

# JavaFX Tutorial

- [JavaFX Features](#)
  - [Third Party JavaFX Resources](#)
- [JavaFX Use Cases](#)
- [JavaFX on Mobile Devices](#)
- [JavaFX in Java 8](#)
  - [JavaFX Replaces Swing](#)
  - [Java Installers](#)
  - [Java WebStart](#)
- [JavaFX in Java 9](#)
- [JavaFX From Java 11](#)
- [JavaFX JavaDoc](#)

Jakob Jenkov

Last update: 2019-08-03

*JavaFX* is a GUI toolkit for Java (GUI is short for Graphical User Interface). JavaFX makes it easier to create desktop applications and games in Java. This JavaFX tutorial is a multi-page tutorial explaining the core features of JavaFX. See the menu in the left side of this page to see all the topics covered in this JavaFX tutorial.

Some applications are just easier to create as standalone desktop applications than as web applications. For instance, applications that need to access the local disk of the computer it runs on, or which needs to communicate with many different remote systems, and sometimes using other protocols than HTTP (e.g. [IAP](#) or streaming protocols etc.). JavaFX is a good option in these cases. We at [Nanosai](#) are actually developing a desktop app using JavaFX for these exact reasons. See [JavaFX use cases](#) for more examples.

JavaFX has replaced Swing as the recommended GUI toolkit for Java. Furthermore, JavaFX is more consistent in its design than Swing, and has more features. It is more modern too, enabling you to design GUI using layout files (XML) and style them with CSS, just like we are used to with web

applications. JavaFX also integrates 2D + 3D graphics, charts, audio, video, and embedded web applications into one coherent GUI toolkit.

The Gluon company has ported JavaFX so it can run on both Android and iOS. See [JavaFX on Mobile Devices](#) for more information.

## JavaFX Features

JavaFX comes with a large set of built-in GUI components, like buttons, text fields, tables, trees, menus, charts and much more. That saves you a lot of time when building a desktop applications.

JavaFX components can be styled using CSS, and you can use FXML to compose a GUI instead of doing it in Java code. This makes it easier to quickly put a GUI together, or change the looks or composition without having to mess around in the Java code.

JavaFX contains a set of ready-to-use chart components, so you don't have to code that from scratch every time you need a basic chart.

JavaFX also comes with support for 2D and 3D graphics as well as audio and video support. This is useful if you are developing a game, or similar media applications.

JavaFX even contains a WebView based on the popular WebKit browser, so you can embed web pages or web applications inside JavaFX.

Here is a more complete list of components and features in JavaFX:

- **Core**
  - [Stage](#)
  - [Scene](#)
  - Node
  - [FXML](#)
- **Layout**
  - [HBox](#)
  - [VBox](#)
  - [FlowPane](#)
  - [TilePane](#)
  - [GridPane](#)
  - [Group](#)
- **Basic Controls**

- [Label](#)
- [Button](#)
- [MenuButton](#)
- [SplitMenuButton](#)
- [ToggleButton](#)
- [RadioButton](#)
- [CheckBox](#)
- [ChoiceBox](#)
- [ComboBox](#)
- [ListView](#)
- [TextField](#)
- [PasswordField](#)
- [TextArea](#)
- [ImageView](#)
- [DatePicker](#)
- [ColorPicker](#)
- [Slider](#)
- [Tooltip](#)
- [Hyperlink](#)
- [ProgressBar](#)
- [MenuBar](#)
- [ContextMenu](#)
- [Separator](#)
- [TableView](#)
- [TreeView](#)
- [TreeTableView](#)
- [HTMLEditor](#)
- [Pagination](#)
- [FileChooser](#)
- [DirectoryChooser](#)
- **Container Controls**
  - [Accordion](#)
  - [TitledPane](#)
  - [TabPane](#)
  - [SplitPane](#)
  - [ScrollPane](#)
- **Web**
  - [WebView](#)
  - [WebEngine](#)
- **Charts**
  - [PieChart](#)

- [BarChart](#)
- [StackedBarChart](#)
- [ScatterChart](#)
- [LineChart](#)
- [AreaChart](#)
- [StackedAreaChart](#)
- BubbleChart
- **Other Concepts**
  - 2D Shapes
  - 3D Shapes
  - Effects
  - Transformations
  - Animation
  - Drag and Drop
  - Audio
  - Video
  - Print API
  - High DPI resolution screen support
  - Concurrency in JavaFX

## Third Party JavaFX Resources

There are a few, cool third party resources available out there. I have listed some of them here:

- [JavaFX Drift](#) - OpenGL etc. support for JavaFX. Seamless integration into the scene graph.
- [ControlsFX](#) - Extra UI controls for JavaFX.

## JavaFX Use Cases

I have been asked several times if not desktop applications are dead - if there are really any use cases left for something like JavaFX. It is true, that many applications fit well as web applications, because you access them rarely, and the resources you access are stored on a server anyways. But, there are also several types of applications that are better implemented as desktop applications.

As mentioned in the introduction we at [Nanosai](#) are actually developing a desktop application using JavaFX. We do so because that app needs access to the local disk, needs to be able to communicate via other network protocols

than HTTP, and needs several other features a standard web browser simply does not provide.

Here is a list of some of general use cases I see for JavaFX:

- Developer tools
  - IDE
  - Editors
  - File compression / encryption tools
  - Tools scanning the local disk
- Local system maintenance tools
  - Backup tools
  - Virus scans
- Utility apps
  - Skype / Messenger / Chat
  - Screen shot tools
  - Photo and video editing
  - Video players
  - Audio editing
  - Audio players
- Games
- Data Science Tools

Here are some of the desktop apps I use regularly:

- IntelliJ IDEA
- Notepad++
- SourceTree
- SnagIt
- Putty
- WinSCP
- Skype
- PhotoShop
- Premiere Pro
- VideoLAN (VLC)
- EDraw

## **JavaFX on Mobile Devices**

There is a community effort to make JavaFX applications run on mobile devices. The project is called [JavaFXPorts](#) and is maintained and supported by Gluon.

## **JavaFX in Java 8**

From Java 8 JavaFX is bundled with the Java platform, so JavaFX is available everywhere Java is.

### **JavaFX Replaces Swing**

JavaFX is intended to replace Swing as the default GUI toolkit in Java. Swing will still be shipped with Java for some time, but you should consider porting your old Java Swing applications to JavaFX some time in the future.

### **Java Installers**

From Java 8 you can also create standalone install packages for Windows, Mac and Linux with Java, which includes the JRE needed to run them. This means that you can distribute JavaFX applications to these platforms even if the platform does not have Java installed already.

### **Java WebStart**

JavaFX applications can also be installed and executed using Java WebStart. To start an application using Java WebStart you need to create a JNLP file (Java Network Launch Protocol) file and put it on a web server, and create a link to it from a web page somewhere. When the user clicks the link to the JNLP file the application is downloaded and started.

Once the JavaFX application is installed it can be started again using the same JNLP link. The application is not downloaded the second time. It is executed from the previous installation.

Java WebStart also makes it possible to upgrade the installed JavaFX applications to newer versions. This is a great way to handle upgrades for internal tools in an enterprise. It is almost as seamless as upgrading web applications.

## **JavaFX in Java 9**

JavaFX did not get a lot of new features in Java 9, but it did get 750+ bug fixes, so if you are planning to build a new JavaFX app from scratch, you might want to consider starting with Java 9 !

## JavaFX From Java 11

From Java 11, JavaFX has been removed from the Java SDK again. JavaFX has been detached into its own open source project. This means that to download JavaFX from Java 11 / JavaFX 11, you have to go to:

<http://openjfx.io>

## JavaFX JavaDoc

Even though JavaFX is part of Java 8, the JavaFX JavaDoc is not included in the standard Java 8 JavaDoc. You can find the JavaFX JavaDoc here:

<https://docs.oracle.com/javase/8/javafx/api/toc.htm>

Next: [JavaFX Overview](#)

# JavaFX Overview

- [Stage](#)
- [Scene](#)
  - [Scene Graph](#)
  - [Nodes](#)
- [Controls](#)
- [Layouts](#)
  - [Nested Layouts](#)
- [Charts](#)
- [2D Graphics](#)
- [3D Graphics](#)
- [Audio](#)
- [Video](#)
- [WebView](#)

Jakob Jenkov

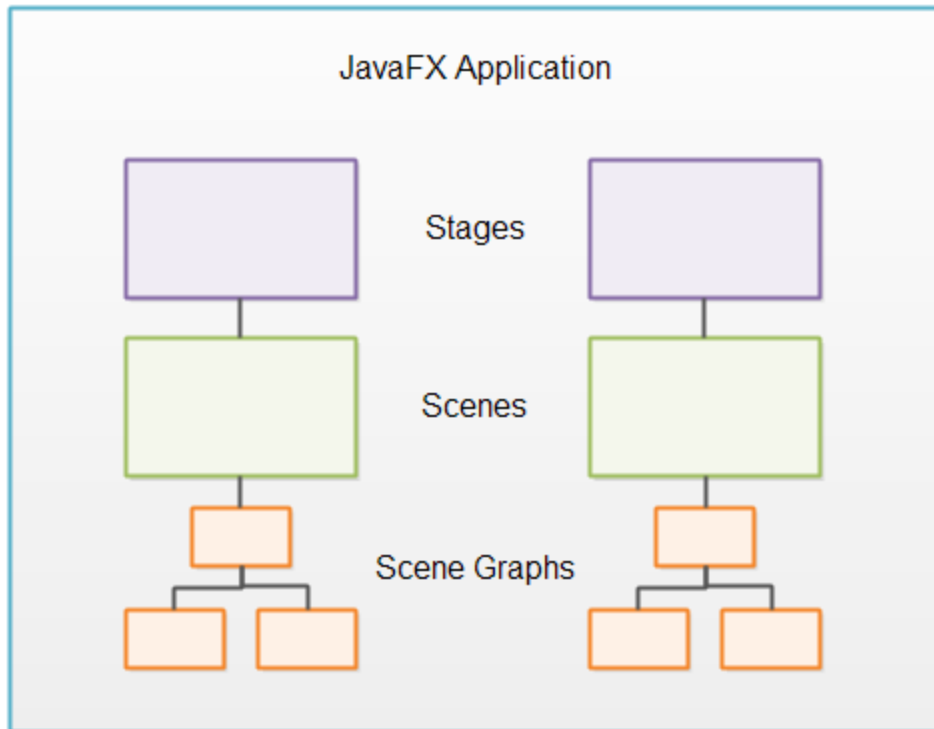
Last update: 2016-05-17

To fully benefit from JavaFX it is useful to understand how JavaFX is designed, and to have a good overview of what features JavaFX contains. The purpose of this text is to give you that JavaFX overview. This text will first look at the general JavaFX design, then look at the various features in JavaFX.

If you are familiar with Flash / Flex, you will see that JavaFX is somewhat inspired by Flash / Flex. Some of the same ideas are found in JavaFX.

In general, a JavaFX application contains one or more stages which corresponds to windows. Each stage has a scene attached to it. Each scene can have an object graph of controls, layouts etc. attached to it, called the scene graph. These concepts are all explained in more detail later. Here is an illustration of the general structure of a JavaFX application:





## Stage

The *stage* is the outer frame for a JavaFX application. The stage typically corresponds to a window. In the early days where JavaFX could run in the browser, the stage could also refer to the area inside the web page that JavaFX had available to draw itself.

Since the deprecation of the Java browser plugin JavaFX is mostly used for desktop applications. Here, JavaFX replaces Swing as the recommended desktop GUI framework. And I must say, that JavaFX looks a whole lot more consistent and feature rich than Swing.

When used in a desktop environment, a JavaFX application can have multiple windows open. Each window has its own stage.

Each stage is represented by a `Stage` object inside a JavaFX application. A JavaFX application has a primary `Stage` object which is created for you by the JavaFX runtime. A JavaFX application can create

additional `Stage` objects if it needs additional windows open. For instance, for dialogs, wizards etc.

## Scene

To display anything on a stage in a JavaFX application, you need a *scene*. A stage can only show one scene at a time, but it is possible to exchange the scene at runtime. Just like a stage in a theater can be rearranged to show multiple scenes during a play, a stage object in JavaFX can show multiple scenes (one at a time) during the life time of a JavaFX application.

You might wonder why a JavaFX application would ever have more than one scene per stage. Imagine a computer game. A game might have multiple "screens" to show to the user. For instance, an initial menu screen, the main game screen (where the game is played), a game over screen and a high score screen. Each of these screens can be represented by a different scene. When the game needs to change from one screen to the next, it simply attaches the corresponding scene to the `Stage` object of the JavaFX application.

A scene is represented by a `Scene` object inside a JavaFX application. A JavaFX application must create all `Scene` objects it needs.

## Scene Graph

All visual components (controls, layouts etc.) must be attached to a scene to be displayed, and that scene must be attached to a stage for the whole scene to be visible. The total object graph of all the controls, layouts etc. attached to a scene is called the *scene graph*.

## Nodes

All components attached to the scene graph are called *nodes*. All nodes are subclasses of a JavaFX class called `javafx.scene.Node`.

There are two types of nodes: Branch nodes and leaf nodes. A branch node is a node that can contain other nodes (child nodes). Branch nodes are also referred to as parent nodes because they can contain child nodes. A leaf node is a node which cannot contain other nodes.

# Controls

JavaFX controls are JavaFX components which provide some kind of control functionality inside a JavaFX application. For instance, a button, radio button, table, tree etc.

For a control to be visible it must be attached to the scene graph of some `Scene` object.

Controls are usually nested inside some JavaFX layout component that manages the layout of controls relative to each other.

JavaFX contains the following controls:

- Accordion
- [Button](#)
- [CheckBox](#)
- ChoiceBox
- ColorPicker
- ComboBox
- DatePicker
- [Label](#)
- ListView
- Menu
- MenuBar
- PasswordField
- ProgressBar
- [RadioButton](#)
- Slider
- Spinner
- SplitMenuButton
- SplitPane
- TableView
- TabPane
- TextArea
- [TextField](#)
- TitledPane
- [ToggleButton](#)
- ToolBar
- TreeTableView
- TreeView

Each of these controls will be explained in separate texts.

## Layouts

*JavaFX layouts* are components which contains other components inside them. The layout component manages the layout of the components nested inside it. JavaFX layout components are also sometimes called *parent components* because they contain child components, and because layout components are subclasses of the JavaFX class `javafx.scene.Parent`.

A layout component must be attached to the scene graph of some `Scene` object to be visible.

JavaFX contains the following layout components:

- Group
- Region
- Pane
- [HBox](#)
- [VBox](#)
- FlowPane
- BorderPane
- BorderPane
- StackPane
- TilePane
- GridPane
- AnchorPane
- TextFlow

Each of these layout components will be covered in separate texts.

## Nested Layouts

It is possible to nest layout components inside other layout components. This can be useful to achieve a specific layout. For instance, to get horizontal rows of components which are not layed out in a grid, but differently for each row, you can nest multiple `HBox` layout components inside a `VBox` component.

## Charts

JavaFX comes with a set of built-in ready-to-use chart components, so you don't have to code charts from scratch everytime you need a basic chart. JavaFX contains the following chart components:

- `AreaChart`
- `BarChart`
- `BubbleChart`
- `LineChart`
- `PieChart`
- `ScatterChart`
- `StackedAreaChart`
- `StackedBarChart`

## 2D Graphics

JavaFX contains features that makes it easy to draw 2D graphics on the screen.

## 3D Graphics

JavaFX contains features that makes it easy to draw 3D graphics on the screen.

## Audio

JavaFX contains features that makes it easy to play audio in JavaFX applications. This is typically useful in games or educational applications.

## Video

JavaFX contains features that makes it easy to play video in JavaFX applications. This is typically useful in streaming applications, games or educational applications.

## WebView

JavaFX contains a `WebView` component which is capable of showing web pages (HTML5, CSS etc.). The JavaFX `WebView` component is based on WebKit - the web page rendering engine also used in Chrome and Safari.

The `WebView` component makes it possible to mix a desktop application with a web application. There are times where that is useful. For instance, if you already have a decent web application, but need some features which can only be provided sensibly with a desktop application - like disk access, communication with other network protocols than HTTP (e.g UDP, IAP etc.) .

Next: [Your First JavaFX Application](#)

# Your First JavaFX Application

- [The JavaFX Application Class](#)
- [Implementing start\(\)](#)
- [Adding a main\(\) Method](#)
- [Adding a Scene](#)

Jakob Jenkov

Last update: 2016-05-11

In this tutorial I will show you how to create your first JavaFX application. This tutorial thus serves both to introduce you to the core JavaFX concepts, as well as to give you a some JavaFX code you can use as template for your own experiments.

## The JavaFX Application Class

A JavaFX application needs a primary launch class. This class has to extend the `javafx.application.Application` class which is a standard class in Java since Java 8.

Here is an example subclass of `Application`:

```
package com.jenkov.javafx.helloworld;

import javafx.application.Application;

public class MyFxApp extends Application {

}
```

## Implementing start()

All subclasses of the `JavaFXApplication` class must implement the abstract `start()` method of the `Application` class (or be an abstract subclass of `Application` itself).

The `start()` method is called when the JavaFX application is started. Here is the example from above, but with the `start()` method implemented:

```
package com.jenkov.javafx.helloworld;

import javafx.application.Application;
import javafx.stage.Stage;

public class MyFxApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("My First JavaFX App");

        primaryStage.show();
    }

}
```

The `start()` method takes a single parameter of the type `Stage`. The stage is where all the visual parts of the JavaFX application are displayed. The `Stage` object is created for you by the JavaFX runtime.

The example above sets a title on the stage object and then calls `show()` on it. That will make the JavaFX application visible in a window with the title visible in the top bar of the window.

If you do not call `show()` on the stage object, nothing is visible. No window is opened. In case your JavaFX application does not become visible when launched, check if you have remembered to call the `Stage show()` method from inside `start()`.

## Adding a `main()` Method

You can actually launch a JavaFX application without a `main()` method. But, if you want to pass command line parameters to the application you need to add a `main()` method. In general I prefer to add a `main()` method because it makes it more explicit which code launches the application.

Here is the example from above with a `main()` method added:

```
package com.jenkov.javafx.helloworld;
```



```
import javafx.application.Application;
import javafx.stage.Stage;

public class MyFxApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("My First JavaFX App");

        primaryStage.show();
    }

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

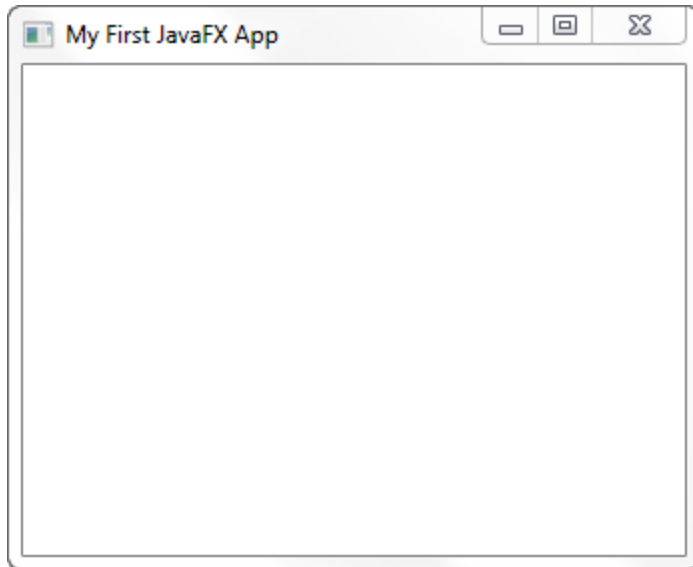
}
```

As you can see, the `main()` method calls the static `launch()` method with the command line parameters. The `launch()` method is a static method located in the `Application` class. This method launches the JavaFX runtime and your JavaFX application.

The `launch()` method will detect from which class it is called, so you don't have to tell it explicitly what class to launch.

That is really all it takes to create a JavaFX application. It is quite simple, isn't it? Now you are ready to start playing around with JavaFX !

Here is a screenshot of the window being opened as a result of running the above JavaFX application:



## Adding a Scene

The previous JavaFX examples only open a window, but nothing is displayed inside this window. To display something inside the JavaFX application window you must add a `Scene` to the `Stage` object. This is done inside the `start()` method.

All components to be displayed inside a JavaFX application must be located inside a scene. The names for "stage" and "scene" are inspired by a theater. A stage can display multiple scenes, just like in a theater play. Similarly, a computer game could have a menu scene, a game scene, a game over scene, a high score scene etc.

Here is an example of how to add a `Scene` object to the `Stage` along with a simple `Label`:

```
package com.jenkov.javafx.helloworld;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class MyFxApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("My First JavaFX App");
    }
}
```

```
Label label = new Label("Hello World, JavaFX !");
Scene scene = new Scene(label, 400, 200);
primaryStage.setScene(scene);

primaryStage.show();
}

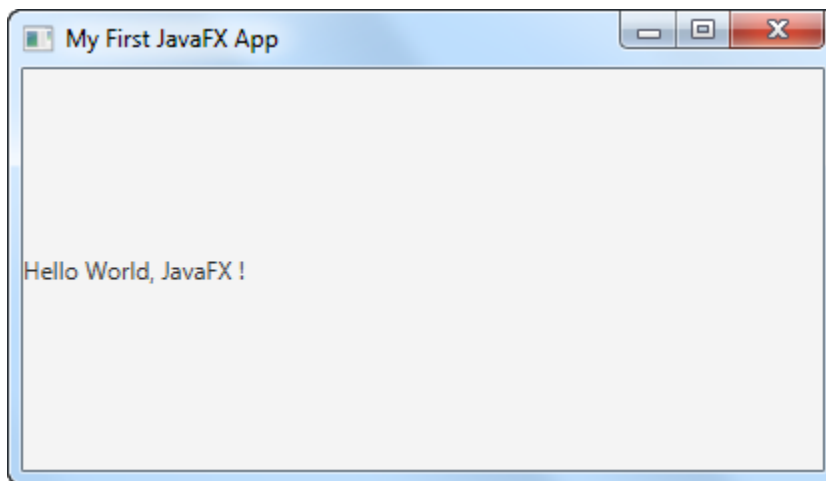
public static void main(String[] args) {
    Application.launch(args);
}
}
```

Three lines have been added to this example. First a `Label` object is created. Then a `Scene` object is created, passing the `Label` as parameter along with two parameters representing the width and height of the scene.

The first parameter of the `Scene` constructor is the root element of the *scene graph*. The scene graph is a graph like object structure containing all the visual components to be displayed in the JavaFX application - for instance GUI components.

The width and height parameters sets the width and height of the JavaFX window when it opened, but the window can be resized by the user.

Here is how the opened window looks with the `Scene` and `Label` added:



Next: [JavaFX Stage](#)

# JavaFX Stage

- [Creating a Stage](#)
- [Showing a Stage](#)
  - [show\(\) vs. showAndWait\(\)](#)
- [Set a Scene on a Stage](#)
- [Stage Title](#)
- [Stage Position](#)
- [Stage Width and Height](#)
- [Stage Modality](#)
- [Stage Owner](#)
- [Stage Style](#)
- [Stage Full Screen Mode](#)

Jakob Jenkov

Last update: 2018-12-28

A JavaFX *Stage*, `javafx.stage.Stage`, represents a window in a JavaFX desktop application. Inside a JavaFX Stage you can insert a JavaFX Scene which represents the content displayed inside a window - inside a Stage.

When a JavaFX application starts up, it creates a root Stage object which is passed to the `start(Stage primaryStage)` method of the root class of your JavaFX application. This Stage object represents the primary window of your JavaFX application. You can create new Stage objects later in your application's life time, in case your application needs to open more windows.

## Creating a Stage

You create a JavaFX Stage object just like any other Java object: Using the `new` command and the Stage constructor. Here is an example of creating a JavaFX Stage object.

```
Stage stage = new Stage();
```

## Showing a Stage

Simple creating a JavaFX Stage object will not show it. In order to make the Stage visible you must call either its `show()` or `showAndWait()` method. Here is an example of showing a JavaFX Stage:

```
Stage stage = new Stage();  
stage.show();
```

### **show() vs. showAndWait()**

The difference between the JavaFX Stage methods `show()` and `showAndWait()` is, that `show()` makes the Stage visible and then exits the `show()` method immediately, whereas the `showAndWait()` shows the Stage object and then blocks (stays inside the `showAndWait()` method) until the Stage is closed.

## Set a Scene on a Stage

In order to display anything inside a JavaFX Stage, you must set a JavaFX Scene object on the Stage. The content of the Scene will then be displayed inside the Stage when the Stage is shown. Here is an example of setting a Scene on a JavaFX Stage:

```
VBox vbox = new VBox(new Label("A JavaFX Label"));  
Scene scene = new Scene(vbox);  
  
Stage stage = new Stage();  
stage.setScene(scene);
```

## Stage Title

You can set the JavaFX Stage title via the Stage setTitle() method. The Stage title is displayed in the title bar of the Stage window. Here is an example of setting the title of a JavaFX Stage:

```
stage.setTitle("JavaFX Stage Window Title");
```

## Stage Position

You can set the position (X,Y) of a JavaFX Stage via its setX() and setY() methods. The setX() and setY() methods set the position of the upper left corner of the window represented by the Stage. Here is an example of setting the X and Y position of a JavaFX Stage object:

```
Stage stage = new Stage();  
  
stage.setX(50);  
stage.setY(50);
```

Please note, that it might be necessary to also set the width and height of the Stage if you set the X and Y position, or the stage window might become very small. See the next section for more information about setting the width and height of a Stage.

## Stage Width and Height

You can set the width and of a JavaFX Stage via its setWidth() and setHeight() methods. Here is an example of setting the width and height of a JavaFX Stage:

```
Stage stage = new Stage();  
  
stage.setWidth(600);  
stage.setHeight(300);
```

## Stage Modality

You can set window modality of a JavaFX Stage. The Stage modality determines if the window representing the Stage will block other windows opened by the same JavaFX application. You set the window modality of a JavaFX Stage via its `initModality()` method. Here is an example of setting the JavaFX Stage modality:

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

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

        Stage stage = new Stage();
        stage.initModality(Modality.APPLICATION_MODAL);
        //stage.initModality(Modality.WINDOW_MODAL);
        //stage.initModality(Modality.NONE);

        primaryStage.show();

        stage.showAndWait();

    }
}
```

Notice how this example is a full JavaFX application. The `start()` method is executed when the JavaFX application is launched (first `main()` is called which calls `launch()` which later calls `start()`).

Notice also, how a new JavaFX Stage object is created, its modality mode set, and then both the primary and the new Stage objects are made visible (shown). The second Stage has its modality set to `Modality.APPLICATION_MODAL` meaning it will block all other windows (stages) opened by this JavaFX application. You cannot access any other windows until this Stage window has been closed.

The `Modality.WINDOW_MODAL` modality option means that the newly created Stage will block the Stage window that "owns" the newly created Stage, but only that. Not all windows in the application. See the next section below to see how to set the "owner" of a Stage.

The `Modality.NONE` modality option means that this Stage will not block any other windows opened in this application.

The `Modality.APPLICATION_MODAL` and `Modality.WINDOW_MODAL` modality modes are useful for Stage objects representing windows that function as "wizards" or "dialogs" which should block the application or window until the wizard or dialog process is completed by the user.

The `Modality.NONE` modality is useful for Stage objects representing windows that can co-exist, like different browser windows in a browser application.

## Stage Owner

A JavaFX Stage can be *owned* by another Stage. You set the owner of a Stage via its `initOwner()` method. Here is an example of initializing the owner of a JavaFX Stage, plus set the modality of the Stage to `Modality.WINDOW_MODAL`:

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

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

        Stage stage = new Stage();
        stage.initModality(Modality.WINDOW_MODAL);

        stage.initOwner(primaryStage);

        primaryStage.show();

        stage.showAndWait();
    }
}
```

This example will open a new Stage which will block the Stage owning the newly created Stage (which is set to the primary stage).

## Stage Style



You can set the style of a JavaFX `Stage` via its `initStyle()` method. There are a set of different styles you can choose from:

- `DECORATED`
- `UNDECORATED`
- `TRANSPARENT`
- `UNIFIED`
- `UTILITY`

A decorated `Stage` is a standard window with OS decorations (title bar and minimize / maximize / close buttons), and a white background.

An undecorated `Stage` is a standard window without OS decorations, but still with a white background.

A transparent `Stage` is an undecorated window with transparent background.

A unified `Stage` is like a decorated stage, except it has no border between the decoration area and the main content area.

A utility `Stage` is a decorated window, but with minimal decorations.

Here is an example of setting the style of a JavaFX `Stage`:

```
stage.initStyle(StageStyle.DECORATED);  
  
//stage.initStyle(StageStyle.UNDECORATED);  
//stage.initStyle(StageStyle.TRANSPARENT);  
//stage.initStyle(StageStyle.UNIFIED);  
//stage.initStyle(StageStyle.UTILITY);
```

Only the first line is actually executed. The rest are commented out. They are just there to show how to configure the other options.

## Stage Full Screen Mode

You can switch a JavaFX `Stage` into full screen mode via the `Stage setFullScreen()` method. Please note, that you may not get the expected result (a window in full screen mode) unless you set

a Scene on the Stage. Here is an example of setting a JavaFX Stage to full screen mode:

```
VBox vbox = new VBox();  
Scene scene = new Scene(vbox);  
  
primaryStage.setScene(scene);  
primaryStage.setFullScreen(true);  
  
primaryStage.show();
```

Next: [JavaFX Scene](#)

# JavaFX Scene

- [Create Scene](#)
- [Set Scene on Stage](#)
- [The Scene Graph](#)
- [Scene Mouse Cursor](#)

Jakob Jenkov

Last update: 2018-12-27

The JavaFX *Scene* object is the root of the JavaFX Scene graph. In other words, the JavaFX Scene contains all the visual JavaFX GUI components inside it. A JavaFX Scene is represented by the class `javafx.scene.Scene`. A Scene object has to be set on a [JavaFX Stage](#) to be visible. In this JavaFX Scene tutorial I will show you how to create a Scene object and add GUI components to it.

## Create Scene

You create a JavaFX Scene object via its constructor. As parameter you must pass the root JavaFX GUI component that is to act as the root view to be displayed inside the Scene. Here is an example of creating a JavaFX Scene object:

```
VBox vBox = new VBox();  
Scene scene = new Scene(vBox);
```

## Set Scene on Stage

In order to make a JavaFX Scene visible, it must be set on a JavaFX Stage. Here is an example of setting a JavaFX Scene on a Stage:

```
VBox vBox = new VBox(new Label("A JavaFX Label"));  
Scene scene = new Scene(vBox);  
  
Stage stage = new Stage();
```

```
stage.setScene(scene);
```

A JavaFX Scene can be attached to only a single Stage at a time, and Stage can also only display one Scene at a time.

## The Scene Graph

As mentioned in the [JavaFX Overview](#), the *scene graph* consists of all the *nodes* which are attached to a given JavaFX Scene object. Each Scene object has its own scene graph.

The scene graph has a single root node. Other nodes can be attached to the root node in a tree-like data structure (a tree is a kind of graph).

## Scene Mouse Cursor

It is possible to set the mouse cursor of a JavaFX Scene. The mouse cursor is the little icon that is being displayed at the location of the mouse cursor (pointer). You set the mouse cursor of a Scene via the `setCursor()` method. Here is an example of setting the mouse cursor of a JavaFX Scene:

```
scene.setCursor(Cursor.OPEN_HAND);
```

The `javafx.scene.Cursor` class contains a lot of constants you can use to specify which mouse cursor you want to display. Some of these constants are:

- `Cursor.OPEN_HAND`
- `Cursor.CLOSED_HAND`
- `Cursor.CROSSHAIR`
- `Cursor.DEFAULT`
- `Cursor.HAND`
- `Cursor.WAIT`
- `Cursor.H_RESIZE`
- `Cursor.V_RESIZE`
- `Cursor.MOVE`
- `Cursor.TEXT`

There are a few more. Just play with the constants found in the `Cursor` class and see for yourself.

