



Introduction à *JavaFX*

v2.16



IENAC SITA

Programmation événementielle

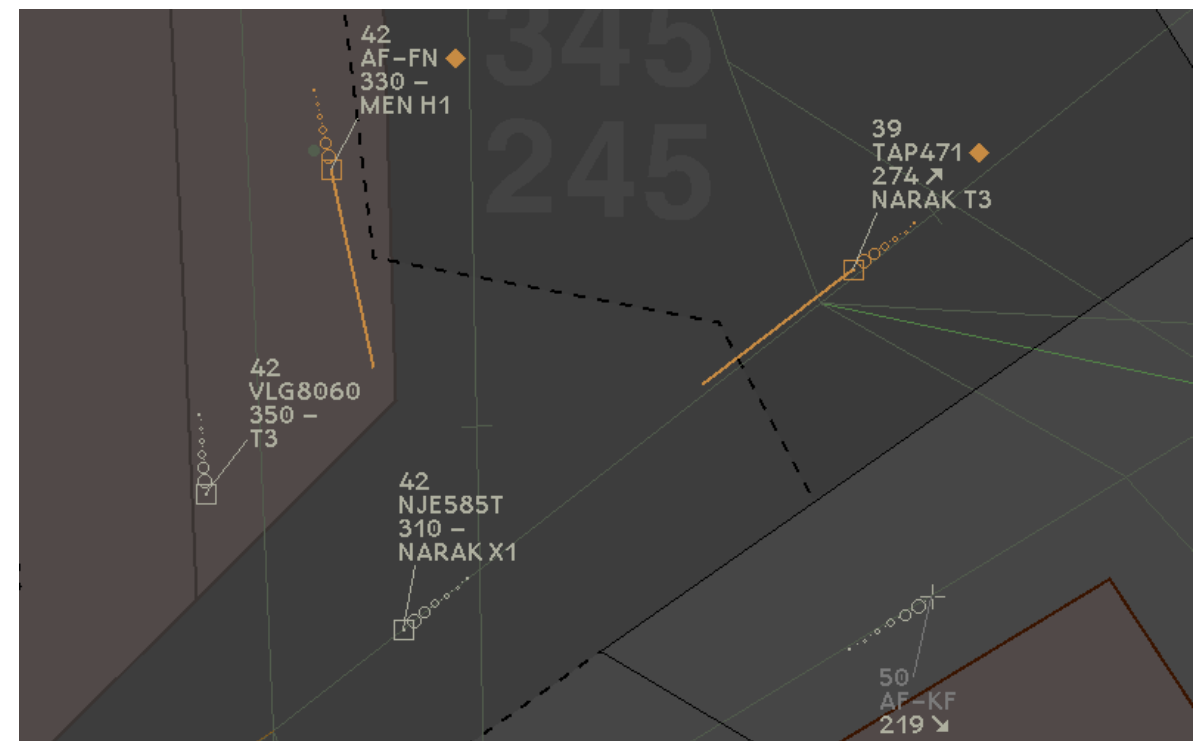
Nicolas Saporito

Objectifs détaillés

- Savoir décrire la structure générale d'une application *JavaFX* et le cycle de vie simplifié de ses composants interactifs,
- savoir créer une application *WIMP* simple en *JavaFX* et gérer statiquement la disposition de ses composants visuels,
- savoir décrire et utiliser les différents mécanismes de **programmation événementielle** fournis par *JavaFX*,
- savoir dessiner des formes géométriques en utilisant les classes *JavaFX* de haut niveau permettant des interactions,
- savoir manipuler le **graphe de scène** en appliquant des transformations à ses noeuds,
- savoir programmer les **interactions de manipulation directe** suivantes : *pan, drag, zoom centré souris* standard et *zoom centré souris* avec grossissement variable selon les composants (*zoom différencié*)

Objectifs

L'atteinte de ces objectifs pédagogiques vous permettra de réaliser la partie IHM de votre projet Java.



Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

1. Intro

JavaFX est une bibliothèque Java orientée graphisme et interaction

- équivalent de *Qt*, sauf que...
Qt n'est pas lié à un langage
JavaFX est intrinsèquement lié à Java
- successeur des bibliothèques graphiques historiques de Java: *awt* et *swing*
- amène des choses en plus :
graphe de scène, transformations graphiques, effets, animations, 3D...

1. Intro

JavaFX est aussi prévu pour les RIA...

- RIA = Rich Internet Application
 - applications riches en termes de présentation et d'interactions, non permises par le html ancestral (hypertexte)
 - historiquement : *Flash, applets java, Flex, Silverlight...*
- Possibilité d'architecture client-serveur

...et unifie donc deux cadres pensés différemment à l'origine.

1. Intro

ANNULÉ**Rester sur Oracle Java SE Development Kit 8u281**<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

Installation

- Depuis sa version 11, le JDK n'inclut plus JavaFX. Ce dernier est open-sourcé sous le nom *OpenJFX*.
- Depuis avril 2019, Oracle n'offre plus de support gratuit sur les précédentes versions de Java incluant encore JavaFX.

⇒ Passage recommandé à : *OpenJDK* + *OpenJFX* :

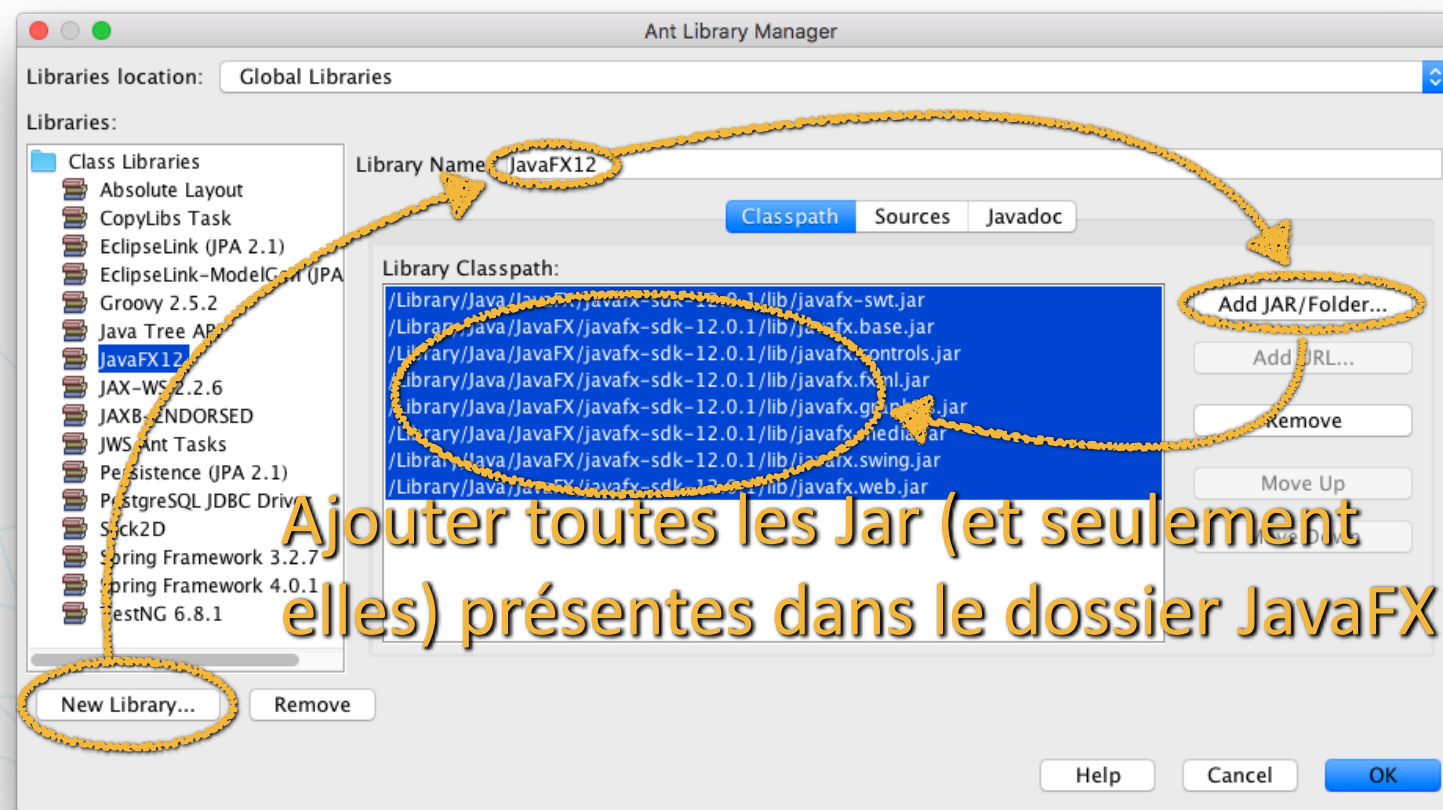
<https://openjfx.io/openjfx-docs/>

1. Intro

ANNULÉ**Rester sur Oracle Java SE Development Kit 8u281**<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

Utilisation avec NetBeans (1/4)

Une fois pour toutes, au premier lancement de NetBeans, déclarez une bibliothèque globale (**Tools > Libraries > New Library**):

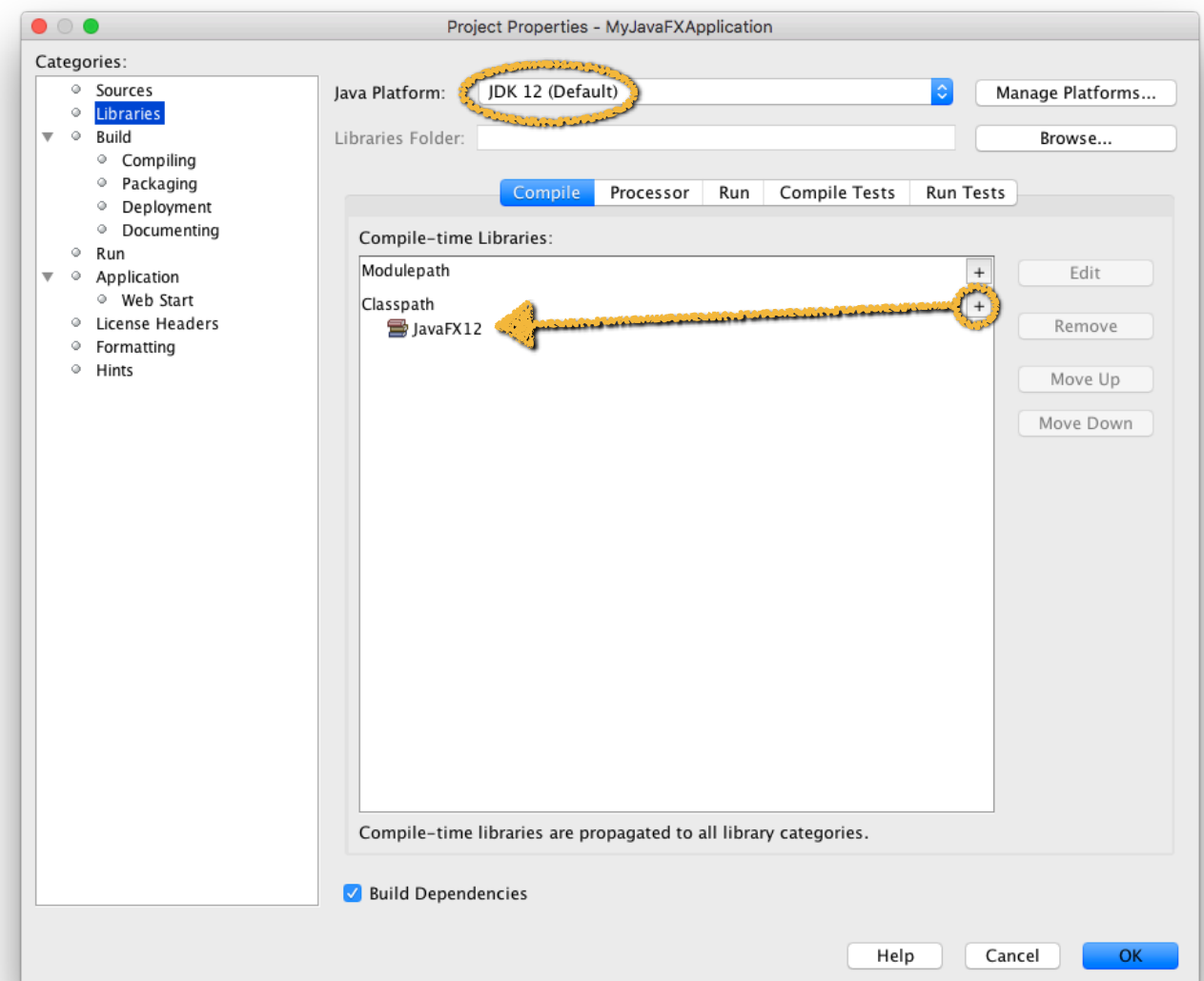


1. Intro

ANNULÉ**Rester sur Oracle Java SE Development Kit 8u281**<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

