# JavaFX Documentation Project

# Table of Contents

# Chapter 1. Introduction

The JavaFX Documentation Project aims to pull together useful information for JavaFX developers from all over the web. The project is open source and encourages community participation to ensure that the documentation is as highly polished and useful as possible.

## 1.1. Contributors

This project would not be possible without the contributors who work hard on the content contained within this documentation. Whenever possible contributors are given attribution within the document when they write a section, but it is also important to gather all names here, at the top of the document, to give the recognition that these contributors deserve.

## 1.2. Contributing

Contributing to this project is easy - fork the GitHub repo, edit the relevant files, and create a pull request! Once merged, your content will form a part of the documentation and you'll have the unending appreciation of the entire community!

The JavaFX Documentation Project uses AsciiDoc as the syntax of choice for writing the documentation. The AsciiDoc Syntax Quick Reference guide is a great resource for those learning how to write AsciiDoc.

Authors are welcome to include a byline beneath the sections that they have authored. To ensure consistency, the recommended format for the byline is the following:

*Contributed by <name> - <website>*

# Chapter 2. Scenegraph

Placeholder whilst things get built...

# Chapter 3. UI Controls

## 3.1. ListView

### 3.1.1. ListView Filtering in JavaFX

(Contributed by Carl Walker, October 1, 2016)

This article demonstrates how to filter a ListView in a JavaFX Application. Two lists are managed by the Application. One list contains all of the items in the data model. The second list contains all of the items currently being viewed. A scrap of comparison logic stored as a filter mediates between the two.

Binding is used heavily to keep the data structures in sync with what the user has selected.

This screenshot shows the Application which contains a top row of ToggleButtons which set the filter and a ListView containing the objects.
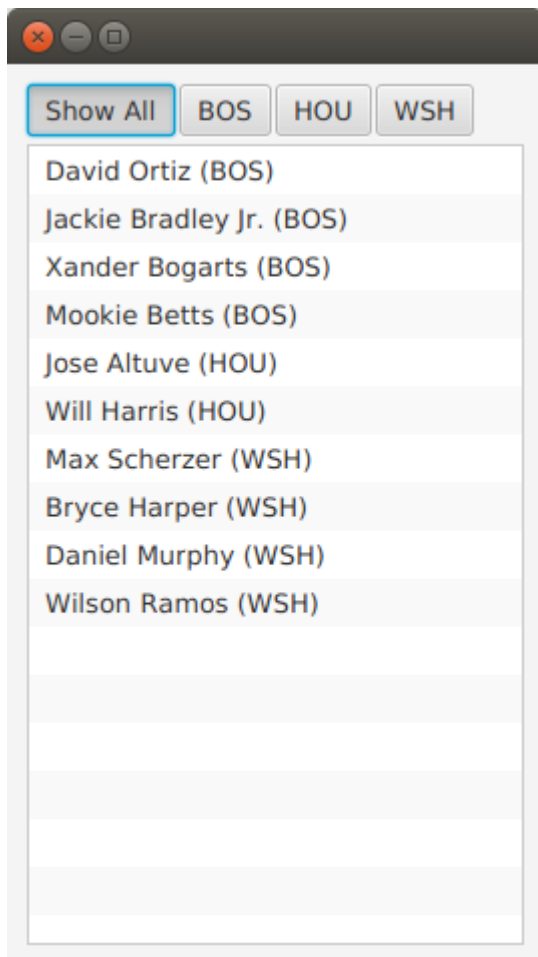


*Figure 1. Screenshot of ListView Filtering App*

The complete code — a single .java file — is listed at the end of the article.

**Data Structures**

The program begins with a domain model Player and an array of Player objects.

```
static class Player {

    private final String team;
    private final String playerName;
    public Player(String team, String playerName) {
        this.team = team;
        this.playerName = playerName;
    }
    public String getTeam() {
        return team;
    }
    public String getPlayerName() {
        return playerName;
    }
    @Override
    public String toString() { return playerName + " (" + team + ")"; }
}
```

The Player class contains a pair of fields, team and playerName. A toString() is provided so that when the object is added to the ListView (presented later), a custom ListCell class is not needed.

The test data for this example is a list of American baseball players.

```
Player[] players = {new Player("BOS", "David Ortiz"),
                    new Player("BOS", "Jackie Bradley Jr."),
                    new Player("BOS", "Xander Bogarts"),
                    new Player("BOS", "Mookie Betts"),
                    new Player("HOU", "Jose Altuve"),
                    new Player("HOU", "Will Harris"),
                    new Player("WSH", "Max Scherzer"),
                    new Player("WSH", "Bryce Harper"),
                    new Player("WSH", "Daniel Murphy"),
                    new Player("WSH", "Wilson Ramos") };
```