Next: [JavaFX FXML](#)

# JavaFX FXML

- [JavaFX FXML Example](#)
- [Loading an FXML File](#)
- [Importing Classes in FXML](#)
- [Creating Objects in FXML](#)
  - [Creating Objects Via FXML Elements and No-arg Constructors](#)
  - [Creating Objects via valueOf\(\) Method](#)
  - [Creating Objects Via Factory Methods](#)
- [Properties in FXML](#)
  - [Property Name Matching](#)
  - [Default Properties](#)
- [FXML Namespace](#)
- [FXML Element IDs](#)
- [FXML Event Handlers](#)
- [FXML CSS Styling](#)
- [FXML Controller Classes](#)
  - [Specifying Controller Class in FXML](#)
  - [Setting a Controller Instance on the FXMLLoader](#)
  - [Binding JavaFX Components to Controller Fields](#)
  - [Referencing Methods in the Controller](#)
  - [Obtaining the Controller Instance From the FXMLLoader](#)

Jakob Jenkov

Last update: 2018-03-24

*JavaFX FXML* is an XML format that enables you to compose JavaFX GUIs in a fashion similar to how you compose web GUIs in HTML. *FXML* thus enables you to separate your JavaFX layout code from the rest of your application code. This cleans up both the layout code and the rest of the application code.

FXML can be used both to compose the layout of a whole application GUI, or just part of an application GUI, e.g. the layout of one part of a form, tab, dialog etc.

## JavaFX FXML Example

The easiest way to start learning about JavaFX FXML is to see an FXML example. Below is a FXML example that composes a simple JavaFX GUI:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox>
  <children>
    <Label text="Hello world FXML"/>
  </children>
</VBox>
```

This example defines a [VBox](#) containing a single [Label](#) as child element. The `VBox` component is a JavaFX layout component. The `Label` just shows a text in the GUI. Don't worry if you do not yet understand all the JavaFX components. You will once you start playing with them all.

The first line in the FXML document is the standard first line of XML documents.

The following two lines are import statements. In FXML you need to import the classes you want to use. Both JavaFX classes and core Java classes used in FXML must be imported.

After the import statements you have the actual composition of the GUI. A `VBox` component is declared, and inside its `children` property is declared a single `Label` component. The result is that the `Label` instance will be added to the `children` property of the `VBox` instance.

## Loading an FXML File

In order to load an FXML file and create the JavaFX GUI components the file declares, you use the `FXMLLoader` (`javafx.fxml.FXMLLoader`) class. Here is a full JavaFX FXML loading example which loads an FXML file and returns the JavaFX GUI component declared in it:

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
```

```

import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;

public class FXMLExample extends Application{

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

    @Override
    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(new URL("file:///C:/data/hello-world.fxml"));
        VBox vbox = loader.<VBox>load();

        Scene scene = new Scene(vbox);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

For this example to work, the FXML file must be located at `C:\data\hello-world.fxml`. As you can see, the location of the file is set via the `setLocation()` method. The root GUI component (the `VBox` object) is obtained via the `load()` method.

## Importing Classes in FXML

In order to use a Java class in FXML, whether a JavaFX GUI component or a regular Java class, the class must be imported in the FXML file. FXML import statements look like this:

```
<?import javafx.scene.layout.VBox?>
```

This FXML import statement imports the class `javafx.scene.layout.VBox`.

## Creating Objects in FXML

FXML can create both JavaFX GUI objects as well as non-JavaFX objects. There are several ways to create objects in FXML. In the following sections we will see what these options are.



## Creating Objects Via FXML Elements and No-arg Constructors

The easiest way to create objects in FXML is via an FXML element in an FXML file. The element names used in FXML are the same names as the Java class names without the package names. Once you have imported a class via an FXML import statement, you can use its name as an FXML element name.

In the following example the element names `VBox` and `Label` are valid because these two classes are declared with import statements earlier in the FXML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox>
  <children>
    <Label text="Hello world FXML"/>
  </children>
</VBox>
```

To create objects using FXML elements like this requires that the class of the created object has a no-argument constructor.

## Creating Objects via valueOf() Method

Another way to create objects in FXML is to call a static `valueOf()` method in the class you want to create the object of. The way to create objects via a `valueOf()` method is to insert a `value` attribute in the FXML element. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import com.jenkov.javafx.MyClass?>

<MyClass value="The Value"/>
```

Here is how the corresponding `MyClass` needs to look for this to work:

```
public MyClass {
    public static MyClass valueOf(String value) {
        return new MyClass(value);
    }
}
```

```
private String value = null;

public MyClass(String value) {
    this.value = value;
}
}
```

Notice the static `valueOf()` method which takes a [Java String](#) as parameter. This method is called by the `FXMLLoader` when it sees the `MyClass` element in the FXML file. The object returned by the `valueOf()` method is what is inserted into the GUI composed in the FXML file. The above FXML doesn't contain any other elements than the `MyClass` element, but it could.

Keep in mind that whatever object is returned by the `valueOf()` method will be used in the object graph (composed GUI). If the object returned is not an instance of the class containing the `valueOf()` method, but an instance of some other class, then that object will still be used in the object graph. The element name is used only to find the class containing the `valueOf()` method (when the FXML element contains a `value` attribute).

## Creating Objects Via Factory Methods

In a sense, a `valueOf()` method is also a factory method that creates objects based on a `String` parameter. But - you can also get the `FXMLLoader` to call other factory methods than a `valueOf()` method.

To call another factory method to create an object, you need to insert an `fx:factory` attribute. The value of the `fx:factory` attribute should be the name of the factory method to call. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import com.jenkov.javafx.MyClass?>

<MyClass fx:factory="instance"/>
```

The `MyClass` class should look like this for the above FXML example to work:

```
public MyClass {  
    public static MyClass instance() {  
        return new MyClass();  
    }  
}
```

Notice the `instance()` method. This method is referenced from the `fx:factory` attribute in the FXML snippet above.

Note, that the factory method must be a no-argument method to call it from a `fx:factory` attribute.

## Properties in FXML

Some JavaFX objects have properties. In fact, most of them do. You can set the values of properties in two ways. The first way is to use an XML attribute to set the property value. The second way is to use a nested XML element to set the property value.

To understand how to set properties in FXML elements better, let us look at an example:

```
<?xml version="1.0" encoding="UTF-8"?>  
<?import javafx.scene.layout.VBox?>  
<?import javafx.scene.control.Label?>  
  
<VBox spacing="20">  
    <children>  
        <Label text="Line 1"/>  
        <Label text="Line 2"/>  
    </children>  
</VBox>
```

This example shows 3 property examples. The first example is the `spacing` attribute in the `VBox` element. The value set in the `spacing` attribute is passed as parameter to the `setSpacing()` method of the `VBox` object created based on the `VBox` element.

The second example is the `children` element nested inside the `VBox` element. This element corresponds to the `getChildren()` method of the `VBox` class. The elements nested

inside the `children` element will be converted to JavaFX components that are added to the collection obtained from the `getChildren()` method of the `VBox` object represented by the parent `VBox` element.

The third example are the `text` attributes of the two `Label` elements nested inside the `children`. The values of the `text` attributes will be passed as parameters to the `setText()` property of the `Label` objects created by the `Label` elements.

## Property Name Matching

FXML considers "properties" to be member variables accessed via getters and setters. E.g. `getText()` and `setText()`.

As you can see from the example in the previous section the property names of JavaFX classes are matched to the attribute and element names by:

- Remove any `get/set` in the property name.
- Convert first remaining character of property name to lowercase.

Thus, the getter method `getChildren` will first be reduced to `Children` and then to `children`. Similarly, the setter method `setText` will be reduced to `Text` and then to `text`.

## Default Properties

A JavaFX component can have a default property. That means, that if a FXML element contains children which are not nested inside a property element, then it is assumed that the children are belonging to the default property.

Let us look at an example. The `VBox` class has the `children` property as default property. That means that we can leave out the `children` element. Thus, this FXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox spacing="20">
  <children>
    <Label text="Line 1"/>
    <Label text="Line 2"/>
  </children>
</VBox>
```

```
    </children>
</VBox>
```

can be shortened to:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox spacing="20">
    <Label text="Line 1"/>
    <Label text="Line 2"/>
</VBox>
```

The two `Label` elements are then assumed to belong to the default property of `VBox`, which is the `children` property.

A default property is marked with the JavaFX annotation `@DefaultProperty(value="propertyName")` where the value is the name of the property that should be the default property. For instance, the `@DefaultProperty(value="children")` declaration would make the `children` property the default property.

## FXML Namespace

FXML has a namespace you can set on the root element of your FXML files. The FXML namespace is needed for some FXML attributes like the `fx:id` attribute (see the next section in this FXML tutorial).

Setting the FXML namespace on the root element of an FXML file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns:fx="http://javafx.com/fxml">
</VBox>
```

The FXML namespace is declared by the attribute declaration `xmlns:fx="http://javafx.com/fxml"`.

## FXML Element IDs

You can assign IDs to FXML elements. These IDs can be used to reference the FXML elements elsewhere in the FXML file. Specifying an ID for an FXML element is done via the `id` attribute from the FXML namespace. Here is an example of specifying an ID for an FXML element:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>

<VBox xmlns:fx="http://javafx.com/fxml">
    <Label fx:id="label1" text="Line 1"/>
</VBox>
```

Notice the attribute declaration `fx:id="label1"` in the `Label` element. This attribute declares the ID of that `Label` element. Now this specific `Label` element can be referenced via the ID `label1` elsewhere in the FXML document. For instance, this ID can be used to reference the FXML element from CSS. You will see examples of referencing FXML elements by their ID later in this FXML tutorial.

## FXML Event Handlers

It is possible to set event handlers on JavaFX objects from inside the FXML file that defines the JavaFX objects. You might prefer to set advanced event handlers from within Java code, but for simple event handlers setting them from within FXML might be fine.

In order to define an event handler you need to use a `script` element. Here is how an FXML script element looks:

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

<VBox xmlns:fx="http://javafx.com/fxml">
    <Label fx:id="label1" text="Button not clicked"/>
    <Button fx:id="button1" text="Click me!" onAction="reactToClick()"/>

    <fx:script>
        function reactToClick() {
            label1.setText("Button clicked");
        }
    </fx:script>
</VBox>
```

```
    }  
    </fx:script>  
  
</VBox>
```

This example shows two interesting FXML concepts. The first concept is adding an event listener to a JavaFX component from within FXML.

The `Button` element declares an event listener via its `onAction` attribute. The attribute value declares a call to the `reactToClick()` function which is defined in the `script` element further down the FXML file.

The second concept is the reference of a JavaFX component via its ID from within the FXML file. Inside the `reactToClick()` method declared in the `script` element, the `Label` element is referenced via its ID `label1`, via this statement:

```
label1.setText("Button clicked");
```

The `onAction` event listener attribute corresponds to the `onAction` event of the `Button` component. You can set this event listener via Java code too, via the `Button setOnAction()` method. You can set listeners for other events in FXML too, by matching their event listener methods from the corresponding JavaFX component with an FXML attribute, using the same name matching rules as for other properties (see earlier section on [property name matching](#)).

## FXML CSS Styling

It is possible to style the JavaFX components declared inside an FXML file. You can do so by embedding a `style` element inside the FXML element. Here is an example of CSS styling a JavaFX button in an FXML file:

```
<?xml version="1.0" encoding="UTF-8"?>  
<?import javafx.scene.layout.VBox?>  
<?import javafx.scene.control.Button?>  
  
<VBox xmlns:fx="http://javafx.com/fxml">  
    <Button text="Click me!" / onAction="reactToClick()">  
        <style>  
            -fx-padding: 10;
```

```
        -fx-border-width: 3;
    </style>
</Button>
</VBox>
```

This example sets the `-fx-padding` CSS property to 10, and the `-fx-border-width` property to 3. Since the `style` element is nested inside the `button` element, these CSS styles will be applied to that `button` element.

## FXML Controller Classes

You can set a controller class for an FXML document. An FXML controller class can bind the GUI components declared in the FXML file together, making the controller object act as a mediator (design pattern).

There are two ways to set a controller for an FXML file. The first way to set a controller is to specify it inside the FXML file. The second way is to set an instance of the controller class on the `FXMLLoader` instance used to load the FXML document. This JavaFX FXML tutorial will show both options in the following sections.

### Specifying Controller Class in FXML

The controller class is specified in the root element of the FXML file using the `fx:controller` attribute. Here is an example of specifying a controller in FXML:

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

<VBox xmlns:fx="http://javafx.com/fxml"
fx:controller="com.jenkov.javafx.MyFXMLController" >
    <Button text="Click me!" onAction="reactToClick()">
    </Button>
</VBox>
```

Notice the `fx:controller` attribute in the root element (the `VBox` element). This attribute contains the name of the controller class.



An instance of this class is created when the FXML file is loaded. For this to work, the controller class must have a no-argument constructor.

## Setting a Controller Instance on the FXMLLoader

When setting a controller instance on an `FXMLLoader` you must first create an instance of the controller class, and then set that instance on the `FXMLLoader`. Here is an example of setting a controller instance on an `FXMLLoader` instance:

```
MyFXMLController controller = new MyFXMLController();

FXMLLoader loader = new FXMLLoader();
loader.setController(controller);
```

## Binding JavaFX Components to Controller Fields

You can bind the JavaFX components in the FXML file to fields in the controller class. To bind a JavaFX component to a field in the controller class, you need to give the FXML element for the JavaFX component an `fx:id` attribute which has the name of the controller field to bind it to as value. Here is an example controller class:

```
public class MyFXMLController {

    public Label label1 = null;

}
```

And here is the FXML file with a `Label` element bound to the `label1` field of the controller class:

```
<VBox xmlns:fx="http://javafx.com/fxml" >
    <Label fx:id="label1" text="Line 1"/>
</VBox>
```

Notice how the value of the `fx:id` attribute has the value `label1` which is the same as the field name in the controller class to which it should be bound.

## Referencing Methods in the Controller

It is possible to reference methods in the controller instance from FXML. For instance, you can bind the events of a JavaFX GUI component to methods of the controller. Here is an example of binding an event of a JavaFX component to a method in the controller:

```
<VBox xmlns:fx="http://javafx.com/fxml"
fx:controller="com.jenkov.javafx.MyFXMLController" spacing="20">
<children>
    <Label fx:id="label1" text="Line 1"/>
    <Label fx:id="label2" text="Line 2"/>
    <Button fx:id="button1" text="Click me!" onAction="#buttonClicked"/>
</children>
</VBox>
```

This example binds the `onAction` event of the `Button` to the method `buttonClicked` in the controller class. Here is how the controller class must look to enable the event binding:

```
import javafx.event.Event;
import javafx.fxml.FXML;
import javafx.scene.control.Label;

public class MyFXMLController {

    @FXML
    public void buttonClicked(Event e){
        System.out.println("Button clicked");
    }

}
```

Notice the `@FXML` annotation above the `buttonClicked` method. This annotation marks the method as a target for binding for FXML. Notice also that the name `buttonClicked` is referenced in the FXML file.

## Obtaining the Controller Instance From the FXMLLoader

Once the `FXMLLoader` instance has loaded the FXML document, you can obtain a reference to the controller instance via the `FXMLLoader.getController()` method. Here is an example:

```
MyFXMLController controllerRef = loader.getController();
```

Next: [JavaFX CSS Styling](#)

# JavaFX CSS Styling

- [CSS Styling Methods](#)
  - [Default CSS Stylesheet](#)
  - [Scene Specific CSS Stylesheet](#)
- [Parent Specific CSS Stylesheets](#)
- [Component Style Property](#)
- [JavaFX CSS Properties](#)

Jakob Jenkov

Last update: 2016-05-31

JavaFX enables you to style JavaFX components using CSS, just like you can style HTML and SVG element in web pages with CSS. JavaFX uses the same CSS syntax as CSS for the web, but the CSS properties are specific to JavaFX and therefore have slightly different names than their web counterparts.

Styling your JavaFX applications using CSS helps you separate styling (looks) from the application code. This results in cleaner application code and makes it easier to change the styling of the application. You do not have to look inside the Java code to change the styling. You can also change the styling for many components at once, by using shared CSS stylesheets.

In this JavaFX CSS tutorial I will take a deeper look at the many different ways you can apply CSS styles to your JavaFX applications, as well as look at what can and cannot be styled. The JavaFX CSS features are quite advanced, so there is a lot you can do with just CSS.

I assume that you are already somewhat familiar with the core concepts of CSS, like CSS syntax, CSS rules, CSS properties etc. If not, it might be a good idea to read the basics about CSS in my [CSS tutorial](#)

## CSS Styling Methods

There are several different methods to apply a CSS style to a JavaFX component. These methods are:

- JavaFX default CSS stylesheet

- Scene specific CSS stylesheet
- Parent specific CSS stylesheet
- Component `style` property

I will briefly explain how each of these CSS styling mechanisms work in the following sections.

## Default CSS Stylesheet

JavaFX applications have a default CSS stylesheet which is applied to all JavaFX components. If you provide no styling of the components, the default CSS stylesheet will style the JavaFX components so they look pretty.

The default stylesheet for JavaFX 8 is called Modena, and is located in the JavaFX JAR file.

## Scene Specific CSS Stylesheet

You can set a CSS stylesheet for a JavaFX `Scene` object. This CSS stylesheet is then applied to all JavaFX components added to the scene graph for that `Scene` object. The scene specific CSS stylesheet will override the styles specified in the default stylesheet, in case both stylesheets sets the same CSS properties.

Here is an example of setting a CSS stylesheet on a `Scene` object:

```
scene.getStylesheets().add("style1/button-styles.css");
```

This example tells JavaFX to look for a stylesheet file called `button-styles.css` which is located in a directory called `style1`. JavaFX looks for this file on the classpath, so the directory `style1` should be located in a directory which is at root of one of the directories (or JAR files) which are included in the classpath for the application.

The string pointing to the CSS stylesheet file is interpreted as a URL. That means that you can also specify full paths to a file in the file system. However, it is good practice to think of CSS files as resources and bundle them with the code for your application. The users running the application will typically not

be changing the styles, so there is no need to distribute the file outside the code (like you would with config file that users were intended to change).

## Parent Specific CSS Stylesheets

It is also possible to set a CSS stylesheet on all subclasses of the `Parent` class. The `Parent` class is a base class for all components that can have children, meaning they can contain other components inside them. CSS properties specified in a stylesheet set on a `Parent` subclass will normally take precedence over CSS rules specified in a scene stylesheet, and the default stylesheet.

The JavaFX layout components are typical examples of `Parent` subclasses. If you set a CSS stylesheet on a layout component, the stylesheet will be applied to all components inside that layout component.

Setting a CSS stylesheet on a `Parent` subclass looks similar to setting it on a `Scene` object. Here is an example of setting a CSS stylesheet on a `VBox` which is a `Parent` subclass:

```
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");

VBox vbox = new VBox(button1, button2);

vbox.getStylesheets().add("style1/button-styles.css");
```

This code will set the `style1/button-styles.css` stylesheet on the `VBox`. The stylesheet will thus be applied to the two buttons inside the `VBox`.

## Component Style Property

You can set CSS styles for a specific component by setting the CSS properties directly on the component. This is done by setting a `String` containing the CSS properties in the component's `style` property.

CSS properties set via the `style` property take precedence over CSS properties specified in any `Parent` subclasses the component is nested inside, the scene stylesheet and the default stylesheet.

Here is an example that sets the `style` property for a JavaFX `Button` :

```
Button button = new Button("Button 2");
button.setStyle("-fx-background-color: #0000ff");
```

This example sets the background color CSS property in the `style` property to a blue color.

You can set multiple CSS properties inside the same style string. Here is an example of how that looks:

```
String styles =
    "-fx-background-color: #0000ff;" +
    "-fx-border-color: #ff0000;" ;

Button button = new Button("Button 2");
button.setStyle(styles);
```

## JavaFX CSS Properties

As mentioned earlier JavaFX contains its own set of CSS properties. The JavaFX CSS properties are named somewhat differently from the CSS properties used with HTML. However, the JavaFX team have kept the names of the JavaFX CSS properties very close to the CSS properties used in HTML. If you are familiar with CSS for the web, you will often be able to guess what the corresponding JavaFX CSS property is called.

Here is a list of the most commonly used JavaFX CSS properties. Not all CSS properties can be applied to all JavaFX components, but many CSS properties can be applied to several JavaFX components. I will update this list over time, by the way, so check back in the future to see a (hopefully) more complete list of JavaFX CSS properties.

JavaFX CSS Property	Description
<code>-fx-background-color</code>	Sets the background color of a JavaFX component (Node subclass).

Next: [JavaFX ImageView](#)

# JavaFX ImageView

- [Creating an ImageView](#)
- [Adding an ImageView to the Scene Graph](#)
- [ImageView in Labels and Buttons](#)

Jakob Jenkov

Last update: 2016-05-13

The JavaFX ImageView control can display an image inside a JavaFX GUI. The ImageView control must be added to the scene graph to be visible. The JavaFX ImageView control is represented by the class `javafx.scene.image.ImageView`.

## Creating an ImageView

You create an ImageView control instance by creating an instance of the ImageView class. The constructor of the ImageView class needs an instance of a `javafx.scene.image.Image` as parameter. The Image object represents the image to be displayed by the ImageView control.

Here is a JavaFX ImageView instantiation example:

```
FileInputStream input = new FileInputStream("resources/images/iconmonstr-home-6-48.png");
Image image = new Image(input);
ImageView imageView = new ImageView(image);
```

First a `FileInputStream` is created which points to the image file of the image to display.

Second an `Image` instance is created, passing the `FileInputStream` as parameter to the `Image` constructor. This way the `Image` class knows where to load the image file from.

Third an `ImageView` instance is created, passing the `Image` instance as parameter to the `ImageView` constructor.

## Adding an ImageView to the Scene Graph

To make the `ImageView` visible you must add it to the scene graph. This means adding it to a `Scene` object. Since `ImageView` is not a subclass of `javafx.scene.Parent` it cannot be added directly to the scene graph. It must be nested inside another component, for instance a layout component.

Here is an example that attaches a JavaFX `ImageView` to the scene graph by nesting it inside a `HBox` layout component:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

import java.io.FileInputStream;

public class ImageViewExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ImageView Experiment 1");

        FileInputStream input = new
FileInputStream("resources/images/iconmonstr-home-6-48.png");
        Image image = new Image(input);
        ImageView imageView = new ImageView(image);

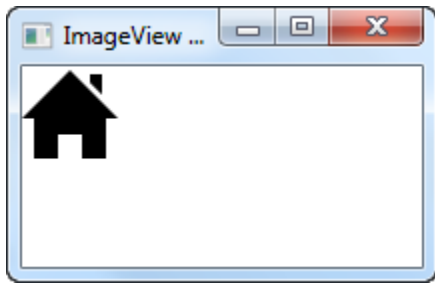
        HBox hbox = new HBox(imageView);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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



The result of running the above JavaFX `ImageView` example is an application that looks like this:



## ImageView in Labels and Buttons

It is possible to use an `ImageView` in both a JavaFX `Label` and `Button`. This will cause the `Label` and `Button` to display the `ImageView` to the left of the text in the `Label` or `Button`. See the texts about [JavaFX Label](#) and [JavaFX Button](#) for information about how to do that.

Next: [JavaFX Label](#)

# JavaFX Label

- [Creating a Label](#)
- [Adding a Label to the Scene Graph](#)
- [Displaying Images in a Label](#)
- [Changing the Text of a Label](#)
- [Set Label Font](#)

Jakob Jenkov

Last update: 2019-08-22

The JavaFX Label control can display a text or image label inside a JavaFX GUI. The label control must be added to the scene graph to be visible. The JavaFX Label control is represented by the class `javafx.scene.control.Label`.

## Creating a Label

You create a label control instance by creating an instance of the `Label` class. Here is a JavaFX Label instantiation example:

```
Label label = new Label("My Label");
```

As you can see, the text to display in the label is passed as parameter to the `Label` constructor.

## Adding a Label to the Scene Graph

To make the `Label` visible you must add it to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `Label` to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class LabelExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        Label label = new Label("My Label");

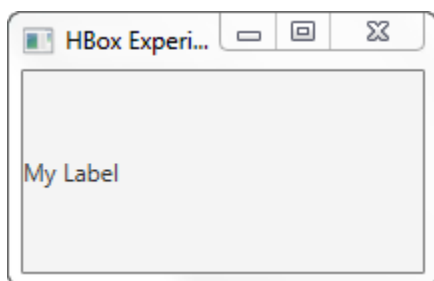
        Scene scene = new Scene(label, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

Notice that the `Label` is added directly to the `Scene` object. Normally you would nest the `Label` inside a layout component of some kind. I have left that out here to keep the example simple. See the tutorials about layout components to see how they work.

The result of running the above JavaFX `Label` example is an application that looks like this:



## Displaying Images in a Label

It is possible to display an image inside a label next to the label text. The JavaFX `Label` class contains a constructor that can take a `Node` as extra parameter. Here is a JavaFX label example that adds an image to the label using an [JavaFX ImageView](#) component:

```

package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

import java.io.FileInputStream;

public class LabelExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        FileInputStream input = new
FileInputStream("resources/images/iconmonstr-home-6-48.png");
        Image image = new Image(input);
        ImageView imageView = new ImageView(image);

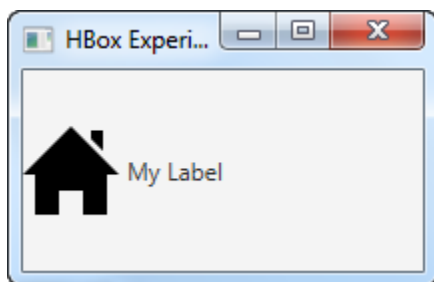
        Label label = new Label("My Label", imageView);

        Scene scene = new Scene(label, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The result of running the above JavaFX Label example is an application that looks like this:



## Changing the Text of a Label

You can change the text of a label using its `setText()` method. This can be done while the application is running. See the [JavaFX Button tutorial](#) for an example that changes the text of a label when a button is clicked.

## Set Label Font

You can change the font used by a JavaFX `Label` by calling its `setFont()` method. This is useful if you need to change the size of the text, or want to use a different text style. Here is an example of setting the font of a JavaFX `Label`:

```
Label label = new Label("A label with custom font set.");  
label.setFont(new Font("Arial", 24));
```

This example tells the `Label` to use the `Arial` font with a size of 24.

Next: [JavaFX Hyperlink](#)

# JavaFX Hyperlink

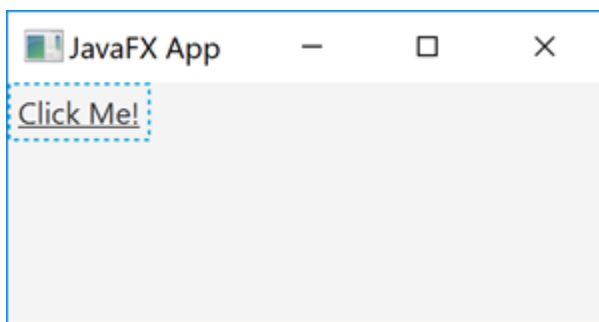
- [JavaFX Hyperlink Example](#)
- [Create a Hyperlink](#)
- [Set Hyperlink Action](#)

Jakob Jenkov

Last update: 2019-05-25

The *JavaFX Hyperlink* control is a text that functions as a button, meaning you can configure a Hyperlink to perform some action when the user clicks it. Just like a hyperlink in a web page. The *JavaFX Hyperlink* control is represented by the class `javafx.scene.control.Hyperlink`.

Here is a screenshot showing how a JavaFX Hyperlink looks:



## JavaFX Hyperlink Example

Here is a full *JavaFX Hyperlink example*:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Hyperlink;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX App");
```

```

Hyperlink link = new Hyperlink("Click Me!");

VBox vBox = new VBox(link);
Scene scene = new Scene(vBox, 960, 600);

primaryStage.setScene(scene);
primaryStage.show();
}
}

```

This example is a full JavaFX application that creates a `Hyperlink`, inserts it into a [JavaFX VBox](#) which is then added to a [JavaFX Scene](#). The `Scene` is then added to a [JavaFX Stage](#) which is then made visible.

## Create a Hyperlink

In order to use a JavaFX `Hyperlink` control you must first create a `Hyperlink` instance. Here is an example of creating a JavaFX `Hyperlink` instance:

```
Hyperlink link = new Hyperlink("Click me!");
```

## Set Hyperlink Action

To respond to clicks on a JavaFX `Hyperlink` you set an action listener on the `Hyperlink` instance. Here is an example of setting an action listener on a JavaFX `Hyperlink` instance:

```
Hyperlink link = new Hyperlink("Click me!");

link.setOnAction(e -> {
    System.out.println("The Hyperlink was clicked!");
});

```

Next: [JavaFX Button](#)

# JavaFX Button

- [Creating a Button](#)
- [Adding a Button to the Scene Graph](#)
- [Button Text](#)
  - [Button Text Size](#)
  - [Button Text Wrap](#)
- [Button Image](#)
- [Button Size](#)
- [Button Events](#)
- [Button Mnemonic](#)
- [Button CSS Styles](#)

Jakob Jenkov

Last update: 2016-06-12

A JavaFX Button control enables a JavaFX application to have some action executed when the application user clicks the button. The JavaFX Button control is represented by the class `javafx.scene.control.Button`.

## Creating a Button

You create a button control by creating an instance of the `Button` class. Here is a JavaFX `Button` instantiation example:

```
Button button = new Button("My Label");
```

The text to be displayed on the button is passed as parameters to the `Button` constructor.

## Adding a Button to the Scene Graph



For a JavaFX `Button` to be visible the button object must be added to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `Button` to the scene graph:

```
package com.jenkov.javafx.controls;

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

public class ButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

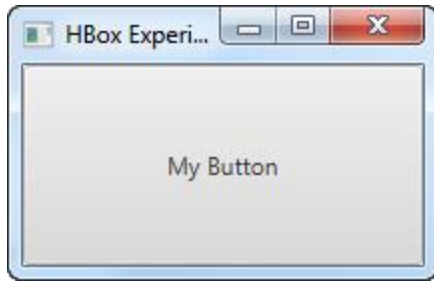
        Button button = new Button("My Button");

        Scene scene = new Scene(button, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Notice that the `Button` is added directly to the `Scene` object. Normally you would nest the `Button` inside a layout component of some kind. I have left that out here to keep the example simple. See the tutorials about layout components to see how they work.

The result of running the above JavaFX `Button` example is an application that looks like this:



Notice that the button takes up all the space available in the window. That is why it is hard to see the edges of the button. When a JavaFX button is added to a layout component you can more easily see the edges of the button.

## Button Text

There are two ways to set the text of a JavaFX button. The first way is to pass the text to the `Button` constructor. You have already seen this in earlier examples.

The second way to set the button text is by calling the `setText()` method on the `Button` instance. This can be done after the `Button` instance is created. Thus it can be used to change the text of a `Button` that is already visible. Here is an example how how calling `setText()` on a JavaFX `Button` looks:

```
button.setText("Click me if you dare!");
```

## Button Text Size

You can set the text size of a JavaFX `Button`. You do so using the CSS property `-fx-text-size`. This CSS property is explained in the section about [Button CSS Styling](#)

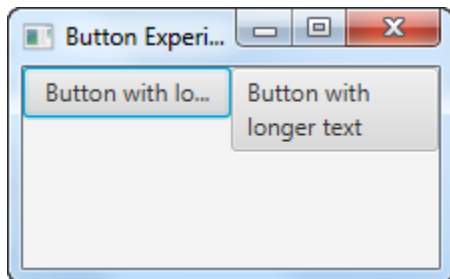
## Button Text Wrap

The JavaFX `Button` control supports text wrapping of the button text. By text wrapping is meant that if the text is too long to be displayed on a single line inside the button, the text is broken onto multiple lines.

You enable text wrapping on a JavaFX Button instance using the method `setWrapText()`. The `setWrapText()` method takes a single boolean parameter. If you pass a value of `true` to `setWrapText()` then you enable text wrapping. If you pass a value of `false` to `setWrapText()` then you disable text wrapping. Here is an example that enables text wrapping on a JavaFX button:

```
button.setWrapText(true);
```

Here is a screenshot of two JavaFX buttons one of which has text wrapping enabled:



## Button Image

It is possible to display an image inside a button next to the button text. The JavaFX Button class contains a constructor that can take a Node as extra parameter. Here is a JavaFX label example that adds an image to the button using an [JavaFX ImageView](#) component:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

import java.io.FileInputStream;

public class ButtonExperiments extends Application {

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

```

        primaryStage.setTitle("HBox Experiment 1");

        FileInputStream input = new
FileInputStream("resources/images/iconmonstr-home-6-48.png");
        Image image = new Image(input);
        ImageView imageView = new ImageView(image);

        Button button = new Button("Home", imageView);

        Scene scene = new Scene(button, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The result of running the above JavaFX Button example is an application that looks like this:



## Button Size

The JavaFX Button class contains a set of methods you can use to set the button size. The methods controlling the button size are:

```

button.setMinWidth()
button.setMaxWidth()
button.setPrefWidth()

button.setMinHeight()
button.setMaxHeight()
button.setPrefHeight()

button.setMinSize()
button.setMaxSize()
button.setPrefSize()

```

The methods `setMinWidth()` and `setMaxWidth()` sets the minimum and maximum width the button should be allowed to have. The method `setPrefWidth()` sets the preferred width of the button. When there is space enough to display a button in its preferred width, JavaFX will do so. If not, JavaFX will scale the button down until it reaches its minimum width.