Utilisation avec NetBeans (2/4)

A la création de chaque projet, ajoutez la bibliothèque précédemment déclarées au chemin pour la compilation (**Project Properties > Libraries > Compile**):

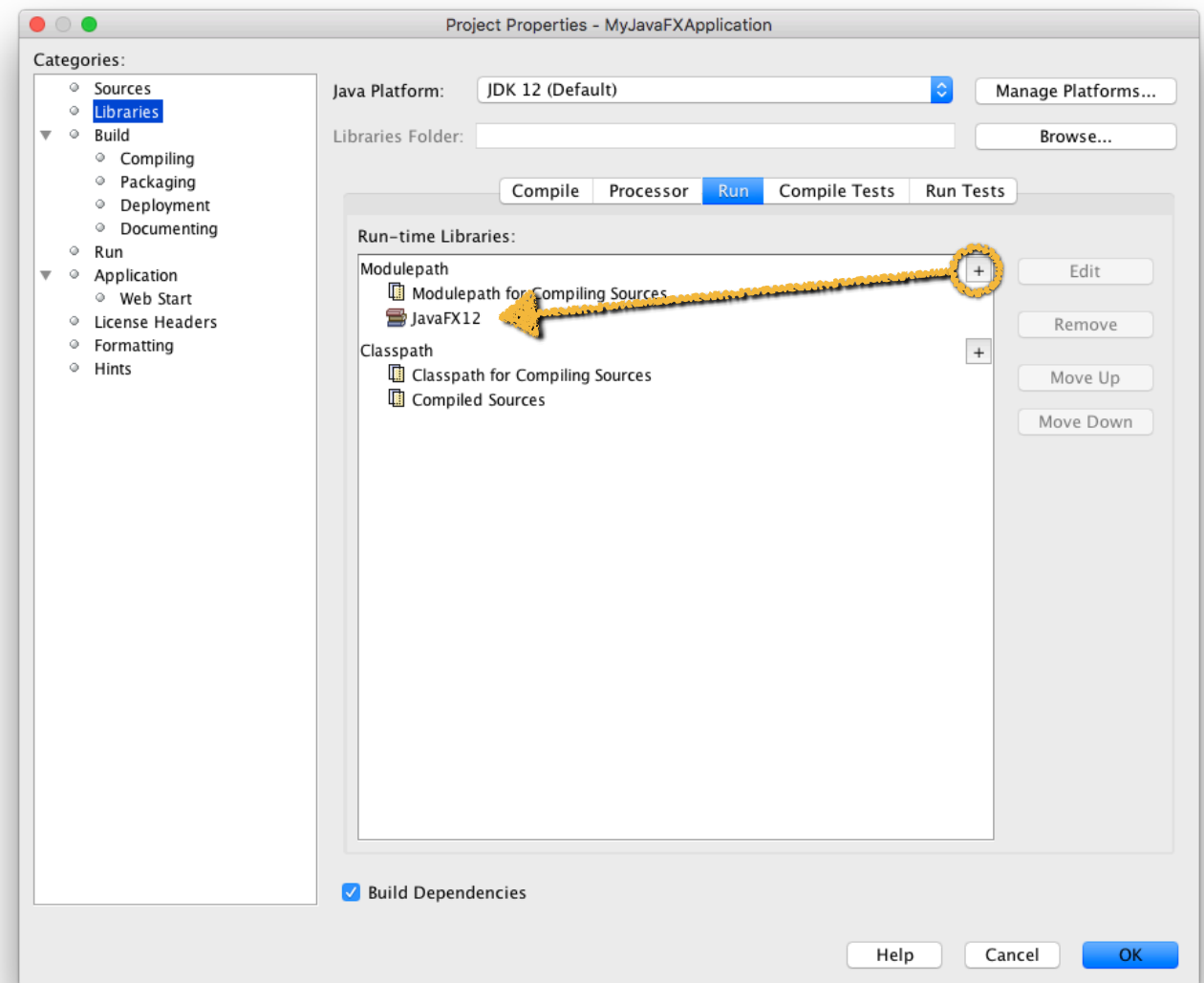


1. Intro

ANNULÉ**Rester sur Oracle Java SE Development Kit 8u281**<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

Utilisation avec NetBeans (3/4)

A la création de chaque projet, ajoutez la bibliothèque précédemment déclarées au chemin pour l'exécution (Project Properties > Libraries > Run) :

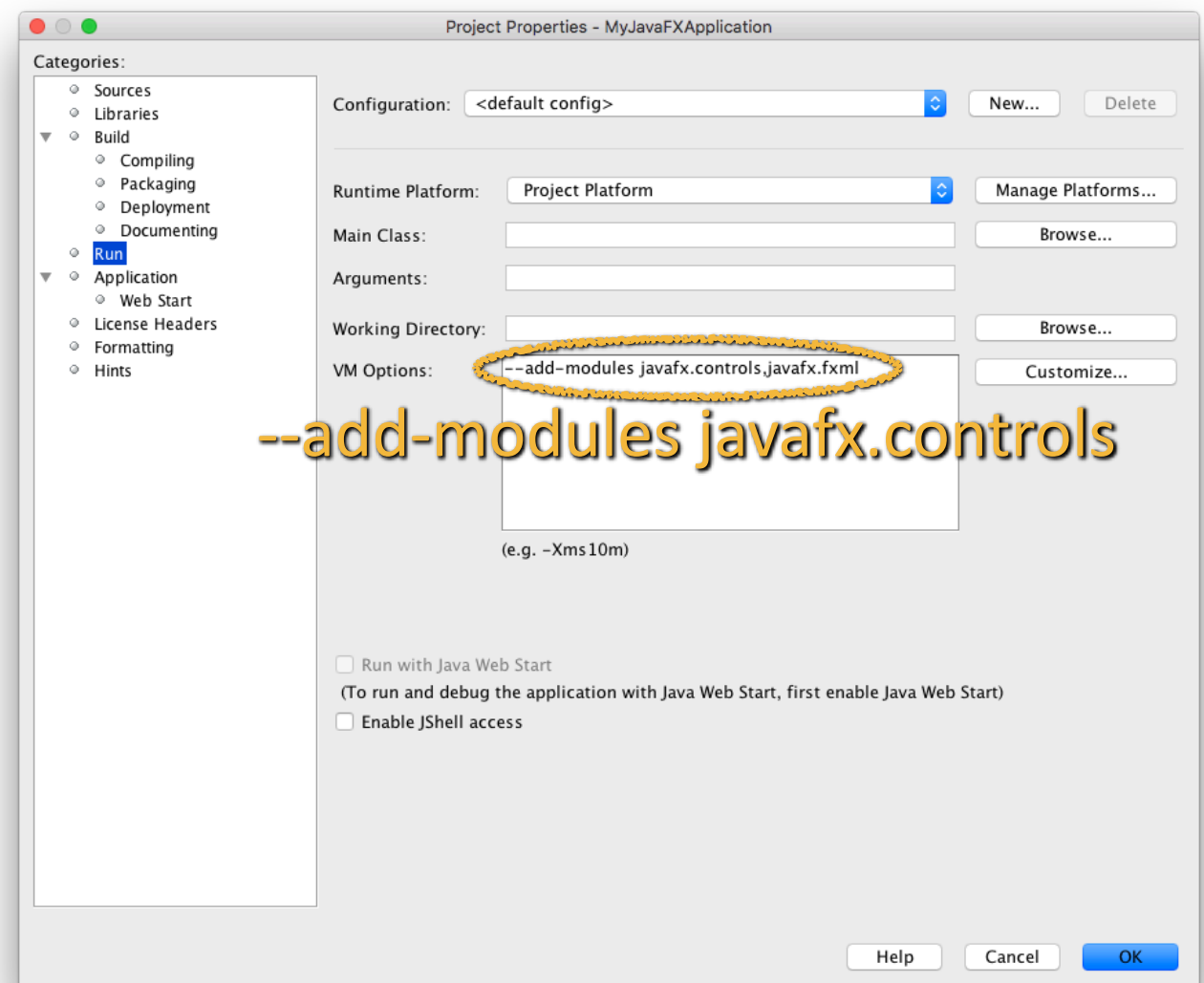


1. Intro

ANNULÉ**Rester sur Oracle Java SE Development Kit 8u281**<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

Utilisation avec NetBeans (4/4)

A la création de chaque projet, ajoutez les options d'exécution pour la machine virtuelle (**Project Properties > Run**):

