

Eventos en JavaFX

1.	Introducción, concepto y clase Event	1
2.	Manejo y filtro de eventos	2
2.1.	Filtros	3
2.2.	Manejadores	5
3.	Tipos de eventos	6
3.1.	Eventos de ratón	6
3.2.	Eventos de teclado	7
3.3.	Eventos de arrastrar y soltar	8
3.4.	Otros eventos: acciones de botones, ventana y muchos más	9
4.	La clase SelectionModel	10
5.	Uso de listeners	11
5.1.	La interfaz InvalidationListener	11
5.2.	La interfaz ChangeListener	13
5.3.	El método removeListener	14

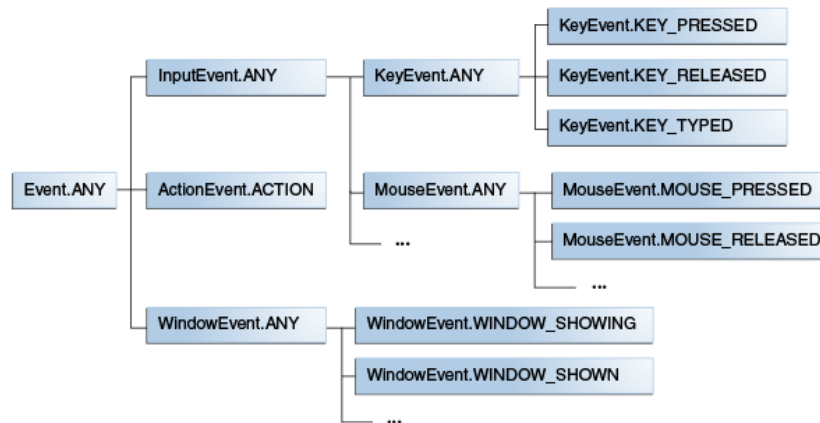
El objetivo de este documento es explicar algunos de los tipos de eventos más importantes en JavaFX y como manejar las acciones asociadas a ellos

1. Introducción, concepto y clase Event

Los **eventos** en la API JavaFX son una de las partes más importantes en el desarrollo de una aplicación. Un evento se produce cuando el usuario interactúa con la aplicación, por ejemplo, al hacer clic sobre un botón, al mover el ratón sobre algún nodo de la escena, al presionar una tecla, o al seleccionar un elemento de una lista, entre muchas otras posibilidades.

Un evento se puede generar en cualquier clase que herede de **Node** o incluso en el propio objeto **Scene**. El código asociado al evento se creará a través de **filtros** o **manejadores** como veremos más adelante.

La clase base que representa los eventos es **javafx.event.Event** y todas las subclases de la misma también representan eventos de diferentes tipos. A lo largo de esta unidad nos vamos a centrar en el estudio de eventos de ratón y teclado, pero hay que tener en cuenta que existen otras muchas acciones que generan eventos. En la siguiente imagen se muestra un ejemplo de la jerarquía de clases y los diferentes eventos asociados a cada subclase.



Como se puede comprobar, cada tipo de eventos tiene una superclase denominada **ANY**. Esta clase sirve para asignar un código común a todos los eventos del mismo tipo (teclado, ratón, ventana, etc.). Por ejemplo, **MouseEvent.ANY** serviría para todos los eventos de ratón.

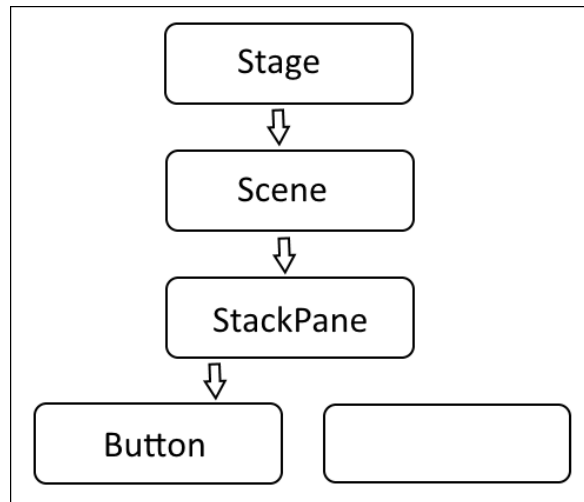
La clase **Event** contiene una serie de propiedades y métodos genéricos para manejar cualquier evento. Las más importantes son:

- **Source:** Método **getSource()**. El nodo que detecta el evento, un botón, la ventana, etc. Más adelante veremos que un evento se puede detectar en un nodo con filtros o manejadores.
- **Target:** Método **getTarget()**. Indica el objeto cuyo estado cambia realmente. Es decir, un evento se puede detectar en un contenedor que sería el objeto **source**. Pero el objeto en el que el usuario ha hecho clic o ha pulsado una tecla puede ser un texto o un botón dentro del mismo, que sería el objeto **target** y es donde se genera.
- **Type:** Método **getEventType()**. Se refiere al tipo de evento producido, presionar un botón del ratón, presionar una tecla, redimensionar la pantalla, etc.
- **consume():** Se trata de un método que marca un evento como realizado. Esto implica que ya no se detectan más acciones asociadas al mismo. Por ejemplo, si se ha pulsado una tecla en un campo de texto situado dentro de un contenedor, ya no se detectarán más eventos de teclado en el resto de los elementos de dicho contenedor.

Más adelante se indicarán otros métodos y propiedades específicos de cada evento.

2. Manejo y filtro de eventos

Cuando se genera un evento, se genera una ruta iniciando en el objeto **Stage** que representa la ventana hasta el objeto en donde se origina el evento. Por ejemplo, supongamos la siguiente aplicación que tiene un botón situado en un **StackPane**.



En la imagen de arriba se construye la ruta hasta el botón y se puede capturar el evento en dos momentos:

- Fase de captura: El nodo raíz distribuye el evento, dicho evento recorre la jerarquía iniciando en la parte superior. Si alguno de los nodos de ha registrado un **filtro**, el mismo es invocado, esto ocurre hasta llegar a la parte inferior del recorrido que termina con el objeto de que originó el evento.
- Fase de propagación: Ocurre el proceso inverso, el evento se distribuye iniciando el recorrido en el origen del evento hasta llegar al nodo raíz, si algún nodo ha registrado un **manejador** de eventos este será invocado, el proceso termina al llegar al nodo raíz.

Por lo que tenemos dos opciones para gestionar los eventos: **filtros** y **manejadores**.

Para especificar las acciones asociadas a cada evento, se requiere implementar la interfaz **EventHandler**. La forma más sencilla es mediante funciones lambda.

```
EventHandler<MouseEvent> manejo = (MouseEvent event) -> {  
    System.out.println("Manejador común para filtros y handlers");  
};
```

La interfaz **EventHandler** tiene un único método denominado **handle** que se puede incluir directamente con la nueva sintaxis a partir de Java 8. Como se puede observar, en el ejemplo de arriba recibe un parámetro de tipo **MouseEvent**. Este parámetro será de la clase que corresponda al tipo de evento que estamos manejando.

El código anterior es común para los filtros y manejadores que veremos a continuación.

2.1. Filtros

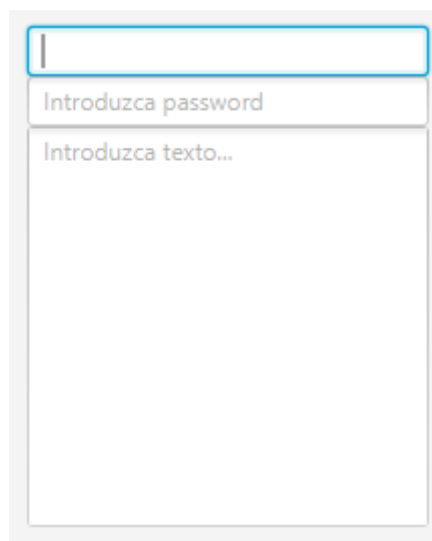
Un filtro de eventos se ejecuta durante la fase de captura. Normalmente se emplea para aplicar un código común de procesamiento de eventos desde un nodo raíz a sus múltiples nodos hijos. Un filtro se registra para un tipo de eventos concreto y ejecuta su código en todos los nodos por los que pasa el evento registrado.

Un nodo puede ejecutar más de un filtro. Cuando más específica sea la clase, más prioridad tendrá. Por ejemplo, según la jerarquía de eventos del apartado 1, el evento de tipo **MouseEvent.MOUSE_PRESSED** tendrá mayor prioridad que **InputEvent.ANY**.