The methods `setMinHeight()` and `setMaxHeight()` sets the minimum and maximum height the button should be allowed to have. The method `setPrefHeight()` sets the preferred height of the button. When there is space enough to display a button in its preferred height, JavaFX will do so. If not, JavaFX will scale the button down until it reaches its minimum height.

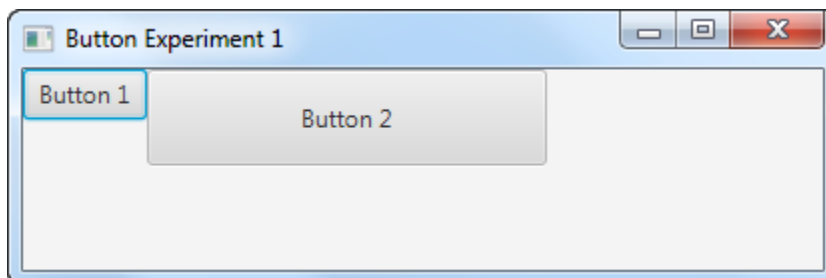
The methods `setMinSize()`, `setMaxSize()` and `setPrefSize()` sets both width and height for the button in a single call. Thus, these methods takes both a width and a height parameter. For instance, calling

```
button.setMaxSize(100, 200);
```

is equivalent to calling

```
button.setMaxWidth(100);  
button.setMaxHeight(200);
```

Here is a screenshot of two JavaFX buttons. The first button has the default size calculated from its button text and the layout component it is nested inside. The second button has a preferred width of 200 and height of 48 set on it:



# Button Events

In order to respond to the click of a button you need to attach an event listener to the `Button` object. Here is how that looks:

```
button.setOnAction(new EventHandler() {  
    @Override  
    public void handle(ActionEvent actionEvent) {  
        //... do something in here.  
    }  
});
```

Here is how attaching a click event listener looks with a [Java Lambda expression](#):

```
button.setOnAction(actionEvent -> {  
    //... do something in here.  
});
```

Finally, let us see a full example that changes the text of a [JavaFX Label](#) when the button is clicked:

```
package com.jenkov.javafx.controls;  
  
import javafx.application.Application;  
import javafx.event.ActionEvent;  
import javafx.event.EventHandler;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.layout.HBox;  
import javafx.stage.Stage;  
  
public class ButtonExperiments extends Application {  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        primaryStage.setTitle("HBox Experiment 1");  
  
        Label label = new Label("Not clicked");  
        Button button = new Button("Click");  
  
        button.setOnAction(value -> {  
            label.setText("Clicked!");  
        });  
  
        HBox hbox = new HBox(button, label);
```

```
        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

## Button Mnemonic

You can set a mnemonic on a JavaFX `Button` instance. A *mnemonic* is a keyboard key which activates the button when pressed in conjunction with the ALT key. Thus, a mnemonic is a keyboard shortcut to activating the button. I will explain how to activate a button via its mnemonic later.

The mnemonic for a button is specified inside the button text. You mark which key is to be used as mnemonic by placing an underscore character (`_`) in front of the character in the button text you want to set as mnemonic for that button. The underscore character will not be displayed in the button text. Here is an example setting a mnemonic for a button:

```
button.setMnemonicParsing(true);

button.setText("_Click");
```

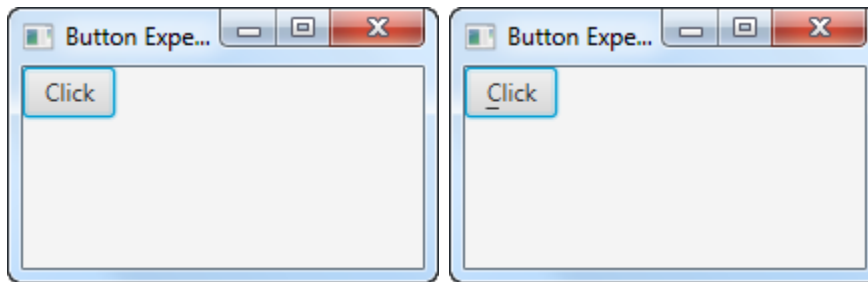
Notice that it is necessary to first call `setMnemonicParsing()` on the button with a value of `true`. This instructs the button to parse mnemonics in the button text. If you call this method with a value of `false` instead, the underscore character in the button text will just show up as text, and will not be interpreted as a mnemonic.

The second line sets the text `_Click` on the button. This tells the button to use the key `c` as mnemonic. Mnemonics are case insensitive, so it does not have to be an uppercase `C` that activates the button.

To activate the button you can now press ALT-C (both at the same time). That will activate the button just as if you had clicked it with the mouse.

You can also first press the ALT key once. That will show the mnemonic of the button in the button text. You can then press the `C` key. If you press ALT and then ALT again, the mnemonic is first shown, then hidden again. When the mnemonic is visible you can activate the button with the mnemonic key alone, without ALT pressed at the same time. When the mnemonic is not visible you have to press both ALT and the mnemonic key at the same time to activate the button.

Here are two screenshots showing what it looks like when the mnemonic is invisible and visible:



## Button CSS Styles

You can style a JavaFX button using CSS styles. The JavaFX `Button` control supports the following CSS styles:

```
-fx-border-width  
-fx-border-color  
-fx-background-color  
-fx-font-size  
-fx-text-fill
```

Here is an example setting the background color of a JavaFX button to red:

```
Button button = new Button("My Button");  
  
button.setStyle("-fx-background-color: #ff0000; ");
```

This example sets the style directly on the button via the `setStyle()` method, but you can also style a JavaFX button via style sheets. See my [JavaFX CSS Styling](#) tutorial for more information about using CSS stylesheets with JavaFX.



Here is a JavaFX button example which creates 4 different buttons. Each button has a CSS style set on them. After the code example I have included a screenshot of how the buttons look with the given styling.

```
package com.jenkov.javafx.controls;

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

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Button Experiment 1");

        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button 2");
        Button button3 = new Button("Button 3");
        Button button4 = new Button("Button 4");

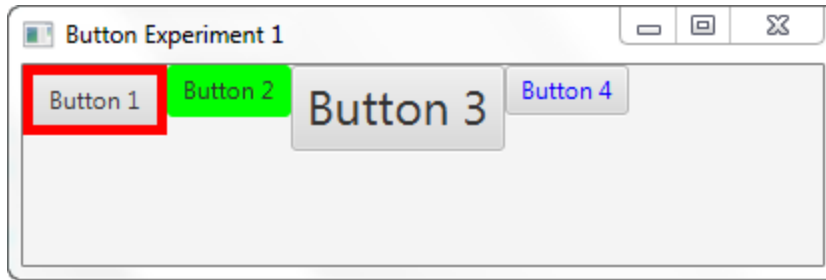
        button1.setStyle("-fx-border-color: #ff0000; -fx-border-width: 5px;");
        button2.setStyle("-fx-background-color: #00ff00");
        button3.setStyle("-fx-font-size: 2em;");
        button4.setStyle("-fx-text-fill: #0000ff");

        HBox hbox = new HBox(button1, button2, button3, button4);

        Scene scene = new Scene(hbox, 400, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Here is a screenshot of the 4 JavaFX buttons with their CSS styling:



The first button has both the `-fx-border-width` and `-fx-border-color` CSS properties set. This results in a 5 pixel wide red border for the button.

The second button has the `-fx-background-color` CSS property set. This results in a green background color for the button.

The third button has the `-fx-font-size` CSS property set. This results in a button with a text that is 2 times as big as normal.

The fourth button has the `-fx-text-fill` CSS property set. This results in a button with a blue text color.

You can combine the CSS styles for a JavaFX button simply by setting multiple CSS properties on it, like the first button in the example above has.

Next: [JavaFX MenuButton](#)

# JavaFX MenuButton

- [Creating a MenuButton](#)
- [Adding a MenuButton to the Scene Graph](#)
- [Responding to Menu Item Selection](#)

Jakob Jenkov

Last update: 2016-05-15

The JavaFX MenuButton control can show a list of menu options which the user can choose from. The JavaFX MenuButton can show or hide the menu items. The menu items are usually shown when a little arrow button is clicked in the MenuButton. The JavaFX MenuButton control is represented by the class `javafx.scene.control.MenuButton`.

## Creating a MenuButton

You create a JavaFX MenuButton by creating an instance of the MenuButton class. Here is a JavaFX MenuButton instantiation example:

```
MenuItem menuItem1 = new MenuItem("Option 1");
MenuItem menuItem2 = new MenuItem("Option 2");
MenuItem menuItem3 = new MenuItem("Option 3");

MenuButton menuButton = new MenuButton("Options", null, menuItem1, menuItem2,
menuItem3);
```

First 3 MenuItem instances are created, each with a different text. Then a MenuButton instance is created, passing a button text, a graphic icon (null) and the 3 MenuItem instances as parameter to the MenuButton constructor.

The second MenuButton constructor parameter is a Node which is used as a graphic icon which is shown next to the MenuButton text. You could use an [ImageView](#) control to display an image next to the MenuButton text. Just create an ImageView instance and pass a reference to that to the MenuButton constructor, instead of null. Here is an example:

```
MenuItem menuItem1 = new MenuItem("Option 1");
MenuItem menuItem2 = new MenuItem("Option 2");
MenuItem menuItem3 = new MenuItem("Option 3");

FileInputStream input = new FileInputStream("resources/images/iconmonstr-
menu-5-32.png");
Image image = new Image(input);
ImageView imageView = new ImageView(image);

MenuBar menuBar = new MenuBar("Options", imageView, menuItem1,
menuItem2, menuItem3);
```

## Adding a MenuButton to the Scene Graph

To make a `MenuBar` visible you must add it to the JavaFX scene graph. This means adding it to a `Scene`, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `MenuBar` to the scene graph:

```
package com.jenkov.javaafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.MenuButton;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

import java.io.FileInputStream;

public class MenuButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ImageView Experiment 1");

        MenuItem menuItem1 = new MenuItem("Option 1");
        MenuItem menuItem2 = new MenuItem("Option 2");
        MenuItem menuItem3 = new MenuItem("Option 3");

        MenuButton menuButton = new MenuButton("Options", null, menuItem1,
menuItem2, menuItem3);

        HBox hbox = new HBox(menuButton);
```

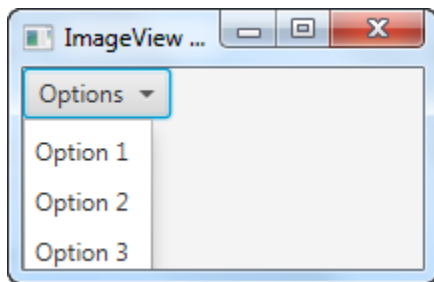
```

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

Here is how the application resulting from the above example looks:



Here is how the same example would look with a graphic icon added to the MenuButton:

```

package com.jenkov.javaafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.MenuButton;
import javafx.scene.control.MenuItem;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

import java.io.FileInputStream;

public class MenuButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ImageView Experiment 1");

        MenuItem menuItem1 = new MenuItem("Option 1");
        MenuItem menuItem2 = new MenuItem("Option 2");
        MenuItem menuItem3 = new MenuItem("Option 3");
    }
}

```

```

        FileInputStream input = new
FileInputStream("resources/images/iconmonstr-menu-5-32.png");
        Image image = new Image(input);
        ImageView imageView = new ImageView(image);

        MenuButton menuButton = new MenuButton("Options", imageView,
menuItem1, menuItem2, menuItem3);

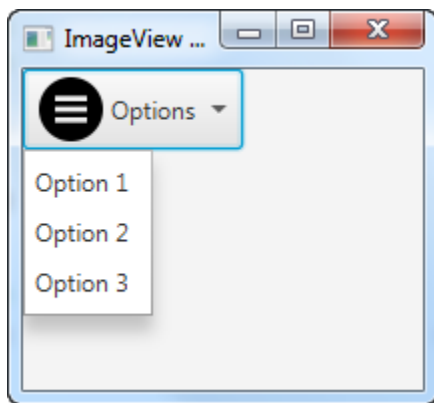
        HBox hbox = new HBox(menuButton);

        Scene scene = new Scene(hbox, 200, 160);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

Here is how the application resulting from the above example looks:



## Responding to Menu Item Selection

To respond to when a user selects a menu item, add an "on action" event listener to the corresponding `MenuItem` object. Here is an example showing you how to add an action event listener to a `MenuItem` object:

```

MenuItem menuItem3 = new MenuItem("Option 3");

menuItem3.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {

```

```
        System.out.println("Option 3 selected");
    }
});
```

You can also use a [Java Lambda expression](#) instead of an anonymous implementation of the `EventHandler` interface. Here is how that looks:

```
menuItem3.setOnAction(event -> {
    System.out.println("Option 3 selected via Lambda");
});
```

Next: [JavaFX SplitMenuButton](#)

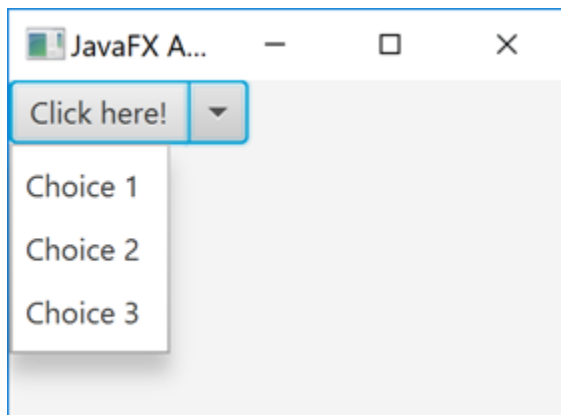
# JavaFX SplitMenuButton

- [Create SplitMenuButton](#)
- [Set SplitMenuButton Text](#)
- [Set SplitMenuButton Menu Items](#)
- [Respond to Menu Item Selection](#)
- [Respond to Button Click](#)
- [SplitMenuButton vs. MenuButton, ChoiceBox and ComboBox](#)

Jakob Jenkov

Last update: 2019-07-10

The *JavaFX SplitMenuButton* control can show a list of menu options which the user can choose from, as well as a button which the user can click on when a menu option has been chosen. The *JavaFX SplitMenuButton* can show or hide the menu items. The menu items are usually shown when a little arrow button is clicked in the *SplitMenuButton*. The *JavaFX SplitMenuButton* control is represented by the class `javafx.scene.control.SplitMenuButton`. Here is a screenshot of a *JavaFX SplitMenuButton*:



## Create SplitMenuButton

Before you can use the *JavaFX SplitMenuButton* you must create an instance of it. Here is an example of creating a *JavaFX SplitMenuButton*:



```
SplitMenuButton splitMenuButton = new SplitMenuButton();
```

## Set SplitMenuButton Text

You can set the `SplitMenuButton`'s button text via its `setText()` method. Here is an example of setting the button text of a JavaFX `SplitMenuButton`:

```
splitMenuButton.setText("Click here!");
```

## Set SplitMenuButton Menu Items

You can set the menu items to display in the menu part of a JavaFX `SplitMenuButton` via its `MenuItem` collection returned by `getItems()`. Each menu item is represented by a `MenuItem` object. Here is an example of setting three menu items on a JavaFX `SplitMenuButton`:

```
MenuItem choice1 = new MenuItem("Choice 1");
MenuItem choice2 = new MenuItem("Choice 2");
MenuItem choice3 = new MenuItem("Choice 3");

button.getItems().addAll(choice1, choice2, choice3);
```

## Respond to Menu Item Selection

The JavaFX `SplitMenuButton` works similarly to the [JavaFX MenuButton](#) when it comes to responding to selected menu items. To respond to selection of a menu item in a JavaFX *SplitMenuButton* you must set an action listener on each `MenuItem` added to the `SplitMenuButton`. Here is an example of responding to menu item selection in a JavaFX `SplitMenuButton` by setting action listeners on its `MenuItem` instances:

```
MenuItem choice1 = new MenuItem("Choice 1");
MenuItem choice2 = new MenuItem("Choice 2");
MenuItem choice3 = new MenuItem("Choice 3");
```

```
choice1.setOnAction((e)-> {  
    System.out.println("Choice 1 selected");  
});  
choice2.setOnAction((e)-> {  
    System.out.println("Choice 2 selected");  
});  
choice3.setOnAction((e)-> {  
    System.out.println("Choice 3 selected");  
});
```

In this example the action listeners simply print a text to the console. In a real application you would probably want to store information about what action was selected, or take some other action, rather than just printing a text out to the console.

## Respond to Button Click

You can respond to JavaFX `SplitMenuButton` button clicks by setting an action listener on it. Here is an example of setting an action listener on a JavaFX `SplitMenuButton`:

```
splitMenuButton.setOnAction((e) -> {  
    System.out.println("SplitMenuButton clicked!");  
});
```

This example uses a [Java Lambda Expression](#) as action listener. When the button is clicked, the text `SplitMenuButton clicked!` will be printed to the console.

## SplitMenuButton vs. MenuButton, ChoiceBox and ComboBox

You might be wondering what the difference is between a JavaFX `SplitMenuButton` and a [JavaFX MenuButton](#), [JavaFX ChoiceBox](#) and a [JavaFX ComboBox](#). I will try to explain that below.

The `SplitMenuButton` and `MenuButton` controls are *buttons*. That means, that they are intended for your application to respond to clicks on either one of the menu items, or in the case of the `SplitMenuButton` - the primary button or one of the menu items. Use one of these two controls when you want an immediate action to follow when the user clicks / selects a

menu item. Use the `SplitMenuButton` when one of the choices is done more often than the rest. Use the button part for the most selected choice, and the menu items for the less often selected choices.

The `ChoiceBox` and `ComboBox` merely store internally what choices the user has made among their menu items. They are not designed for immediate action upon menu item selection. Use these controls in forms where the user has to make several choices before finally clicking either an "OK" or "Cancel" button. When one of these buttons are clicked, you can read what menu item is chosen from the `ChoiceBox` or `ComboBox`.

Next: [JavaFX ToggleButton](#)

# JavaFX ToggleButton

- [Creating a ToggleButton](#)
- [Adding a ToggleButton to the Scene Graph](#)
- [Reading Selected State](#)
- [ToggleGroup](#)
- [Reading Selected State of a ToggleGroup](#)

Jakob Jenkov

Last update: 2016-05-15

A JavaFX ToggleButton is a button that can be selected or not selected. Like a button that stays in when you press it, and when you press it the next time it comes out again. Toggled - not toggled. The JavaFX ToggleButton is represented by the `class javafx.scene.control.ToggleButton`.

## Creating a ToggleButton

You create a JavaFX ToggleButton by creating an instance of the `ToggleButton` class. Here is an example of creating a JavaFX ToggleButton instance:

```
ToggleButton toggleButton1 = new ToggleButton("Left");
```

This example creates a `ToggleButton` with the text `Left` on.

## Adding a ToggleButton to the Scene Graph

To make a `ToggleButton` visible you must add it to the JavaFX scene graph. This means adding it to a `Scene`, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `ToggleButton` to the scene graph:

```

package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ToggleButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        ToggleButton toggleButton1 = new ToggleButton("Left");

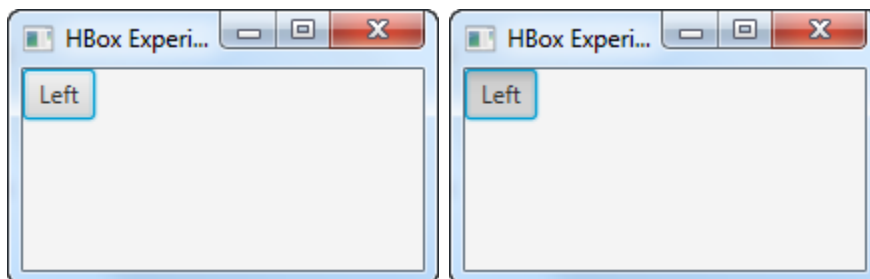
        HBox hbox = new HBox(toggleButton1);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The application resulting from running the above example code is illustrated in the following two screenshots. The first screenshot shows a `ToggleButton` which is not pressed, and the second screenshot shows the same `ToggleButton` pressed (selected, activated etc.):



## Reading Selected State

The `ToggleButton` class has a method named `isSelected` which lets you determine if the `ToggleButton` is selected (pressed) or not. The `isSelected()` method returns a boolean with the value `true` if the `ToggleButton` is selected, and `false` if not. Here is an example:

```
boolean isSelected = toggleButton1.isSelected();
```

## ToggleGroup

You can group JavaFX `ToggleButton` instances into a `ToggleGroup`. A `ToggleGroup` allows at most one `ToggleButton` to be toggled (pressed) at any time. The `ToggleButton` instances in a `ToggleGroup` thus functions similarly to radio buttons.

Here is a JavaFX `ToggleGroup` example:

```
ToggleButton toggleButton1 = new ToggleButton("Left");
ToggleButton toggleButton2 = new ToggleButton("Right");
ToggleButton toggleButton3 = new ToggleButton("Up");
ToggleButton toggleButton4 = new ToggleButton("Down");

ToggleGroup toggleGroup = new ToggleGroup();

toggleButton1.setToggleGroup(toggleGroup);
toggleButton2.setToggleGroup(toggleGroup);
toggleButton3.setToggleGroup(toggleGroup);
toggleButton4.setToggleGroup(toggleGroup);
```

Here is a full example that adds the 4 `ToggleButton` instances to a `ToggleGroup`, and adds them to the scene graph too:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ToggleButtonExperiments extends Application {
```

```

@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("HBox Experiment 1");

    ToggleButton toggleButton1 = new ToggleButton("Left");
    ToggleButton toggleButton2 = new ToggleButton("Right");
    ToggleButton toggleButton3 = new ToggleButton("Up");
    ToggleButton toggleButton4 = new ToggleButton("Down");

    ToggleGroup toggleGroup = new ToggleGroup();

    toggleButton1.setToggleGroup(toggleGroup);
    toggleButton2.setToggleGroup(toggleGroup);
    toggleButton3.setToggleGroup(toggleGroup);
    toggleButton4.setToggleGroup(toggleGroup);

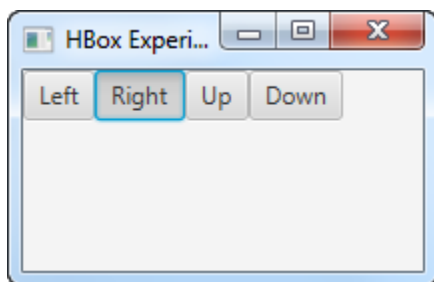
    HBox hbox = new HBox(toggleButton1, toggleButton2, toggleButton3,
toggleButton4);

    Scene scene = new Scene(hbox, 200, 100);
    primaryStage.setScene(scene);
    primaryStage.show();
}

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

```

The resulting applications looks like this:



## Reading Selected State of a ToggleGroup

You can read which `ToggleButton` of a `ToggleGroup` is selected (pressed) using the `getSelectedToggle()` method, like this:

```
ToggleButton selectedToggleButton =  
    (ToggleButton) toggleGroup.getSelectedToggle();
```

If no `ToggleButton` is selected  
the `getSelectedToggle()` method returns `null`.

Next: [JavaFX RadioButton](#)



# JavaFX RadioButton

- [Creating a RadioButton](#)
- [Adding a RadioButton to the Scene Graph](#)
- [Reading Selected State](#)
- [ToggleGroup](#)
- [Reading Selected State of a ToggleGroup](#)

Jakob Jenkov

Last update: 2016-05-15

A JavaFX `RadioButton` is a button that can be selected or not selected. The `RadioButton` is very similar to the [JavaFX ToggleButton](#), but with the difference that a `RadioButton` cannot be "unselected" once selected. If `RadioButtons` are part of a `ToggleGroup` then once a `RadioButton` has been selected for the first time, there must be one `RadioButton` selected in the `ToggleGroup`.

The JavaFX `RadioButton` is represented by the class `javafx.scene.control.RadioButton`. The `RadioButton` class is a subclass of the `ToggleButton` class.

## Creating a RadioButton

You create a JavaFX `RadioButton` using its constructor. Here is a JavaFX `RadioButton` instantiation example:

```
RadioButton radioButton1 = new RadioButton("Left");
```

The String passed as parameter to the `RadioButton` constructor is displayed next to the `RadioButton`.

## Adding a RadioButton to the Scene Graph

To make a `RadioButton` visible you must add it to the scene graph of your JavaFX application. This means adding the `RadioButton` to a `Scene`, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `RadioButton` to the scene graph:

```
package com.jenkov.javaafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class RadioButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

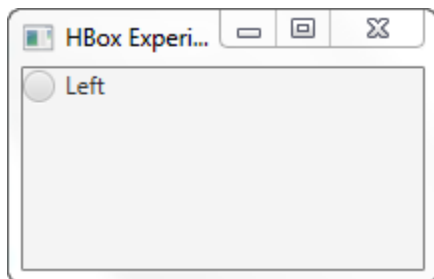
        RadioButton radioButton1 = new RadioButton("Left");

        HBox hbox = new HBox(radioButton1);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from running this example looks like this:



## Reading Selected State

The `RadioButton` class has a method named `isSelected` which lets you determine if the `RadioButton` is selected or not.

The `isSelected()` method returns a boolean with the value `true` if the `RadioButton` is selected, and `false` if not. Here is an example:

```
boolean isSelected = radioButton1.isSelected();
```

## ToggleGroup

You can group JavaFX `RadioButton` instances into a `ToggleGroup`. A `ToggleGroup` allows at most one `RadioButton` to be selected at any time.

Here is a JavaFX `ToggleGroup` example:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class RadioButtonExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        RadioButton radioButton1 = new RadioButton("Left");
        RadioButton radioButton2 = new RadioButton("Right");
        RadioButton radioButton3 = new RadioButton("Up");
        RadioButton radioButton4 = new RadioButton("Down");

        ToggleGroup radioGroup = new ToggleGroup();

        radioButton1.setToggleGroup(radioGroup);
        radioButton2.setToggleGroup(radioGroup);
        radioButton3.setToggleGroup(radioGroup);
        radioButton4.setToggleGroup(radioGroup);
    }
}
```

```

        HBox hbox = new HBox(radioButton1, radioButton2, radioButton3,
radioButton4);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();

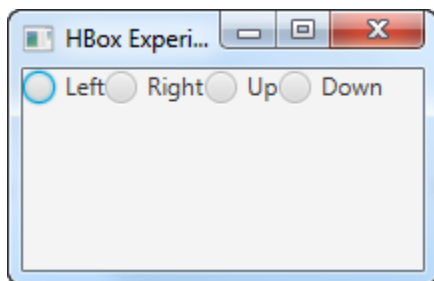
    }

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

}

```

The application resulting from running this example looks like this:



## Reading Selected State of a ToggleGroup

You can read which `RadioButton` of a `ToggleGroup` is selected using the `getSelectedToggle()` method, like this:

```

RadioButton selectedRadioButton =
    (RadioButton) toggleGroup.getSelectedToggle();

```

If no `RadioButton` is selected the `getSelectedToggle()` method returns `null`.

Next: [JavaFX CheckBox](#)

# JavaFX CheckBox

- [Creating a CheckBox](#)
- [Adding a CheckBox to the Scene Graph](#)
- [Reading Selected State](#)
- [Allowing Indeterminate State](#)
- [Reading Indeterminate State](#)

Jakob Jenkov

Last update: 2016-05-15

A JavaFX `CheckBox` is a button which can be in three different states: Selected, not selected and unknown (indeterminate). The JavaFX `CheckBox` control is represented by the class `javafx.scene.control.CheckBox`.

## Creating a CheckBox

You create a JavaFX `CheckBox` control via the `CheckBox` constructor. Here is a JavaFX `CheckBox` instantiation example:

```
CheckBox checkBox1 = new CheckBox("Green");
```

The String passed to the `CheckBox` constructor is displayed next to the `CheckBox` control.

## Adding a CheckBox to the Scene Graph

To make a JavaFX `CheckBox` control visible you must add it to the scene graph of your JavaFX application. That means adding the `CheckBox` control to a `Scene` object, or to some layout component which is itself added to a `Scene` object.

Here is an example showing how to add a `CheckBox` to the scene graph:

```
package com.jenkov.javafx.controls;
```

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class CheckBoxExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("CheckBox Experiment 1");

        CheckBox checkBox1 = new CheckBox("Green");

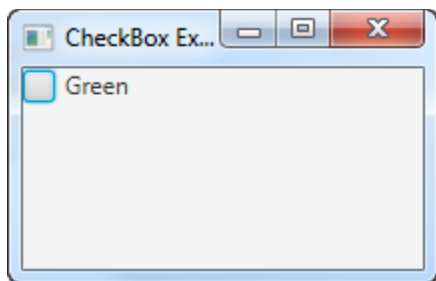
        HBox hbox = new HBox(checkBox1);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The application resulting from running this code looks like this:



## Reading Selected State

You can read the selected state of a `CheckBox` via its method `isSelected()`. Here is an example of how calling `isSelected()` looks:

```
boolean isSelected = checkBox1.isSelected();
```

## Allowing Indeterminate State

As mentioned earlier a JavaFX `CheckBox` can be in an *indeterminate state* which means that it is neither selected, nor not selected. The user simply has not interacted with the `CheckBox` yet.

By default a `CheckBox` is not allowed to be in the indeterminate state. You can set if a `CheckBox` is allowed to be in an indeterminate state using the method `setAllowIndeterminate()`. Here is an example of allowing the indeterminate state for a `CheckBox`:

```
checkBox1.setAllowIndeterminate(true);
```

## Reading Indeterminate State

You can read if a `CheckBox` is in the indeterminate state via its `isIndeterminate()` method. Here is an example of checking if a `CheckBox` is in the indeterminate state:

```
boolean isIndeterminate = checkBox1.isIndeterminate();
```

Note, that if a `CheckBox` is not in the indeterminate state, it is either selected or not selected, which can be seen via its `isSelected()` method shown earlier.

Next: [JavaFX ChoiceBox](#)

# JavaFX ChoiceBox

- [Creating a ChoiceBox](#)
- [Adding Choices to a ChoiceBox](#)
- [Adding a ChoiceBox to the Scene Graph](#)
- [Reading the Selected Value](#)

Jakob Jenkov

Last update: 2016-05-17

The JavaFX ChoiceBox control enables users to choose an option from a predefined list of choices. The JavaFX ChoiceBox control is represented by the class `javafx.scene.control.ChoiceBox`. This JavaFX ChoiceBox tutorial will explain how to use the ChoiceBox class.