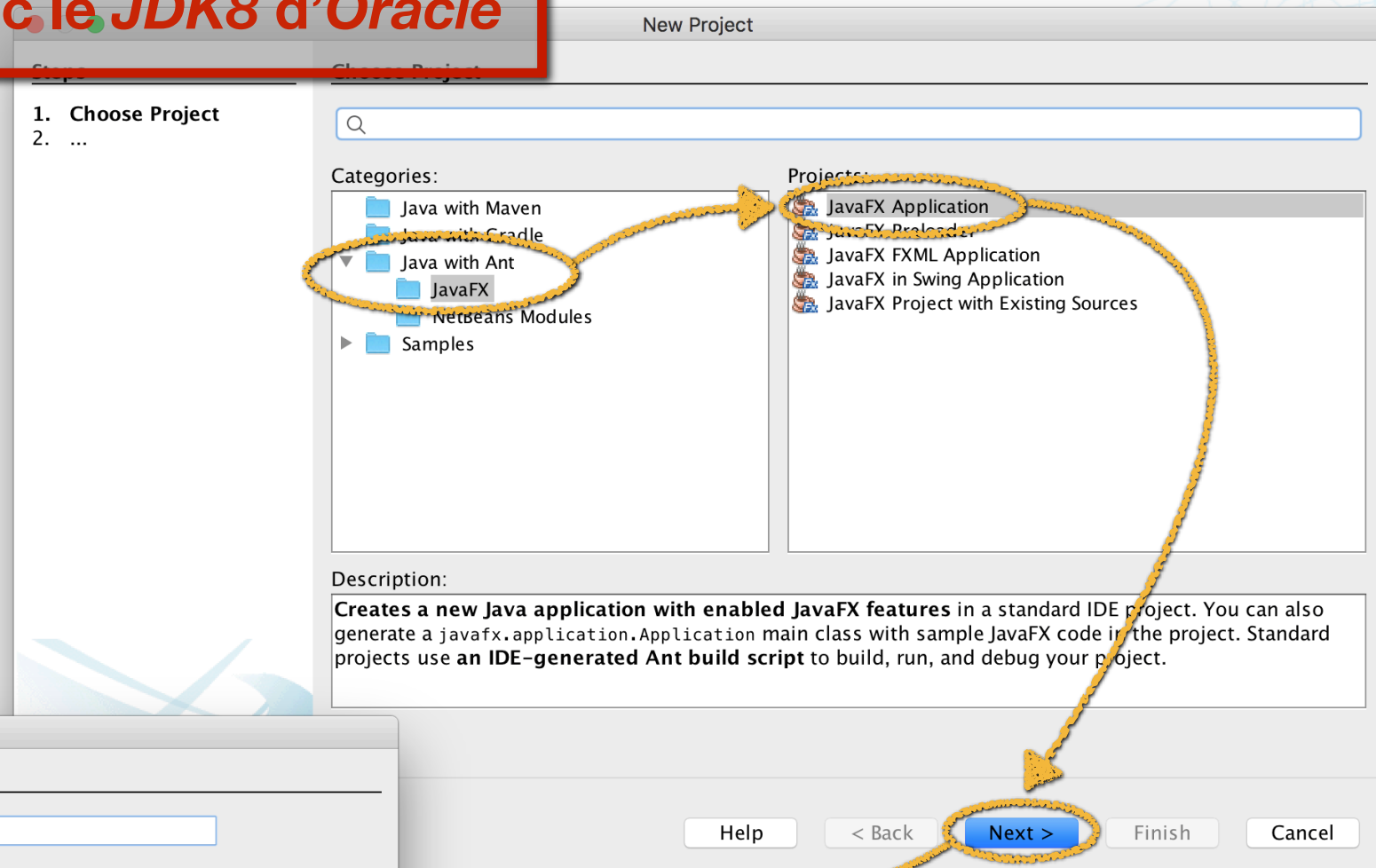
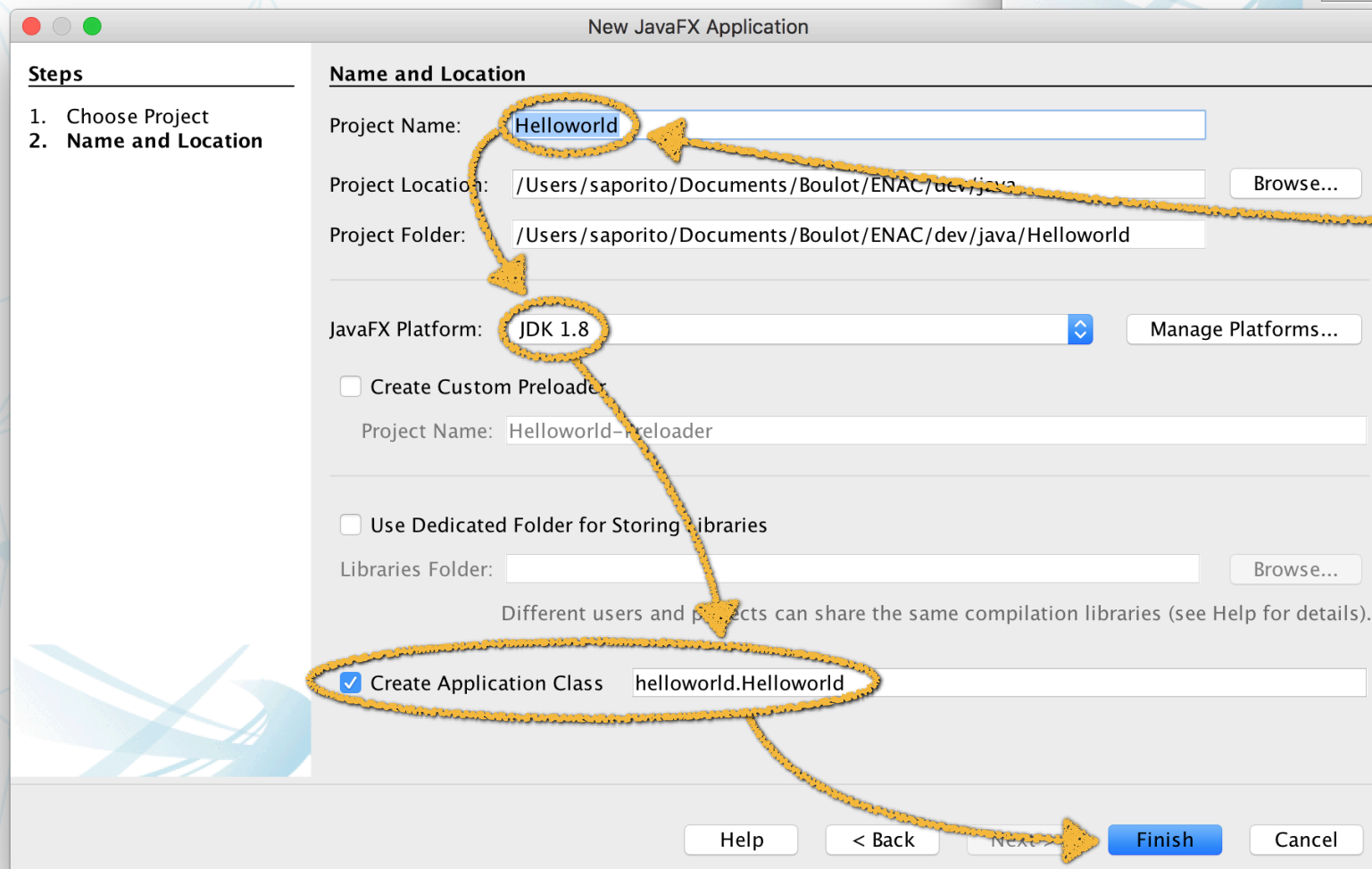


1. Intro

Helloworld!

Créez un nouveau projet
JavaFX : **File > New project**

Avec le JDK8 d'Oracle



1. Intro

Documentation officielle

- Javadoc (*à part de celle de JavaSE*)
<https://docs.oracle.com/javase/8/javafx/api/toc.htm>
- *UI Components* tutorial
<https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/index.html>
- *Layouts* tutorial
<http://docs.oracle.com/javase/8/javafx/layout-tutorial/index.html>
- *Handling events* tutorial
<http://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm>

Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

2.1. Classes constitutives d'une application

Les classes qui constituent l'architecture de base d'une application *JavaFX* sont :

- *Application*
- *Stage*
- *Scene*

2.1. Classes constitutives d'une application

Application

- La classe principale d'une application JavaFX doit hériter de :
javafx.application.Application
- comme toute application java elle a sa méthode *main (String[])*
- qui ne fait en général qu'appeler la méthode *launch (String[])*
 - qui prépare l'application
 - puis appelle la méthode *start (Stage primaryStage)*
 - que l'on doit redéfinir,
 - et qui décrit notre application.

2.1. Classes constitutives d'une application

Application

```
public class MyJavaFXApplication extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        // L'application doit être décrite ici  
    }  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

2.1. Classes constitutives d'une application

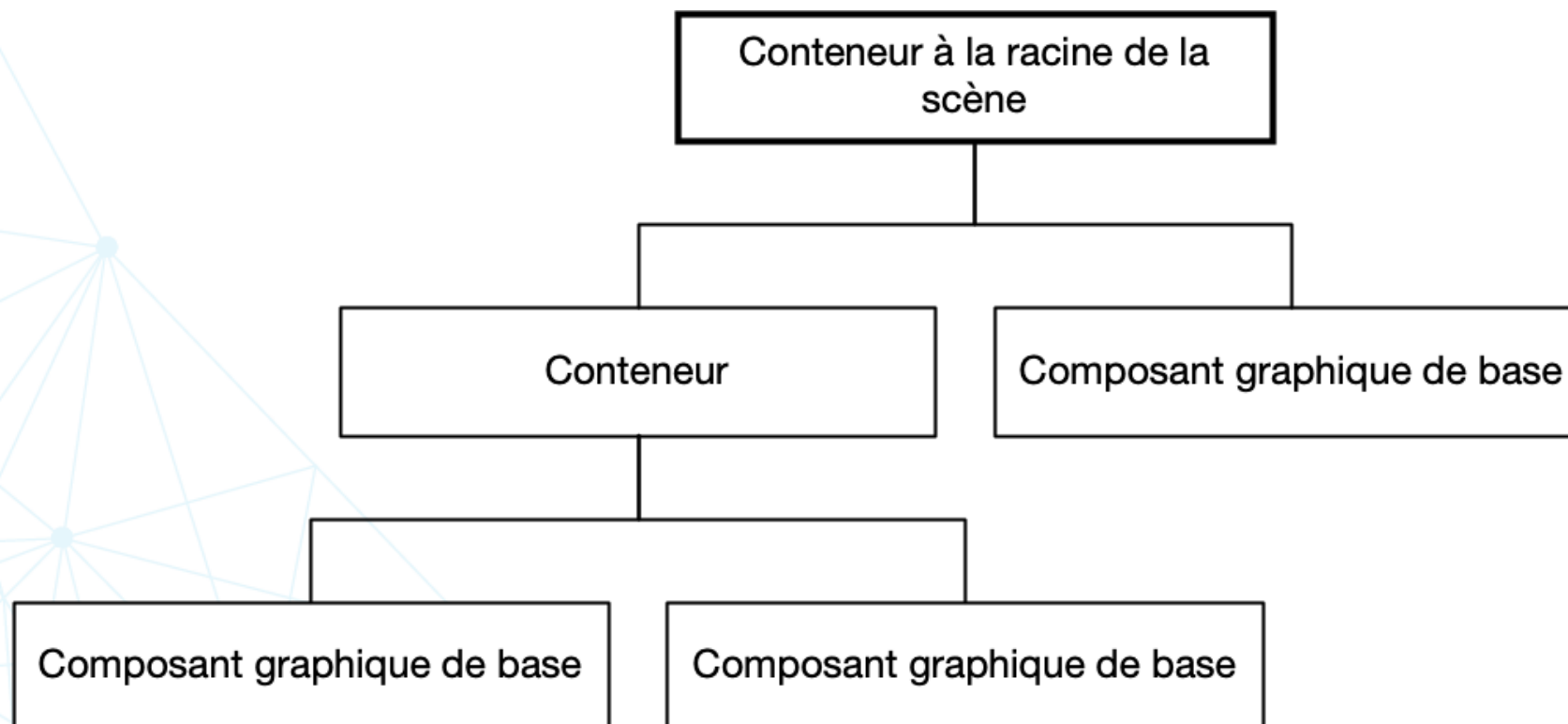
Stage

- Conteneur de plus haut niveau de l'application,
- décoré de manière différente en fonction du cadre d'utilisation :
 - client léger (appli web) : fenêtre du navigateur
 - application desktop : fenêtre classique (influence de l'OS)
- il n'est pas instancié manuellement mais automatiquement passé par l'application en paramètre de la méthode *start(Stage)*
- dedans, on place une ou plusieurs instances de *Scene*.

2.1. Classes constitutives d'une application

Scene

- Référence les composants visuels,
- décrit le **graphe de scène** = la hiérarchie des composants visuels que contient la scène



2.1. Classes constitutives d'une application

Stage & Scene

```
@Override
public void start(Stage primaryStage) {
    // composant constituant la racine du graphe de scène
    StackPane root = new StackPane();

    // ...

    // la scène, créée avec sa racine passée en paramètre
    Scene scene = new Scene(root, 300, 250);

    // la fenêtre, contenant une ou plusieurs scènes
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

2.2. Construction du graphe de scène

Ajouter/enlever des enfants à un conteneur

- on récupère la liste des enfants d'un conteneur grâce à sa méthode *getChildren()*
- on lui ajoute un enfant grâce aux méthodes génériques *add(E)* ou *addAll(E...)* appliquées sur sa liste d'enfants,
- on lui enlève un enfant grâce aux méthodes génériques *remove(Object)* ou *removeAll(E...)* appliquées sur sa liste d'enfants.