Para comprender mejor la utilidad de los filtros, veamos el siguiente ejemplo:

```
scene.addEventFilter(KeyEvent.KEY_TYPED, e -> {  
  
    String type = e.getEventType().getName();  
    String source = e.getSource().getClass().getSimpleName();  
    String target = e.getTarget().getClass().getSimpleName();  
  
    System.out.println("filter: " + type + ", " + source + ", " + target);  
  
    if (Character.isDigit(e.getCharacter().charAt(0))) {  
        System.out.println("caracter: " + e.getCharacter() + ", no  
            permitido.");  
        e.consume();  
    }  
});
```

Por un lado, el código del filtro se crea sin almacenarlo en una variable como se hacía al final del apartado 2. Además, el filtro se define directamente en el código de **Scene**. Por ejemplo, se puede aplicar en una pantalla como la siguiente.



De esta manera, con el evento **KEY_TYPED** que veremos en detalle más adelante se detecta cualquier carácter válido en los tres elementos que admiten texto de la página. En el código de evento se comprueba si el carácter pulsado es numérico. En caso afirmativo, se consume el evento. Esto implica que el resto del código asociado al evento **KEY_TYPED** ya no se ejecutará y ni siquiera se mostrará la tecla, ya que hemos “cancelado” el resto del código que procesa el evento y se encarga de llevar a cabo las acciones asociadas al teclado.

A tener en cuenta que el método **consume** solo afecta el tipo de evento actual. Si por ejemplo se lleva a cabo el evento un evento de ratón al mismo tiempo sí que se continúa su manejo con normalidad.

En conclusión, los filtros son especialmente útiles para realizar validaciones y evitar que se lleve a cabo un evento, o las acciones asociadas con los nodos hijos. En el resto de casos es indiferente emplear un filtro o manejador.

En el código anterior se ha creado el filtro añadiendo directamente el código. Si creamos el manejador previamente y lo pasamos como parámetro, existe la posibilidad de eliminar los filtros de eventos. Por ejemplo:

```
EventHandler<MouseEvent> manejo = (MouseEvent event) -> {  
    System.out.println("Manejador común para filtros y handlers");  
};  
  
txtField.addEventFilter(MouseEvent.MOUSE_PRESSED, manejo);
```

Como tenemos identificado el código del evento, lo podemos eliminar con el método **removeEventFilter**

```
txtField.removeEventFilter(MouseEvent.MOUSE_PRESSED, manejo);
```

2.2. Manejadores

Un manejador se lleva a cabo en el orden inverso a los filtros, es decir, cuando el evento ha recorrido toda la jerarquía y ha llegado al nodo hijo. Entonces se realiza el recorrido “de vuelta” hasta llegar al nodo raíz. Si algún nodo ha registrado un manejador de eventos este será invocado.

Por tanto, si se llega a ejecutar un manejador es porque no se ha bloqueado previamente el evento con ningún filtro.

A diferencia de los filtros, un manejador debe asociarse directamente al nodo donde ocurre el evento. No se puede incluir en el contenedor u objeto **Scene**. Es decir, en el código siguiente se define un manejador en el campo de texto donde ocurre un evento de ratón. No funcionará sobre los posibles nodos hijos que pudiera tener.

```
txtField.addEventHandler(MouseEvent.DRAG_DETECTED, e -> {  
    String type = e.getEventType().getName();  
    String source = e.getSource().getClass().getSimpleName();  
    String target = e.getTarget().getClass().getSimpleName();  
  
    System.out.println("Manejador específico de: " + type + ", " + source +  
        ", " + target);  
});
```

Teniendo en cuenta esto, un manejador es especialmente útil para ejecutar cualquier código asociado a las acciones que no implique validaciones o bloquear el evento. Por ejemplo, cuando se hace clic en un botón y se abre una ventana, cuando nos posicionamos sobre un elemento y se muestra un mensaje al usuario, arrastrar y soltar, etc.

De la misma manera que los filtros, podemos eliminar las acciones asociadas a un manejador siempre que se hayan almacenado previamente en una variable.