## Creating a ChoiceBox

You create a ChoiceBox simply by creating a new instance of the ChoiceBox class. Here is a JavaFX ChoiceBox instantiation example:

```
ChoiceBox choiceBox = new ChoiceBox();
```

## Adding Choices to a ChoiceBox

You can add choices to a ChoiceBox by obtaining its item collection and add items to it. Here is an example that adds choices to a JavaFX ChoiceBox:

```
choiceBox.getItems().add("Choice 1");  
choiceBox.getItems().add("Choice 2");  
choiceBox.getItems().add("Choice 3");
```

## Adding a ChoiceBox to the Scene Graph



To make a `ChoiceBox` visible you must add it to the scene graph. This means that you must add the `ChoiceBox` to a `Scene` object or to some layout component which is then attached to the `Scene` object.

Here is an example showing how to add a JavaFX `ChoiceBox` to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ChoiceBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ChoiceBoxExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ChoiceBox Experiment 1");

        ChoiceBox choiceBox = new ChoiceBox();

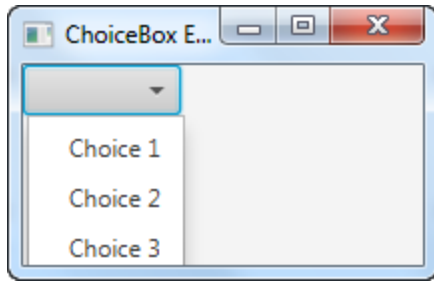
        choiceBox.getItems().add("Choice 1");
        choiceBox.getItems().add("Choice 2");
        choiceBox.getItems().add("Choice 3");

        HBox hbox = new HBox(choiceBox);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from running this example would look similar to this:



## Reading the Selected Value

You can read the selected value of a `ChoiceBox` via its `getValue()` method. If no choice is selected, the `getValue()` method returns `null`. Here is an example of calling `getValue()`:

```
String value = (String) choiceBox.getValue();
```

Next: [JavaFX ComboBox](#)

# JavaFX ComboBox

- [Creating a ComboBox](#)
- [Adding Choices to a ComboBox](#)
- [Adding a ComboBox to the Scene Graph](#)
- [Reading the Selected Value](#)
- [Making the ComboBox Editable](#)

Jakob Jenkov

Last update: 2016-05-17

The JavaFX ComboBox control enables users to choose an option from a predefined list of choices, or type in another value if none of the predefined choices matches what the user want to select. The JavaFX ComboBox control is represented by the class `javafx.scene.control.ComboBox`. This JavaFX ComboBox tutorial will explain how to use the `ComboBox` class.

## Creating a ComboBox

You create a `ComboBox` simply by creating a new instance of the `ComboBox` class. Here is a JavaFX `ComboBox` instantiation example:

```
ComboBox comboBox = new ComboBox();
```

## Adding Choices to a ComboBox

You can add choices to a `ComboBox` by obtaining its item collection and add items to it. Here is an example that adds choices to a JavaFX `ComboBox` :

```
comboBox.getItems().add("Choice 1");  
comboBox.getItems().add("Choice 2");  
comboBox.getItems().add("Choice 3");
```

## Adding a ComboBox to the Scene Graph

To make a `ComboBox` visible you must add it to the scene graph. This means that you must add the `ComboBox` to a `Scene` object or to some layout component which is then attached to the `Scene` object.

Here is an example showing how to add a JavaFX `ComboBox` to the scene graph:

```
package com.jenkov.javaafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ComboBoxExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ComboBox Experiment 1");

        ComboBox comboBox = new ComboBox();

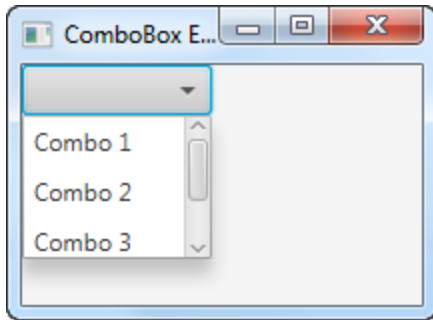
        comboBox.getItems().add("Choice 1");
        comboBox.getItems().add("Choice 2");
        comboBox.getItems().add("Choice 3");

        HBox hbox = new HBox(comboBox);

        Scene scene = new Scene(hbox, 200, 120);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from running this example would look similar to this:



## Reading the Selected Value

You can read the selected value of a `ComboBox` via its `getValue()` method. If no choice is selected, the `getValue()` method returns `null`. Here is an example of calling `getValue()`:

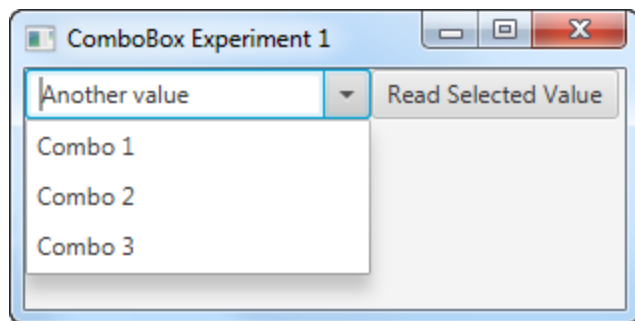
```
String value = (String) comboBox.getValue();
```

## Making the ComboBox Editable

A `ComboBox` is not editable by default. That means, that by default the user cannot enter anything themselves, but only choose from the predefined list of options. To make a `ComboBox` editable you must call the `setEditable()` method of the `ComboBox`. Here is an example making a JavaFX `ComboBox` editable:

```
comboBox.setEditable(true);
```

Once the `ComboBox` is editable the user can type in values into the `ComboBox`. The entered value is also read via the `getValue()` method as explained earlier. The following screenshot shows a JavaFX `ComboBox` which is editable, and with a custom value entered:



Next: [JavaFX ListView](#)

# JavaFX ListView

- [Creating a ListView](#)
- [Adding Items to a ListView](#)
- [Adding a ListView to the Scene Graph](#)
- [Reading the Selected Value](#)
- [Allowing Multiple Items to be Selected](#)

Jakob Jenkov

Last update: 2016-05-18

The JavaFX `ListView` control enables users to choose one or more options from a predefined list of choices. The JavaFX `ListView` control is represented by the class `javafx.scene.control.ListView`. This JavaFX `ListView` tutorial will explain how to use the `ListView` class.

## Creating a ListView

You create a `ListView` simply by creating a new instance of the `ListView` class. Here is a JavaFX `ListView` instantiation example:

```
ListView listView = new ListView();
```

## Adding Items to a ListView

You can add items (options) to a `ListView` by obtaining its item collection and add items to it. Here is an example that adds items to a JavaFX `ListView`:

```
listView.getItems().add("Item 1");  
listView.getItems().add("Item 2");  
listView.getItems().add("Item 3");
```

## Adding a ListView to the Scene Graph

To make a `ListView` visible you must add it to the scene graph. This means that you must add the `ListView` to a `Scene` object or to some layout component which is then attached to the `Scene` object.

Here is an example showing how to add a JavaFX `ListView` to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ListViewExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ListView Experiment 1");

        ListView listView = new ListView();

        listView.getItems().add("Item 1");
        listView.getItems().add("Item 2");
        listView.getItems().add("Item 3");

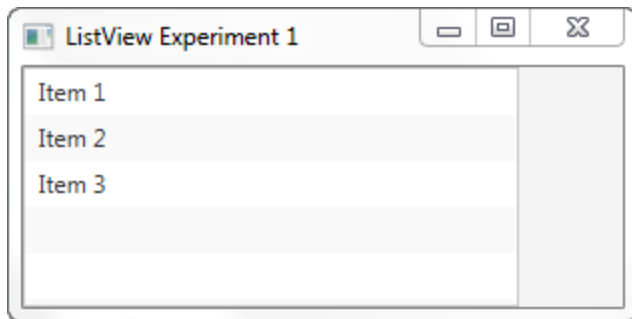
        HBox hbox = new HBox(listView);

        Scene scene = new Scene(hbox, 300, 120);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from running this example would look similar to this screenshot:





Notice how the `ListView` shows multiple options by default. You can set a height and width for a `ListView`, but you cannot set explicitly how many items should be visible. The height determines that based on the height of each item displayed.

If there are more items in the `ListView` than can fit into its visible area, the `ListView` will add scroll bars so the user can scroll up and down over the items.

## Reading the Selected Value

You can read the selected indexes of a `ListView` via its `SelectionModel`. Here is an example showing how to read the selected indexes of a JavaFX `ListView`:

```
ObservableList selectedIndices =  
    listView.getSelectionModel().getSelectedIndices();
```

The `ObservableList` will contain `Integer` objects representing the indexes of the selected items in the `ListView`.

Here is a full JavaFX example with a button added which reads the selected items of the `ListView` when clicked:

```
package com.jenkov.javafx.controls;  
  
import javafx.application.Application;  
import javafx.collections.ObservableList;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.ListView;  
import javafx.scene.control.SelectionMode;
```

```

import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ListViewExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ListView Experiment 1");

        ListView listView = new ListView();

        listView.getItems().add("Item 1");
        listView.getItems().add("Item 2");
        listView.getItems().add("Item 3");

        Button button = new Button("Read Selected Value");

        button.setOnAction(event -> {
            ObservableList selectedIndices =
listView.getSelectionModel().getSelectedIndices();

            for(Object o : selectedIndices){
                System.out.println("o = " + o + " (" + o.getClass() + ")");
            }
        });

        VBox vBox = new VBox(listView, button);

        Scene scene = new Scene(vBox, 300, 120);
        primaryStage.setScene(scene);
        primaryStage.show();

    }

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

```

## Allowing Multiple Items to be Selected

To allow multiple items in the `ListView` to be selected you need to set the corresponding selection mode on the `ListView` selection model. Here is an example of setting the selection mode on the JavaFX `ListView`:

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

Once you have set the `SelectionMode.MULTIPLE` on the `ListView` selection model, the user can select multiple items in the `ListView` by holding down **SHIFT** or **CTRL** when selecting additional items after the first selected item.

Here is a full JavaFX example that shows how to set a `ListView` into multiple selection mode, including a button which when clicked will write out the indices of the selected items in the `ListView`:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ListViewExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("ListView Experiment 1");

        ListView listView = new ListView();

        listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);

        listView.getItems().add("Item 1");
        listView.getItems().add("Item 2");
        listView.getItems().add("Item 3");

        Button button = new Button("Read Selected Value");

        button.setOnAction(event -> {
            ObservableList selectedIndices =
listView.getSelectionModel().getSelectedIndices();

            for(Object o : selectedIndices){
                System.out.println("o = " + o + " (" + o.getClass() + ")");
            }
        });
    }
}
```

```
        VBox vBox = new VBox(listView, button);

        Scene scene = new Scene(vBox, 300, 120);
        primaryStage.setScene(scene);
        primaryStage.show();

    }

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

Next: [JavaFX DatePicker](#)

# JavaFX DatePicker

- [Creating a DatePicker](#)
- [Adding a DatePicker to the Scene Graph](#)
- [Reading the Selected Date](#)

Jakob Jenkov

Last update: 2016-05-18

A JavaFX DatePicker control enables the user to enter a date or choose a date from a wizard-like popup dialog. The popup dialog shows only valid dates, so this is an easier way for users to choose a date and ensure that both the date and date format entered in the date picker text field is valid. The JavaFX DatePicker is represented by the class `javafx.scene.control.DatePicker`.

The `DatePicker` is a subclass of the `ComboBox` class, and thus shares some similarities with this class.

## Creating a DatePicker

You create a `DatePicker` control via the constructor of the `DatePicker` class. Here is a JavaFX `DatePicker` instantiation example:

```
DatePicker datePicker = new DatePicker();
```

## Adding a DatePicker to the Scene Graph

To make a `DatePicker` visible it must be added to the JavaFX scene graph. This means adding it to a `Scene` object, or to a layout component which is added to a `Scene` object.

Here is an example showing how to add a JavaFX `DatePicker` to the scene graph:

```
package com.jenkov.javafx.controls;
```

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.DatePicker;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class DatePickerExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Button Experiment 1");

        DatePicker datePicker = new DatePicker();

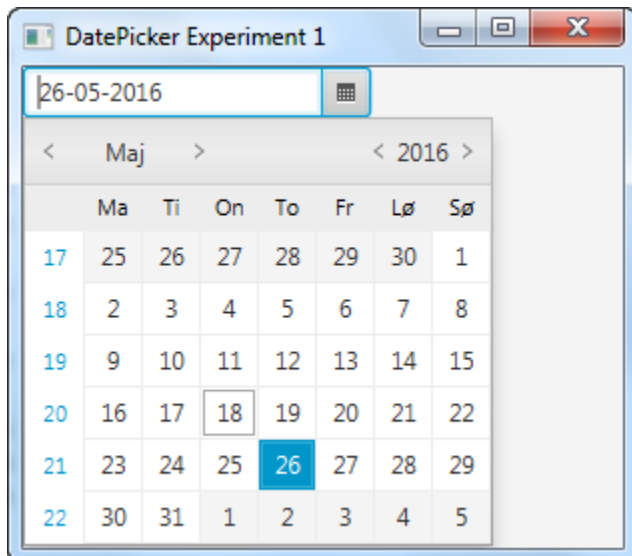
        HBox hbox = new HBox(datePicker);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The application resulting from running this example would look similar to this:



## Reading the Selected Date

Reading the date selected in the `DatePicker` can be done using its `getValue()` method. Here is an example of reading the selected date from a `DatePicker`:

```
LocalDate value = datePicker.getValue();
```

The `getValue()` returns a [LocalDate](#) object representing the date selected in the `DatePicker`.

Here is a full example with a button added to extract the selected date in the `DatePicker` when the button is clicked:

```
package com.jenkov.javafx.controls;

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

import java.time.LocalDate;

public class DatePickerExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("DatePicker Experiment 1");

        DatePicker datePicker = new DatePicker();

        Button button = new Button("Read Date");

        button.setOnAction(action -> {
            LocalDate value = datePicker.getValue();
        });

        HBox hbox = new HBox(datePicker);

        Scene scene = new Scene(hbox, 300, 240);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```
    }  
}
```

Next: [JavaFX ColorPicker](#)



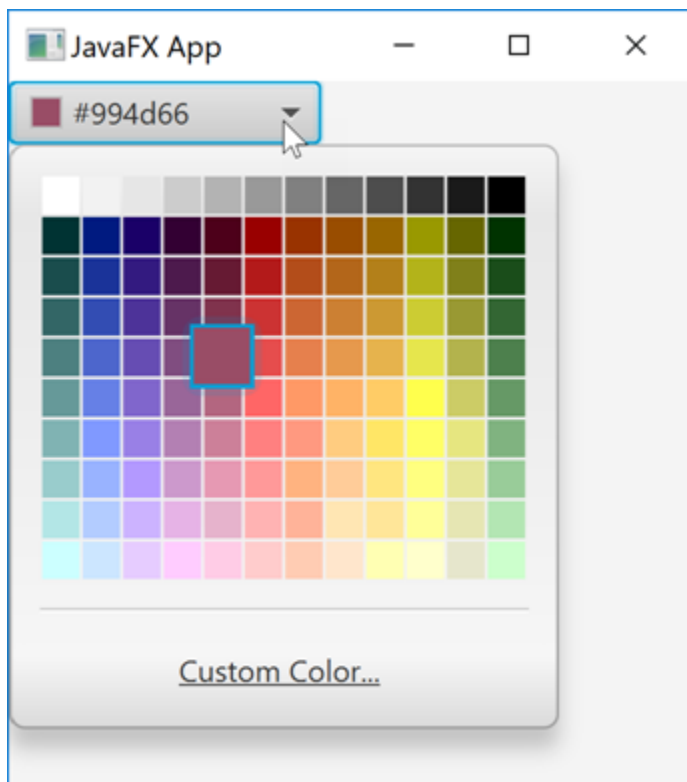
# JavaFX ColorPicker

- [Full ColorPicker Example](#)
- [Create a ColorPicker](#)
- [Get Chosen Color](#)

Jakob Jenkov

Last update: 2019-07-26

The *JavaFX ColorPicker* control enables the user to choose a color in a popup dialog. The chosen color can later be read from the ColorPicker by your JavaFX application. The JavaFX ColorPicker control is represented by the class `javafx.scene.control.ColorPicker`. Here is a screenshot of an opened JavaFX ColorPicker:



## Full ColorPicker Example

Here is a full JavaFX ColorPicker example so you can see what the code looks like:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

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

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

        ColorPicker colorPicker = new ColorPicker();

        Color value = colorPicker.getValue();

        VBox vBox = new VBox(colorPicker);
        //HBox hBox = new HBox(button1, button2);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

## Create a ColorPicker

In order to use a JavaFX `ColorPicker` you must first create an instance of the `ColorPicker` class. Here is an example of creating a JavaFX `ColorPicker`:

```
ColorPicker colorPicker = new ColorPicker();
```

## Get Chosen Color

To read the color chosen in a JavaFX `ColorPicker` you call its `getValue()` method. Here is an example of getting the chosen color in a JavaFX `ColorPicker`:

```
Color value = colorPicker.getValue();
```

Next: [JavaFX TextField](#)

# JavaFX TextField

- [Creating a TextField](#)
- [Adding a TextField to the Scene Graph](#)
- [Getting the Text of a TextField](#)
- [Setting the Text of a TextField](#)

Jakob Jenkov

Last update: 2016-05-13

A JavaFX TextField control enables users of a JavaFX application to enter text which can then be read by the application. The JavaFX TextField control is represented by the class `javafx.scene.control.TextField`.

## Creating a TextField

You create a TextField control by creating an instance of the `TextField` class. Here is a JavaFX `TextField` instantiation example:

```
TextField textField = new TextField();
```

## Adding a TextField to the Scene Graph

For a JavaFX `TextField` to be visible the `TextField` object must be added to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `TextField` to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
```

```

public class TextFieldExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        TextField textField = new TextField();

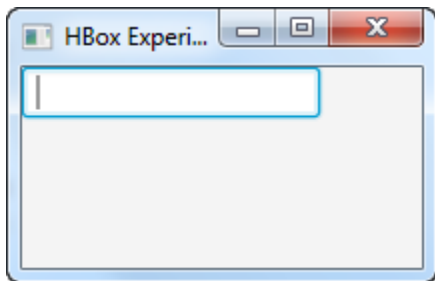
        HBox hbox = new HBox(textField);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The result of running the above JavaFX `TextField` example is an application that looks like this:



## Getting the Text of a TextField

You can get the text entered into a `TextField` using its `getText()` method which returns a `String`. Here is a full example that shows a `TextField` and a `Button` and which reads the text entered into the `TextField` when the button is clicked:

```

package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

```

```

import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TextFieldExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        TextField textField = new TextField();

        Button button = new Button("Click to get text");

        button.setOnAction(action -> {
            System.out.println(textField.getText());
        });

        HBox hbox = new HBox(textField, button);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

## Setting the Text of a TextField

You can set the text of a `TextField` using its `setText()` method. This is often useful when you need to set the initial value for a text field that is part of a form. For instance, editing an existing object or record. Here is a simple example of setting the text of a JavaFX `TextField`:

```

textField.setText("Initial value");

```

Next: [JavaFX Slider](#)

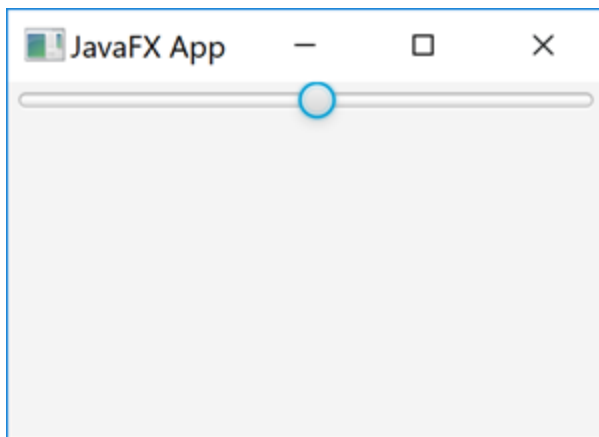
# JavaFX Slider

- [JavaFX Slider Example](#)
- [Create a Slider](#)
- [Reading Slider Value](#)
- [Major Tick Unit](#)
- [Minor Tick Count](#)
- [Snap Handle to Ticks](#)
- [Show Tick Marks](#)
- [Show Tick Labels](#)

Jakob Jenkov

Last update: 2019-05-30

The *JavaFX Slider* control provides a way for the user to select a value within a given interval by sliding a handle to the desired point representing the desired value. The *JavaFX Slider* is represented by the JavaFX class `javafx.scene.control.Slider`. Here is a screenshot of how a JavaFX Slider looks:



## JavaFX Slider Example

Here is a full JavaFX `Slider` code example:

```
import javafx.application.Application;
import javafx.scene.Scene;
```

```

import javafx.scene.control.Slider;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

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

        Slider slider = new Slider(0, 100, 0);

        VBox vBox = new VBox(slider);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

## Create a Slider

To use a JavaFX `Slider` you must first create an instance of the `Slider` class. Here is an example of creating a JavaFX `Slider` instance:

```
Slider slider = new Slider(0, 100, 0);
```

The `Slider` constructor used above takes three parameters: The min value, the max value and the initial value. The min value is the value sliding the handle all the way to the left represents. This is the beginning of the interval the user can select a value in. The max value is the value sliding the handle all the way to the right represents. This the end of the interval the user can select a value in. The initial value is the value that the handle should be located at, when presented to the user at first.

## Reading Slider Value



You can read the value of a `Slider` as selected by the user via the `getValue()` method. Here is an example of reading the selected value of a JavaFX `Slider`:

```
double value = slider.getValue();
```

## Major Tick Unit

You can set the major tick unit of a JavaFX `Slider` control. The major tick unit is how many units the value changes every time the user moves the handle of the `Slider` one tick. Here is an example that sets the major tick unit of a JavaFX `Slider` to 8:

```
Slider slider = new Slider(0, 100, 0);  
slider.setMajorTickUnit(8.0);
```

This `Slider` will have its value change with 8.0 up or down whenever the handle in the `Slider` is moved.

## Minor Tick Count

You can set the minor tick count of a JavaFX `Slider` via the `setMinorTickCount()` method. The minor tick count specifies how many minor ticks there are between two of the major ticks. Here is an example that sets the minor tick count to 2:

```
Slider slider = new Slider(0, 100, 0);  
slider.setMajorTickUnit(8.0);  
slider.setMinorTickCount(3);
```

The `Slider` configured here has 8.0 value units between each major tick, and in between each of these major ticks it has 3 minor ticks.

## Snap Handle to Ticks

You can make the handle of the JavaFX `Slider` snap to the ticks using the `Slider setSnapToTicks()` method, passing a parameter value of `true` it. Here is an example of making the JavaFX `Slider` snap its handle to the ticks:

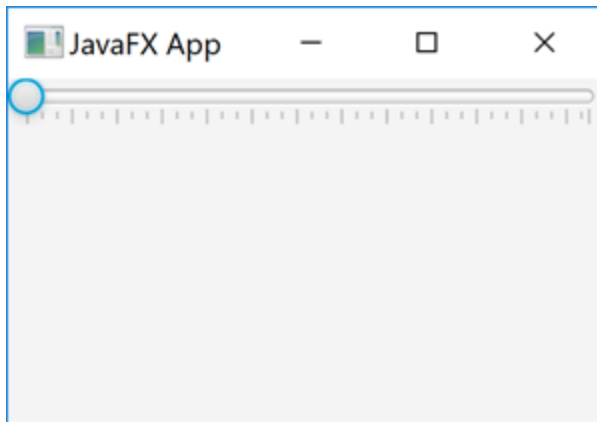
```
slider.setSnapToTicks(true);
```

## Show Tick Marks

You can make the *JavaFX Slider* show marks for the ticks when it renders the slider. You do so using its `setShowTickMarks()` method. Here is an example of making a JavaFX `Slider` show tick marks:

```
slider.setShowTickMarks(true);
```

Here is a screenshot of how a JavaFX `Slider` looks with tick marks shown:

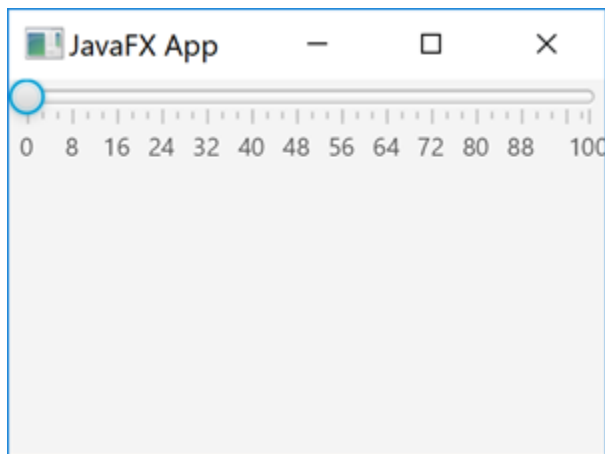


## Show Tick Labels

You can make the JavaFX `Slider` show tick labels for the ticks when it renders the slider. You do so using its `setShowTickLabels()` method. Here is an example of making a JavaFX `Slider` show tick labels:

```
slider.setShowTickLabels(true);
```

Here is a screenshot of how a JavaFX `Slider` looks with tick marks and labels shown:



Next: [JavaFX PasswordField](#)

# JavaFX PasswordField

- [Creating a PasswordField](#)
- [Adding a PasswordField to the Scene Graph](#)
- [Getting the Text of a PasswordField](#)

Jakob Jenkov

Last update: 2016-05-19

A JavaFX PasswordField control enables users of a JavaFX application to enter password which can then be read by the application.

The PasswordField control does not show the text entered into it. Instead it shows a circle for each character entered. The JavaFX PasswordField control is represented by the class `javafx.scene.control.PasswordField`.

## Creating a PasswordField

You create a PasswordField control by creating an instance of the PasswordField class. Here is a JavaFX PasswordField instantiation example:

```
PasswordField passwordField = new PasswordField();
```

## Adding a PasswordField to the Scene Graph

For a JavaFX PasswordField to be visible the PasswordField object must be added to the scene graph. This means adding it to a Scene object, or as child of a layout which is attached to a Scene object.

Here is an example that attaches a JavaFX PasswordField to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class PasswordFieldExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("PasswordField Experiment 1");

        PasswordField passwordField = new PasswordField();

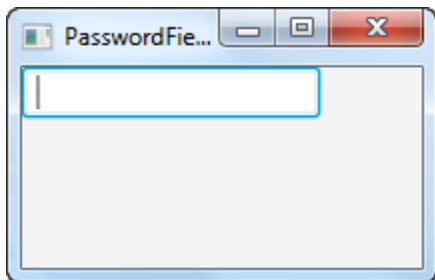
        HBox hbox = new HBox(passwordField);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The result of running the above JavaFX PasswordField example is an application that looks like this:



## Getting the Text of a PasswordField

You can get the text entered into a PasswordField using its `getText()` method which returns a `String`. Here is a full example that shows a PasswordField and a Button and which reads the text entered into the PasswordField when the button is clicked:

```

package com.jenkov.javafx.controls;

```

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class PasswordFieldExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("PasswordField Experiment 1");

        PasswordField passwordField = new PasswordField();

        Button button = new Button("Click to get password");

        button.setOnAction(action -> {
            System.out.println(passwordField.getText());
        });

        HBox hbox = new HBox(passwordField, button);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Next: [JavaFX TextArea](#)

# JavaFX TextArea

- [Creating a TextArea](#)
- [Adding a TextArea to the Scene Graph](#)
- [Reading the Text of a TextArea](#)
- [Setting the Text of a TextArea](#)

Jakob Jenkov

Last update: 2016-05-19

A JavaFX TextArea control enables users of a JavaFX application to enter text spanning multiple lines, which can then be read by the application. The JavaFX TextArea control is represented by the class `javafx.scene.control.TextArea`.

## Creating a TextArea

You create a TextArea control by creating an instance of the `TextArea` class. Here is a JavaFX TextArea instantiation example:

```
TextArea textArea = new TextArea();
```

## Adding a TextArea to the Scene Graph

For a JavaFX TextArea to be visible the `TextArea` object must be added to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX TextArea to the scene graph:

```
package com.jenkov.javafx.controls;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
```

```

public class TextAreaExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("TextArea Experiment 1");

        TextArea textArea = new TextArea();

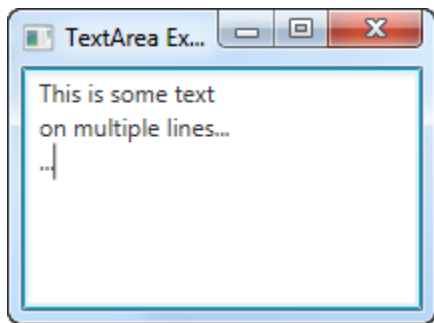
        VBox vbox = new VBox(textArea);

        Scene scene = new Scene(vbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The result of running the above JavaFX `TextArea` example is an application that looks like this:



## Reading the Text of a TextArea

You can read the text entered into a `TextArea` via its `getText()` method. Here is an example of reading text of a JavaFX `TextArea` control via its `getText()` method:

```
String text = textArea.getText();
```

Here is a full example that shows a `TextArea` and a `Button` and which reads the text entered into the `TextArea` when the button is clicked:



```

package com.jenkov.javafx.controls;

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

public class TextAreaExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("TextArea Experiment 1");

        TextArea textArea = new TextArea();

        Button button = new Button("Click to get text");
        button.setMinWidth(50);

        button.setOnAction(action -> {
            System.out.println(textArea.getText());

            textArea.setText("Clicked!");
        });

        VBox vbox = new VBox(textArea, button);

        Scene scene = new Scene(vbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

## Setting the Text of a TextArea

You can set the text of a `TextArea` control via its `setText()` method. Here is an example of setting the text of a `TextArea` control via `setText()` :

```

textArea.setText("New Text");

```

Next: [JavaFX ToolBar](#)

# JavaFX ToolBar

- [Creating a ToolBar](#)
- [Adding Items to a ToolBar](#)
- [Adding a ToolBar to the Scene Graph](#)
- [Vertical Oriented ToolBar](#)
- [Separating Items in a ToolBar](#)

Jakob Jenkov

Last update: 2018-03-29

The *JavaFX ToolBar* class (`javafx.scene.control.ToolBar`) is a horizontal or vertical bar containing buttons or icons that are typically used to select different tools of a JavaFX application. Actually, a `JavaFX ToolBar` can contain other JavaFX controls than just buttons and icons. In fact, you can insert any JavaFX control into a `ToolBar`.

## Creating a ToolBar

In order to create a `JavaFX ToolBar` you must first instantiate it. Here is an example of creating a `JavaFX ToolBar` instance:

```
ToolBar toolBar = new ToolBar();
```

That is all it takes to create a `JavaFX ToolBar`.

## Adding Items to a ToolBar

Once a `JavaFX ToolBar` has been created, you can add items (JavaFX components) to it. You add items to a `ToolBar` by obtaining its collection of items and adding the new item to that collection. Here is an example of adding an item to a `ToolBar`:

```
Button button = new Button("Click Me");  
  
toolBar.getItems().add(button);
```

## Adding a ToolBar to the Scene Graph

In order to make a JavaFX `ToolBar` visible, it must be added to the JavaFX scene graph. Here is a full example that shows the creation of a JavaFX `ToolBar` and adding it to the JavaFX scene graph:

```
package com.jenkov.javafx;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

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

        ToolBar toolBar = new ToolBar();

        Button button1 = new Button("Button 1");
        toolBar.getItems().add(button1);

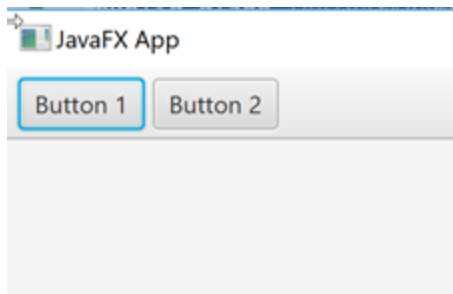
        Button button2 = new Button("Button 2");
        toolBar.getItems().add(button2);

        VBox vBox = new VBox(toolBar);

        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

The JavaFX GUI resulting from this `ToolBar` example would look similar to this:

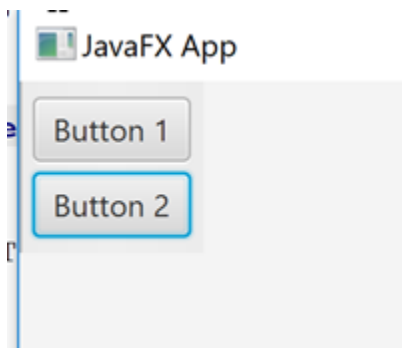


## Vertical Oriented Toolbar

By default a JavaFX `ToolBar` displays the items added to it in a horizontal row. It is possible to get the `ToolBar` to display the items vertically instead, so the `ToolBar` becomes a vertical toolbar. To make the `ToolBar` display its items vertically, you call its `setOrientation()` method. Here is an example of setting the orientation of a `ToolBar` to vertical:

```
toolbar.setOrientation(Orientation.VERTICAL);
```

Here is a screenshot of how the JavaFX `ToolBar` from the previous section looks in vertical orientation:



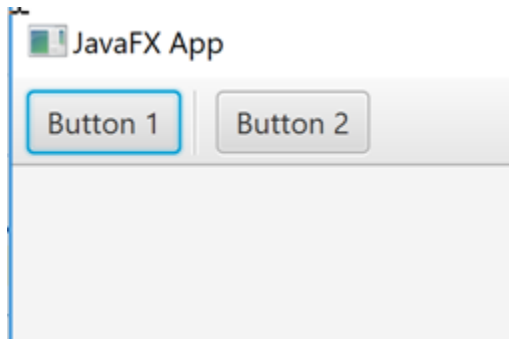
## Separating Items in a Toolbar

You can add a visual separator to a JavaFX `ToolBar`. The visual separator is typically displayed as a vertical or horizontal line between the items in the `ToolBar`. Here is an example of adding a separator to a `ToolBar`:

```
Button button1 = new Button("Button 1");
```

```
toolBar.getItems().add(button1);  
  
toolBar.getItems().add(new Separator());  
  
Button button2 = new Button("Button 2");  
toolBar.getItems().add(button2);
```

Here is a screenshot of how a visual separator between items in a ToolBar looks:



Next: [JavaFX Tooltip](#)

# JavaFX Tooltip

- [Creating a Tooltip Instance](#)
- [Adding a Tooltip to a JavaFX Component](#)
- [Text Alignment](#)
- [Tooltip Graphics](#)

Jakob Jenkov

Last update: 2018-03-31

The *JavaFX Tooltip* class (`javafx.scene.control.Tooltip`) can display a small popup with explanatory text when the user hovers the mouse over a JavaFX control. A `Tooltip` is a well-known feature of modern desktop and web GUIs. A `Tooltip` is useful to provide extra help text in GUIs where there is not space enough available to have an explanatory text visible all the time, e.g. in the button text.