2.2. Construction du graphe de scène

Ajouter/enlever des enfants à un conteneur

```
@Override
public void start(Stage primaryStage) {
    // root of the scene graph
    StackPane root = new StackPane();

    // exemple d'enfant ajouté à la racine
    Button btn = new Button("Click me!");
    root.getChildren().add(btn);
    // ...

    // the scene, created with its root as a parameter
    Scene scene = new Scene(root, 300, 250);

    // the stage, containing one or more scenes
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

2.3. Cycle de vie des composants

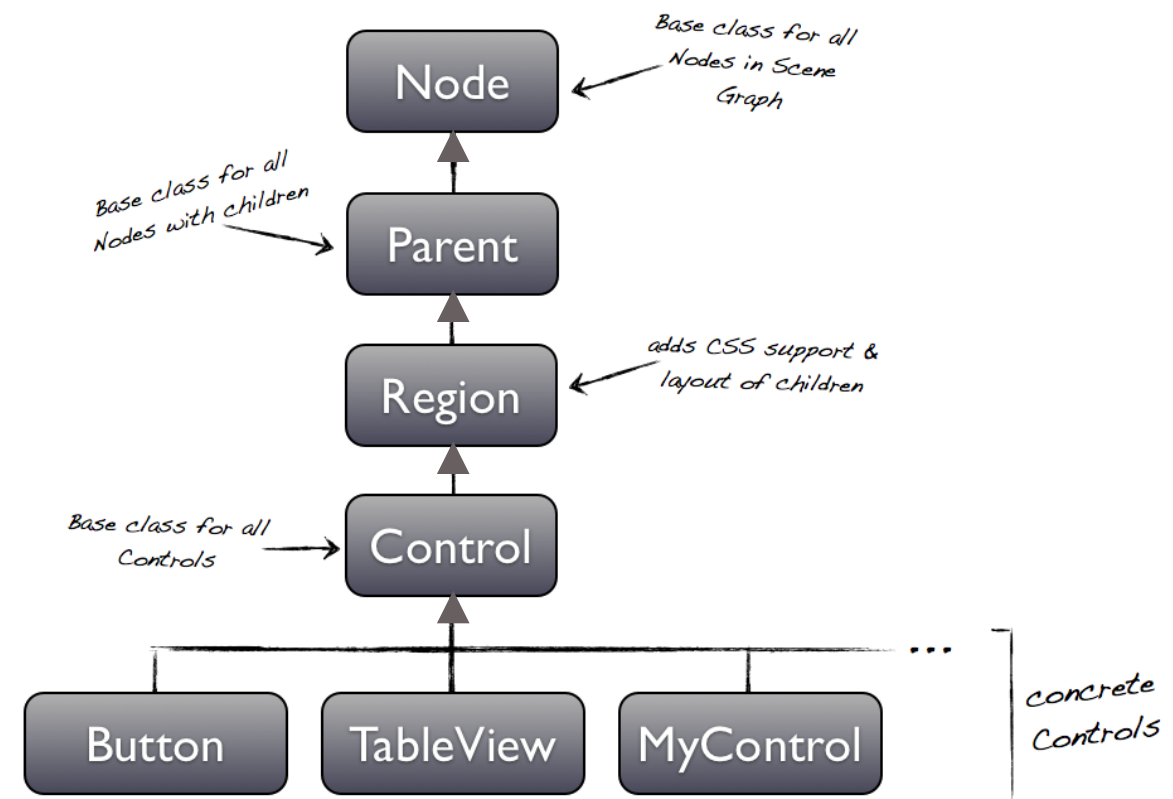
Le cycle de vie d'un composant *JavaFX* est très similaire à celui d'un widget *Qt*

- Le composant est **créé** et **paramétré** via son constructeur,
- **paramétré** par appel de méthodes (en général des *setters*) sur l'objet,
- **parenté** (ajouté à un conteneur, pour contrôler l'imbrication, la géométrie),
- **montré** ou **caché** (montré par défaut si son parent l'est),
- **abonné** à un ou plusieurs *signals events*,
- **réagit** aux événements utilisateur au sein ~~de la boucle principale~~ du thread de l'application,
- est **détruit** (explicitement ou automatiquement par le garbage collector).

2.4. Classes de base du graphe de scène

Les Classes de base du graphe de scène :

- *Node*
- *Parent*
- *Region*



2.4. Classes de base du graphe de scène

Node

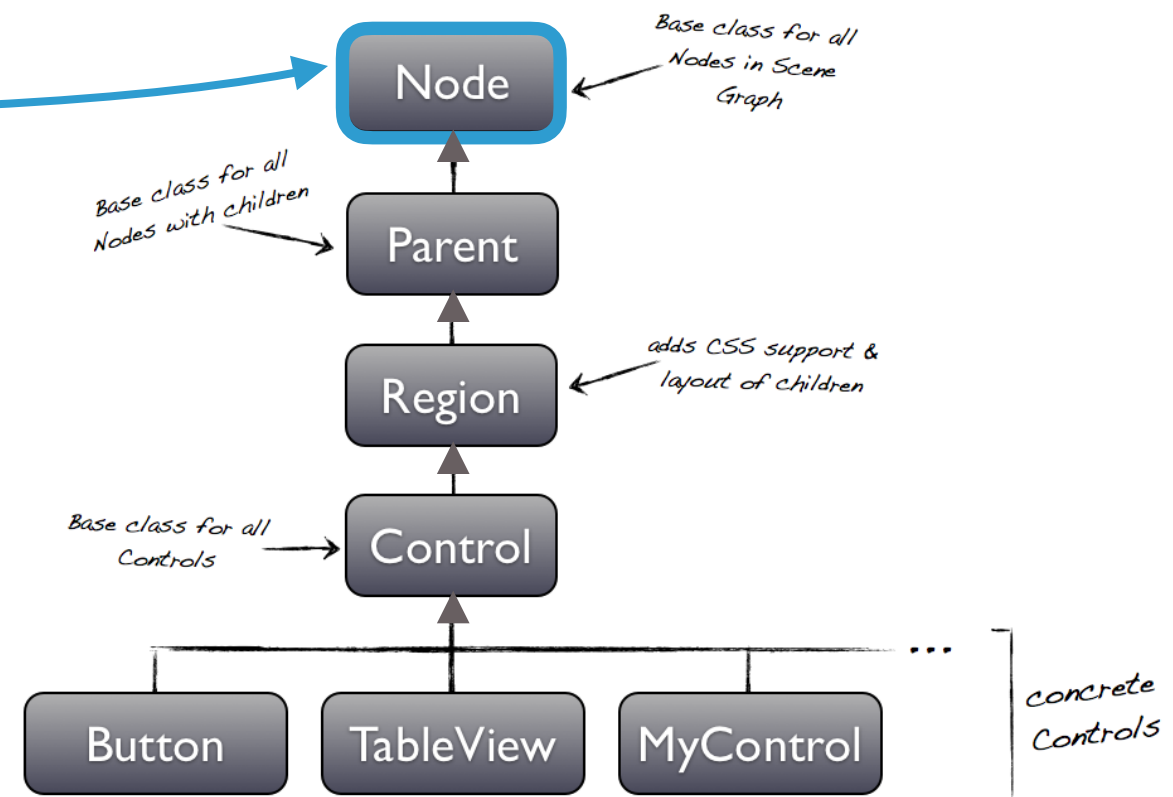
`java.lang.Object`

`javafx.scene.Node`

`javafx.scene.Parent`

`javafx.scene.layout.Region`

- Classe de base de tous les noeuds du graphe de scène,
- elle ne sera jamais utilisée directement mais sera spécialisée en plusieurs types.



2.4. Classes de base du graphe de scène

Parent

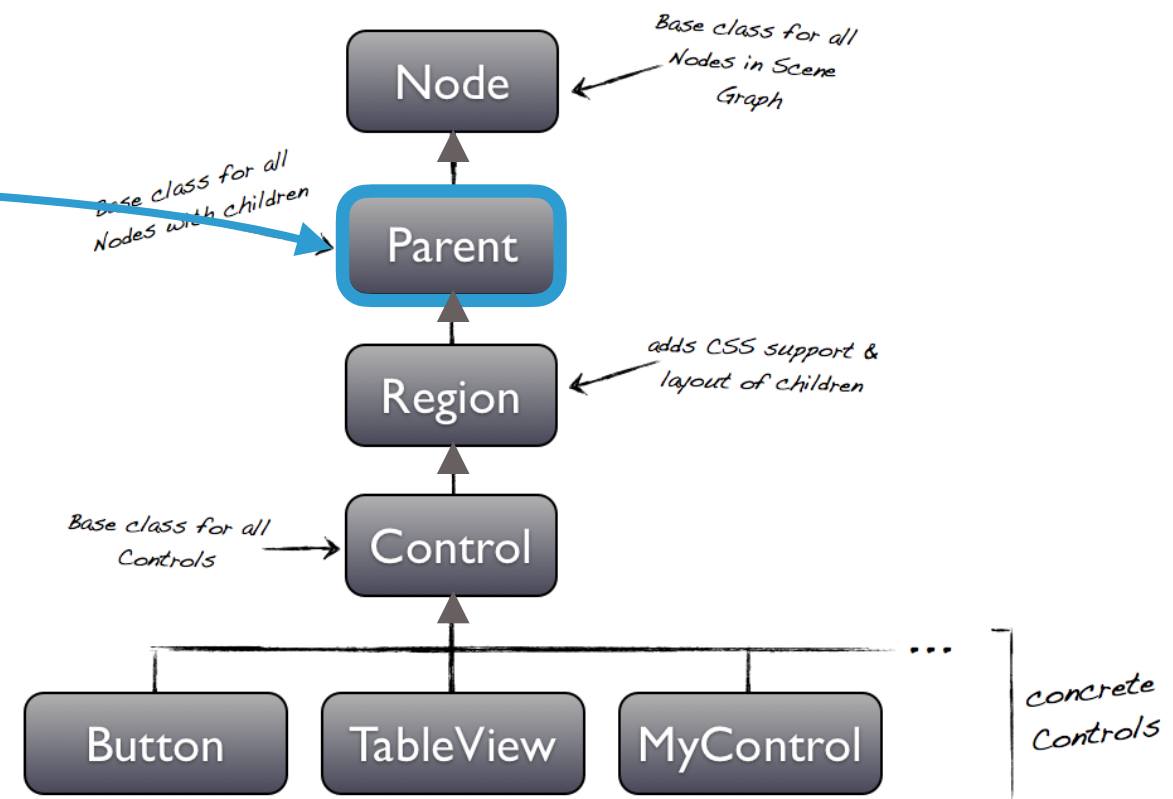
`java.lang.Object`

`javafx.scene.Node`

`javafx.scene.Parent`

`javafx.scene.layout.Region`

- Classe de base pour les noeuds qui ont des enfants,
- contient les mécanismes :
 - d'ajout/retrait d'enfants
 - de calcul de taille et de positionnement (layout)
 - de picking

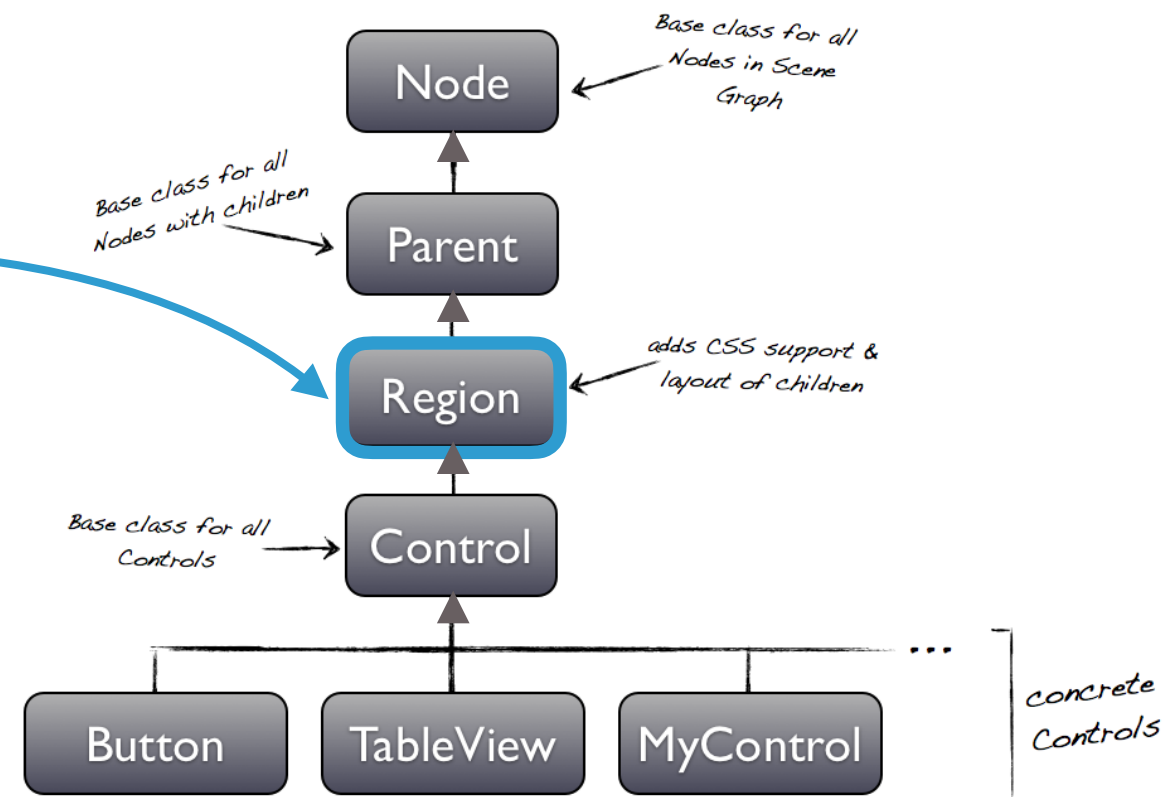


2.4. Classes de base du graphe de scène

Region

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

- Classe de base des noeuds de contrôle interactifs (*UI Components* : *Button*...) ou des conteneurs (*Layout containers* : *StackPane*...),
- tous les composants graphiques héritent donc directement de *Region* ou d'une de ses sous-classes.



2.4. Classes de base du graphe de scène

Noeuds de contrôle interactifs (*UI Components*)

- Un *UI Component* est un composant qui affiche quelque chose, et avec lequel on peut éventuellement interagir.
⇒ équivalent du *widget en Qt*
- Guide d'utilisation des principaux *UI Components* :
<https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/index.html>

2.4. Classes de base du graphe de scène

Conteneurs (*Layout containers*) : Enfants et contraintes de placement

- Un *Layout container* est un composant capable de contenir d'autres composants (UI Components ou Layout containers).
⇒ équivalent du layout en Qt
- Peut utiliser des contraintes de placement paramétrables (dépend du conteneur) : ligne, colonne, différents types de grille...
- Guide d'utilisation des principaux conteneurs :
<http://docs.oracle.com/javase/8/javafx/layout-tutorial/index.html>

2.4. Classes de base du graphe de scène

Conteneurs (*Layout containers*) : Positionnement absolu (1/3)

- Certains conteneurs ne mettent pas en place de contraintes de placement :
Pane, StackPane...
- Leurs enfants sont alors placés manuellement dans le système de coordonnées de leur parent grâce aux méthodes :

setLayoutX(double) / setLayoutY(double)

setTranslateX(double) / setTranslateY(double)

JavaFX 8 *Node* javadoc: *layoutX* establishes the node's stable position and *translateX* optionally makes dynamic adjustments to that position.

2.4. Classes de base du graphe de scène

Conteneurs (*Layout containers*) : Positionnement absolu (2/3)

- Chaque noeud a son propre système de coordonnées (repère),
- les coordonnées de chaque noeud sont exprimées dans le repère de son parent direct,
- tout déplacement d'un noeud provoque mécaniquement en cascade le déplacement de tous ses enfants sans pour autant changer leurs coordonnées (dans le repère de leur parent, donc du noeud déplacé).

2.4. Classes de base du graphe de scène

Conteneurs (*Layout containers*) : Positionnement absolu (3/3)

- Chaque noeud sait faire les conversions dans les deux sens entre son propre repère et celui de son parent / de la scène / de l'écran :

localToParent (...) / *localToScene (...)* / *localToScreen (...)*

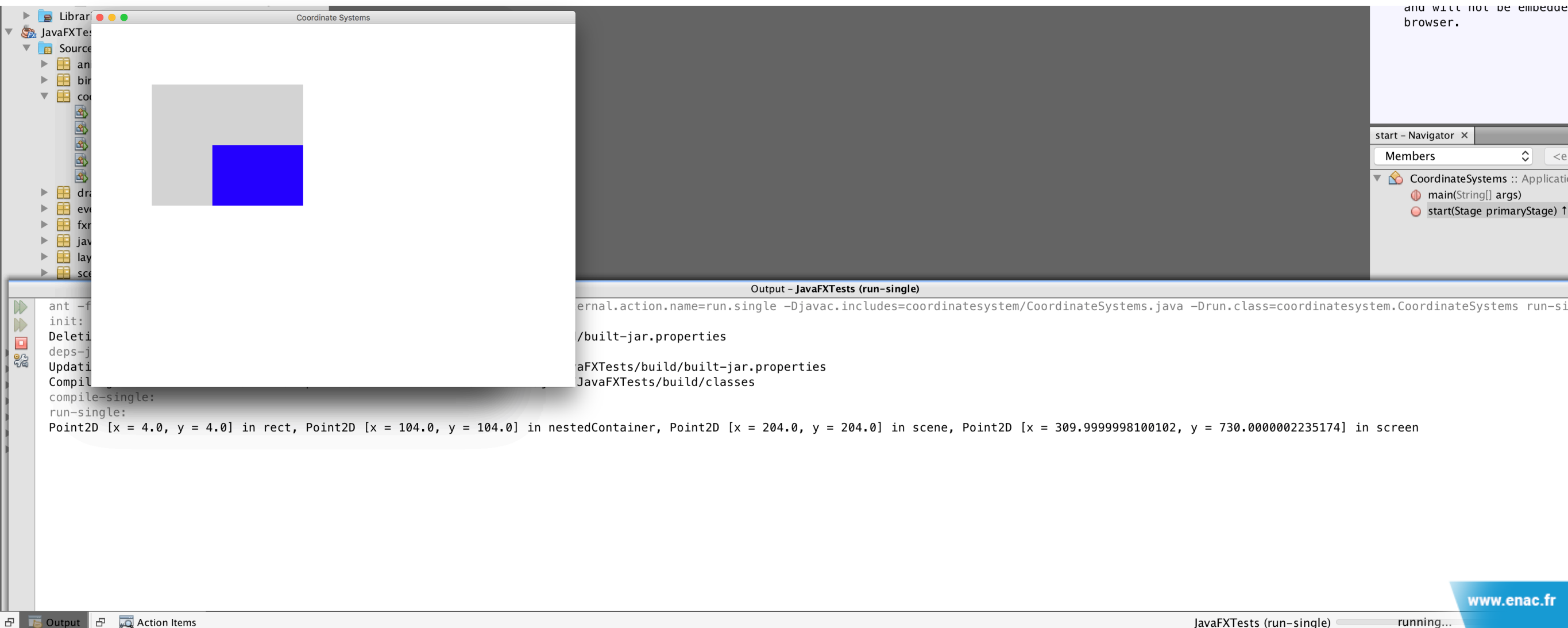
parentToLocal (...) / *sceneToLocal (...)* / *screenToLocal (...)*

- Cela simplifie considérablement les calculs car il ne s'agit pas toujours de changements de repères orthonormés et de même échelle.

2.4. Classes de base du graphe de scène

TP 1 - Exercice 1

Graphe de scène, noeuds et systèmes de coordonnées



Coordinate Systems

CoordinateSystems :: Application

- main(String[] args)
- start(Stage primaryStage)

Output - JavaFXTests (run-single)