**Model**

As mentioned at the start of the article, the ListView filtering is centered around the management of two lists. All the objects are stored in a wrapped ObservableList playersProperty and the objects that are currently viewable are stored in a wrapped ObservableList, viewablePlayersProperty.

```
ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
        new SimpleObjectProperty<>(FXCollections.observableArrayList());

ReadOnlyObjectProperty<ObservableList<Player>> viewablePlayersProperty =
        new SimpleObjectProperty<>(FXCollections.observableArrayList());
```

Coordinating the model, is a filter objects which is a wrapped Predicate. A listener is attached to

the filter such that when the Predicate changes, a new viewable set of items is available for the ListView. This is done with the filtered() method of the playersProperty ObservableList.

```
ObjectProperty<Predicate<Player>> filterProperty = new SimpleObjectProperty<>((p) ->
true);

filterProperty.addListener(
        (obs,ov,nv) -> {
            viewablePlayersProperty.get().setAll(
                    playersProperty.get().filtered( nv )
                    );
        });
```

The UI root is a VBox which contains an HBox of ToggleButtons and a ListView.

```
VBox vbox = new VBox();
vbox.setPadding( new Insets(10));
vbox.setSpacing(4);

HBox hbox = new HBox();
hbox.setSpacing( 2 );

ToggleGroup filterTG = new ToggleGroup();
```

**Filtering Action**

A handler is attached the ToggleButtons which will modify the filter property. Each ToggleButton is supplied a Predicate in the userData field. toggleHandler uses this supplied Predicate when setting the filter property. This code sets the special case "Show All" ToggleButton.

```
@SuppressWarnings("unchecked")
EventHandler<ActionEvent> toggleHandler = (event) -> {
        ToggleButton tb = (ToggleButton)event.getSource();
        Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
        filterProperty.set( filter );
    };

ToggleButton tbShowAll = new ToggleButton("Show All");
tbShowAll.setSelected(true);
tbShowAll.setToggleGroup( filterTG );
tbShowAll.setOnAction(toggleHandler);
tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);
```

The ToggleButtons that filter a specific team are created at runtime based on the Players array. This Stream does the following.

1. Distill the list of Players down to a distinct list of team Strings

2. Create a ToggleButton for each team String

3. Set a Predicate for each ToggleButton to be used as a filter

4. Collect the ToggleButtons for addition into the HBox container

```
List<ToggleButton> tbs = Arrays.asList( players)
        .stream()
        .map( (p) -> p.getTeam() )
        .distinct()
        .map( (team) -> {
            ToggleButton tb = new ToggleButton( team );
            tb.setToggleGroup( filterTG );
            tb.setOnAction( toggleHandler );
            tb.setUserData( (Predicate<Player>) (Player p) -> team.equals(p.getTeam())
);
            return tb;
        })
        .collect(Collectors.toList());

hbox.getChildren().add( tbShowAll );
hbox.getChildren().addAll( tbs );
```

**ListView**

The next step creates the ListView and binds the ListView to the viewablePlayersProperty. This enables the ListView to receive updates based on the changing filter.

```
ListView<Player> lv = new ListView<>();
lv.itemsProperty().bind( viewablePlayersProperty );
```

The remainder of the program creates a Scene and shows the Stage. onShown loads the data set into the playersProperty and the viewablePlayersProperty lists. Although both lists are in sync in this partcular version of the program, if the stock filter is every different than "no filter", this code would not need to be modified.

```
    vbox.getChildren().addAll( hbox, lv );

    Scene scene = new Scene(vbox);

    primaryStage.setScene( scene );
    primaryStage.setOnShown((evt) -> {

        //
        // Initialize the UI with test data by adding everything to playersProperty
        // and only those Players that meet the filter criteria to viewablePlayersProperty
        //
        // Note that this implementation has the initial filter set to "Show All" so
        // players==viewablePlayers
        //
        Arrays.asList( players ).forEach( (p) -> {

            playersProperty.get().add( p );

            if( filterProperty.get().test(p) ) {
                viewablePlayersProperty.get().add( p );
            }
        });
    });

    primaryStage.show();
```

This article used binding to tie a list of viewable Player objects to a ListView. The viewable Players were updated when a ToggleButton is selected. The selection applied a filter to a full set of Players which was maintained separately. Binding was used to keep the UI in sync and to allow for a separation of concerns in the design.

**Further Reading**

To see how such a design would implement basic add and remove functionality, visit the following page https://courses.bekwam.net/public_tutorials/bkcourse_filterlistapp.php.