## Creating a Tooltip Instance

To use the JavaFX `Tooltip` class you must create a `Tooltip` instance. Here is an example of creating a JavaFX `Tooltip` instance:

```
Tooltip tooltip1 = new Tooltip("Creates a new file");
```

The text passed as parameter to the `Tooltip` constructor is the text displayed when the `Tooltip` is visible.

## Adding a Tooltip to a JavaFX Component

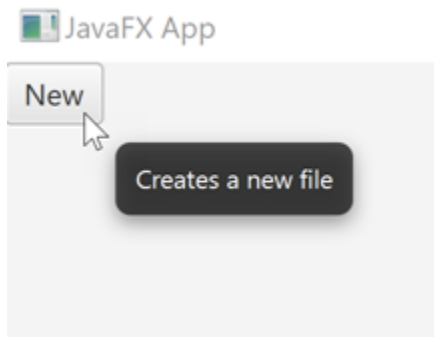
Once you have created a `Tooltip` instance you need to add it to a JavaFX component to make it active. Here is an example of adding a `Tooltip` instance to a JavaFX `Button`:

```
Tooltip tooltip1 = new Tooltip("Creates a new file");

Button button1 = new Button("New");
button1.setTooltip(tooltip1);
```

Notice the call to `Button's setTooltip()` method. This is what causes the `Tooltip` instance to be visible when the mouse is hovered over the button.

Here is a screenshot showing how the resulting `Tooltip` could look:



## Text Alignment

You can set the text alignment of the text inside the `Tooltip` box via its `setTextAlignment()` method. Here is an example of setting the text alignment of a `Tooltip`:

```
tooltip1.setTextAlignment(TextAlignment.LEFT);
```

The class `javafx.scene.text.TextAlignment` contains four different constants that represent different kinds of text alignment. The four constants are:

- `LEFT`
- `RIGHT`
- `CENTER`
- `JUSTIFY`

The first three constants represent the left, right and center justification of text within the popup box. The last constant, `JUSTIFY`, will align the text with both the left and right edges of the popup box by increasing the space in between the words to make the text fit.

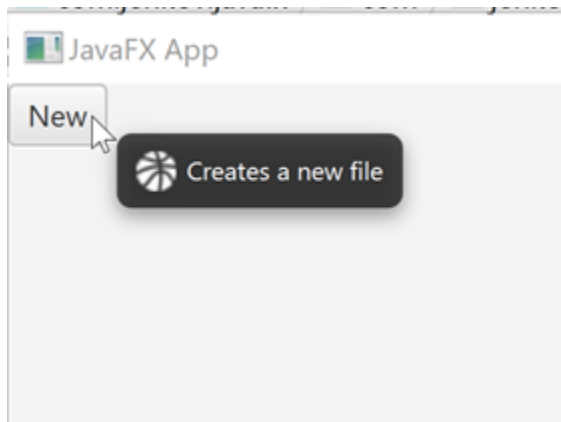
Notice that setting the text alignment may not result in a visible effect on the text alignment. That is because by default width of the popup box around the text is calculated based on the width of the text. If your text is just a single line, the text will almost always appear centered within the popup box. Text alignment first really takes effect when the popup box contains multiple lines of text, or if you set the width of the `Tooltip` explicitly (manually).

## Tooltip Graphics

You can set a graphic icon for a `Tooltip` via the `setGraphic()` method. Here is an example of setting a graphic icon for a `Tooltip`:

```
tooltip1.setGraphic(new ImageView("file:iconmonstr-basketball-1-16.png"));
```

Here is an example screenshot illustrating how a `Tooltip` graphic could look:



Next: [JavaFX ProgressBar](#)



# JavaFX ProgressBar

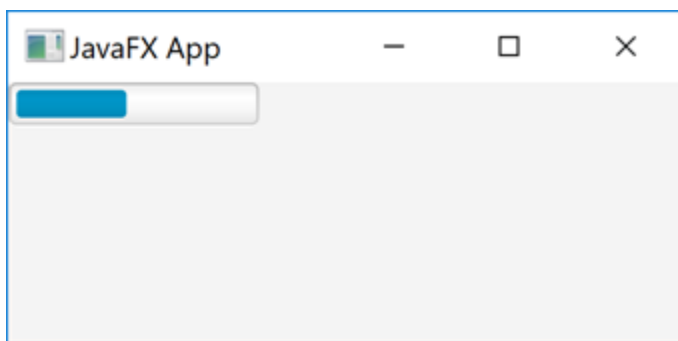
- [JavaFX ProgressBar Example](#)
- [Create a ProgressBar](#)
- [Setting the Progress Level](#)

Jakob Jenkov

Last update: 2019-05-25

The *JavaFX ProgressBar* is a control capable of displaying the progress of some task. The progress is set as a `double` value between 0 and 1, where 0 means no progress and 1 means full progress (task completed).

The *JavaFX ProgressBar* control is represented by the `javafx.scene.control.ProgressBar` class. Here is a screenshot of how a *JavaFX ProgressBar* looks:



The `ProgressBar` in the above screenshot has its progress set to 0.5.

## JavaFX ProgressBar Example

Here is a full JavaFX `ProgressBar` code example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Hyperlink;
import javafx.scene.control.ProgressBar;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

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

```

    }

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

        ProgressBar progressBar = new ProgressBar(0);

        progressBar.setProgress(0.5);

        VBox vBox = new VBox(progressBar);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

## Create a ProgressBar

In order to use a JavaFX `ProgressBar` you must first create an instance of the `ProgressBar` class. Here is how you create an instance of a JavaFX `ProgressBar`:

```
ProgressBar progressBar = new ProgressBar();
```

This example creates a `ProgressBar` in indeterminate mode, meaning its progress level is not known. In indeterminate mode the JavaFX `ProgressBar` displays an animation.

You can create a `ProgressBar` instance with a determinate progress level by passing the progress value as parameter to its constructor, like this:

```
ProgressBar progressBar = new ProgressBar(0);
```

## Setting the Progress Level

You set the progress level of a `ProgressBar` via the `setProgress()` method. Here is an example of how you set the progress level of a JavaFX `ProgressBar`:

```
ProgressBar progressBar = new ProgressBar(0);  
progressBar.setProgress(0.5);
```

Next: [JavaFX FileChooser](#)

# JavaFX FileChooser

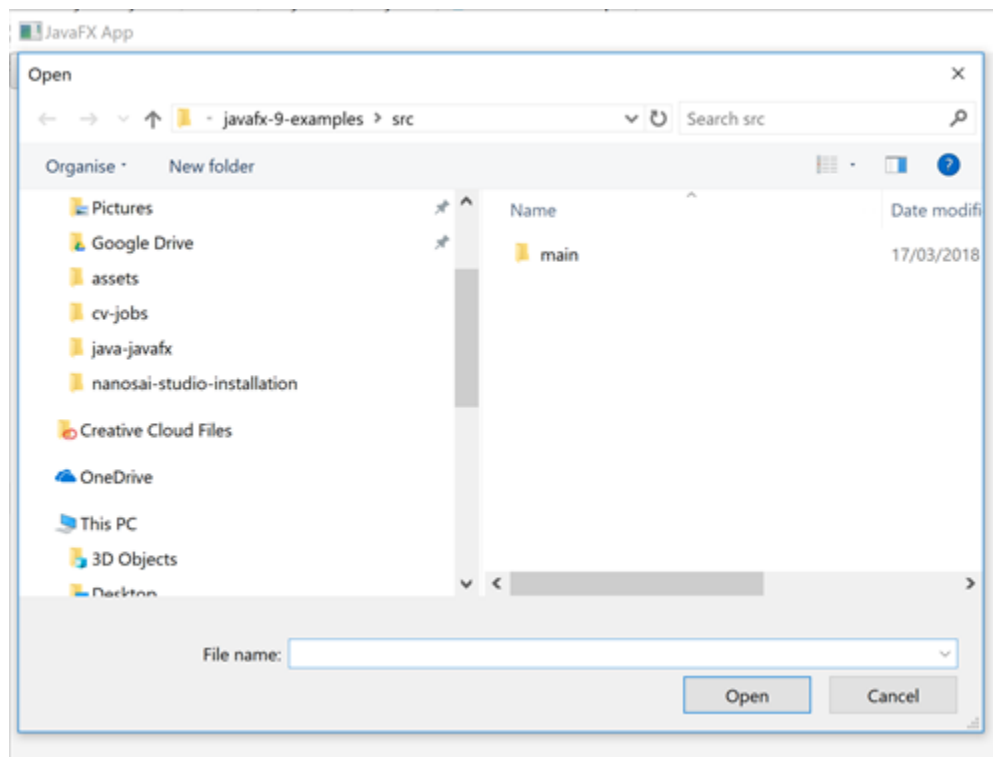
- [Creating a FileChooser](#)
- [Showing the FileChooser Dialog](#)
- [Setting Initial Directory](#)
- [Setting Initial File Name](#)
- [Adding File Name Filters](#)

Jakob Jenkov

Last update: 2019-01-24

A *JavaFX FileChooser* class (`javafx.stage.FileChooser`) is a dialog that enables the user to select one or more files via a file explorer from the user's local computer. The JavaFX FileChooser is implemented in the class `javafx.stage.FileChooser`. In this JavaFX FileChooser tutorial I will show you how to use the JavaFX FileChooser dialog.

Here is an example screenshot of how a JavaFX FileChooser looks:



## Creating a FileChooser

In order to use the JavaFX `FileChooser` dialog you must first create a `FileChooser` instance. Here is an example of creating a JavaFX `FileChooser` dialog:

```
FileChooser fileChooser = new FileChooser();
```

As you can see, it is pretty easy to create a `FileChooser` instance.

## Showing the FileChooser Dialog

Showing the JavaFX `FileChooser` dialog is done by calling its `showOpenDialog()` method. Here is an example of showing a `FileChooser` dialog:

```
File selectedFile = fileChooser.showOpenDialog(stage);
```

The `File` returned by the `showOpenDialog()` method is the file the user selected in the `FileChooser`.

The `stage` parameter is the JavaFX `Stage` that should "own" the `FileChooser` dialog. By "owning" is meant what `Stage` from which the `FileChooser` dialog is shown. This will typically be the `Stage` in which the button sits that initiates the showing of the `FileChooser`.

Showing a `FileChooser` is typically done as a result of a click on a button or menu item. Here is a full JavaFX example that shows a button that opens a `FileChooser` when it is clicked:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

public class FileChooserExample extends Application {
    public static void main(String[] args) {
```

```

        launch(args);
    }

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

        FileChooser fileChooser = new FileChooser();

        Button button = new Button("Select File");
        button.setOnAction(e -> {
            File selectedFile = fileChooser.showOpenDialog(primaryStage);
        });

        VBox vBox = new VBox(button);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

This example creates a full JavaFX application with a `Button` that when clicked opens a `FileChooser`. Notice how the primary `Stage` for the JavaFX application is passed as parameter to the `FileChooser.showOpenDialog()` method.

## Setting Initial Directory

You can set the initial directory displayed in the JavaFX `FileChooser` via its `setInitialDirectory()` method. Here is an example of setting the initial directory of a `FileChooser` dialog:

```
fileChooser.setInitialDirectory(new File("data"));
```

This example sets the initial directory displayed by the `FileChooser` to `data`.

## Setting Initial File Name

You can set the initial file name to display in the `FileChooser`. Some platforms (e.g. Windows) may ignore this setting, though. Here is an example of setting the initial file name of a `FileChooser`:

```
fileChooser.setInitialFileName("myfile.txt");
```

This example sets the initial file name to `myfile.txt`.

## Adding File Name Filters

It is possible to add file name filters to a JavaFX `FileChooser`. File name filters are used to filter out what files are shown in the `FileChooser` when the user browses around the file system. Here is an example of adding file name filters:

```
FileChooser fileChooser = new FileChooser();

fileChooser.getExtensionFilters().addAll(
    new FileChooser.ExtensionFilter("Text Files", "*.txt")
    ,new FileChooser.ExtensionFilter("HTML Files", "*.htm")
);
```

This examples adds two file name filters to the `FileChooser`. The user can choose between these file name filters inside the `FileChooser` dialog.

Next: [JavaFX DirectoryChooser](#)

# JavaFX DirectoryChooser

- [Creating a DirectoryChooser](#)
- [Showing the DirectoryChooser Dialog](#)
- [Setting Initial Directory](#)

Jakob Jenkov

Last update: 2019-01-24

A JavaFX *DirectoryChooser* is a dialog that enables the user to select a directory via a file explorer from the user's local computer. The JavaFX *DirectoryChooser* is implemented in the class `javafx.stage.DirectoryChooser`. In this JavaFX *DirectoryChooser* tutorial I will show you how to use the *DirectoryChooser* dialog.

Here is an example screenshot of how a JavaFX *DirectoryChooser* looks:



## Select Folder



« tutorial-projects » javafx-9


Organise ▾

New folder


### ★ Quick access

 Desktop



 Downloads



 Documents





 Pictures





 Google Drive




 data

 data-streaming

 java-try-with-resources

 regnskaber

 Creative Cloud Files

## Creating a DirectoryChooser

In order to use the `DirectoryChooser` you must first create a `DirectoryChooser` instance. Here is an example of creating a JavaFX `DirectoryChooser`:

```
DirectoryChooser directoryChooser = new DirectoryChooser();
```

## Showing the DirectoryChooser Dialog

In order to make the `DirectoryChooser` visible you must call its `showDialog()` method. Here is an example of showing a JavaFX `DirectoryChooser`:

```
File selectedDirectory = directoryChooser.showDialog(primaryStage);
```

The `File` returned by the `showDialog()` method represents the directory the user selected in the `DirectoryChooser`.

The `stage` parameter is the JavaFX `Stage` that should "own" the `DirectoryChooser` dialog. By "owning" is meant what `Stage` from which the `DirectoryChooser` dialog is shown. This will typically be the `Stage` in which the button sits that initiates the showing of the `DirectoryChooser`.

Showing a `DirectoryChooser` is typically done as a result of a click on a button or menu item. Here is a full JavaFX example that shows a button that opens a `DirectoryChooser` when it is clicked:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.DirectoryChooser;
import javafx.stage.Stage;

import java.io.File;

public class DirectoryChooserExample extends Application {
```

```

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

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

    DirectoryChooser directoryChooser = new DirectoryChooser();
    directoryChooser.setInitialDirectory(new File("src"));

    Button button = new Button("Select Directory");
    button.setOnAction(e -> {
        File selectedDirectory =
directoryChooser.showDialog(primaryStage);

        System.out.println(selectedDirectory.getAbsolutePath());
    });

    VBox vBox = new VBox(button);
    //HBox hBox = new HBox(button1, button2);
    Scene scene = new Scene(vBox, 960, 600);

    primaryStage.setScene(scene);
    primaryStage.show();
}
}

```

## Setting Initial Directory

You can set the initial directory of the JavaFX `DirectoryChooser`, meaning the root directory the `DirectoryChooser` will be located at when opened. This is also shown in the example above. You set the initial directory via the method `setInitialDirectory()`. Here is an example of setting the initial directory of a JavaFX `DirectoryChooser`:

```

directoryChooser.setInitialDirectory(new File("data/json/invoices"));

```

This example will set the initial directory of the given `DirectoryChooser` to `data/json/invoices`.

Next: [JavaFX TitledPane](#)

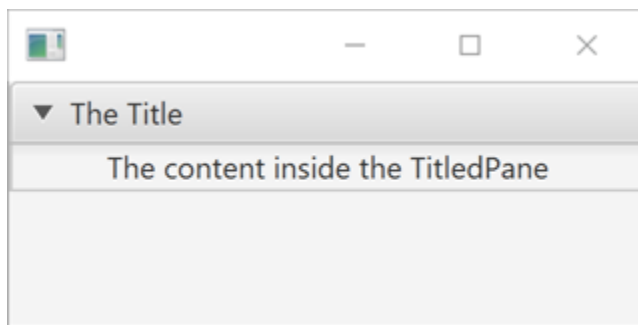
# JavaFX TitledPane

- [Creating a JavaFX TitledPane](#)
- [Adding the TitledPane to the JavaFX Scene Graph](#)
- [Collapse and Expand a TitledPane](#)
- [Disable Collapse](#)

Jakob Jenkov

Last update: 2019-04-22

The *JavaFX TitledPane* control is a container control which displays its content inside a a pane (box) which at the top contains a title - hence the name TitledPane. The TitledPane control is implemented by the `javafx.scene.control.TitledPane` class. In this JavaFX TitledPane tutorial we will look at how to use the TitledPane control. Here is a JavaFX TitledPane screenshot showing how it looks:



A TitledPane can be collapsed so only the title bar is visible. This functionality is used inside the JavaFX Accordion control. The TitledPane can of course be expanded too. I will show how that works later in this tutorial.

## Creating a JavaFX TitledPane

In order to use a JavaFX TitledPane you must first create a TitledPane instance. Here is an example of creating a JavaFX TitledPane:

```
Label label = new Label("The content inside the TitledPane");

TitledPane titledPane = new TitledPane("The Title", label);
```

Notice the second line in the code example. This is the line that creates the `TitledPane` instance. Notice how the title to display in the `TitledPane` is passed as a parameter to the constructor. Notice also, how the content to display, a JavaFX Node, is also passed as a parameter to the constructor. In this example the content is just a simple [JavaFX Label](#).

## Adding the TitledPane to the JavaFX Scene Graph

To make a JavaFX `TitledPane` instance visible, it must be added to a [JavaFX scene graph](#). Here is a full example of adding a JavaFX `TitledPane` to a JavaFX scene graph:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TitledPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TitledPaneExample extends Application {

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

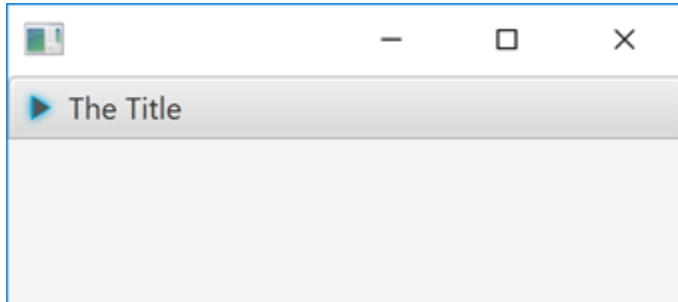
    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("The content inside the TitledPane");
        TitledPane titledPane = new TitledPane("The Title", label);

        Scene scene = new Scene(new VBox(titledPane));
        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

## Collapse and Expand a TitledPane

The user can collapse and expand a JavaFX `TitledPane` using the small triangle next to the title in the title bar of the `TitledPane`. Here is an example of how a collapsed `TitledPane` looks:



Notice how the content of the `TitledPane` is no longer visible.

It is also possible to collapse and expand a `TitledPane` programmatically. You do so by calling its `setExpanded()` method. Here is an example of expanding and collapsing a `TitledPane` programmatically:

```
titledPane.setExpanded(true);  
titledPane.setExpanded(false);
```

## Disable Collapse

It is possible to disable the collapse functionality of a JavaFX `TitledPane`. You do so by calling its `setCollapsible()` method, passing a value of `false` as parameter. Here is how switching off the collapsible functionality of a `TitledPane` looks:

```
Label label = new Label("The content inside the TitledPane");  
TitledPane titledPane = new TitledPane("The Title", label);  
  
titledPane.setCollapsible(false);
```

Next: [JavaFX Accordion](#)

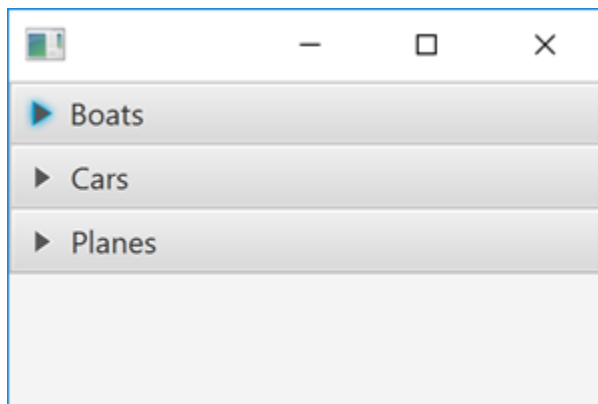
# JavaFX Accordion

- [JavaFX Accordion Example](#)
- [Create an Accordion](#)
- [Add TitledPane Objects to Accordion](#)
- [Add Accordion to Scene Graph](#)

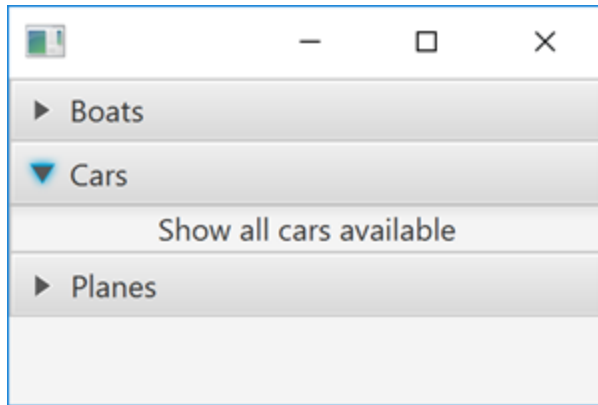
Jakob Jenkov

Last update: 2019-04-27

The *JavaFX Accordion* control is a container control which can contain several sections internally, each of which can have their content expanded or collapsed. The Accordion control is implemented by the JavaFX class `javafx.scene.control.Accordion`. The sections displayed inside it are made up of [JavaFX TitledPane](#) controls. Here is a screenshot of a JavaFX Accordion control:



Notice that none of the sections are expanded. You can expand a section by clicking on the little triangle next to the title for each section. Expanding a section will reveal its content. Here is a screenshot of a JavaFX Accordion with a section expanded:



## JavaFX Accordion Example

Here is a full JavaFX `Accordion` example so you can quickly get an overview of what its usage looks like:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Accordion;
import javafx.scene.control.Label;
import javafx.scene.control.TitledPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class AccordionExample extends Application {

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

    public void start(Stage primaryStage) {

        Accordion accordion = new Accordion();

        TitledPane panel1 = new TitledPane("Boats" , new Label("Show all boats
available"));
        TitledPane pane2 = new TitledPane("Cars" , new Label("Show all cars
available"));
        TitledPane pane3 = new TitledPane("Planes", new Label("Show all
planes available"));

        accordion.getPanes().add(panel1);
        accordion.getPanes().add(pane2);
        accordion.getPanes().add(pane3);

        VBox vBox = new VBox(accordion);
        Scene scene = new Scene(vBox);

        primaryStage.setScene(scene);
```



```
        primaryStage.show();  
    }  
}
```

## Create an Accordion

Before you can use the JavaFX `Accordion` control you must first instantiate it. You instantiate it simply using the Java `new` command, like this:

```
Accordion accordion = new Accordion();
```

## Add TitledPane Objects to Accordion

Each section displayed inside a JavaFX `Accordion` is represented by a [JavaFX TitledPane](#). To add sections to the `Accordion` control, you create one `TitledPane` per section, and add it to the `Accordion`. Here is an example of adding `TitledPane` sections to a JavaFX `Accordion`:

```
Accordion accordion = new Accordion();  
  
TitledPane panel1 = new TitledPane("Boats" , new Label("Show all boats  
available"));  
TitledPane panel2 = new TitledPane("Cars"  , new Label("Show all cars  
available"));  
TitledPane panel3 = new TitledPane("Planes", new Label("Show all planes  
available"));  
  
accordion.getPanes().add(panel1);  
accordion.getPanes().add(panel2);  
accordion.getPanes().add(panel3);
```

## Add Accordion to Scene Graph

To make a JavaFX `Accordion` visible, you must add it to the [scene graph](#). Here is an example of adding a JavaFX `Accordion` to the JavaFX scene graph:

```
Accordion accordion = new Accordion();  
  
VBox vbox = new VBox(accordion);
```

```
Scene scene = new Scene(vBox);  
primaryStage.setScene(scene);  
primaryStage.show();
```

Next: [JavaFX SplitPane](#)

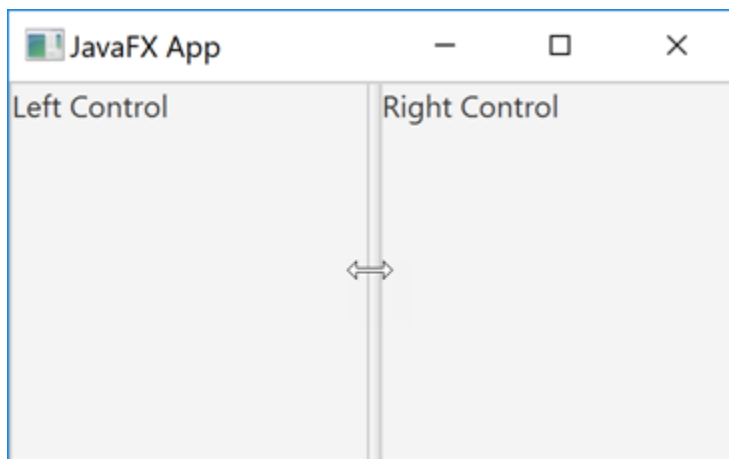
# JavaFX SplitPane

- [Full JavaFX SplitPane Example](#)
- [Create a SplitPane](#)
- [Adding Controls to the SplitPane](#)
  - [Adding More Than Two Controls to a SplitPane](#)

Jakob Jenkov

Last update: 2019-06-16

The *JavaFX SplitPane* is a container control that can contain multiple other components inside it. In other words, the `SplitPane` is *split* between the controls it contains. Between the controls in the `SplitPane` is a divider. The user can move the divider to set how much space is allocated to each control. Here is a screenshot of a JavaFX `SplitPane`:



## Full JavaFX SplitPane Example

The JavaFX *SplitPane* is represented by the JavaFX class `javafx.scene.control.SplitPane`. Here is a full JavaFX `SplitPane` example so you can get an idea about how using it looks:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.SplitPane;
```

```

import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SplitPaneExample extends Application {

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

    public void start(Stage primaryStage) {

        SplitPane splitPane = new SplitPane();

        VBox leftControl  = new VBox(new Label("Left Control"));
        VBox rightControl = new VBox(new Label("Right Control"));

        splitPane.getItems().addAll(leftControl, rightControl);

        Scene scene = new Scene(splitPane);

        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App");

        primaryStage.show();
    }
}

```

## Create a SplitPane

Before you can use a JavaFX `SplitPane` you must first create a `SplitPane` instance. Here is an example of creating a JavaFX `SplitPane`:

```
SplitPane splitPane = new SplitPane();
```

## Adding Controls to the SplitPane

In order to show anything inside the JavaFX `SplitPane` you must add some JavaFX controls to it. You do so via the `SplitPane.getItems().add(...)` method. Here is an example of adding two controls to a JavaFX `SplitPane`:

```

SplitPane splitPane = new SplitPane();

VBox leftControl  = new VBox(new Label("Left Control"));

```

```
VBox rightControl = new VBox(new Label("Right Control"));

splitPane.getItems().addAll(leftControl, rightControl);
```

## Adding More Than Two Controls to a SplitPane

You can add more than two controls to a JavaFX `SplitPane`. If you do, there will be a divider in-between each two controls. Here is a Java code example of adding 3 controls to a JavaFX `SplitPane`:

```
SplitPane splitPane = new SplitPane();

VBox leftControl  = new VBox(new Label("Left Control"));
VBox midControl   = new VBox(new Label("Mid Control"));
VBox rightControl = new VBox(new Label("Right Control"));

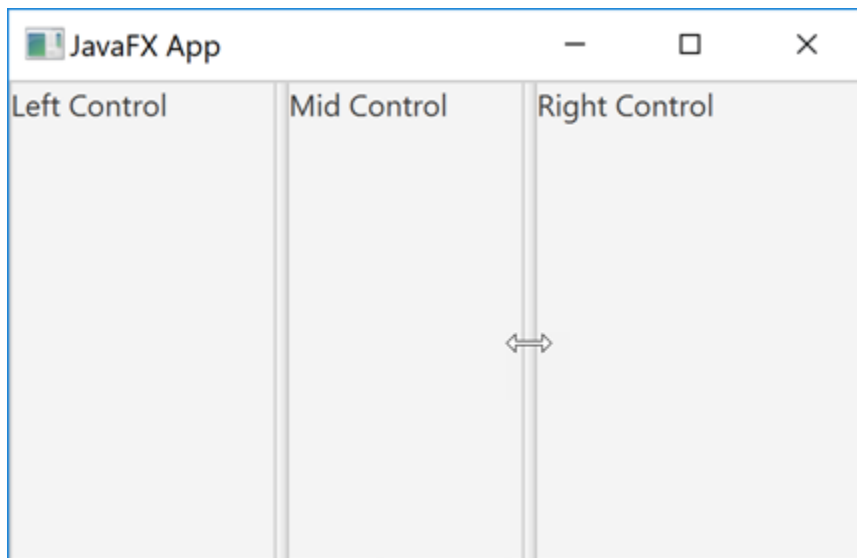
splitPane.getItems().addAll(leftControl, midControl, rightControl);

Scene scene = new Scene(splitPane);

primaryStage.setScene(scene);
primaryStage.setTitle("JavaFX App");

primaryStage.show();
```

Here is a screenshot of how such a `SplitPane` with 3 controls added looks:



Next: [JavaFX TabPane](#)

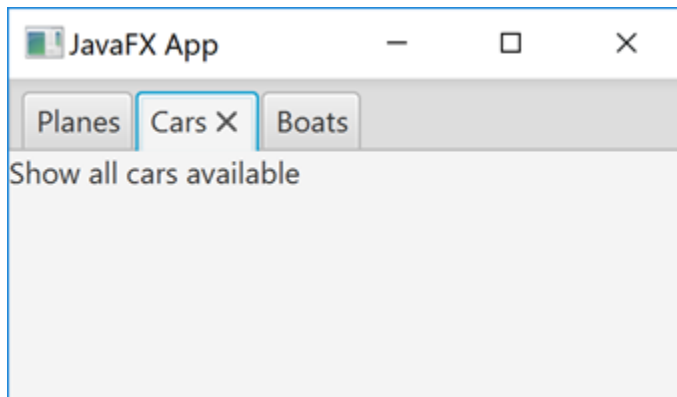
# JavaFX TabPane

- [Full JavaFX TabPane Example](#)
- [Create a TabPane](#)
- [Add Tabs to TabPane](#)
- [Get Selected Tab](#)

Jakob Jenkov

Last update: 2019-06-10

The *JavaFX TabPane* is a container control which can contain multiple tabs (sections) internally, which can be displayed by clicking on the tab with the title on top of the TabPane. Only one tab is displayed at a time. It is like paper folders where one of the folders is open. The *JavaFX TabPane* control is implemented by the `javafx.scene.control.TabPane` class. Here is a screenshot of a *JavaFX TabPane*:



In this screenshot the middle tab has focus, meaning the title of the middle tab has been clicked.