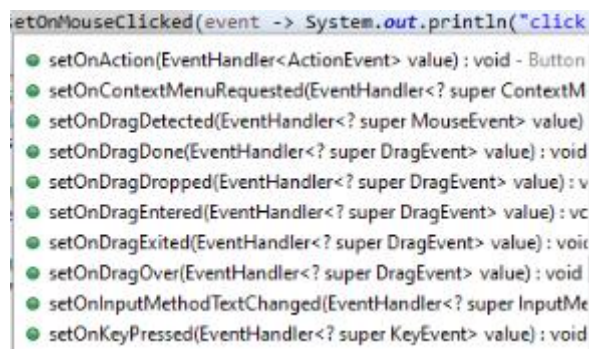
```
EventHandler<MouseEvent> manejo = (MouseEvent event) -> {
    System.out.println("Manejador común para filtros y handlers");
};

button.addEventFilter(MouseEvent.DRAG_DETECTED, manejo);
```

El método **removeEventHandler** permite eliminar el manejador de eventos creado anteriormente.

```
button.removeEventHandler(MouseEvent.DRAG_DETECTED, manejo);
```

Dado que los manejadores se crean directamente en el nodo objeto del evento, se pueden aplicar los métodos **setOnXXX** presentes en cualquier nodo. Cada elemento contiene solamente los métodos del tipo de evento que pueden aceptar.



```
setOnMouseClicked(event -> System.out.println("click
● setOnAction(EventHandler<ActionEvent> value) : void - Button
● setOnContextMenuRequested(EventHandler<? super ContextM
● setOnDragDetected(EventHandler<? super MouseEvent> value)
● setOnDragDone(EventHandler<? super DragEvent> value) : void
● setOnDragDropped(EventHandler<? super DragEvent> value) : v
● setOnDragEntered(EventHandler<? super DragEvent> value) : vc
● setOnDragExited(EventHandler<? super DragEvent> value) : voic
● setOnDragOver(EventHandler<? super DragEvent> value) : void
● setOnInputMethodTextChanged(EventHandler<? super InputMe
● setOnKeyPressed(EventHandler<? super KeyEvent> value) : void
```

Por ejemplo, **setOnMouseClicked** cuando el usuario hace clic en el ratón.

```
btnProbar.setOnMouseClicked(event -> System.out.println("click 1"));
```

3. Tipos de eventos

Una vez que se ha estudiado cómo manejar las acciones de un evento, vamos a ver con más detalle alguno de los tipos más empleados y los métodos específicos de las subclases de **Event** que heredan.

3.1. Eventos de ratón

La clase que genera este tipo de eventos es **MouseEvent**.

A continuación se indican los eventos más empleados y las acciones que los generan:

- **MOUSE_PRESSED**: Cuando se presiona cualquier botón del ratón. El objeto **MouseEvent** puede detectar el botón clickado y la posición exacta.

- MOUSE RELEASED: Cuando el usuario deja de pulsar la tecla del ratón. Esto puede ser útil cuando, por ejemplo, un usuario comienza a hacer click en una posición, pero termina en otra. Así podemos saber en qué momento exacto se ha terminado el evento.
- MOUSE CLICKED: Es una versión de alto nivel que combina las acciones de los dos eventos anteriores. Solamente funciona cuando se hace click y se deja de presionar sobre el mismo nodo.
- MOUSE MOVED: Cualquier movimiento del ratón sobre un elemento genera este evento.
- MOUSE ENTERED: Se lleva a cabo cuando se detecta que el ratón se posiciona sobre un nodo concreto al que se ha asociado código para manejar el evento
- MOUSE EXITED: Cuando el ratón deja de posicionarse sobre un nodo.
- DRAG DETECTED y MOUSE DRAGGED lo veremos en el apartado posterior específico sobre eventos relacionados con arrastrar y soltar.

Los métodos más comunes de la clase **MouseEvent** son los que se indican debajo:

- getSceneX(): Devuelve la coordenada horizontal relativa al objeto **Scene** del nodo donde se ha generado el evento.
- getSceneY(): Devuelve la coordenada vertical relativa al objeto **Scene** del nodo donde se ha generado el evento.
- getScreenX(): Devuelve la posición horizontal absoluta del evento respecto a la pantalla.
- getScreenY(): Devuelve la posición vertical absoluta del evento respecto a la pantalla.

3.2. Eventos de teclado

La clase que genera este tipo de eventos es **KeyEvent**.