**Complete Code**

The code can be tested in a single .java file.

```
public class FilterListApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        //
        // Test data
        //
        Player[] players = {new Player("BOS", "David Ortiz"),
```

```java
                         new Player("BOS", "Jackie Bradley Jr."),
                         new Player("BOS", "Xander Bogarts"),
                         new Player("BOS", "Mookie Betts"),
                         new Player("HOU", "Jose Altuve"),
                         new Player("HOU", "Will Harris"),
                         new Player("WSH", "Max Scherzer"),
                         new Player("WSH", "Bryce Harper"),
                         new Player("WSH", "Daniel Murphy"),
                         new Player("WSH", "Wilson Ramos") };

    //
    // Set up the model which is two lists of Players and a filter criteria
    //
    ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
            new SimpleObjectProperty<>(FXCollections.observableArrayList());

    ReadOnlyObjectProperty<ObservableList<Player>> viewablePlayersProperty =
            new SimpleObjectProperty<>(FXCollections.observableArrayList());

    ObjectProperty<Predicate<Player>> filterProperty = new
SimpleObjectProperty<>((p) -> true);

    filterProperty.addListener(
            (obs,ov,nv) -> {
                viewablePlayersProperty.get().setAll(
                        playersProperty.get().filtered( nv )
                        );
            });

    //
    // Build the UI
    //
    VBox vbox = new VBox();
    vbox.setPadding( new Insets(10));
    vbox.setSpacing(4);

    HBox hbox = new HBox();
    hbox.setSpacing( 2 );

    ToggleGroup filterTG = new ToggleGroup();

    //
    // The toggleHandler action wills set the filter based on the TB selected
    //
    @SuppressWarnings("unchecked")
    EventHandler<ActionEvent> toggleHandler = (event) -> {
            ToggleButton tb = (ToggleButton)event.getSource();
            Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
            filterProperty.set( filter );
        };
```

```java
        ToggleButton tbShowAll = new ToggleButton("Show All");
        tbShowAll.setSelected(true);
        tbShowAll.setToggleGroup( filterTG );
        tbShowAll.setOnAction(toggleHandler);
        tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);

        //
        // Create a distinct list of teams from the Player objects, then create
        // ToggleButtons
        //
        List<ToggleButton> tbs = Arrays.asList( players)
                .stream()
                .map( (p) -> p.getTeam() )
                .distinct()
                .map( (team) -> {
                    ToggleButton tb = new ToggleButton( team );
                    tb.setToggleGroup( filterTG );
                    tb.setOnAction( toggleHandler );
                    tb.setUserData( (Predicate<Player>) (Player p) ->
team.equals(p.getTeam()) );
                    return tb;
                })
                .collect(Collectors.toList());

        hbox.getChildren().add( tbShowAll );
        hbox.getChildren().addAll( tbs );

        //
        // Create a ListView bound to the viewablePlayers property
        //
        ListView<Player> lv = new ListView<>();
        lv.itemsProperty().bind( viewablePlayersProperty );

        vbox.getChildren().addAll( hbox, lv );

        Scene scene = new Scene(vbox);

        primaryStage.setScene( scene );
        primaryStage.setOnShown((evt) -> {

            //
            // Initialize the UI with test data by adding everything to
playersProperty
            // and only those Players that meet the filter criteria to
viewablePlayersProperty
            //
            // Note that this implementation has the initial filter set to "Show All"
so
            // players==viewablePlayers
            //
            Arrays.asList( players ).forEach( (p) -> {
```

```java
                playersProperty.get().add( p );

                if( filterProperty.get().test(p) ) {
                    viewablePlayersProperty.get().add( p );
                }
            });
        });

        primaryStage.show();

    }

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

    static class Player {

        private final String team;
        private final String playerName;
        public Player(String team, String playerName) {
            this.team = team;
            this.playerName = playerName;
        }
        public String getTeam() {
            return team;
        }
        public String getPlayerName() {
            return playerName;
        }
        @Override
        public String toString() { return playerName + " (" + team + ")"; }
    }
}
```

# Chapter 4. CSS

Placeholder whilst things get built...

# Chapter 5. Performance

Placeholder whilst things get built...

# Chapter 6. Application Structure

Placeholder whilst things get built...

# Chapter 7. Best Practices

Placeholder whilst things get built…

1. Styleable Properties

## 7.1. 1. Styleable Properties

Author: Gerrit Grunwald

```java
/* Member variables for StyleablePropertyFactory
 * and StyleableProperty
 */
private static final StyleablePropertyFactory<MY_CTRL> FACTORY =
    new StyleablePropertyFactory<>(Control.getClassCssMetaData());

private static final CssMetaData<MY_CTRL, Color> COLOR =
    FACTORY.createColorCssMetaData("-color", s -> s.color, Color.RED, false);
private final StyleableProperty<Color> color = new
SimpleStyleableObjectProperty<>(COLOR, this, "color");


// Getter, Setter and Property method
public Color getColor() {
    return this.color.getValue();
}

public void setColor(final Color color) {
    this.color.setValue(COLOR);
}

public ObjectProperty<Color> colorProperty() {
    return (ObjectProperty<Color>) this.color;
}


// Return CSS Metadata
public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
    return FACTORY.getCssMetaData();
}

@Override public List<CssMetaData<? extends Styleable, ?>> getControlCssMetaData() {
    return getClassCssMetaData();
}
```

# Chapter 8. Contributing

Placeholder whilst things get built...

# Chapter 9. License