```

external.action.name=run.single -Djavac.includes=coordinatesystem/CoordinateSystems.java -Drun.class=coordinatesystem.CoordinateSystems run-si
/built-jar.properties
aFXTests/build/built-jar.properties
JavaFXTests/build/classes
compile-single:
run-single:
Point2D [x = 4.0, y = 4.0] in rect, Point2D [x = 104.0, y = 104.0] in nestedContainer, Point2D [x = 204.0, y = 204.0] in scene, Point2D [x = 309.9999998100102, y = 730.0000002235174] in screen

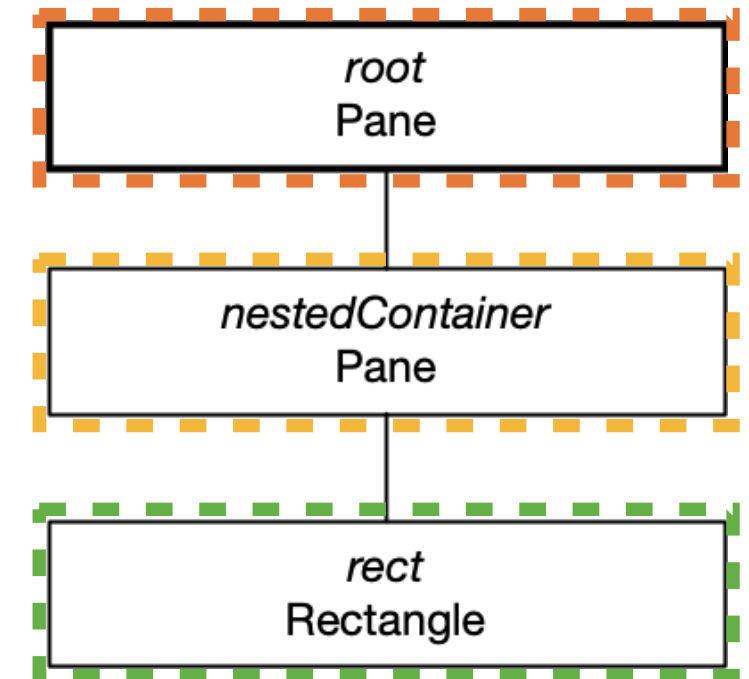
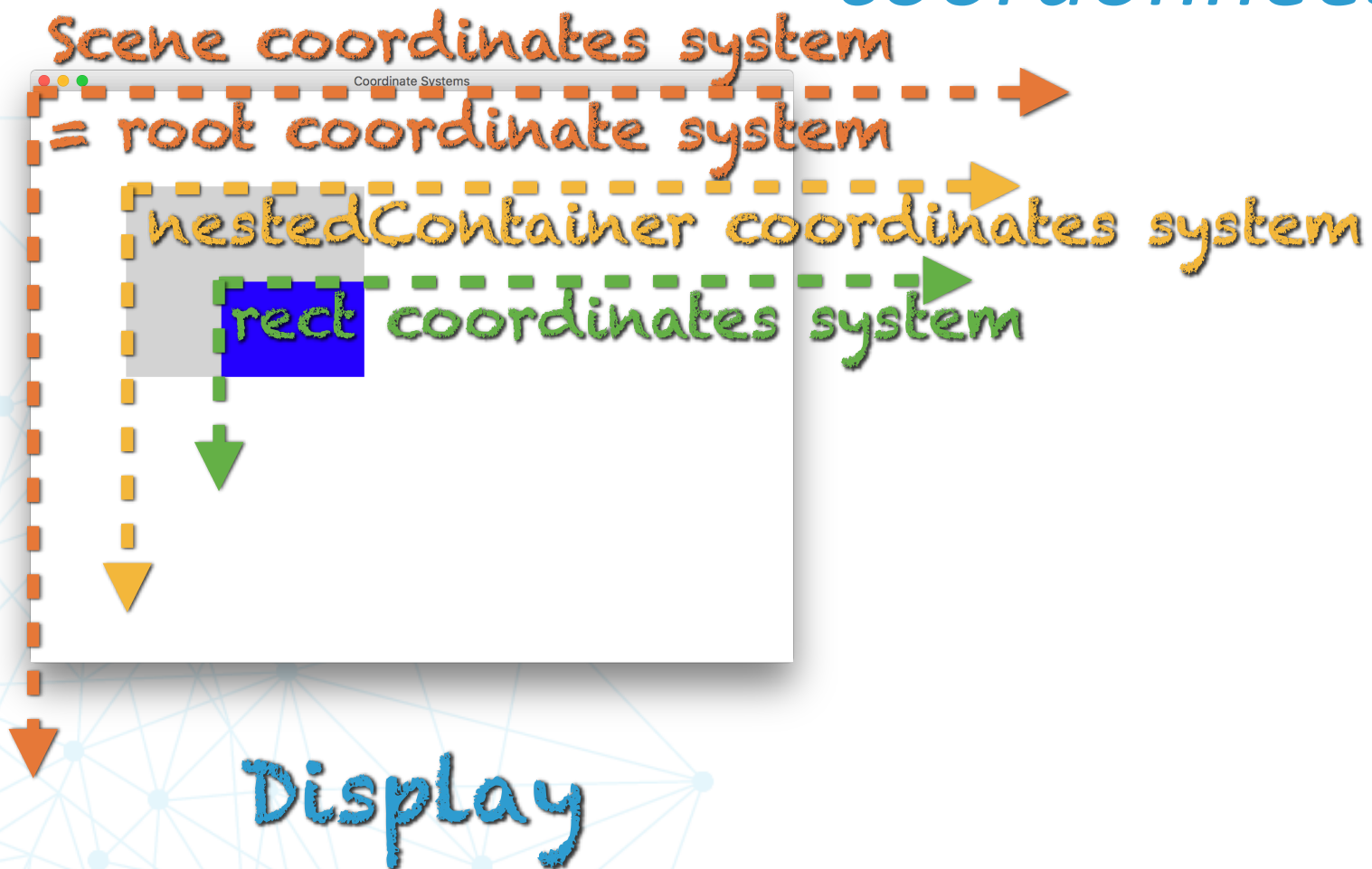
```

JavaFXTests (run-single) running...

2.4. Classes de base du graphe de scène

TP 1 - Exercice 1

Graphe de scène, noeuds et systèmes de coordonnées



Scene graph

Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

3. Transformations

On peut appliquer toutes sortes de transformations à un noeud :

- translations (*Translate*),
- mises à l'échelle (*Scale*),
- rotations (*Rotate*),
- mais aussi cisaillement (*Shear*),
- ou toute autre transformation affine customisée (*Affine*).

3. Transformations

Transformations et coordonnées

- Les déplacements déjà vus sont un cas particulier des transformations.
- De façon prévisiblement analogue **toute transformation effectuée sur un noeud se répercute en cascade sur tous ses enfants** sans pour autant changer leurs coordonnées (dans le repère de leur parent).
- Les calculs de changement de repère ne sont alors plus du tout triviaux...
- ...et les méthodes de conversion prennent tout leur sens !

3. Transformations

Appliquer une transformation (1/2)

Il existe des méthodes de commodité (convenience methods) héritées de la classe **Node** :

setTranslateX(double) / setTranslateY(double)
setScaleX(double) / setScaleY(double)
setRotate(double)...

Mais les transformations obtenues par ces méthodes sont **peu paramétrables**. Par exemple la mise à l'échelle et la rotation ne peuvent s'effectuer qu'autour du centre de la bounding box du noeud.

3. Transformations

Appliquer une transformation (2/2)

Pour dépasser les limitations des méthodes de commodité on doit utiliser le mécanisme général d'application des transformations :

- Instancier une transformation et l'ajouter à la liste des transformations du noeud :