## Full JavaFX TabPane Example

Here is a full JavaFX TabPane code example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TabPane;
import javafx.scene.control.Tab;
import javafx.scene.control.Label;
```

```

import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TabPaneExample extends Application {

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

    public void start(Stage primaryStage) {

        TabPane tabPane = new TabPane();

        Tab tab1 = new Tab("Planes", new Label("Show all planes available"));
        Tab tab2 = new Tab("Cars" , new Label("Show all cars available"));
        Tab tab3 = new Tab("Boats" , new Label("Show all boats available"));

        tabPane.getTabs().add(tab1);
        tabPane.getTabs().add(tab2);
        tabPane.getTabs().add(tab3);

        VBox vBox = new VBox(tabPane);
        Scene scene = new Scene(vBox);

        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App");

        primaryStage.show();
    }
}

```

## Create a TabPane

In order to use a JavaFX TabPane You must first create an instance of the TabPane class. Here is an example of creating an instance of the JavaFX TabPane class:

```
TabPane tabPane = new TabPane();
```

## Add Tabs to TabPane

To display any content, you must add one or more *tabs* to the JavaFX TabPane. A tab is represented by the `javafx.scene.control.Tab` class. Here is an example of adding 3 tabs to a JavaFX TabPane:

```
TabPane tabPane = new TabPane();

Tab tab1 = new Tab("Planes", new Label("Show all planes available"));
Tab tab2 = new Tab("Cars" , new Label("Show all cars available"));
Tab tab3 = new Tab("Boats" , new Label("Show all boats available"));

tabPane.getTabs().add(tab1);
tabPane.getTabs().add(tab2);
tabPane.getTabs().add(tab3);
```

The `Tab` constructor used in the above example takes two parameters. The first parameter is the title to display at the top of the tab, in the "handle" where you click to show the tab. The second parameter is the root JavaFX control containing the content to display inside the body part of the tab. In the example above a simple [JavaFX Label](#) is used, but in a real application it would be more normal to use a container control which can contain other nested controls inside it. For instance, a [JavaFX VBox](#), [JavaFX HBox](#), [JavaFX Flowpane](#), [JavaFX TilePane](#) or [JavaFX GridPane](#).

## Get Selected Tab

You can obtain the `Tab` that is currently selected (visible) in a JavaFX `TabPane` via the `TabPane.getSelectionModel().getSelectedItem()` method calls. Here is an example of obtaining the currently selected `Tab` from a JavaFX `TabPane`:

```
Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
```

Next: [JavaFX ScrollPane](#)



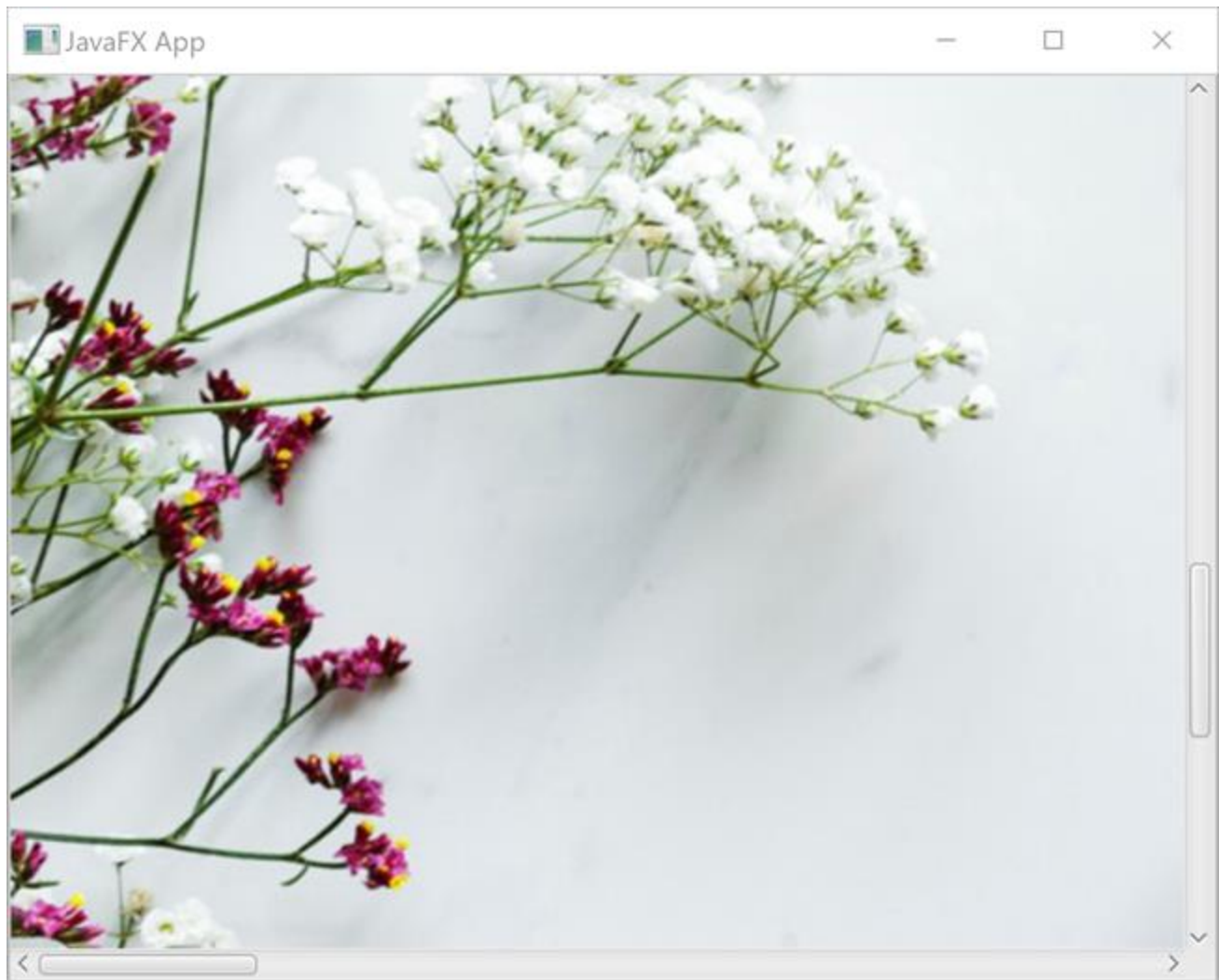
# JavaFX ScrollPane

- [Create a ScrollPane](#)
- [Set ScrollPane Content](#)
- [ScrollPane Viewport](#)
- [Content With Effects or Transforms](#)
- [Pannable ScrollPane](#)
- [Fit To Width](#)
- [Fit To Height](#)
- [Showing and Hiding Scrollbars via ScrollBar Policies](#)

Jakob Jenkov

Last update: 2019-07-07

The *JavaFX ScrollPane* control is a container that has two scrollbars around the component it contains if the component is larger than the visible area of the ScrollPane. The scrollbars enable the user to scroll around the component shown inside the ScrollPane, so different parts of the component can be seen. The JavaFX ScrollPane controls is represented by the JavaFX class `javafx.scene.control.ScrollPane`. Here is a screenshot of a JavaFX ScrollPane with a [JavaFX ImageView](#) inside:



## Create a ScrollPane

To use a JavaFX `ScrollPane` you must first create a `ScrollPane` instance. Here is an example of creating a JavaFX `ScrollPane` instance:

```
ScrollPane scrollPane = new ScrollPane();
```

## Set ScrollPane Content

Once you have created a JavaFX `ScrollPane` instance you can set the content you want it to display via its `setContent()` method. Here is an

example that sets a JavaFX `ImageView` as content of a `ScrollPane`:

```
ScrollPane scrollPane = new ScrollPane();

String imagePath = "images/aerial-beverage-caffeine-972533.jpg";
ImageView imageView = new ImageView(new Image(new
FileInputStream(imagePath)));

scrollPane.setContent(imageView);
```

## ScrollPane Viewport

The visible part of a JavaFX `ScrollPane` is called the *ScrollPane viewport*. As you scroll around the content displayed inside the `ScrollPane` using the scrollbars, the viewport is moved around the content too, making different parts of the content visible.

## Content With Effects or Transforms

If the content (JavaFX control) you want to display inside the JavaFX `ScrollPane` uses effects or transforms, you must first wrap these controls in a [JavaFX Group](#). Otherwise the content won't be displayed correctly.

## Pannable ScrollPane

By default the user can only navigate around the content displayed in a JavaFX `ScrollPane` using its scrollbars. However, it is possible to make a JavaFX `ScrollPane` *pannable*. A *pannable* `ScrollPane` enables the user to navigate its content by holding down the left mouse button and move the mouse around. This will have the same effect as using the scrollbars. However, using panning you can move the content along both X and Y axis simultaneously. This is not possible using the scrollbars, where the user can only operate one scrollbar at a time.

To switch a JavaFX `ScrollPane` into pannable mode you must set its `pannableProperty` to the value `true`. Here is an example of switching a JavaFX `ScrollPane` into pannable mode:

```
scrollPane.pannableProperty().set(true);
```

## Fit To Width

The JavaFX `ScrollPane` `fitToWidth` property can make the `ScrollPane` fit its content to the width of the `ScrollPane` viewport. To do so, the `fitToWidth` property must be set to the value `true`. This property is ignored if the content node is not resizable. Here is an example of setting the JavaFX `ScrollPane` `fitToWidth` property to `true`:

```
scrollPane.fitToWidthProperty().set(true);
```

## Fit To Height

The JavaFX `ScrollPane` `fitToHeight` property can make the `ScrollPane` fit its content to the height of the `ScrollPane` viewport. To do so, the `fitToHeight` property must be set to the value `true`. This property is ignored if the content node is not resizable. Here is an example of setting the JavaFX `ScrollPane` `fitToHeight` property to `true`:

```
scrollPane.fitToHeightProperty().set(true);
```

## Showing and Hiding Scrollbars via ScrollBar Policies

It is possible to specify when the JavaFX `ScrollPane` is to show the vertical and horizontal scrollbars. You do so via the `ScrollPane` `hbarPolicyProperty` and `vbarPolicyProperty` properties. These properties can be set to one of the `ScrollPane.ScrollBarPolicy` enum values. You can choose from the values `ALWAYS`, `AS_NEEDED` and `NEVER`. Here is an example of setting the `hbarPolicyProperty` and `vbarPolicyProperty` to `ScrollBarPolicy.NEVER`:

```
scrollPane.hbarPolicyProperty().setValue(ScrollPane.ScrollBarPolicy.NEVER);  
scrollPane.vbarPolicyProperty().setValue(ScrollPane.ScrollBarPolicy.NEVER);
```

The above example removes the vertical and horizontal scrollbar from the `ScrollPane`. Without the scrollbars the user cannot use them to scroll around the content of the `ScrollPane`. However, if the `ScrollPane` is in pannable mode (see earlier sections in this JavaFX `ScrollPane` tutorial) the user can still grab the content and scroll around it with the mouse.

Next: [JavaFX Group](#)

# JavaFX Group

- [Creating a Group](#)
- [Adding Components to a Group](#)
- [Adding a Group to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-21

The JavaFX Group component is a container component which applies no special layout to its children. All child components (nodes) are positioned at 0,0 . A JavaFX Group component is typically used to apply some effect or transformation to a set of controls as a whole - as a group. If you need some layout to the children inside the Group, nest them inside layout components and add the layout components to the Group. The JavaFX Group component is represented by the class `javafx.scene.Group`.

## Creating a Group

You create a JavaFX Group instance via its constructor. Here is a JavaFX Group instantiation example:

```
Group group = new Group();
```

## Adding Components to a Group

You can add components to a JavaFX Group by obtaining its list of children and adding the children to that list. Here is an example of adding children to a JavaFX Group:

```
Button button1 = new Button("Button Number 1");  
Button button2 = new Button("Button 2");  
  
Group group = new Group();  
  
group.getChildren().add(button1);  
group.getChildren().add(button2);
```

## Adding a Group to the Scene Graph

To make a JavaFX `Group` instance visible it must be added to the JavaFX scene graph. That means adding the `Group` instance to a `Scene` object or adding the `Group` instance to a layout component which is then added to a `Scene` object.

Here is an example of adding a JavaFX `Group` instance to the JavaFX scene graph:

```
package com.jenkov.javafx.layouts;

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

public class GroupExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        Button button1 = new Button("Button Number 1");
        Button button2 = new Button("Button 2");

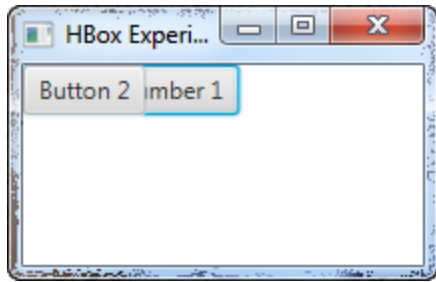
        Group group = new Group();

        group.getChildren().add(button1);
        group.getChildren().add(button2);

        Scene scene = new Scene(group, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from running the above code would look similar to this:



As you can see, the two buttons are positioned on top of each other, because both buttons are positioned at 0,0 inside the `Group` component.

Next: [JavaFX HBox](#)



# JavaFX HBox

- [Creating a HBox](#)
- [Adding a HBox to the Scene Graph](#)
- [Space Between Nodes](#)

Jakob Jenkov

Last update: 2016-05-14

The JavaFX HBox component is a layout component which positions all its child nodes (components) in a horizontal row. The Java HBox component is represented by the class `javafx.scene.layout.HBox`.

## Creating a HBox

You create an `HBox` using its constructor like this:

```
HBox hbox = new HBox();
```

`HBox` also has a constructor which takes a variable length list of components it should layout. Here is an example of how to do that:

```
Button button1 = new Button("Button Number 1");  
Button button2 = new Button("Button Number 2");  
  
HBox hbox = new HBox(button1, button2);
```

This `HBox` example will layout the two [Button](#) instances next to each other in a horizontal row.

## Adding a HBox to the Scene Graph

For an `HBox` to be visible it must be added to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX `HBox` with the two `Button` instances to the scene graph:

```
package com.jenkov.javafx.layouts;

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

public class HBoxExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

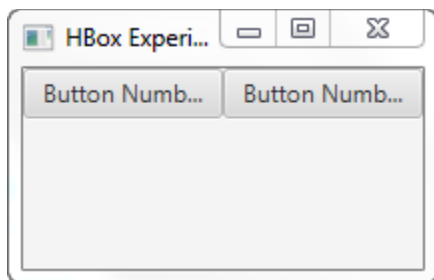
        Button button1 = new Button("Button Number 1");
        Button button2 = new Button("Button Number 2");

        HBox hbox = new HBox(button1, button2);

        Scene scene = new Scene(hbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The result of running the above JavaFX `HBox` example is an application that looks like this:



Notice that the two `Button` controls are kept on the same horizontal row even if there is not enough space to show them in their fully preferred widths. The buttons do not "wrap" down on the next line.

## Space Between Nodes

In the earlier example the `HBox` positioned the nodes (button controls) right next to each other. You can make the `HBox` insert some space between its nested controls by providing the space in the `HBox` constructor. Here is an example of setting the space between nested controls in an `HBox`:

```
HBox hbox = new HBox(20, button1, button2);
```

This example sets the spacing between the controls in the `HBox` layout component to 20.

You can also set the space between the nested controls using the `setSpacing()` method, like this:

```
hbox.setSpacing(50);
```

This example will set the spacing between nested controls to 50.

Next: [JavaFX VBox](#)

# JavaFX VBox

- [Creating a VBox](#)
- [Adding a VBox to the Scene Graph](#)
- [Space Between Nodes](#)

Jakob Jenkov

Last update: 2016-05-14

The JavaFX VBox component is a layout component which positions all its child nodes (components) in a vertical row. The Java VBox component is represented by the class `javafx.scene.layout.VBox`.

## Creating a VBox

You create an `VBox` using its constructor like this:

```
VBox vbox = new VBox();
```

`VBox` also has a constructor which takes a variable length list of components it should layout. Here is an example of how to do that:

```
Button button1 = new Button("Button Number 1");  
Button button2 = new Button("Button Number 2");  
  
VBox vbox = new VBox(button1, button2);
```

This `VBox` example will layout the two [Button](#) instances under each other in a vertical row.

## Adding a VBox to the Scene Graph

For an `VBox` to be visible it must be added to the scene graph. This means adding it to a `Scene` object, or as child of a layout which is attached to a `Scene` object.

Here is an example that attaches a JavaFX VBox with the two Button instances to the scene graph:

```
package com.jenkov.javafx.layouts;

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

public class VBoxExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

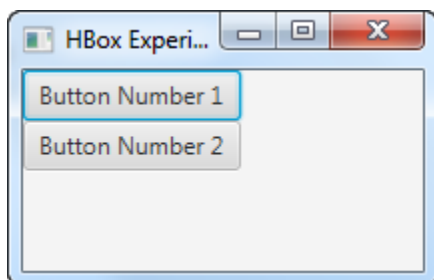
        Button button1 = new Button("Button Number 1");
        Button button2 = new Button("Button Number 2");

        VBox vbox = new VBox(button1, button2);

        Scene scene = new Scene(vbox, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The result of running the above JavaFX VBox example is an application that looks like this:



## Space Between Nodes

In the earlier example the `VBox` positioned the nodes (button controls) right under the other. You can make the `VBox` insert some space between its nested controls by providing the space in the `VBox` constructor. Here is an example of setting the space between nested controls in an `VBox`:

```
VBox vbox = new VBox(20, button1, button2);
```

This example sets the spacing between the controls in the `VBox` layout component to 20.

You can also set the space between the nested controls using the `setSpacing()` method, like this:

```
vbox.setSpacing(50);
```

This example will set the spacing between nested controls to 50.

Next: [JavaFX Separator](#)

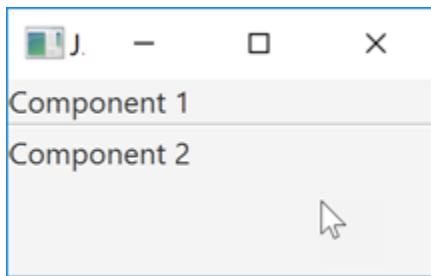
# JavaFX Separator

- [Full JavaFX Separator Example](#)
- [Separator Orientation](#)

Jakob Jenkov

Last update: 2019-07-24

The *JavaFX Separator* component shows a visual divider between groups of components - e.g. between groups of controls inside a [JavaFX VBox](#) or [JavaFX VBox](#). The JavaFX Separator is represented by the class `javafx.scene.control.Separator`. Here is a screenshot of a JavaFX application containing a VBox with a Label, a Separator and a Label:



## Full JavaFX Separator Example

Here is a full JavaFX Separator example to give you an idea about how using it looks in code:

```
import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Separator;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SeparatorExample extends Application {

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

    public void start(Stage primaryStage) {
```

```
Label label1 = new Label("Component 1");
Label label2 = new Label("Component 2");

Separator separator = new Separator(Orientation.HORIZONTAL);

VBox vbox = new VBox(label1, separator, label2);
Scene scene = new Scene(vbox);

primaryStage.setScene(scene);
primaryStage.setTitle("JavaFX App");

primaryStage.show();
}
```

Notice how the `Separator` is passed as second parameter to the `VBox` component, between the first and second `Label`.

## Separator Orientation

You can specify whether the JavaFX `Separator` is supposed to be vertical or horizontal. You do so by passing a parameter to the `Separator` constructor. Here are two examples that set the orientation of the `Separator` created to horizontal and vertical:

```
Separator separator = new Separator(Orientation.HORIZONTAL);

Separator separator = new Separator(Orientation.VERTICAL);
```

Next: [JavaFX FlowPane](#)



# JavaFX FlowPane

- [Creating a FlowPane](#)
- [Adding Children to a FlowPane](#)
- [Adding a FlowPane to the Scene Graph](#)
- [Horizontal and Vertical Spacing](#)
- [Orientation](#)

Jakob Jenkov

Last update: 2016-05-22

A JavaFX `FlowPane` is a layout component which lays out its child components either vertically or horizontally, and which can wrap the components onto the next row or column if there is not enough space in one row. The JavaFX `FlowPane` layout component is represented by the class `javafx.scene.layout.FlowPane`

## Creating a FlowPane

You create a JavaFX `FlowPane` via its constructor. Here is a JavaFX `FlowPane` instantiation example:

```
FlowPane flowpane = new FlowPane();
```

## Adding Children to a FlowPane

You can add children to a `FlowPane` by obtaining its child collection and add adding the components to it you want the `FlowPane` to layout. Here is an example of adding 3 buttons to a `FlowPane`:

```
Button button1 = new Button("Button Number 1");  
Button button2 = new Button("Button Number 2");  
Button button3 = new Button("Button Number 3");  
  
FlowPane flowpane = new FlowPane();  
  
flowpane.getChildren().add(button1);  
flowpane.getChildren().add(button2);
```

```
flowpane.getChildren().add(button3);
```

## Adding a FlowPane to the Scene Graph

To make a `FlowPane` visible you must add it to the JavaFX scene graph. To do so you must add the `FlowPane` instance to a `Scene` object, or add the `FlowPane` to a layout component which is added to a `Scene` object.

Here is an example of adding a JavaFX `FlowPane` to the scene graph:

```
package com.jenkov.javaafx.layouts;

import javafx.application.Application;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class FlowPaneExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("HBox Experiment 1");

        Button button1 = new Button("Button Number 1");
        Button button2 = new Button("Button Number 2");
        Button button3 = new Button("Button Number 3");

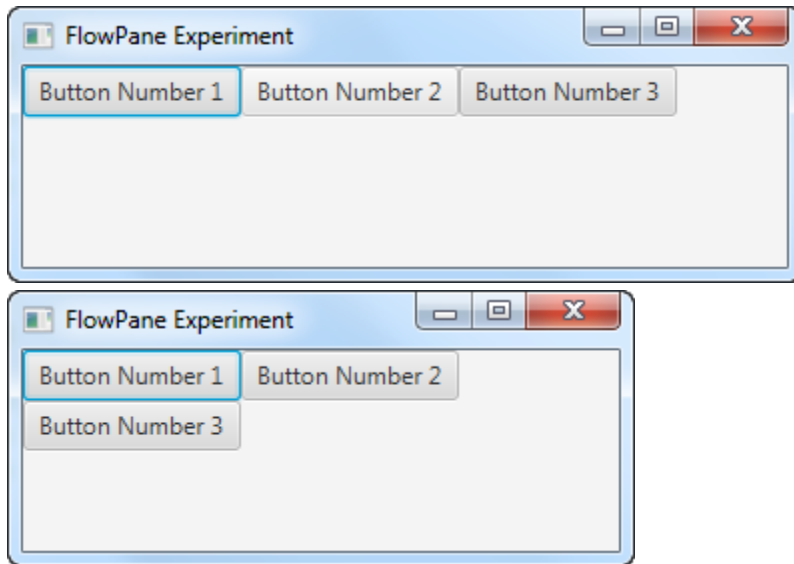
        FlowPane flowpane = new FlowPane();

        flowpane.getChildren().add(button1);
        flowpane.getChildren().add(button2);
        flowpane.getChildren().add(button3);

        Scene scene = new Scene(flowpane, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

The application resulting from this application looks like the following screen shots. Notice how the buttons flow down onto the next horizontal line when the window becomes too small to show them all in a single horizontal row.

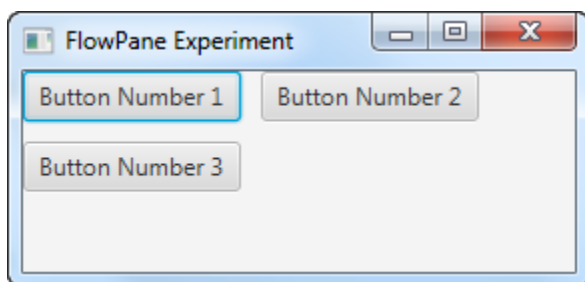


## Horizontal and Vertical Spacing

You can set the horizontal and vertical spacing between the components shown inside a JavaFX `FlowPane` using its `setHGap()` and `setVGap()` methods. Here is an example that shows how to set the horizontal and vertical gap between components in a `FlowPane`:

```
flowpane.setHgap(10);  
flowpane.setVgap(10);
```

When added to the example earlier, the resulting application would look like this:



Notice the horizontal and vertical gaps between the buttons now.

## Orientation

By default the components in a `FlowPane` are layed out horizontally, wrapping onto the next horizontal line when there is no longer space enough inside the `FlowPane` to show more components horizontally.

You can change the flow orientation (direction) of a `FlowPane` using its `setOrientation()` method. You can force the components to be layed out in columns from top to bottom, and then change column when there is no more space in the height to show more components. Here is how you do that:

```
flowpane.setOrientation(Orientation.VERTICAL);
```

Next: [JavaFX TilePane](#)

# JavaFX TilePane

- [Creating a TilePane](#)
- [Adding Children to a TilePane](#)
- [Adding a TilePane to the Scene Graph](#)
- [Horizontal and Vertical Spacing](#)

Jakob Jenkov

Last update: 2016-05-23

A JavaFX `TilePane` is a layout component which lays out its child components in a grid of equally sized cells. The JavaFX `TilePane` layout component is represented by the class `javafx.scene.layout.TilePane`

## Creating a TilePane

You create a JavaFX `TilePane` via its constructor. Here is a JavaFX `TilePane` instantiation example:

```
TilePane tilePane = new TilePane();
```

## Adding Children to a TilePane

You can add children to a `TilePane` by obtaining its child collection and add adding the components to it you want the `TilePane` to layout. Here is an example of adding 6 buttons to a `TilePane`:

```
primaryStage.setTitle("TilePane Experiment");

Button button1 = new Button("Button 1");
Button button2 = new Button("Button Number 2");
Button button3 = new Button("Button No 3");
Button button4 = new Button("Button No 4");
Button button5 = new Button("Button 5");
Button button6 = new Button("Button Number 6");

TilePane tilePane = new TilePane();

tilePane.getChildren().add(button1);
tilePane.getChildren().add(button2);
```

```
tilePane.getChildren().add(button3);
tilePane.getChildren().add(button4);
tilePane.getChildren().add(button5);
tilePane.getChildren().add(button6);

tilePane.setTileAlignment(Pos.TOP_LEFT);
```

## Adding a TilePane to the Scene Graph

To make a `TilePane` visible you must add it to the JavaFX scene graph. To do so you must add the `TilePane` instance to a `Scene` object, or add the `TilePane` to a layout component which is added to a `Scene` object.

Here is an example of adding a JavaFX `TilePane` to the scene graph:

```
package com.jenkov.javafx.layouts;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.TilePane;
import javafx.stage.Stage;

public class TilePaneExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("TilePane Experiment");

        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button Number 2");
        Button button3 = new Button("Button No 3");
        Button button4 = new Button("Button No 4");
        Button button5 = new Button("Button 5");
        Button button6 = new Button("Button Number 6");

        TilePane tilePane = new TilePane();

        tilePane.getChildren().add(button1);
        tilePane.getChildren().add(button2);
        tilePane.getChildren().add(button3);
        tilePane.getChildren().add(button4);
        tilePane.getChildren().add(button5);
        tilePane.getChildren().add(button6);

        tilePane.setTileAlignment(Pos.TOP_LEFT);
```

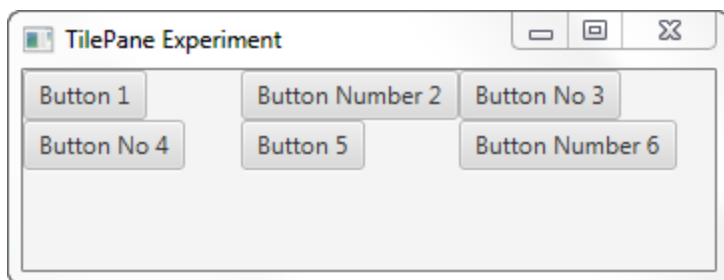
```

        Scene scene = new Scene(tilePane, 200, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The application resulting from this application looks like the following screen shots.



## Horizontal and Vertical Spacing

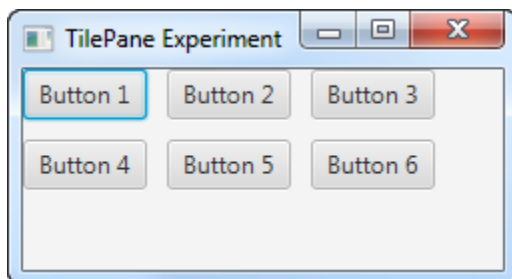
You can set the horizontal and vertical spacing between the components shown inside a JavaFX `TilePane` using its `setHGap()` and `setVGap()` methods. Here is an example that shows how to set the horizontal and vertical gap between components in a `TilePane`:

```

tilePane.setHgap(10);
tilePane.setVgap(10);

```

When added to the example earlier, the resulting application would look like this:



Notice the horizontal and vertical gaps between the buttons. If there were no gaps set on the `TilePane` the buttons would have been positioned next to each other.

Next: [JavaFX GridPane](#)



# JavaFX GridPane

- [Creating a GridPane](#)
- [Adding Children to a GridPane](#)
- [Adding a GridPane to the Scene Graph](#)
- [Spanning Multiple Rows and Columns](#)
- [Horizontal and Vertical Spacing](#)

Jakob Jenkov

Last update: 2016-05-24

A JavaFX GridPane is a layout component which lays out its child components in a grid. The size of the cells in the grid depends on the components displayed in the GridPane, but there are some rules. All cells in the same row will have the same height, and all cells in the same column will have the same width. Different rows can have different heights and different columns can have different widths.

The JavaFX GridPane is different from the [TilePane](#) in that a GridPane allows different size of cells, whereas a TilePane makes all tiles the same size.

The number of rows and columns in a GridPane depends on the components added to it. When you add a component to a GridPane you tell in what cell (row, column) the component should be inserted, and how many rows and columns the component should span.