A continuación se indican los eventos más empleados y las acciones que los generan:

- KEY_PRESSED: Cuando el usuario presiona una tecla.
- KEY_RELEASED: Cuando el usuario deja de presionar una tecla.
- KEY_TYPED: Cuando el usuario presiona un carácter Unicode válido. Este evento combina **KEY_PRESSED** y **KEY_RELEASED**, pero solo cuando se detectan caracteres válidos.

Se podría considerar que **KEY_PRESSED** y **KEY_RELEASED** son las alternativas de bajo nivel. Se llaman en cuanto el usuario presiona cualquier tecla, mientras que **KEY_TYPED** solamente con caracteres válidos. Por lo que si queremos validar caracteres alfanuméricos la mejor opción es **KEY_TYPED**. Sin embargo, si el objetivo es detectar teclas no asociadas a caracteres como las teclas de función (F1, F2, etc.), ENTER, SHIFT, CTRL, etc., entonces habrá que manejar el resto de alternativas.

Los métodos más empleados de **KeyEvent** son los siguientes:

- getCode: Devuelve el código asociado a cada carácter. Se almacena en una variable de tipo **KeyCode**. Por cada tecla existe una constante en la clase **KeyCode**. Se emplea especialmente para conocer si el usuario ha empleado alguna tecla con caracteres no imprimibles, como CTRL, SHIFT, flechas, etc. Solamente funciona con los eventos **KEY_PRESSED** y **KEY_RELEASED**

- getText: Devuelve el texto que el usuario ha generado a través del teclado. Solamente funciona con los eventos **KEY_PRESSED** y **KEY_RELEASED**.
- getCharacter: Solamente funciona con el evento **KEY_TYPED**. Sirve para conocer la secuencia de caracteres UNICODE que el usuario ha generado, en caso de que sea un carácter válido. En realidad, es muy parecido a **getText** pero cada alternativa funciona en el tipo de eventos indicado.

3.3. Eventos de arrastrar y soltar

Para detectar las acciones relacionadas con arrastrar y soltar un elemento de un lugar a otro de pantalla se emplean dos eventos de tipo **MouseEvent**

- MOUSE_DRAGGED: Se asocia al nodo origen. Se detecta cuando el usuario ha presionado un nodo y mueve el ratón.
- DRAG_DETECTED: Se asocia al nodo origen. Se envía a un nodo que es detectado como el origen de una operación de arrastrar y soltar. A diferencia del anterior, nos permite ejecutar el método **startDragAndDrop** que veremos más adelante.

El resto de eventos son del tipo **DragEvent**. Se indicarán en el orden en el que se ejecutan:

- DRAG_OVER: Se asocia al nodo destino. Se genera cuando el usuario alcanza el nodo destino con el objeto origen que ha arrastrado.
- DRAG_ENTERED: Se asocia al nodo destino. Se lanza cuando la aplicación detecta que el nodo origen ha entrado completamente en el destino.
- DRAG_DROPPED: Se asocia al nodo destino. Ocurre cuando el usuario suelta el botón del ratón una vez que se ha arrastrado el nodo origen al destino. Es donde se suele implementar el código más crítico, por ejemplo, mover o copiar el origen al destino, indicar en el nodo destino que la operación se ha llevado a cabo correctamente, etc.
- DRAG_EXITED: Se asocia al nodo destino. Cuando el usuario ha soltado el ratón y abandonado el nodo destino.
- DRAG_DONE: Se asocia al nodo origen. Cuando ha finalizado la operación en su totalidad.

Por otro lado, los métodos más importantes para hacer efectiva la operación son los siguientes:

- startDragAndDrop: Se ejecuta **sobre el objeto con el nodo origen**. Solo funciona desde el evento de ratón **DRAG_DETECTED**. Sirve para detectar en el nodo origen la operación de arrastrar y soltar. Recibe como parámetro un objeto **TransferMode** que permite indicar el tipo de operaciones soportadas: copiar, mover, enlazar, copiar y mover, todas las operaciones anteriores (ANY). Este método devuelve un objeto de tipo **Dragboard** que permite transferir datos entre los diferentes eventos drag&drop. Por ejemplo, en el código de debajo se inicia una operación en la que solamente es válido el movimiento de un nodo del origen al destino y se almacena el texto del nodo origen en el objeto **Dragboard**.


```
source.setOnDragDetected((event) -> {  
    // Se indica que el modo de transferencia será del tipo movimiento  
    Dragboard db = source.startDragAndDrop(TransferMode.MOVE);  
  
    // Se transfiere el String del texto inicial  
    ClipboardContent content = new ClipboardContent();  
    content.putString(source.getText());  
    db.setContent(content);  
});
```