```
Rotate myRotation = new Rotate(angle, pivotX, pivotY);  
myNode.getTransforms().add(myRotation);
```

- puis modifier au besoin la transformation :

```
myRotation.setAngle(45);
```

...

3. Transformations

Accumuler des transformations

Pour accumuler des transformations successives on peut aussi :

- Ajouter une transformation affine unique dans la liste des transformations :

```
Affine myTransforms = new Affine();  
myNode.getTransforms().add(myTransforms);
```

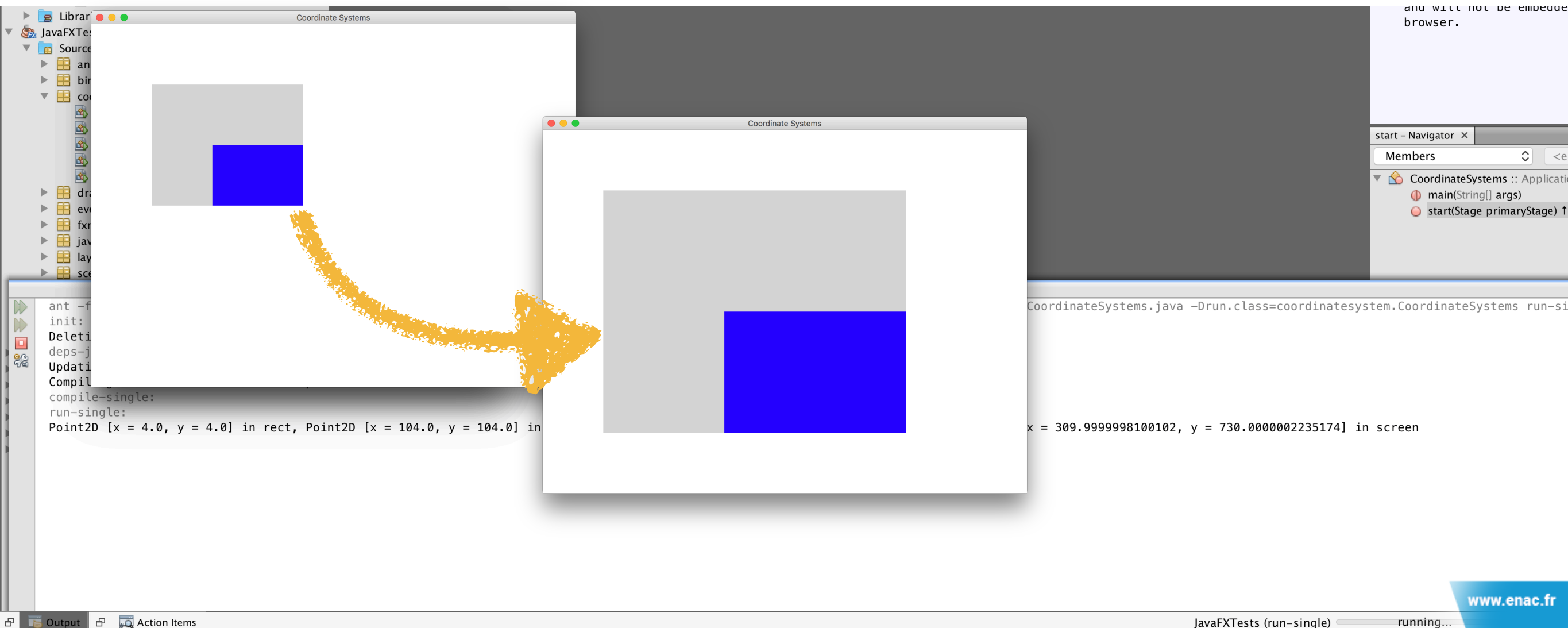
- puis la modifier grâce à ses méthodes ***prependXxx()*** ou ***appendXxx()*** :

```
myTransforms.appendScale(scaleX, scaleY, pivotX, pivotY);  
myTransforms.appendRotation...
```

3. Transformations

TP 1 - Exercice 2

Transformations et systèmes de coordonnées



Coordinate Systems

Coordinate Systems

CoordinateSystems.java -Drun.class=coordinatesystem.CoordinateSystems run-si

x = 309.9999998100102, y = 730.0000002235174] in screen

ant -f
init:
Deleti
deps-j
Updati
Compil
compile-single:
run-single:
Point2D [x = 4.0, y = 4.0] in rect, Point2D [x = 104.0, y = 104.0] in

start - Navigator x
Members
CoordinateSystems :: Applicati
main(String[] args)
start(Stage primaryStage) t

JavaFXTests (run-single) running...

Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

4. Programmation événementielle

La *programmation événementielle* est un paradigme dans lequel l'exécution du programme est déterminée par des événements.

Un **événement** peut être une **action de l'utilisateur** (clic souris, touche enfoncée...), une **mesure d'un capteur**, un **message** depuis un autre composant logiciel...

Un composant logiciel peut **s'abonner** à un événement, ce qui lui permettra de réagir quand l'événement surviendra.

Dans les systèmes interactifs, tout est guidé par des **réactions à des événements**...

4.1. Property

Quand on a besoin de réagir aux changements de valeur d'une donnée particulière...

Que ces changements soient provoqués par une action de l'utilisateur ou pas, on doit :

- encapsuler la donnée à observer dans une *Property* (si ce n'est pas déjà le cas),
- écouter les changements de cette *Property* grâce à l'un des deux mécanismes suivants
 - *binding* (simple recopie de donnée d'un composant vers un autre),
 - *listener* (réactions plus complexes).

4.1. Property

Property (1/3)

Property est une *interface* spécifiant le mécanisme nécessaire pour écouter les changements de valeur d'une donnée.

- Chacune de ses réalisations encapsule une donnée d'un type particulier.
- On peut donc écouter tout type de donnée en utilisant la réalisation de ***Property*** qui y correspond, par exemple :
 - ***SimpleBooleanProperty*** pour observer un booléen,
 - ***SimpleIntegerProperty*** pour observer un entier,
 - ***SimpleDoubleProperty*** pour observer un double,
 - ...

4.1. Property

Property (2/3)

- Les attributs des composants *JavaFX* sont des *Property*. Elles sont donc liables à d'autres *Property*, que ces dernières proviennent d'un composant fourni par *JavaFX* ou qu'elles soient créées par le développeur.
- On peut accéder ou modifier la valeur encapsulée dans une *Property* grâce à ses méthodes standard *get()* et *set(...)*
- ...mais il vaut mieux réserver ces dernières à un usage interne à la classe où la *Property* est définie et créer des accesseurs/modificateurs spécifiques (conventions *JavaFX* ≠ *Java* classique)...

4.1. Property

Property (3/3)

- La *Property* sera stockée comme un attribut privé,
- et sera exposée selon les conventions de nommage utilisées dans les composants JavaFX :
 - *xxx* étant le nom de la poignée sur la *Property*,
 - *getXxx()* permet d'obtenir la valeur encapsulée (ce n'est donc pas un accesseur au sens conventionnel du terme),
 - *setXxx(...)* permet de fixer la valeur encapsulée (ce n'est donc pas un modifieur au sens conventionnel du terme),
 - *xxxProperty()* permet d'obtenir la *xxx Property* elle-même.

4.2. Réagir à un changement de valeur: *Binding*

Liaison de données : *binding*

Le mécanisme de *binding* crée une liaison entre les valeurs de deux *Property* :

- La liaison peut être unidirectionnelle ou bidirectionnelle.
- Elle se met en oeuvre grâce à l'une des deux méthodes suivantes spécifiée dans l'interface *Property* :
 - *prop1.bind(prop2)* pour une **liaison unidirectionnelle** (la valeur de *prop2* est systématiquement recopiée dans celle de *prop1* en cas de changement),
 - *prop1.bindBidirectional(prop2)* pour une **liaison bidirectionnelle** (toute nouvelle valeur de l'une des deux est recopiée dans l'autre).

4.2. Réagir à un changement de valeur: *Binding*

Liaison de données : *binding* et conversions de type (1/2)

Que se passe-t-il si les *Property* à lier ne sont pas du même type ?

- Erreur de compilation : types incompatibles !
- Il faut intégrer un mécanisme de conversion au *binding*.
- On trouve les mécanismes utilisables dans la doc de l'implémentation spécifique de la *Property* que l'on souhaite utiliser (ils dépendent évidemment du type de la valeur encapsulée).

4.2. Réagir à un changement de valeur: *Binding*

Liaison de données : *binding* et conversions de type (2/2)

Exemples pour une liaison entre *Double* et *String* :

- **unidirectionnelle**

méthode *asString()* définie dans la classe *NumberExpressionBase* (dont héritent les propriétés encapsulant un nombre)

```
myStringProperty.bind(myDoubleProperty.asString());
```

- **bidirectionnelle**

classe *StringConverter* assurant la conversion dans les deux sens

```
StringConverter sc = new DoubleStringConverter();
```

```
myStringProperty.bindBidirectional(myDoubleProperty, sc);
```

4.2. Réagir à un changement de valeur: *Binding*

Liaison de données : *binding* et opérations

Le mécanisme de *binding* permet aussi d'effectuer un certain nombre d'opérations mathématiques dépendant du type de la valeur à recopier (*add*, *subtract*, *multiply*...).

Exemple:

La copie réactive de la somme des valeurs de *prop1* et *prop2* dans la valeur de *prop3* se fait en créant le *binding* *prop3.bind(prop1.add(prop2))* ;

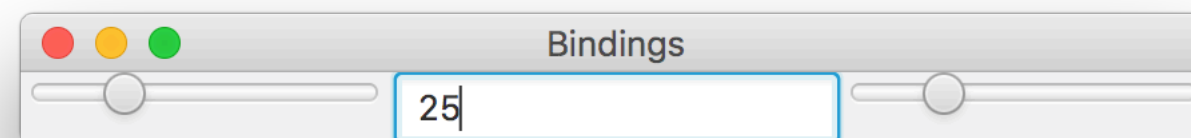
4.2. Réagir à un changement de valeur: *Binding*

TP 1 - Exercice 3

Liaison de données

TP 1 - Exercice 4

Modèle de données



4.3. Réagir à un changement de valeur : *ChangeListener*

Pour des réactions plus complexes aux changements de valeur d'une *Property* (ou si il y a nécessité de créer des effets de bord), on fait appel au mécanisme des *listeners* :

- Creation d'un *ChangeListener*,
- abonnement de ce *ChangeListener* sur la *Property* grâce à sa méthode *addListener()*.

4.3. Réagir à un changement de valeur : *ChangeListener*

ChangeListener

- *ChangeListener<T>* est une interface fonctionnelle qui permet de décrire la réaction à un changement de valeur,
- elle comporte une unique méthode *changed(ObservableValue<? extends T> observable, T oldValue, T newValue)*
- pour décrire la réaction il faut créer et instancier une classe qui implémente l'interface et placer le traitement à effectuer dans sa méthode *changed(...)*,
- les informations passées en paramètres permettent de récupérer une poignée sur la donnée observée, son ancienne valeur et sa nouvelle valeur.

4.3. Réagir à un changement de valeur : *ChangeListener*

Implémenter *ChangeListener* et s'abonner à un changement de valeur (instance nommée)