The JavaFX GridPane layout component is represented by the class `javafx.scene.layout.GridPane`

## Creating a GridPane

You create a JavaFX GridPane via its constructor. Here is a JavaFX GridPane instantiation example:

```
GridPane gridPane = new GridPane();
```

## Adding Children to a GridPane

You can add children to a JavaFX `GridPane` in several ways. The easiest way is to use the `add()` of the `GridPane`. Here is an example of adding 6 buttons to a `GridPane`:

```
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");
Button button3 = new Button("Button 3");
Button button4 = new Button("Button 4");
Button button5 = new Button("Button 5");
Button button6 = new Button("Button 6");

GridPane gridPane = new GridPane();

gridPane.add(button1, 0, 0, 1, 1);
gridPane.add(button2, 1, 0, 1, 1);
gridPane.add(button3, 2, 0, 1, 1);
gridPane.add(button4, 0, 1, 1, 1);
gridPane.add(button5, 1, 1, 1, 1);
gridPane.add(button6, 2, 1, 1, 1);
```

The first parameter of the `add()` method is the component (node) to add to the `GridPane`.

The second and third parameter of the `add()` method is the column index and row index of the cell in which the component should be displayed. Column and row indexes start from 0.

The fourth and fifth parameter of the `add()` method are the row span and column span of the component, meaning how many rows and columns the component should extend to. Row span and column span works similarly to `rowspan` and `colspan` in HTML tables.

## Adding a GridPane to the Scene Graph

To make a JavaFX `GridPane` visible you must add it to the JavaFX scene graph. To do so you must add the `GridPane` instance to a `Scene` object, or add the `GridPane` to a layout component which is added to a `Scene` object.

Here is an example of adding a JavaFX `GridPane` to the scene graph:

```

package com.jenkov.javafx.layouts;

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

public class GridPaneExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("GridPane Experiment");

        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button 2");
        Button button3 = new Button("Button 3");
        Button button4 = new Button("Button 4");
        Button button5 = new Button("Button 5");
        Button button6 = new Button("Button 6");

        GridPane gridPane = new GridPane();

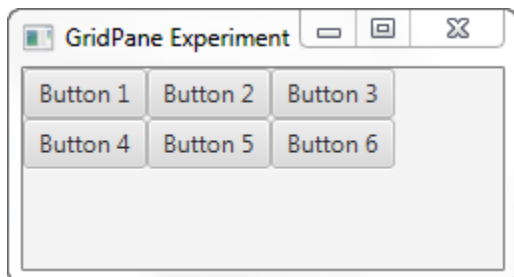
        gridPane.add(button1, 0, 0, 1, 1);
        gridPane.add(button2, 1, 0, 1, 1);
        gridPane.add(button3, 2, 0, 1, 1);
        gridPane.add(button4, 0, 1, 1, 1);
        gridPane.add(button5, 1, 1, 1, 1);
        gridPane.add(button6, 2, 1, 1, 1);

        Scene scene = new Scene(gridPane, 240, 100);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

The application resulting from this application looks like the following screen shots.

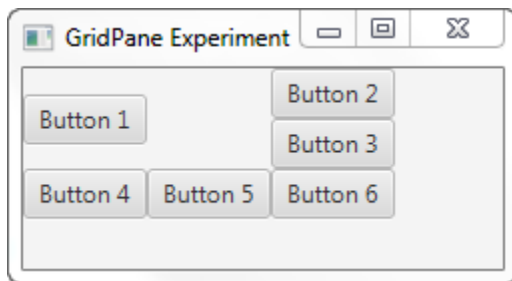


## Spanning Multiple Rows and Columns

To see how to make a component span multiple columns and rows, look at this modification of the 6 buttons added to the `GridPane`:

```
gridPane.add(button1, 0, 0, 2, 2);  
  
gridPane.add(button2, 2, 0, 1, 1);  
gridPane.add(button3, 2, 1, 1, 1);  
gridPane.add(button4, 0, 2, 1, 1);  
gridPane.add(button5, 1, 2, 1, 1);  
gridPane.add(button6, 2, 2, 1, 1);
```

Notice how the first button added is given a column span and row span of 2. Notice how the rest of the buttons are added outside of the top left 2 x 2 columns. The layout resulting from these settings looks like this:

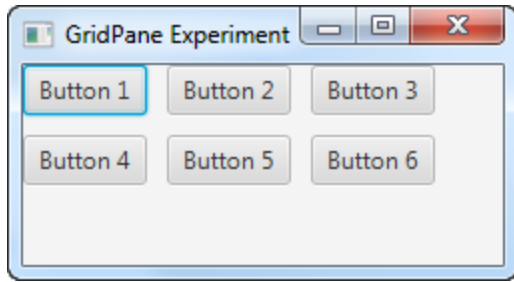


## Horizontal and Vertical Spacing

You can set the horizontal and vertical spacing between the components shown inside a JavaFX `GridPane` using its `setHGap()` and `setVGap()` methods. Here is an example that shows how to set the horizontal and vertical gap between components in a `GridPane`:

```
gridPane.setHgap(10);  
gridPane.setVgap(10);
```

When added to the example earlier, the resulting application would look like this:



Notice the horizontal and vertical gaps between the buttons. If there were no gaps set on the `GridPane` the buttons would have been positioned next to each other.

Next: [JavaFX MenuBar](#)

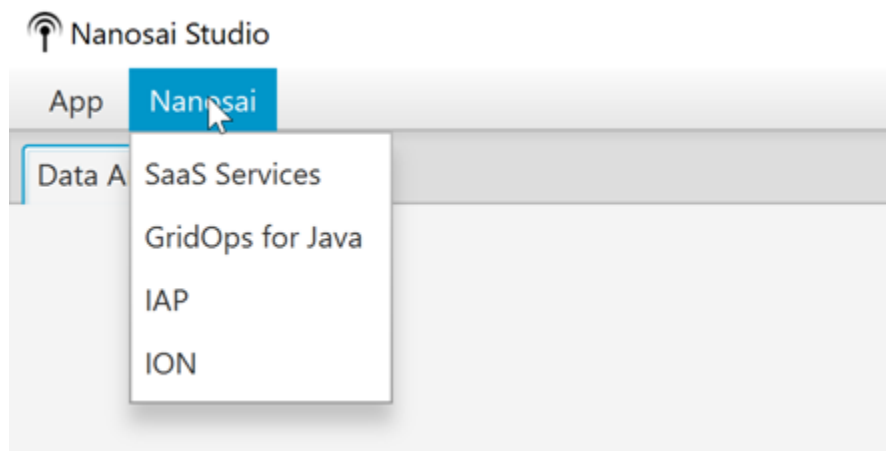
# JavaFX MenuBar

- [Creating a MenuBar Instance](#)
- [Adding a MenuBar to the Scene Graph](#)
- [Creating Menu Instances](#)
- [Menu Graphics](#)
- [Menu Events](#)
- [Adding Menu Items](#)
- [MenuItem Graphics](#)
- [MenuItem Events](#)
- [Submenus](#)
- [Check Menu Items](#)
- [Radio Menu Item](#)
- [Menu Item Separators](#)
- [Custom Control Menu Items](#)

Jakob Jenkov

Last update: 2018-02-25

The *JavaFX MenuBar* provides JavaFX applications with a visual drop down menu similar to that most desktop applications have at the top of their application window. The JavaFX MenuBar is represented by the class `javafx.scene.control.MenuBar`. Here is an example screenshot of what a JavaFX MenuBar can look like:



## Creating a MenuBar Instance

Before you can use the JavaFX MenuBar you must create a MenuBar instance. Here is an example of creating a JavaFX MenuBar instance:

```
MenuBar menuBar = new MenuBar();
```

## Adding a MenuBar to the Scene Graph

Before a MenuBar becomes visible you will have to add it to the JavaFX scene graph. Here is an example of adding a JavaFX MenuBar to the scene graph:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class JavaFXApp extends Application {

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

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

        MenuBar menuBar = new MenuBar();

        VBox vBox = new VBox(menuBar);

        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Notice how the MenuBar is added to the root layout (VBox) of the JavaFX scene. This places the MenuBar at the top of the application window.

Note that the above example does not add any menus or menu items to the `MenuBar`, so if you run the example you will not actually see the `MenuBar`. We will see how to add menus and menu items in the following sections.

## Creating Menu Instances

Once the `MenuBar` instance is created, you can add `Menu` instances to it (`javafx.scene.control.Menu`). A `Menu` instance represents a single vertical menu with nested menu items. Thus, you can add multiple `MenuBar` instances to a `MenuBar` to add multiple vertical drop down menus.

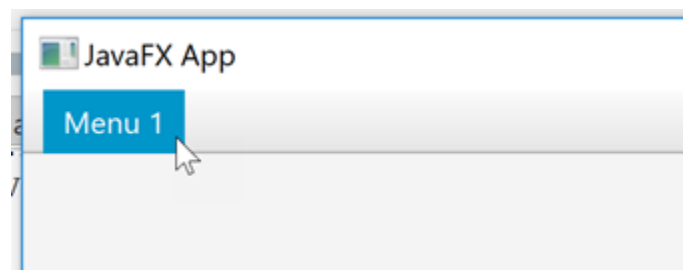
Here is an example of adding a `Menu` to a `MenuBar` :

```
Menu menu1 = new Menu("Menu 1");

MenuBar menuBar = new MenuBar();

menuBar.getMenus().add(menu1);
```

Here is a screenshot showing the JavaFX `MenuBar` as configured by the example code above:



As you can see, there is only a single menu in the `MenuBar` titled "Menu 1". This menu has no menu items nested under it. We will see how to add menu items to a `Menu` in the following sections.

## Menu Graphics

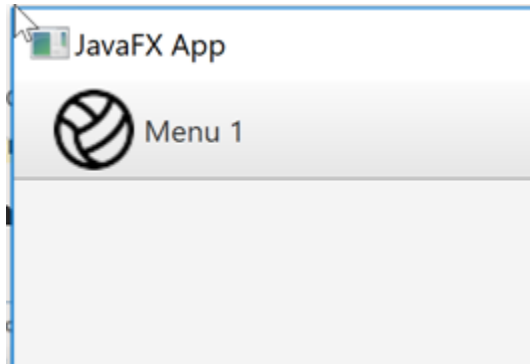
You can set a graphic icon for a `Menu` by calling its `setGraphic()` method. The graphic icon will be displayed next to the



text label of the menu. Here is an example of setting a graphic icon for a JavaFX Menu instance:

```
Menu menu = new Menu("Menu 1");  
menu.setGraphic(new ImageView("file:volleyball.png"));
```

Here is how the resulting menu could look in a JavaFX application:



## Menu Events

A JavaFX Menu instance can fire several events which you can listen for in your application. The most commonly used events are:

- `onShowing`
- `onShown`
- `onHiding`
- `onHidden`

When a Menu is clicked with the mouse it shows its contents. This action fires the event `onShowing` before the Menu starts showing its menu items. Once the menu is fully visible the `onShown` event is fired.

When an shown (open) Menu is clicked with the mouse it hides its contents again. This action fires the event `onHiding` before the Menu starts hiding its menu items. Once the menu is fully hidden the `onHidden` event is fired.

You can set Menu event listeners for the events above using the methods `setOnShowing()`, `setOnShown()`, `setOnHiding()` and

d `setOnHidden()`. Here is an example of setting event listeners for these events on a JavaFX Menu :

```
Menu menu = new Menu("Menu 1");

menu.setOnShowing(e -> { System.out.println("Showing Menu 1"); });
menu.setOnShown(e -> { System.out.println("Shown Menu 1"); });
menu.setOnHiding(e -> { System.out.println("Hiding Menu 1"); });
menu.setOnHidden(e -> { System.out.println("Hidden Menu 1"); });
```

The Menu event listeners set above only print out a message to the console when the events fired. You could do something more advanced in case you needed to.

## Adding Menu Items

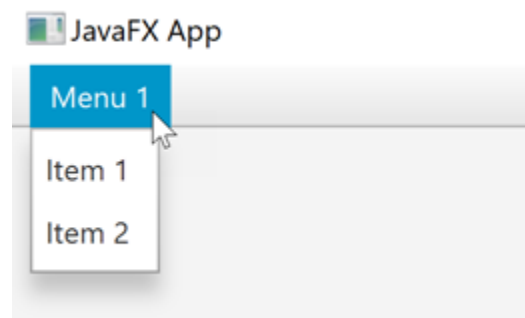
Once you have created a Menu instance you must add one or more MenuItem instances to it. Each MenuItem corresponds to a menu item in the menu it is added to. Here is an example of adding 2 MenuItem instances to a Menu, which is then added to a MenuBar:

```
Menu menu = new Menu("Menu 1");
MenuItem menuItem1 = new MenuItem("Item 1");
MenuItem menuItem2 = new MenuItem("Item 2");

menu.getItems().add(menuItem1);
menu.getItems().add(menuItem2);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

Here is what the resulting JavaFX MenuBar would look like, if used in a JavaFX application:



## MenuItem Graphics

You can add an icon to a menu item. You add a graphic icon to a `MenuItem` by calling its `setGraphic()` method, passing as parameter the graphic you want to use for the given `MenuItem`. Here is an example that adds images to the menu items created in the example in the previous section:

```
Menu menu = new Menu("Menu 1");

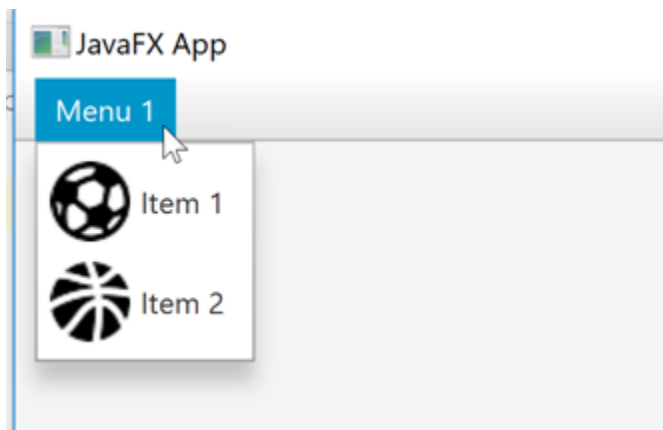
MenuItem menuItem1 = new MenuItem("Item 1");
menuItem1.setGraphic(new ImageView("file:soccer.png"));

MenuItem menuItem2 = new MenuItem("Item 2");
menuItem2.setGraphic(new ImageView("file:basketball.png"));

menu.getItems().add(menuItem1);
menu.getItems().add(menuItem2);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

Here is how a JavaFX `MenuBar` looks with graphic icons added to its menu items:



## MenuItem Events

The `MenuBar` configurations created in the previous examples do not react if you select any of the menu items. In order to respond to the selection of a `MenuItem` you must set an event listener on the `MenuItem`. Here is an example of adding an event listener to a JavaFX `MenuItem`:

```
MenuItem menuItem1 = new MenuItem("Item 1");

menuItem1.setOnAction(e -> {
    System.out.println("Menu Item 1 Selected");
});
```

Notice the [Java Lambda](#) added as parameter to the `setOnAction()` method of the `MenuItem`. This lambda expression is executed when the menu item is selected.

## Submenus

The JavaFX `MenuBar` supports multiple layers of menus. A menu nested inside another menu is called a submenu. The `Menu` class extends the `MenuItem` class and can therefore be used as a menu item inside another `Menu` instance. Here is an example that creates a single JavaFX menu with a submenu inside:

```
Menu menu = new Menu("Menu 1");

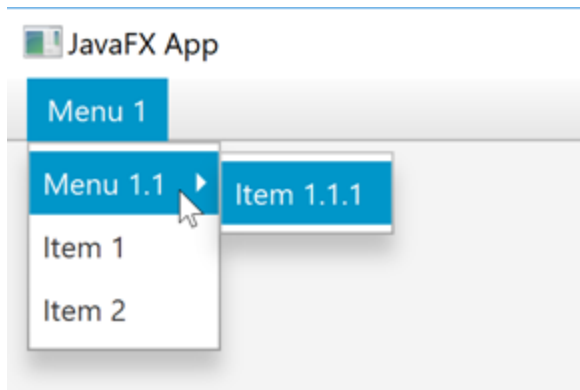
Menu subMenu = new Menu("Menu 1.1");
MenuItem menuItem11 = new MenuItem("Item 1.1.1");
subMenu.getItems().add(menuItem11);
menu.getItems().add(subMenu);

MenuItem menuItem1 = new MenuItem("Item 1");
menu.getItems().add(menuItem1);

MenuItem menuItem2 = new MenuItem("Item 2");
menu.getItems().add(menuItem2);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

The JavaFX `MenuBar` resulting from the above example will look similar to this:



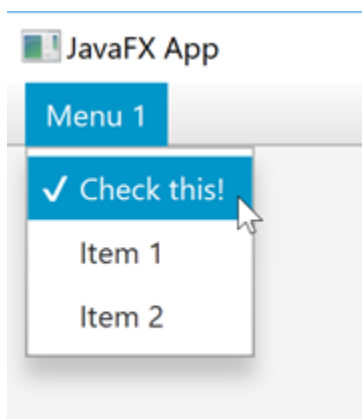
## Check Menu Items

The JavaFX `MenuBar` supports using check menu items in a menu. A check menu item is a menu item that can be "selected" and remain selected until unselected later. A small check mark is displayed next to the check menu item as long as it remains selected.

The check menu item is represented by the `CheckMenuItem` (`javafx.scene.control.CheckMenuItem`) class. Here is an example of a JavaFX menu with a `CheckMenuItem` in:

```
CheckMenuItem checkMenuItem = new CheckMenuItem("Check this!");  
  
menu.getItems().add(checkMenuItem);
```

The `Menu` instance then need to be added to a `MenuBar` to be visible, as you have seen in earlier examples. Here is how the resulting menu looks, with the check menu menu item checked:



## Radio Menu Item

The JavaFX `MenuBar` also supports radio menu items. Radio menu items are menu items of which only one of a set of menu items can be selected - just like standard [JavaFX radio buttons](#).

The radio menu item is represented by the `RadioMenuItem`.

The `RadioMenuItem` instance must be added to a `ToggleGroup` to make them mutually exclusive. That is how JavaFX knows which `RadioMenuItem` instance belong together. Here is an example of a JavaFx menu that uses a set of radio menu items:

```
Menu menu = new Menu("Menu 1");

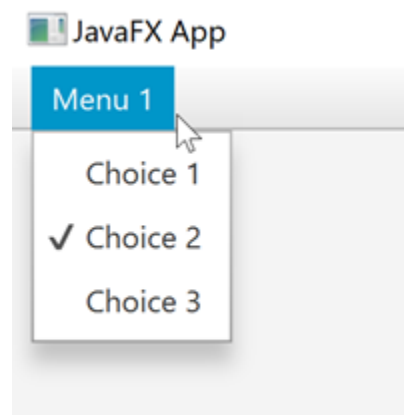
RadioMenuItem choice1Item = new RadioMenuItem("Choice 1");
RadioMenuItem choice2Item = new RadioMenuItem("Choice 2");
RadioMenuItem choice3Item = new RadioMenuItem("Choice 3");

ToggleGroup toggleGroup = new ToggleGroup();
toggleGroup.getToggles().add(choice1Item);
toggleGroup.getToggles().add(choice2Item);
toggleGroup.getToggles().add(choice3Item);

menu.getItems().add(choice1Item);
menu.getItems().add(choice2Item);
menu.getItems().add(choice3Item);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

Here is how the JavaFx menu resulting from this example code looks:



## Menu Item Separators

The `MenuBar` supports menu item separators. A separator is a horizontal line that separates groups of menu items. A separator is often used to signal to users what menu items are related to each other.

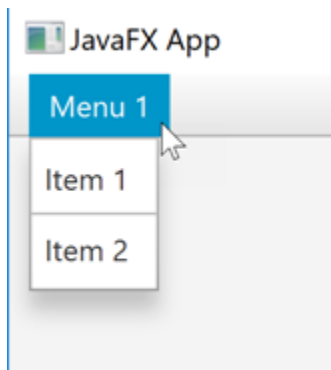
Menu item separators are represented by the `SeparatorMenuItem` class. Here is an example of a menu with two menu items separated by a `SeparatorMenu`:

```
MenuItem item1 = new MenuItem("Item 1");
MenuItem item2 = new MenuItem("Item 2");
SeparatorMenuItem separator = new SeparatorMenuItem();

menu.getItems().add(item1);
menu.getItems().add(separator);
menu.getItems().add(item2);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

Here is how the resulting JavaFX menu would look like:



## Custom Control Menu Items

The JavaFX `MenuBar` also supports using custom JavaFX controls as menu items. To do so you need to use the `CustomMenuItem` (`javafx.scene.control.CustomMenuItem`) class.

The `CustomMenuItem` class has a `setContent()` method which you can use to set the custom JavaFX control to show in the menu. Here is an

example that shows both a [JavaFX Button](#) and a JavaFX Slider as custom menu items:

```
Menu menu = new Menu("Menu 1");

Slider slider = new Slider(0, 100, 50);

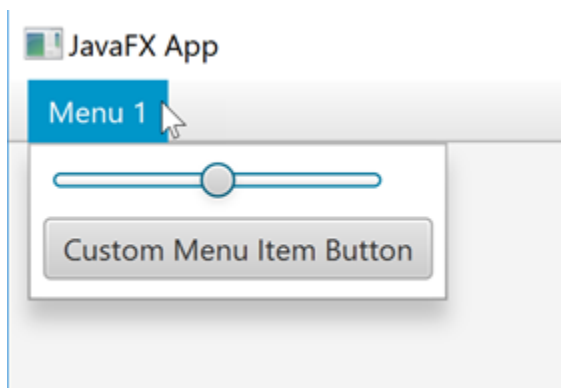
CustomMenuItem customMenuItem = new CustomMenuItem();
customMenuItem.setContent(slider);
customMenuItem.setHideOnClick(false);
menu.getItems().add(customMenuItem);

Button button = new Button("Custom Menu Item Button");
CustomMenuItem customMenuItem2 = new CustomMenuItem();
customMenuItem2.setContent(button);
customMenuItem2.setHideOnClick(false);
menu.getItems().add(customMenuItem2);

MenuBar menuBar = new MenuBar();
menuBar.getMenus().add(menu);
```

Notice the call to `CustomMenuItem setHideOnClick()` with the value `false` as parameter. This is done to keep the menu open while the user interacts with the custom menu item control. If you set the value to `true` the menu will close as soon as the user clicks the control the first time, making further interaction impossible. For normal menu items you actually do want the menu to close immediately, but for some custom menu items you may not want that. The menu can still be closed by clicking on the menu title again.

Here is how the resulting menu looks:



Next: [JavaFX ContextMenu](#)



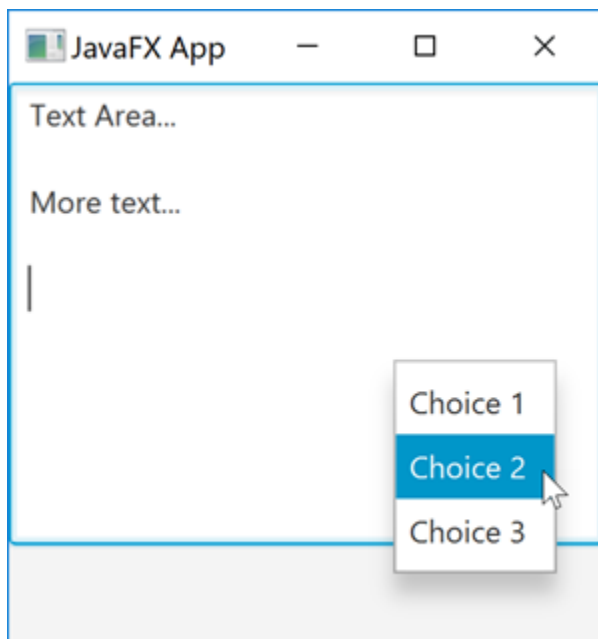
# JavaFX ContextMenu

- [Full ContextMenu Example](#)
- [Create a ContextMenu](#)
- [Add Menu Items to the ContextMenu](#)
- [Add ContextMenu to JavaFX Control](#)

Jakob Jenkov

Last update: 2019-07-21

The *JavaFX ContextMenu* component provides a standard right click menu for JavaFX controls. The JavaFX ContextMenu is represented by the class `javafx.scene.control.ContextMenu`. You create a ContextMenu instance and attach it to the JavaFX control you want the ContextMenu to be active for. Here is a screenshot of a [JavaFX TextArea](#) with a ContextMenu attached and shown:



## Full ContextMenu Example

Here is a full JavaFX ContextMenu example to give you a quick overview of how using the ContextMenu looks:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ContextMenuExample extends Application {

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

    public void start(Stage primaryStage) {

        ContextMenu contextMenu = new ContextMenu();
        MenuItem menuItem1 = new MenuItem("Choice 1");
        MenuItem menuItem2 = new MenuItem("Choice 2");
        MenuItem menuItem3 = new MenuItem("Choice 3");

        menuItem3.setOnAction((event) -> {
            System.out.println("Choice 3 clicked!");
        });

        contextMenu.getItems().addAll(menuItem1, menuItem2, menuItem3);

        TextArea textArea = new TextArea();

        textArea.setContextMenu(contextMenu);

        VBox vBox = new VBox(textArea);
        Scene scene = new Scene(vBox);

        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App");

        primaryStage.show();
    }
}

```

## Create a ContextMenu

To use a JavaFX ContextMenu you must first create a ContextMenu instance. Here is an example of creating a JavaFX ContextMenu:

```

ContextMenu contextMenu = new ContextMenu();

```

## Add Menu Items to the ContextMenu

The `ContextMenu` needs one or more `MenuItem` instances which are displayed in the menu. To detect which menu item was clicked, you add listeners to the `MenuItem` instances. Here is an example of adding `MenuItem` instances to a `ContextMenu`:

```
ContextMenu contextMenu = new ContextMenu();

MenuItem menuItem1 = new MenuItem("Choice 1");
MenuItem menuItem2 = new MenuItem("Choice 2");
MenuItem menuItem3 = new MenuItem("Choice 3");

menuItem3.setOnAction((event) -> {
    System.out.println("Choice 3 clicked!");
});

contextMenu.getItems().addAll(menuItem1, menuItem2, menuItem3);
```

Notice how there is an `onAction` listener set on the third `MenuItem`, using `setOnAction()`, passing [Java Lambda Expression](#) as listener. If this `MenuItem` is clicked, the action listener is executed.

## Add ContextMenu to JavaFX Control

A JavaFX `ContextMenu` needs to be attached to a JavaFX control to be active. You add a `ContextMenu` to a control via the `setContextMenu()` of a JavaFX control. Here is an example of adding a JavaFX `ContextMenu` to a JavaFX control:

```
TextArea textArea = new TextArea();

textArea.setContextMenu(contextMenu);
```

Next: [JavaFX WebView](#)

# JavaFX WebView

- [JavaFX WebView Example](#)
- [WebView WebEngine](#)
  - [Obtaining the WebEngine](#)
  - [Load a Web Page](#)
  - [Load Local Content](#)
  - [Reload Content](#)
- [WebView Zoom](#)
- [WebView Font Scale](#)
- [Set User-Agent HTTP Header](#)
- [Disable WebView Context Menu](#)
- [Browsing History](#)
  - [Browsing History Entries](#)
  - [Iterate Browsing History Entries](#)
  - [WebHistory.Entry](#)
  - [Go Forward and Backward in History](#)
  - [Current History Entry Index](#)
- [Listening for State Changes When Loading Document](#)
- [Execute JavaScript From Java](#)
  - [Executing From a WebEngine Listener](#)
  - [JavaScript Return Values](#)
- [Executing Java From JavaScript](#)
- [Access the DOM](#)
- [Web Page CSS Style Sheet](#)
- [WebView CSS Styles](#)

Jakob Jenkov

Last update: 2018-10-22

The JavaFX *WebView* (`javafx.scene.web.WebView`) component is capable of showing web pages (HTML, CSS, SVG, JavaScript) inside a JavaFX application. As such, the JavaFX *WebView* is a mini browser.

The `WebView` component is very handy when you need to show documentation (e.g. Help texts), news, blog posts or other content which needs to be downloaded from a web server at runtime.

The JavaFX `WebView` uses the WebKit open source browser engine internally to render the web pages.

## JavaFX WebView Example

The `WebView` component is a JavaFX `Node` so it can be included in the scene graph like any other JavaFX component which is also a `Node`. Here is a simple JavaFX `WebView` example:

```
package com.jenkov.javafx;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class WebViewExample extends Application {

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

    public void start(Stage primaryStage) {
        primaryStage.setTitle("JavaFX WebView Example");

        WebView webView = new WebView();

        webView.getEngine().load("http://google.com");

        VBox vBox = new VBox(webView);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

This example shows a JavaFX application that creates a `WebView` which is inserted into a [JavaFX VBox](#) layout component - which is again placed inside a JavaFX `Stage` - which is set on the primary `Stage`.

# WebView WebEngine

The JavaFX `WebView WebEngine (javafx.scene.web.WebEngine)` is an internal component used by the `WebView` to load the data that is to be displayed inside the `WebView`. To make the `WebView WebEngine` load data, you must first obtain the `WebEngine` instance from the `WebView`.

## Obtaining the WebEngine

You obtain the `WebEngine` from the `WebView` by calling the `WebView getEngine()` method. Here is an example of obtaining the `WebEngine` from a JavaFX `WebView`:

```
WebView webView = new WebView();

WebEngine webEngine = webView.getEngine();
```

## Load a Web Page

Once you have obtained a `WebEngine` instance from the `WebView` you can load data by calling its `load()` method. The `load()` method takes a URL as parameter. Here is an example of loading a web page via the `WebEngine load()` method:

```
webEngine.load("http://google.com");
```

## Load Local Content

The `WebView WebEngine` can load local content too - meaning content that is supplied to it directly in a method call (not loaded over the Internet). The `WebEngine` loads local content via the `loadContent()` method. Here is an example of loading local content in a JavaFX `WebView` using `loadContent()`

```
String content =
    "Hello World!";
```

```
webEngine.loadContent(content, "text/html");
```

The first parameter to the `loadContent()` call in the example above is the content itself. In this example it is a very simple HTML document contained in a Java String.

The second parameter to the `loadContent()` call in the example above is the content type (mime type) of the content. Since we are loading an HTML document, the standard content type for that is `text/html`.

## Reload Content

It is possible to reload the content currently loaded in a JavaFX `WebView`. You do so using the `WebEngine reload()` method. Here is an example of reloading content in a JavaFX `WebView` using the `WebEngine reload()` method:

```
webEngine.reload();
```

## WebView Zoom

It is possible to set the zoom level of a JavaFX `WebView`. For instance, you can specify that the `WebView` should always zoom in 25%, or zoom out 10% etc. Zooming in scales up or down all of the content displayed inside the `WebView`. You set the `WebView` zoom level via the `setZoom()` method. Here is an example of setting the JavaFX `WebView` zoom level:

```
webView.setZoom(1.25); //zoom in 25%.
```

The `setZoom()` method takes a double value. A value of 1.0 means a zoom level of 100% which means no zoom. A value of 0.5 means a zoom level of 50%, which means zoom out to 50% of original size. In the example above, the value of 1.25 means a zoom level of 125%, meaning zoom in until a size of 125% of the original size.

## WebView Font Scale

It is also possible to only scale the text displayed inside a JavaFX `WebView` without scaling any of the non-text content (e.g. images) displayed inside the `WebView`. You set the font scaling property via the `setFontSize()` method. Here is an example of scaling up the text displayed in a JavaFX `WebView` by 25% (125% total):

```
webView.setFontScale(1.25);
```

The `setFontSize()` method takes a `double` parameter which specifies the font scale value. A scale value of 1.0 means no scaling up or down. A value of 0.5 means scaling down to half size, and a value of 2.0 means scaling up to double size.