- acceptTransferModes: Se ejecuta **sobre el objeto con el evento DRAG_OVER**. Se debe indicar un modo de transferencia válido según lo indicado en **startDragAndDrop**. La operación no se detecta en el nodo destino si no se ejecuta este método.

```
target.setOnDragOver((event) -> {  
    // Solo se acepta si se detecta texto en el objeto Dragboard  
    if (event.getDragboard().hasString()) {  
        // Se permite movimiento drag and drop  
        event.acceptTransferModes(TransferMode.MOVE);  
    }  
});
```

- Obtener contenido de Dragboard: En los ejemplos anteriores se obtiene el contenido del objeto **Dragboard** creado para validar que es de tipo texto. Un **Dragboard** puede contener diferentes tipos de datos: texto, ficheros, imágenes o código HTML. Se puede acceder en todo momento a través del objeto de la clase **DragEvent**. Por ejemplo, en **DRAG_DROPPED** es cuando nos encargaremos de efectuar cambios sobre el nodo destino en el cual se ha ejecutado la operación. En este caso se cambia el texto por el del nodo origen que hemos transportado en el propio objeto **Dragboard** que a su vez contiene un método get por cada tipo de dato, en este caso **getString**.

```
target.setOnDragDropped((event) -> {  
    // Si se ha almacenado texto se copia al destino  
    Dragboard db = event.getDragboard();  
    if (db.hasString()) {  
        target.setText(db.getString());  
    }  
});
```

3.4. Otros eventos: acciones de botones, ventana y muchos más

Un evento bastante común que no se ha mencionado hasta ahora es **ACTION** del tipo **ActionEvent**. Se emplea principalmente cuando el usuario pulsa un botón.

Un ejemplo muy típico es la operación de mostrar un diálogo desde el código Java. Al hacer click en un botón a través del método **setOnAction** (es el manejador asociado al evento ACTION) se visualiza el diálogo denominado infoAlert.

```
infoButton.setOnAction(e -> {  
    infoAlert.showAndWait();  
});
```

JavaFX permite muchos más tipos de eventos: la clase **WindowEvent** relacionada con operaciones al mostrar u ocultar ventanas, la clase **TouchEvent** para el uso de pantallas táctiles, entre otras muchas posibilidades.

4. La clase SelectionModel

En esta unidad estamos estudiando cómo modificar dinámicamente una aplicación con JavaFX, ya sea con eventos o detectando los cambios de una propiedad como veremos a continuación.

Hay controles como listados, tablas o cuadros desplegables en los que necesitamos conocer cómo acceder en todo momento a los elementos seleccionados. Para ello, JavaFX nos proporciona la clase abstracta **SelectionModel** de la que heredan **MultipleSelectionModel** y **SingleSelectionModel** en función de si podemos seleccionar o no varios ítems a la vez.

A continuación se indican los métodos más empleados y comunes a ambas clases:

- **getSelectedItem:** Nos devuelve el ítem seleccionado y con el tipo de datos que corresponda. Por ejemplo, para un **ComboBox** con String nos devolvería una cadena de caracteres.

```
String seleccionado = comboBox.getSelectionModel().getSelectedItem();
```

- **getSelectedIndex:** En lugar de retornar el ítem seleccionado, nos dice la posición del array de ítems que se ha seleccionado actualmente (índice de 0 a n).
- **selectedIndexProperty** o **selectedItemProperty:** Nos permite acceder a la propiedad real observable de cara a trabajar con eventos o listeners y detectar cambios dinámicamente como veremos más adelante.
- **clearSelection:** Permite deseleccionar todos los ítems, o existe una versión que recibe un índice como parámetro que se puede desmarcar.
- **select:** Permite seleccionar dinámicamente un índice o directamente un ítem que pasamos como parámetro.