```
DoubleProperty myDoubleProperty = new SimpleDoubleProperty();

ChangeListener myListener = new ChangeListener<Number>() {
    @Override
    public void changed(ObservableValue<? extends Number> obs,
                       Number oldValue, Number newValue) {
        // réaction aux changements de valeur de myDoubleProperty
    }
};

myDoubleProperty.addListener(myListener) ;
```

4.3. Réagir à un changement de valeur : *ChangeListener*

Implémenter *ChangeListener* et s'abonner à un changement de valeur (classe interne anonyme)

```
DoubleProperty myDoubleProperty = new SimpleDoubleProperty();  
  
myDoubleProperty.addListener(new ChangeListener<Number>() {  
    @Override  
    public void changed(ObservableValue<? extends Number> obs,  
                        Number oldValue, Number newValue) {  
        // réaction aux changements de valeur de myDoubleProperty  
    }  
});
```

4.3. Réagir à un changement de valeur : *ChangeListener*

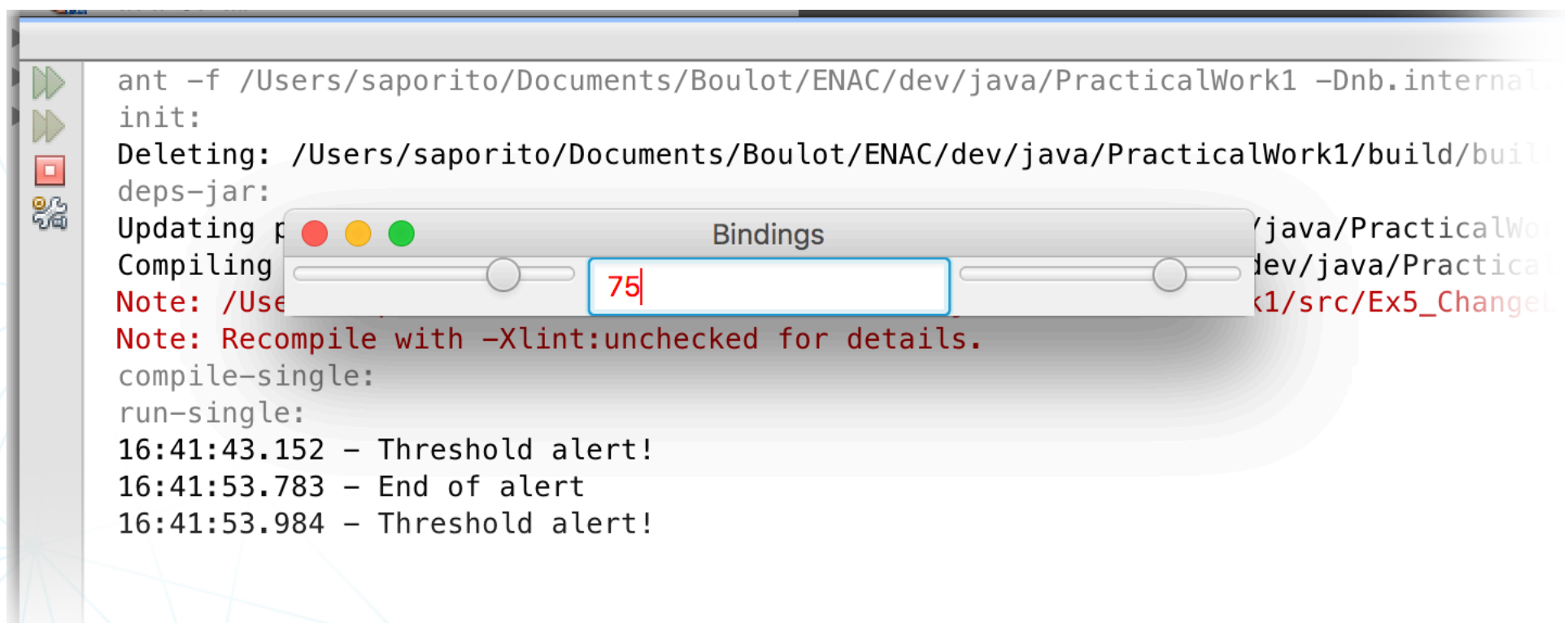
Implémenter *ChangeListener* et s'abonner à un changement de valeur (fonction lambda)

```
DoubleProperty myDoubleProperty = new SimpleDoubleProperty();  
myDoubleProperty.addListener((obs, oldValue, newValue) -> {  
    // réaction aux changements de valeur de myDoubleProperty  
});
```


4.3. Réagir à un changement de valeur : *ChangeListener*

TP 1 - Exercice 5

Réagir à un changement de valeur



The screenshot shows an IDE window with a terminal output on the left and a 'Bindings' dialog box in the foreground. The terminal output shows the following commands and messages:

```
ant -f /Users/saporito/Documents/Boulot/ENAC/dev/java/PracticalWork1 -Dnb.internal  
init:  
Deleting: /Users/saporito/Documents/Boulot/ENAC/dev/java/PracticalWork1/build/build  
deps-jar:  
Updating p  
Compiling  
Note: /Use  
Note: Recompile with -Xlint:unchecked for details.  
compile-single:  
run-single:  
16:41:43.152 - Threshold alert!  
16:41:53.783 - End of alert  
16:41:53.984 - Threshold alert!
```

The 'Bindings' dialog box is titled 'Bindings' and has a text input field containing the value '75'. The dialog box also has a 'Bindings' label and a 'Bindings' button.

4.3. Réagir à un changement de valeur : *ChangeListener*

On peut également observer les changements au sein d'un objet complexe : *ArrayList*

```
// ArrayList Java standard
private List<Flight> list = new ArrayList<>();

// rendre la liste observable
private ObservableList<Flight> observableList =
    FXCollections.observableArrayList(list);

private ListChangeListener<Flight> myListener = new
    ListChangeListener<Flight>() {
    // utiliser la complétion et la doc pour réagir
    // aux ajouts/suppressions d'éléments
};

observableList.addListener(myListener);
```

4.3. Réagir à un changement de valeur : *ChangeListener*

On peut également observer les changements au sein d'un objet complexe : *HashMap*

```
// HashMap Java standard
private Map<Integer, Flight> map = new HashMap<>();

// rendre la HashMap observable
private ObservableMap<Integer, Flight> observableMap =
    FXCollections.observableMap(map);

private MapChangeListener<Integer, Flight> myListener = new
    MapChangeListener<Integer, Flight>() {
    // utiliser la complétion et la doc pour réagir
    // aux ajouts/suppressions d'éléments
};

observableMap.addListener(myListener);
```

4.3. Réagir à un changement de valeur : *ChangeListener*

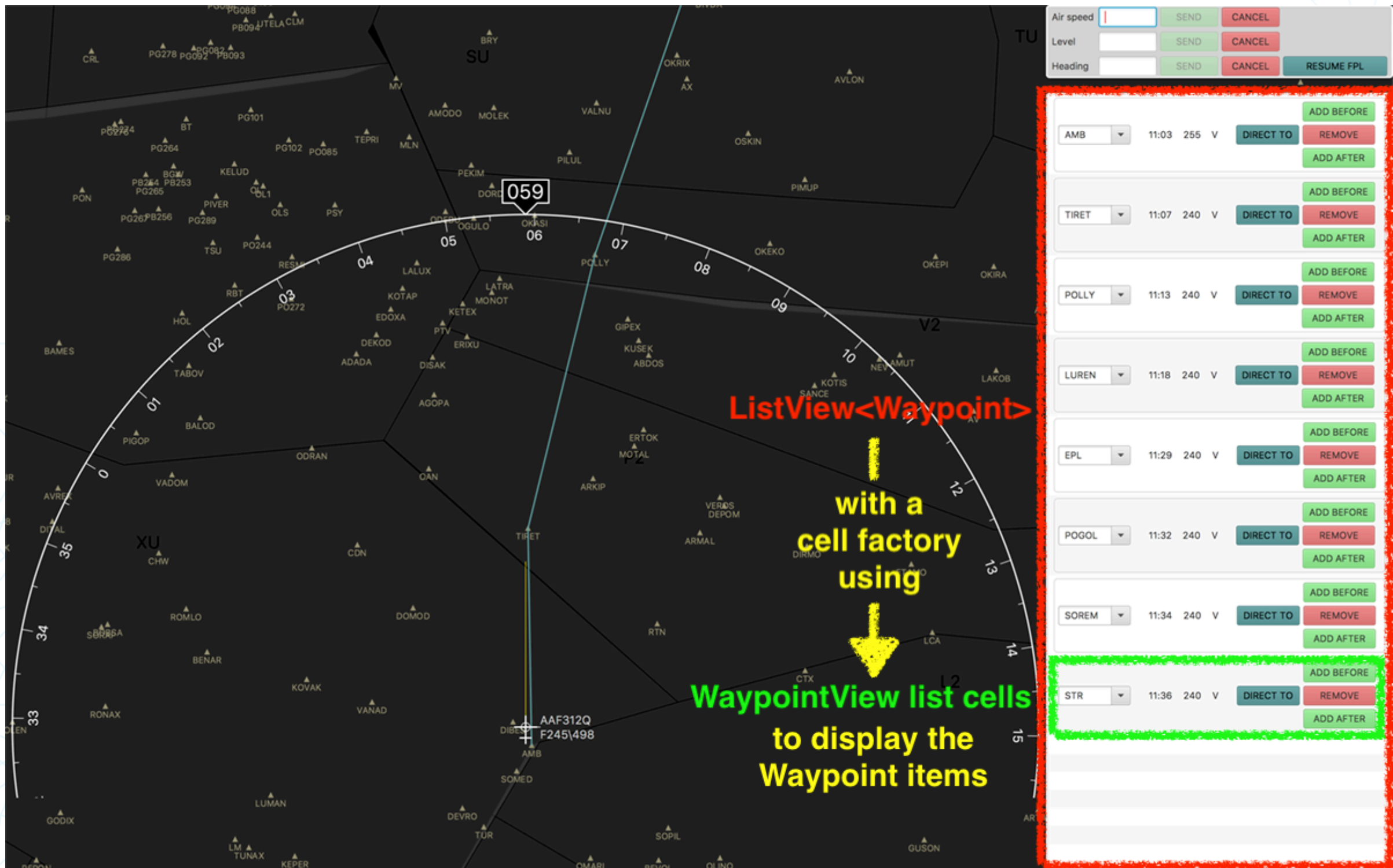
De nombreux composants *JavaFX* utilisent ces services d'observabilité : *ListView*

```
// Liste observable jouant le rôle de modèle de données
private ObservableList<Waypoint> route = ...;

// La ListView en constitue une vue se mettant automatiquement
// à jour en cas d'ajout/suppression d'éléments à la liste.
private ListView<Waypoint> waypointList = new
    ListView<>(route);

