiement d'applications JavaFX

- présentation
- outils de déploiement
- déploiement avec NetBeans IDE
- déploiement avec Ant
- déploiement avec Java Packager
- packaging autonome

Présentation

- la même application JavaFX peut être exécutée de différentes manières:
  - comme application desktop
  - en ligne de commande via le Java launcher
  - via un lien hypertexte dans un navigateur pour télécharger et exécuter l'application
  - en consultant une page web

Outils de déploiement

- plusieurs outils permettent de packager une application JavaFX:
  - NetBeans IDE
  - Ant
  - Java packager
- tous ces outils génèrent un ensemble de fichiers permettant d'exécuter l'application
  - un ou plusieurs fichiers JAR
  - un fichier JNLP avec descripteur de déploiment
  - une page web avec code JavaScript

Déploiement avec NetBeans IDE

- une fois l'application développée, le dossier **`dist`** du projet contient:
  - un fichier HTML
  - un fichier JNLP
  - un fichier JAR
- chacun de ces fichiers permet de lancer l'application par un double-clic

Déploiement avec Ant

- les tâches Ant pour JavaFX nécessitent le fichier **`ant-javafx.jar`** présent dans le dossier **`lib`** du JDK
- les tâches Ant permettent de:
  - générer des fichiers JAR exécutables
  - générer des pages web avec descripteur de déploiement pour Java Web Start, ou des applications intégrées à des pages web
  - signer une application si nécessaire
  - convertir des fichiers CSS au format binaire
  - créer des applications autonomes

Déploiement avec Ant

- exemple de fichier **`build.xml`**

```
<taskdef resource="com/sun/javafx/tools/ant/antlib.xml"
       uri="javafx:com.sun.javafx.tools.ant"
       classpath="${javafx.lib.ant-javafx.jar}"/>

<fx:application id="sampleApp" name="Some sample app"
       mainClass="test.MyApplication"
       <!-- This application has a preloader class -->
       preloaderClass="testpreloader.Preloader"
       fallbackClass="test.UseMeIfNoFX"/>

<fx:resources id="appRes">
     <fx:fileset dir="dist" requiredFor="preloader"
             includes="mypreloader.jar"/>
     <fx:fileset dir="dist" includes="myapp.jar"/>
</fx:resources>
```

Déploiement avec Ant

- exemple (suite)

```
<fx:jar destfile="dist/myapp.jar">
    <fx:application refid="sampleApp"/>
    <fx:resources refid="appRes"/>
    <manifest>
        <attribute name="Implementation-Vendor"
        value="${application.vendor}"/>
        <attribute name="Implementation-Title"
        value="${application.title}"/>
        <attribute name="Implementation-Version"
        value="1.0"/>
    </manifest>
    <fileset dir="${build.classes.dir}"/>
</fx:jar>
```

Déploiement avec Ant

- exemple (suite)

```
<fx:signjar keyStore="${basedir}/sample.jks"
destdir="dist" alias="javafx"
       storePass="****" keyPass="****">
     <fileset dir='dist/*.jar'/>
</fx:signjar>
```

Déploiement avec Ant

- exemple (suite)

```
<fx:deploy width="${applet.width}"
       height="${applet.height}"
       outdir="${basedir}/${dist.dir}" embedJNLP="true"
       outfile="${application.title}">
    <fx:application refId="sampleApp"/>
    <fx:resources refid="appRes"/>
    <fx:info title="Sample app: ${application.title}"
       vendor="${application.vendor}"/>
       <!-- Request elevated permissions -->
    <fx:permissions elevated="true"/>
</fx:deploy>
```

  - plus d'informations à l'URL:

http://docs.oracle.com/javafx/2/deployment/javafx_ant_tasks.htm

Déploiement avec Java Packager

- **`javapackager`** est une commande en ligne présente dans le dossier **`bin`** du JDK
  - elle permet de compiler, packager, signer et déployer une application JavaFX en ligne de commandes
  - syntaxe:

    `javapackager -command [-options]`

    - toutes les options apparaissent lorsque la commande **`javapackager`** est lancée sans paramètres

Déploiement avec Java Packager

- principales valeurs de **`command`**:

`-createbss`	Converts a CSS file into binary form
`-createjar`	Produces a JAR archive according to other parameters specified as options.
`-deploy`	Assembles the application package for redistribution. By default, the deploy task will generate the base application package, but it can also generate self-contained application packages if requested.
`-makeall`	Compiles source code and combines the -createjar and -deploy commands, with simplified options.
`-signJar`	Digitally signs JAR files and attaches a certificate.

  - plus d'informations sur:

http://docs.oracle.com/javase/8/docs/technotes/tools/unix/javafxpackager.html

Packaging autonome

- il est possible de créer des applications JavaFX packagées de façon autonome (*self-contained application packaging*), pour une plateforme donnée
  - elles sont alors installées comme d'autres applications natives
  - l'environnement d'exécution est intégré à l'installation
  - le déploiement peut être effectué même en l'absence de droits administrateur si le format .zip est utilisé

Packaging autonome

- le packaging contient:
  - le code Java de l'application, packagé dans un jeu de fichiers JAR
  - toutes les ressources (images, fichiers de propriétés,...)
  - une copie privée de l'environnement d'exécution Java et JavaFX
  - un launcher natif
  - des métadonnées (icônes,..)

Packaging autonome

- self-contained application pakaging:
  https://docs.oracle.com/javase/8/docs/technotes/guides/deploy/self-contained-packaging.html
- native packaging avec Ant:
  http://docs.oracle.com/javafx/2/deployment/self-contained-packaging.htm#BCGIBBCI
- native packaging avec JavaFX Packager:
  http://docs.oracle.com/javafx/2/deployment/self-contained-packaging.htm#BCGIBBCI