En la especificación podemos ver el resto de métodos y las subclases **MultipleSelectionModel** y **SingleSelectionModel** que añaden otros métodos y propiedades específicos: <https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/SelectionModel.html>

Por ejemplo, para un **MultipleSelectionModel** conviene tener en cuenta los siguientes métodos específicos de la subclase.

- **getSelectedItems:** Devuelve una lista observable con todos los ítems seleccionados. Por ejemplo, en el código de debajo creamos un listener para detectar cualquier cambio en los ítems (veremos más adelante cómo funciona).

```
11.getSelectionModel().getSelectedItems().addListener((Observable o) -> {  
    System.out.println("Invalidation -> " + o);  
});
```

- **getSelectedIndices:** Similar al método anterior, pero en este caso devuelve solo los índices seleccionados.
- **setSelectionMode:** Un `MultipleSelectionModel` se emplea para controles que permiten seleccionar varios elementos a la vez. Por defecto no es posible, salvo que lo configuremos con esta propiedad como se indica en el código de debajo.

```
listNumeros.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

- **selectAll, selectFirst, selectLast:** Permite seleccionar todos los ítems, el primero o el último respectivamente.

Por último, existe una clase específica para seleccionar filas en tablas denominada **TableSelectionModel** que funciona de manera similar y añade algunos métodos para trabajar con filas, columnas y celdas:

<https://openjfx.io/javadoc/17/javafx.controls/javafx/scene/control/TableSelectionModel.html>

5. Uso de listeners

Las propiedades JavaFX cuentan con el método **addListener** que podemos usar para registrar un método que será notificado cuando el valor de una propiedad cambie. La sobrecarga de este método nos permite utilizar dos interfaces, **ChangeListener** y **InvalidationListener**, para indicar cual será el método que recibirá las notificaciones. Cuando ya no deseemos recibir notificaciones usaremos **removeListener**.

Los controles de JavaFX tienen diferentes propiedades cuyo valor podemos “observar”. Este es el motivo por el cual en unidades anteriores se creaban tablas o listas con valores que heredan de **Observable** como **StringProperty**, en lugar de emplear directamente un **String**. Los tipos básicos de Java no permiten añadir listeners.

Por ejemplo, para un **TextField** es posible acceder al valor de tipo **StringProperty** y detectar cuándo ha cambiado.

```
textField.textProperty().addListener(textListener);
```

O incluso en listado o `ComboBox`, el objeto que representa la selección del usuario (`getSelectedItems` en el `selectionModel`) es un **ObservableList** que nos permite detectar cualquier modificación con un listener.

```
listNumeros.getSelectionModel().getSelectedItems().addListener(  
    ...  
);
```

5.1. La interfaz `InvalidationListener`

Por definición, este método se emplea para recibir notificaciones cuando el valor de una propiedad se cambia de estado inválido a válido. Pero en realidad su funcionamiento es muy

parecido en cierto modo a **ChangeListener**, ya que JavaFX detecta un cambio de estado casi siempre que se modifica un valor.

Hay dos formas de definir un **InvalidationListener** como se explica en el siguiente ejemplo en el que se prueba con un **IntegerProperty**.

```
IntegerProperty age = new SimpleIntegerProperty();

// Con clase anónima que implemente InvalidationListener
age.addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable observable) {
        System.out.println("Property is invalid.");
    }
});

// Con función lambda
age.addListener((Observable observable) -> {
    System.out.println("Property is invalid.");
});
```

En la práctica el **InvalidationListener** se llama prácticamente cada vez que se modifica la variable de tipo **IntegerProperty** del ejemplo anterior, aunque a diferencia de **ChangeListener** solo podemos recoger el objeto con el valor actual con el parámetro de tipo **Observable**.

No obstante, una importante característica de **InvalidationListener** es que soporta “Lazy evaluation”. Veamos un ejemplo para entenderlo mejor.

```
IntegerProperty age = new SimpleIntegerProperty();

// Definimos listener
age.addListener((Observable observable) -> {
    System.out.println("Cambio de propiedad");
});

// Cambiamos el valor varias veces seguidas pero solo notifica una
age.set(100);
age.set(101);
age.set(102);
```

En la consola se muestra el texto del listener solo una vez porque Java detecta el cambio de propiedad varias veces seguidas y se ahorra varias llamadas.

Sin embargo, si accedemos con **get** cada vez que cambiamos la propiedad entonces, entonces sí que se llama al **InvalidationListener** después de cada **get** como ocurre a continuación

```
// Sin embargo, si preguntamos por la propiedad sí que se notifica
age.set(100);

age.get();
age.set(101);

age.get();
age.set(102);
```

También pasaría si accedemos al objeto **Observable** desde la definición del listener.