// Cette vue est sous forme textuelle par défaut,
// ou sous forme graphique si on ajoute une CellFactory
waypointList.setCellFactory((ListView<Waypoint> listView) ->
    new WaypointView());
```


4.3. Réagir à un changement de valeur : *ChangeListener*



WaypointView list cells to display the Waypoint items

ListView<Waypoint>

with a cell factory using

Waypoint	Time	Altitude	Speed	Direction	Buttons
AMB	11:03	255	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
TIRET	11:07	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
POLLY	11:13	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
LUREN	11:18	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
EPL	11:29	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
POGOL	11:32	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
SOREM	11:34	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER
STR	11:36	240	V	DIRECT TO	REMOVE, ADD BEFORE, ADD AFTER

4.4. Event

Dans beaucoup de cas il n'est pas suffisant de juste réagir à un changement de valeur. *JavaFX* offre un autre mécanisme similaire à celui du signal/slot *Qt*, mais en plus complet.

- *signal* Event
- *slot* EventHandler
- Il y a des *signaux* événements de base
- on peut créer nos propres *signaux* événements
- les *signaux* événements ont un cycle de propagation très riche : on peut s'y abonner de plusieurs façons et dans plusieurs phases

4.4. *Event*

JavaFX fournit des événements standards

- *ActionEvent* pour une action simple (clic bouton par ex),
- *MouseEvent* pour les survols/clics souris,
- *ScrollEvent* pour les défilements à l'aide de la molette souris, du trackpad, d'un écran tactile...
- *TouchEvent* pour les touchers sur écran tactile,
- ...

4.4. Event

Un événement convoie des informations

- **type** : information additionnelle sur le type (*`MouseEvent.MOUSE_PRESSED`*, ...)
- **target** : cible, dépend du type d'événement (pour les événements souris c'est le noeud le plus haut placé sous le pointeur)
- **source** : le noeud sur lequel on s'est abonné
- **position** : position du pointeur dans le repère de la source / la scène / l'écran
- ...
- et éventuellement des infos spécifiques à l'événement.

Toutes ces informations sont accessible par des *getters* :

`getType()` , *`getTarget()`* , *`getSource()`* , *`getX()`* , *`getSceneX()`*...

4.4. *Event*

On peut aussi créer nos propres événements

- Il suffit de créer une classe héritant de `javafx.event.Event` ou d'une de ses sous-classes.
- Intérêt ?
 - convoyer des informations particulières
 - obtenir un couplage faible entre les composants

4.5. *EventHandler*

EventHandler

- *EventHandler*<*T extends Event*> est une interface fonctionnelle qui spécifie la façon de réagir à un événement,
- elle comporte une unique méthode *handle(T event)* qui a pour paramètre l'événement géré,
- pour décrire la réaction à un événement il faut créer et instancier une classe qui implémente l'interface et placer le traitement à effectuer dans sa méthode *handle(...)*,
- toutes les informations convoyées par l'événement y seront récupérables.

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (1/6)

- La méthode la plus courte et la plus facile à mémoriser pour s'abonner à un événement consiste à utiliser les convenience methods.

- Syntaxe :

setOnXXX (EventHandler<T>)

où **XXX** est le type de l'événement (sans la terminaison Event),
et **T** la classe de l'événement écouté.

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (2/6)

Instance nommée d'une classe implémentant ***EventHandler*** :

```
Button btn = new Button("Click me!");

EventHandler<ActionEvent> myHandler =
    new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent e) {
            System.out.println("Hello World!");
        }
    };

btn.setOnAction(myHandler) ;
```

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (3/6)

Classe interne anonyme :

```
Button btn = new Button("Click me!");  
  
btn.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Hello World!");  
    }  
});
```

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (4/6)

Expression lambda :

```
Button btn = new Button("Click me!");  
  
btn.setOnAction(event) -> {  
    System.out.println("Hello World!");  
});
```

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (5/6)

Les *convenience methods* ont deux inconvénients :

- elles ne fonctionnent qu'avec les événements standards, pas avec ceux que l'on crée,
- les abonnements à un événement donné ne sont pas cumulables sur un composant...

4.6. Réagir à un événement : abonnement simplifié

Réagir à un événement : abonnement simplifié (*convenience methods*) (6/6)

...Pourtant il peut arriver que l'on veuille :

- créer plusieurs réactions différentes pour le même événement,
- les dispatcher dans plusieurs écouteurs...
- eux-même potentiellement dispersés dans plusieurs classes.

Pour cela il faut utiliser le mécanisme complet de propagation des événements.

4.7. Propagation des événements

La chaîne de propagation des événements comprend 3 phases :

- *target selection*

Le système détermine la cible de l'événement selon des règles différentes en fonction du type d'événement (pour les événements souris c'est le noeud le plus haut placé sous le pointeur souris).

- *event capturing phase*

L'événement est propagé en descendant le graphe de scène depuis la racine jusqu'à la cible, offrant l'opportunité aux noeuds de chaque étape de réagir à l'événement.

- *event bubbling phase*

L'événement remonte le graphe depuis la cible jusqu'à la racine offrant une deuxième opportunité aux noeuds de chaque étape de réagir à l'événement.

4.8. Réagir à un événement: abonnement complet

En utilisant le mécanisme complet on peut :

- ajouter autant d'écouteurs que l'on veut,
- choisir les endroits/étape où l'on souhaite écouter,
 - sur la cible ou sur n'importe lequel de ses parents dans le graphe de scène
 - sur l'une ou les deux phases de la chaîne de propagation
 - capturing phase*: `addEventFilter(EventType<T>, EventHandler<? super T>)`
 - bubbling phase*: `addEventHandler(EventType<T>, EventHandler<? super T>)`
- interrompre la propagation quand on veut
en appelant la méthode `consume()` sur l'événement écouté dans l'`EventHandler` de notre choix.

4.8. Réagir à un événement: abonnement complet

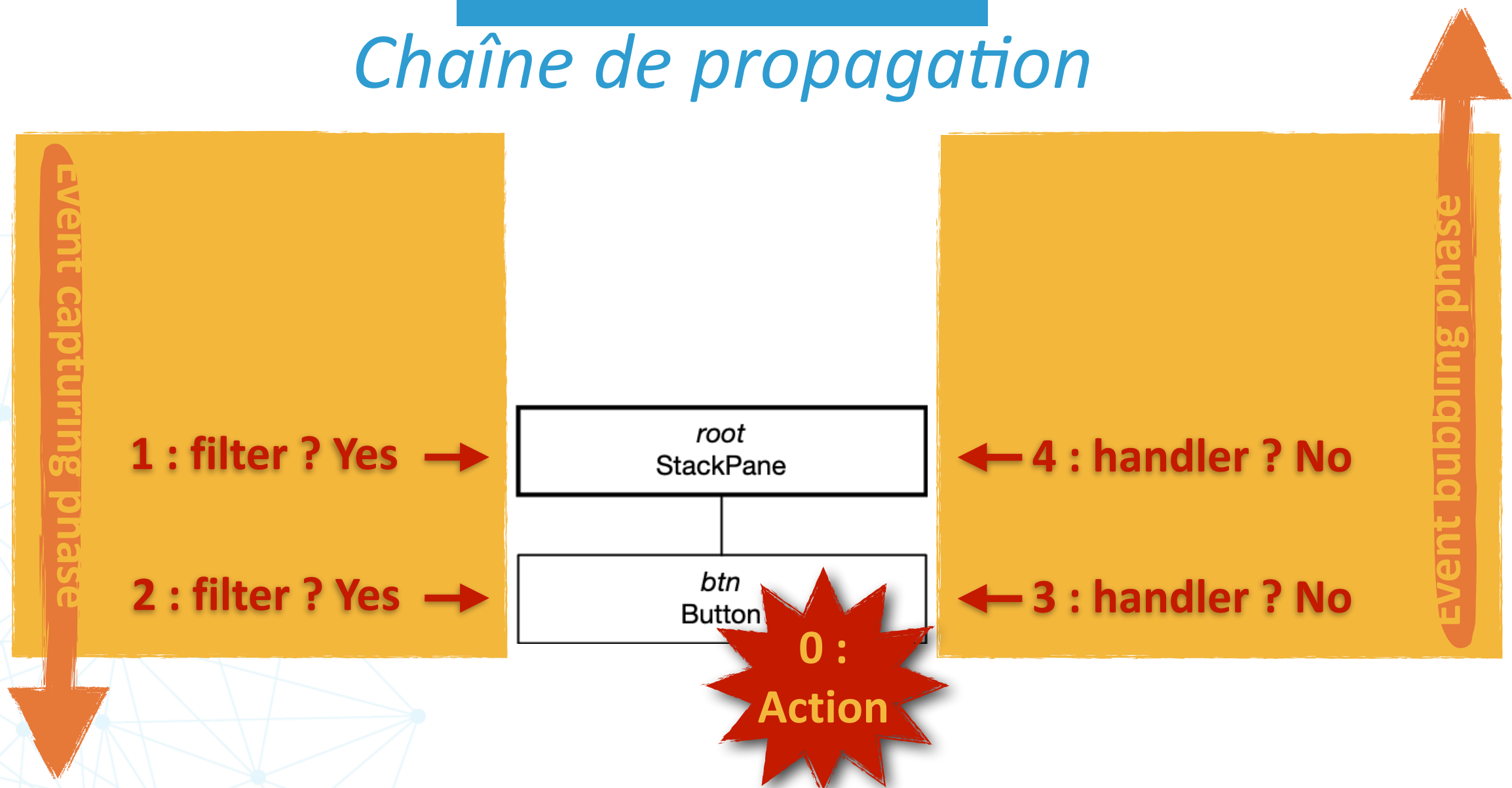
TP 1 - Exercice 6

Chaîne de propagation

```
@Override
public void start(Stage primaryStage) {
    Button btn = new Button("Click me!");
    btn.addEventFilter(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Action due to btn subscription");
        }
    });

    StackPane root = new StackPane();
    root.getChildren().add(btn);
    root.addEventFilter(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            System.out.println("Action due to root subscription");
        }
    });
    ...
}
```

4.8. Réagir à un événement: abonnement complet

TP 1 - Exercice 6*Chaîne de propagation*

4.8. Réagir à un événement: abonnement complet

Réagir à un événement : bilan (1/2)

Le mécanisme complet est **très souple** mais **vite complexe** à gérer...

- en cours de standardisation ?
 - initialement adopté par la plateforme Flash/Flex,
 - adopté dans les specs du w3c pour les standards html5/js,
 - d'autres technologies suivront sans doute la tendance (convergence web/desktop)

- doc complète

<http://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm>

4.8. Réagir à un événement: abonnement complet

Réagir à un événement : bilan (2/2)

...donc sauf besoin très particulier on fait souvent au plus simple :

- utiliser l'abonnement simplifié (*convenience methods*)
- sauf si nécessité de cumuler certains abonnements ou d'utiliser des événements personnalisés, auquel cas écouter
 - si possible toujours lors de la même phase (*capturing* ou *bubbling*),
 - de préférence sur la cible elle-même sauf si besoin de traitement sur plusieurs niveaux,
 - en interrompant si besoin la propagation pour ne pas générer de conflits le long de la chaîne.

Plan

1. Intro

2. Architecture d'une application JavaFX

- 2.1. Classes constitutives d'une application
- 2.2. Construction du graphe de scène
- 2.3. Cycle de vie des composants
- 2.4. Classes de base du graphe de scène

3. Transformations

4. Programmation événementielle

- 4.1. Property
- 4.2. Réagir à un changement de valeur: Binding
- 4.3. Réagir à un changement de valeur : ChangeListener
- 4.4. Event
- 4.5. EventHandler
- 4.6. Réagir à un événement: abonnement simplifié
- 4.7. Propagation des événements
- 4.8. Réagir à un événement: abonnement complet

5. Bilan

5. Bilan

Ce qu'on a vu :

- Architecture d'une application *JavaFX*
 - Classes constitutives d'une application
 - Construction du graphe de scène
 - Cycle de vie des composants
 - Classes de base du graphe de scène
- Transformations
- Programmation événementielle
 - Réagir à un changement de valeur: *Property*, *Binding* et *ChangeListener*
 - *Event*
 - *EventHandler*
 - Réagir à un événement : abonnement simplifié
 - Propagation des événements
 - Réagir à un événement : abonnement complet

5. Bilan

Ce qu'on n'a pas vu :

- un autre langage basé sur xml et destiné à la description du graphe de scène et des interactions de base : *FXML*,
- un *GUI Builder* pour concevoir les interfaces graphiques de type *wimp* par D&D : *Scene Builder* <https://gluonhq.com/products/scene-builder/>,
- la gestion de la *3D*,
- des *effets* et *animations* sur n'importe quel élément du graphe de scène (dessins et widgets),
- habillages CSS...

A vous de découvrir le reste !

5. Bilan

Tous les concepts vus dans ce cours sont réutilisables dans beaucoup d'autres technologies/languages.