```
// Definimos listener
age.addListener((Observable observable) -> {
    System.out.println(observable);
});
```

En este caso, aunque haya tres métodos set seguidos se muestra por consola la línea definida anteriormente porque la evaluación “lazy” deja de aplicarse si se detecta que el usuario accede al objeto con el valor modificado.

Por otro lado, ahora definimos un **InvalidationListener** para un control de Java como en el TextField anterior.

```
// Cada vez que se cambia el TextField se notifica por consola
textField.textProperty().addListener((Observable observable) -> {
    System.out.println("Llamada a observable");
});
```

En este caso, al tratarse de un control gráfico se llama al **InvalidationListener** cada vez que se modifica un valor, aunque sean varias veces seguidas, ya que cada modificación implica internamente llamar al get y set y no se puede aplicar la evaluación de tipo “lazy”.

En conclusión, **InvalidationListener** se suele emplear si no necesitamos conocer el valor antiguo (que veremos que se puede acceder con **ChangeListener**), o cuando queramos simplemente detectar un cambio en un valor aunque no sepamos exactamente el número de veces qué ha ocurrido (Java intenta ahorrar llamadas al listener).

5.2. La interfaz **ChangeListener**

El listener de tipo **ChangeListener** es mucho más sencillo de entender, ya que se llama siempre que se detecta cualquier cambio en una propiedad y además podemos acceder al valor antiguo y al nuevo. La desventaja es que puede ser menos eficiente y el código consume más recursos.

Al igual que **InvalidationListener**, se puede definir con la notación clásica o con las funciones lambda a partir de Java 8.

```
// Notación clásica
comboBox.getSelectionModel().selectedItemProperty().addListener(new
ChangeListener<String>() {
    public void changed(ObservableValue obs, String old, String new) {
        resultado.setText("Antiguo -> " + old + "\n" + "Nuevo -> " + new);
    }
});

// Con función lambda
comboBox.getSelectionModel().selectedItemProperty().addListener(
    (observable, oldValue, newValue) -> {
        resultado.setText("Antiguo->" + oldValue + "Nuevo->" + newValue);
    });
```

Supongamos el ejemplo del apartado anterior, aunque en este caso con **ChangeListener**.

```
IntegerProperty age = new SimpleIntegerProperty();

age.addListener((observable, oldValue, newValue) -> {
    System.out.print("Propiedad cambiada: ");
    System.out.println("old = " + oldValue + ", new = " + newValue);
});

// Prueba de notificación de ChangeListener
age.set(101);
age.set(102);

age.setValue(103);
age.setValue(104);
```

En este caso se llamará al **ChangeListener** tantas veces como se cambie el valor de la propiedad “age”.

Además, como se puede observar en la definición, lo interesante de este tipo de listener es que podemos acceder al valor anterior y nuevo con los parámetros `oldValue` y `newValue`. Es más ineficiente porque JavaFX necesita generar el valor anterior y el nuevo y además no se ahorran llamadas al listener, pero nos puede resultar de utilidad en muchas ocasiones.

Por último, el primer parámetro de tipo **ObservableValue** en la definición es similar a **InvalidationListener** y nos permite obtener el objeto sobre el cual se ha definido el **ChangeListener**.

5.3. El método `removeListener`

Es posible eliminar un listener que ya no quedamos utilizar. Pero Java nos permite solamente borrar los que tengan una referencia en el código. Aquellos creados con clases anónimas como anteriormente no se podrían eliminar

A continuación, se muestra un ejemplo para eliminar un listener. El código es el mismo para **InvalidationListener** y **ChangeListener**.

```
// Listener para el campo de texto
InvalidationListener textListener = new InvalidationListener() {
    @Override
    public void invalidated(Observable observable) {
        StringProperty sp = (StringProperty) observable;
        System.out.println(observable.toString() + ", " + sp.get());
    }
};

textField.textProperty().removeListener(textListener);
```