## Set User-Agent HTTP Header

You can set the `User-Agent` HTTP header sent to the web servers your `WebView` instance is loading web pages from. You set the `User-Agent` HTTP header via the `WebEngine setUserAgent()` method. Here is an example of setting the `User-Agent` HTTP header of a `WebView`:

```
webEngine.setUserAgent("MyApp Web Browser 1.0");
```

You do not need to set the `User-Agent` HTTP Header, but you might be interested in having your particular app show up as a separate browser. You might want to include the version of WebKit your application is using. This gives web servers a better chance of optimizing their website for their visitor browsers, including yours. Here is an example of including the JavaFX and WebKit version in the `User-Agent`:

```
webEngine.setUserAgent("MyApp Web Browser 1.0 - AppleWebKit/555.99 JavaFX 8.0");
```

## Disable WebView Context Menu



A JavaFX `WebView` has a default context menu (right click menu) which is displayed when you right click (context click) on the `WebView` in the JavaFX application. You can disable the `WebView` context menu by calling the `WebView` `setContextMenuEnabled()` method with a parameter value of `false`. Here is an example of disabling the context menu of a JavaFX `WebView`:

```
webView.setContextMenuEnabled(false);
```

## Browsing History

You can access the browsing history of a JavaFX `WebView`. The browsing history consists of the pages the user has visited while browsing inside the `WebView`. You access the browsing history of a JavaFX `WebView` via its `WebEngine` object. You call the `WebEngine` `getHistory()` method, and you get a `WebHistory` object back. Here is an example of obtaining the `WebHistory` object from a `WebView` `WebEngine` object:

```
WebEngine webEngine = webView.getEngine();  
WebHistory history = webEngine.getHistory();
```

Once you have access to the `WebHistory` object, you can start inspecting and manipulating the browsing history. We will see how in the following sections.

### Browsing History Entries

You can access the browsing history entries kept inside a `WebHistory` object by calling its `getEntries()` method. Here is an example of obtaining a list of the browsing history entries from a `WebHistory` object:

```
ObservableList<WebHistory.Entry> entries = history.getEntries();
```

The list returned is a list of `WebHistory.Entry` objects which can be inspected for information about each entry.

## Iterate Browsing History Entries

You can access the browsing history entries (the pages visited) kept inside a `WebHistory` object. Here is an example of iterating through all the browsing history entries of a JavaFX `WebHistory` object:

```
Iterator<WebHistory.Entry> iterator = entries.iterator();
while(iterator.hasNext()){
    WebHistory.Entry entry = iterator.next();
}
```

You can also iterate the browsing history entries using a for-each loop, like this:

```
for(WebHistory.Entry entry : entries){
    //do something with the entry
}
```

## WebHistory.Entry

A `WebHistory.Entry` contains the following information:

- URL
- Title
- Last visited date

You can access the URL, title and last visited date from a `WebHistory.Entry` using the following methods:

```
String url          = entry.getUrl();
String title        = entry.getTitle();
Date lastVisitedDate = entry.getLastVisitedDate();
```

## Go Forward and Backward in History

Once you have an instance of the `WebHistory` object you can actually manipulate the history. You can force the `WebView` to go forward and back in the browsing history. You do so by calling the `WebHistory.go()` method. Here are two examples of going forward and backward in the browsing history:

```
history.go(-1);  
history.go( 1);
```

If you pass a negative integer to the `go()` method the browser will move backward in the browsing history to the page (URL) visited just before the currently displayed page.

If you pass a positive integer to the `go()` method, you will make the `WebView` go one entry forward in the browsing history. That only works if the user has already visited at least 2 pages, and gone back from the first page.

## Current History Entry Index

If you have moved back and forth a bit in the browsing history, you might be interested in seeing what index in the browsing history the current history entry has. You can see the index of the current browsing history entry via the method `WebHistory.getCurrentIndex()` method. Here is an example of reading the index of the current browsing history entry:

```
int currentIndex = history.getCurrentIndex();
```

## Listening for State Changes When Loading Document

When you tell the `WebEngine` to load a document, the document is loaded in the background via another thread. You can listen for changes in the document loading status, so you can be notified when the document has finished loading. Here is an example of listening for document load status changes of a `WebView WebEngine`:

```
webEngine.getLoadWorker().stateProperty().addListener(  
    new ChangeListener() {  
        @Override  
        public void changed(ObservableValue observable, Object oldValue,  
Object newValue) {  
            System.out.println("oldValue: " + oldValue);  
            System.out.println("newValue: " + newValue);  
  
            if (newValue == Worker.State.SUCCEEDED) {  
                //document finished loading  
            }  
        }  
    })
```

```
}  
);
```

## Execute JavaScript From Java

It is possible to execute JavaScript embedded in the HTML page displayed inside a JavaFX `WebView` from Java. You execute JavaScript in a `WebView` from Java via the `WebEngine executeScript()` method. Here is a simple example of executing JavaScript embedded in a `WebView` from Java code:

```
webEngine.executeScript("myFunction()");
```

The `executeScript()` method takes a [Java String](#) as parameter which contains the JavaScript to execute. The example above calls a JavaScript function named `myFunction()`. This function has to be defined inside the web page displayed in the `WebView` the above `WebEngine` to.

## Executing From a WebEngine Listener

So far I have had some problems executing JavaScript functions from Java, unless I call `executeScript()` from within a `WebEngine` listener. Here is an example of executing JavaScript from Java from within a `WebEngine` listener:

```
webEngine.getLoadWorker().stateProperty().addListener(  
    new ChangeListener() {  
        @Override  
        public void changed(ObservableValue observable, Object oldValue,  
Object newValue) {  
            System.out.println("oldValue: " + oldValue);  
            System.out.println("newValue: " + newValue);  
  
            if (newValue != Worker.State.SUCCEEDED) {  
                return;  
            }  
            System.out.println("Succeeded!");  
            String hello = (String) webEngine.executeScript("myFunction()");  
            System.out.println("hello: " + hello);  
        }  
    }  
);
```

If you know what causes the problem, or why this "limitation" exists, I would appreciate an email explaining it :-)

## JavaScript Return Values

If a JavaScript function returns a value, that value will be converted to a Java data type and returned from the `executeScript()` method. Imagine that the `myFunction()` JavaScript function returns a `String`. In that case, a Java `String` would be returned from the `executeScript()` method. Here is how catching that `String` would look:

```
String returnValue = (String) webEngine.executeScript("myFunction()");
```

The following table shows what Java types various JavaScript return types are converted to:

JavaScript Type	Java Type
null	null
boolean	Boolean
int32	Integer
number	Double
string	String
object	JSObject(netscape.javascript.JSObject)

## Executing Java From JavaScript

It is also possible to call Java code from JavaScript running inside a `JavaFX WebView`. In order to do that you must make a Java object available to the JavaScript running inside the `WebView`. The easiest way to do that is to set the Java object as a member of the `window` object in the document displayed in the `WebView`. Here is how that is done:

```
webEngine.getLoadWorker().stateProperty().addListener(  
    new ChangeListener() {  
        @Override  
        public void changed(ObservableValue observable, Object oldValue,  
Object newValue) {  
            if (newValue != Worker.State.SUCCEEDED) { return; }  

```

```

        JSONObject window = (JSONObject) webEngine.executeScript("window");
        window.setMember("myObject", new MyObject());
    }
};

```

The example above first gets access to the `window` object via the `WebEngine executeScript()` method. Second, the example sets an instance of the `MyObject` class as member on the `window` object. JavaScript running inside the `WebView` can now call methods on this object, as if it was a JavaScript object.

The `MyObject` class looks like this:

```

public static class MyObject {
    public void doIt() {
        System.out.println("doIt() called");
    }
}

```

Once an object of this class has been exposed as a member of the `window` object named `myObject`, you can call its `doIt()` method like this:

```

window.myObject.doIt();

```

Please keep in mind that when the document first loads in the `WebView`, the `myObject` is not yet exposed on the `window` object. Therefore, if you try to call a method on it immediately, it may fail.

## Access the DOM

You can access the DOM of the web page displayed inside a `JavaFX WebView` by calling the `WebEngine getDocument()` method. Here is an example of accessing the DOM of a `WebView`:

```

Document document = webEngine.getDocument();

```

The `Document` object returned is  
a `org.w3c.dom.Document` instance.

## Web Page CSS Style Sheet

Normally a web page provides its own CSS style sheet. However, in case a web page has no CSS style sheet you can set a CSS style sheet for it using the `WebEngine.setUserStyleSheetLocation()`. Here is an example of setting the CSS style sheet for a web page using `WebEngine.setUserStyleSheetLocation()`;

```
webEngine.setUserStyleSheetLocation("stylesheet.css");
```

The String passed as parameter to the `setUserStyleSheetLocation()` method should be the path in the file system to where the CSS style sheet file is located,

## WebView CSS Styles

It is possible to style a JavaFX `WebView` component with CSS, just like you can style any other JavaFX component. You can set the following CSS properties for a JavaFX `WebView`:

CSS Property	Description
<code>-fx-context-menu-enabled</code>	Accepts the value of <code>true</code> or <code>false</code> - which specifies whether or not the context menu (right click menu) is enabled or not.
<code>-fx-font-smoothing-type</code>	Specifies the kind of font smoothing to apply.
<code>-fx-font-scale</code>	A decimal number (e.g. <code>1.4</code> ) setting the font scale of this <code>WebView</code>

More coming soon...

Next: [JavaFX PieChart](#)

# JavaFX PieChart

- [Creating a PieChart](#)
- [Adding Data to a PieChart](#)
- [Adding a PieChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-25

The JavaFX PieChart component is capable of drawing pie charts in your JavaFX application based on data you supply it. The PieChart component is really easy to use. The JavaFX PieChart component is represented by the class `javafx.scene.chart.PieChart`.

## Creating a PieChart

You create a JavaFX PieChart component by creating an instance of the `PieChart` class. Here is a JavaFX `PieChart` instantiation example:

```
PieChart pieChart = new PieChart();
```

## Adding Data to a PieChart

To display anything you must add data to the `PieChart`. Pie chart data is represented by the `PieChart.Data` class. Each slice in the pie chart is represented by one `PieChart.Data` instance. Here is an example of adding data to a JavaFX `PieChart` component:

```
PieChart pieChart = new PieChart();

PieChart.Data slice1 = new PieChart.Data("Desktop", 213);
PieChart.Data slice2 = new PieChart.Data("Phone", 67);
PieChart.Data slice3 = new PieChart.Data("Tablet", 36);

pieChart.getData().add(slice1);
pieChart.getData().add(slice2);
pieChart.getData().add(slice3);
```



## Adding a PieChart to the Scene Graph

In order to make a JavaFX `PieChart` component visible it must be added to the JavaFX scene graph. This means adding the `PieChart` instance to a `Scene` object, or adding it to a layout component which is added to a `Scene` object.

Here is a full example of adding a `PieChart` to the JavaFX scene graph:

```
package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class PieChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("My First JavaFX App");

        PieChart pieChart = new PieChart();

        PieChart.Data slice1 = new PieChart.Data("Desktop", 213);
        PieChart.Data slice2 = new PieChart.Data("Phone", 67);
        PieChart.Data slice3 = new PieChart.Data("Tablet", 36);

        pieChart.getData().add(slice1);
        pieChart.getData().add(slice2);
        pieChart.getData().add(slice3);

        VBox vbox = new VBox(pieChart);

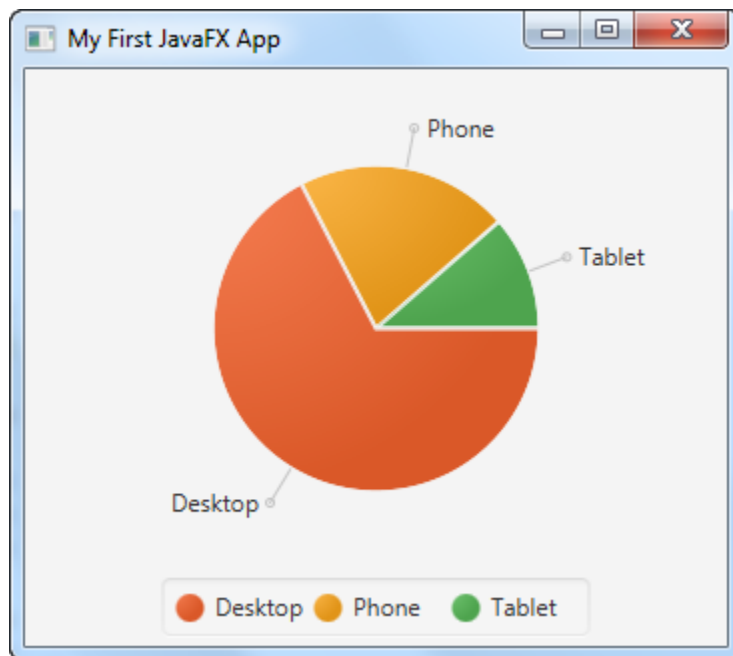
        Scene scene = new Scene(vbox, 400, 200);

        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

The application resulting from running the above code would look similar to this:



Next: [JavaFX TableView](#)

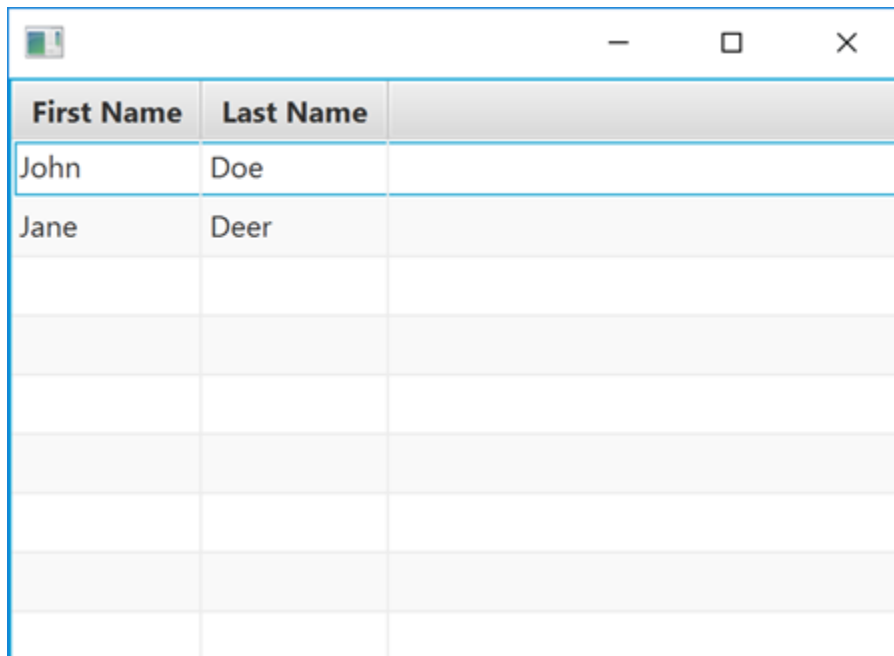
# JavaFX TableView

- [JavaFX TableView Example](#)
- [Create a TableView](#)
- [Add TableColumn to the TableView](#)
  - [TableColumn Cell Value Factory](#)
- [Add Data to TableView](#)
- [Set Placeholder When No Rows to Display](#)

Jakob Jenkov

Last update: 2019-08-24

The JavaFX *TableView* enables you to display table views inside your JavaFX applications. The JavaFX *TableView* is represented by the class `javafx.scene.control.TableView`. Here is a screenshot of a JavaFX *TableView*:



First Name	Last Name	
John	Doe	
Jane	Deer	

## JavaFX TableView Example

Here is a full, but simple JavaFX *TableView* code example:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TableViewExample extends Application {

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

    @Override
    public void start(Stage primaryStage) {

        TableView tableView = new TableView();

        TableColumn<String, Person> column1 = new TableColumn<>("First
Name");
        column1.setCellValueFactory(new PropertyValueFactory<>("firstName"));

        TableColumn<String, Person> column2 = new TableColumn<>("Last Name");
        column2.setCellValueFactory(new PropertyValueFactory<>("lastName"));

        tableView.getColumns().add(column1);
        tableView.getColumns().add(column2);

        tableView.getItems().add(new Person("John", "Doe"));
        tableView.getItems().add(new Person("Jane", "Deer"));

        VBox vbox = new VBox(tableView);

        Scene scene = new Scene(vbox);

        primaryStage.setScene(scene);

        primaryStage.show();
    }
}

```

Here is the Person class used in this example:

```

public class Person {

    private String firstName = null;
    private String lastName = null;

    public Person() {

```

```
}

public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

## Create a TableView

In order to use a JavaFX `TableView` component you must first create a `TableView` instance. Here is an example of creating a JavaFX `TableView` instance:

```
TableView tableView = new TableView();
```

## Add TableColumn to the TableView

Having created a `TableView` you need to add one or more `TableColumn` instances to the `TableView` instance.

A `TableColumn` represents a vertical column of values. Each value is displayed on its own row, and is typically extracted from the list of objects being displayed in the `TableView`. Here is an example of adding two `TableColumn` instances to a JavaFX `TableView` instance:

```
TableView tableView = new TableView();
```

```
TableColumn<String, Person> firstNameColumn = new TableColumn<>("First Name");
firstNameColumn.setCellValueFactory(new PropertyValueFactory<>("firstName"));

TableColumn<String, Person> lastNameColumn = new TableColumn<>("Last Name");
lastNameColumn.setCellValueFactory(new PropertyValueFactory<>("lastName"));
```

Notice the [Java String](#) parameter passed to the constructor of the `TableColumn` class. This string will be displayed as the column header above the column. You can see an example of such a column header title in the screenshot at the top of this page.

## TableColumn Cell Value Factory

A `TableColumn` must have a *cell value factory* set on it. The cell value factory extracts the value to be displayed in each cell (on each row) in the column. In the example above a `PropertyValueFactory` is used. The `PropertyValueFactory` factory can extract a property value (field value) from a Java object. The name of the property is passed as a parameter to the `PropertyValueFactory` constructor, like this:

```
PropertyValueFactory factory = new PropertyValueFactory<>("firstName");
```

The property name `firstName` will match the getter method `getFirstName()` of the `Person` objects which contain the values are displayed on each row.

In the example shown earlier, a second `PropertyValueFactory` is set on the second `TableColumn` instance. The property name passed to the second `PropertyValueFactory` is `lastName`, which will match the getter method `getLastName()` of the `Person` class.

## Add Data to TableView

Once you have added `TableColumn` instances to the JavaFX `TableView`, you can add the data to be displayed to the `TableView`. The data is typically contained in a list of regular Java

objects (POJOs). Here is an example of adding two `Person` objects (class shown earlier in this JavaFX TableView tutorial) to a `TableView`:

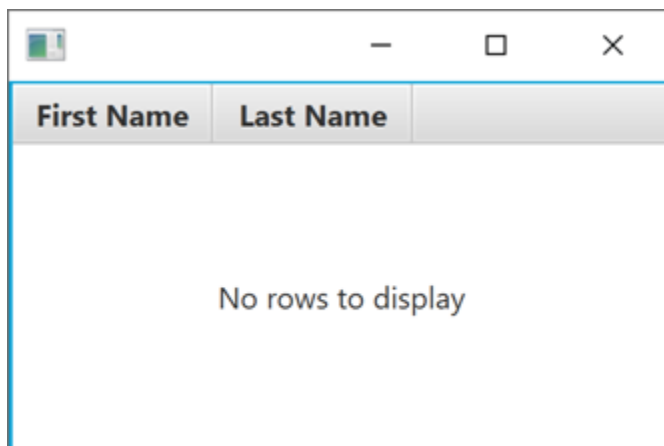
```
tableView.getItems().add(new Person("John", "Doe"));
tableView.getItems().add(new Person("Jane", "Deer"));
```

## Set Placeholder When No Rows to Display

You can set a placeholder to be displayed when the JavaFX `TableView` has no rows to display. The placeholder must be an instance of the JavaFX `Node` class, which most (if not all) JavaFX controls are. Thus, you can use an [JavaFX ImageView](#) or [JavaFX Label](#) as placeholder. Here is an example of using a `Label` as placeholder in a JavaFX `TableView`:

```
tableView.setPlaceholder(new Label("No rows to display"));
```

And here is how the corresponding `TableView` looks with the placeholder displayed:



Next: [JavaFX TreeView](#)

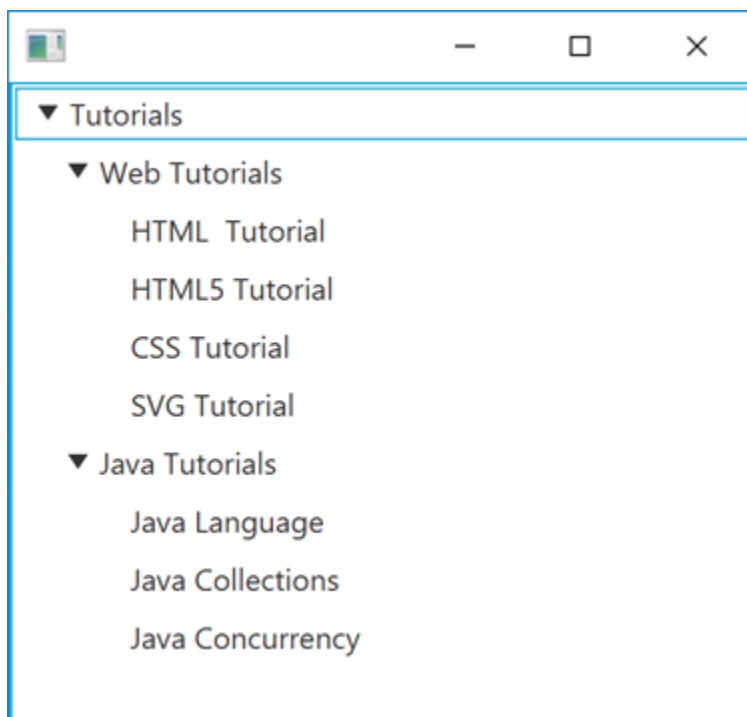
# JavaFX TreeView

- [Create a TreeView](#)
- [Add TreeView to Scene Graph](#)
- [Add Tree Items to TreeView](#)
  - [Add Children to a TreeItem](#)
- [Hide Root Item of TreeView](#)

Jakob Jenkov

Last update: 2019-03-17

The JavaFX *TreeView* enables you to display tree views inside your JavaFX applications. The JavaFX TreeView is represented by the class `javafx.scene.control.TreeView`. Here is a screenshot of a *JavaFX TreeView*:



## Create a TreeView



You create a JavaFX `TreeView` simply by creating a new instance of the `TreeView` class. Here is an example of creating a new JavaFX `TreeView` instance:

```
TreeView treeView = new TreeView();
```

## Add TreeView to Scene Graph

To make a JavaFX `TreeView` visible it must be added to the [JavaFX scene graph](#). Here is an example showing how to add a JavaFX `TreeView` to the JavaFX scene graph:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TreeViewExample extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        TreeView treeView = new TreeView();

        VBox vbox = new VBox(treeView);

        Scene scene = new Scene(vbox);

        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

Notice, that the `TreeView` created in the above example will not have any items (nodes) to display. We will see how to add tree items to a `TreeView` in the next section.

## Add Tree Items to TreeView

The items in the tree displayed by a JavaFX `TreeView` are represented by the `TreeItem` class (`javafx.scene.control.TreeItem`). Here is an example of creating a set of `TreeItem` instances and adding them to a JavaFX `TreeView` instance:

```
TreeItem rootItem = new TreeItem("Tutorials");

TreeItem webItem = new TreeItem("Web Tutorials");
webItem.getChildren().add(new TreeItem("HTML Tutorial"));
webItem.getChildren().add(new TreeItem("HTML5 Tutorial"));
webItem.getChildren().add(new TreeItem("CSS Tutorial"));
webItem.getChildren().add(new TreeItem("SVG Tutorial"));
rootItem.getChildren().add(webItem);

TreeItem javaItem = new TreeItem("Java Tutorials");
javaItem.getChildren().add(new TreeItem("Java Language"));
javaItem.getChildren().add(new TreeItem("Java Collections"));
javaItem.getChildren().add(new TreeItem("Java Concurrency"));
rootItem.getChildren().add(javaItem);

TreeView treeView = new TreeView();
treeView.setRoot(rootItem);
```

## Add Children to a `TreeItem`

If you look at the example in the previous section, you can see that a `TreeItem` can have other `TreeItem` instances as children. This parent-child relationship can continue recursively, indefinitely. This is how you structure the tree nodes logically in your JavaFX application. Here is an example that shows how to add child `TreeItem` instances to a parent `TreeItem`:

```
TreeItem javaItem = new TreeItem("Java Tutorials");
javaItem.getChildren().add(new TreeItem("Java Language"));
javaItem.getChildren().add(new TreeItem("Java Collections"));
javaItem.getChildren().add(new TreeItem("Java Concurrency"));

TreeItem rootItem = new TreeItem("Tutorials");
rootItem.getChildren().add(javaItem);
```

This example creates a tree with a single root `TreeItem` which has one child `TreeItem` set on it, and this child `TreeItem` has itself 3 child `TreeItem` instances added to it.

## Hide Root Item of TreeView

You can hide the root item (root node) of a JavaFX `TreeView`. You do so by calling the `setShowRoot()` method, passing the boolean value `false` to it as parameter. Here is an example of hiding the root `TreeItem` of a JavaFX `TreeView`:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TreeViewExample extends Application {

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

    @Override
    public void start(Stage primaryStage) {

        TreeItem rootItem = new TreeItem("Tutorials");

        TreeItem webItem = new TreeItem("Web Tutorials");
        webItem.getChildren().add(new TreeItem("HTML Tutorial"));
        webItem.getChildren().add(new TreeItem("HTML5 Tutorial"));
        webItem.getChildren().add(new TreeItem("CSS Tutorial"));
        webItem.getChildren().add(new TreeItem("SVG Tutorial"));
        rootItem.getChildren().add(webItem);

        TreeItem javaItem = new TreeItem("Java Tutorials");
        javaItem.getChildren().add(new TreeItem("Java Language"));
        javaItem.getChildren().add(new TreeItem("Java Collections"));
        javaItem.getChildren().add(new TreeItem("Java Concurrency"));
        rootItem.getChildren().add(javaItem);

        TreeView treeView = new TreeView();
        treeView.setRoot(rootItem);

        treeView.setShowRoot(false);

        VBox vbox = new VBox(treeView);

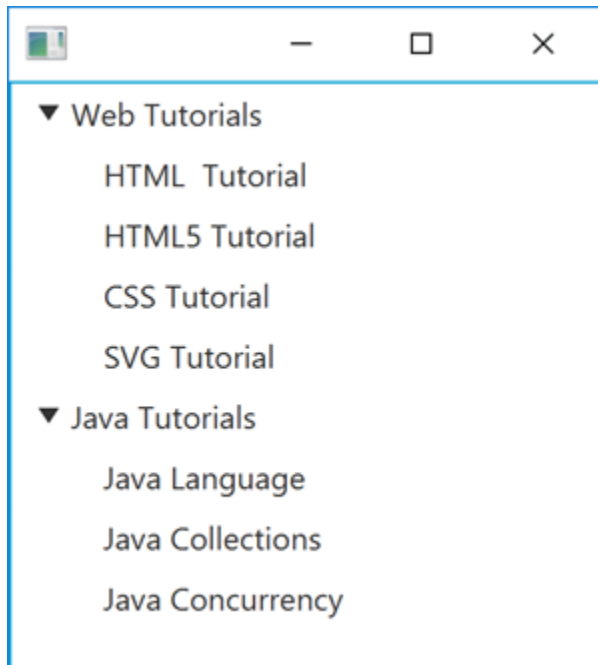
        Scene scene = new Scene(vbox);

        primaryStage.setScene(scene);

        primaryStage.show();
    }
}
```

```
}  
  
}
```

Notice the line in bold. That is the line that instructs the JavaFX `TreeView` to hide the root node. Here is a screenshot showing how the resulting JavaFX `TreeView` looks, with all nodes expanded:



Notice how the root `TreeItem` with the text `Tutorials` is not displayed.

Next: [JavaFX TreeTableView](#)

# JavaFX TreeTableView

- [Create a TreeTableView](#)
- [Add TreeTableColumn to TreeTableView](#)
- [Add TreeItem to TreeTableView](#)

Jakob Jenkov

Last update: 2019-03-24

The *JavaFX TreeTableView* class is a combination of a [JavaFX TreeView](#) and a [JavaFX TableView](#). Overall, the *JavaFX TreeTableView* is a `TableView` which contains a tree of items as its leftmost column. The rest of the columns are normal table columns.

The `JavaFX TreeTableView` shows one row per item in its tree. In other words, the columns shown to the right of each tree node belong to the item in the tree on the left. The tree items in the `JavaFX TreeTableView`'s leftmost column can be expanded and collapsed. As tree items are expanded and collapsed, the rows for any shown or hidden tree items are shown or hidden in the columns on the right side too.

Here is a `JavaFX TreeTableView` screenshot:

JavaFX TreeTableView Example		
Brand	Model	
▼ Cars	...	
▼ Audi	...	
Audi	A1	
Audi	A5	
Audi	A7	
▼ Mercedes	...	
Mercedes	SL500	
Mercedes	SL500 AMG	

## Create a TreeTableView

To use a JavaFX TreeTableView you must create an instance of the class `javafx.scene.control.TreeTableView`. Here is an example of creating a JavaFX TreeTableView:

```
TreeTableView<Car> treeTableView = new TreeTableView<Car>();
```

This example creates a TreeTableView instance which is intended to display Car objects. The Car class used looks like this:

```
public class Car {

    private String brand = null;
    private String model = null;

    public Car() {
    }

    public Car(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

}
```

```

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
}

```

## Add TreeTableColumn to TreeTableView

As you can see from the screenshot earlier in this tutorial, the JavaFX `TreeTableView` component displays the values from its items in horizontal rows divided into vertical columns. The first column is a tree structure. The rest of the columns are normal table columns that display values from the items in the tree structure displayed in the first column.

For the `TreeTableView` to display these values in columns, you must add one or more `TreeTableColumn` instances to the `TreeTableView`. Here is an example of adding two `TreeTableColumn` instances to a JavaFX `TreeTableView`:

```

TreeTableView<Car> treeTableView = new TreeTableView<Car>();

TreeTableColumn<Car, String> treeTableColumn1 = new
TreeTableColumn<>("Brand");
TreeTableColumn<Car, String> treeTableColumn2 = new
TreeTableColumn<>("Model");

treeTableColumn1.setCellValueFactory(new
TreeItemPropertyValueFactory<>("brand"));
treeTableColumn2.setCellValueFactory(new
TreeItemPropertyValueFactory<>("model"));

treeTableView.getColumns().add(treeTableColumn1);
treeTableView.getColumns().add(treeTableColumn2);

```

## Add TreeItem to TreeTableView

For the JavaFX `TreeTableView` to display any data you must add one or more `TreeItem` instances to it. Here is an example of adding 9 `TreeItem` objects to a JavaFX `TreeTableView`:

```
TreeItem mercedes1 = new TreeItem(new Car("Mercedes", "SL500"));
TreeItem mercedes2 = new TreeItem(new Car("Mercedes", "SL500 AMG"));
TreeItem mercedes3 = new TreeItem(new Car("Mercedes", "CLA 200"));

TreeItem mercedes = new TreeItem(new Car("Mercedes", "..."));
mercedes.getChildren().add(mercedes1);
mercedes.getChildren().add(mercedes2);

TreeItem audi1 = new TreeItem(new Car("Audi", "A1"));
TreeItem audi2 = new TreeItem(new Car("Audi", "A5"));
TreeItem audi3 = new TreeItem(new Car("Audi", "A7"));

TreeItem audi = new TreeItem(new Car("Audi", "..."));
audi.getChildren().add(audi1);
audi.getChildren().add(audi2);
audi.getChildren().add(audi3);

TreeItem cars = new TreeItem(new Car("Cars", "..."));
cars.getChildren().add(audi);
cars.getChildren().add(mercedes);

treeTableView.setRoot(cars);
```

Notice how the `TreeItem` instances are organized into a tree, with the "Cars" node at the root, then the two nodes "Mercedes" and "Audi", and then below them three `TreeItem` instances each, with concrete car model nodes.

Notice the last line - the `TreeTableView setRoot()` method call. It is this method call that sets the tree of `TreeItem` instances on the `TreeTableView`.

Next: [JavaFX HTML Editor](#)



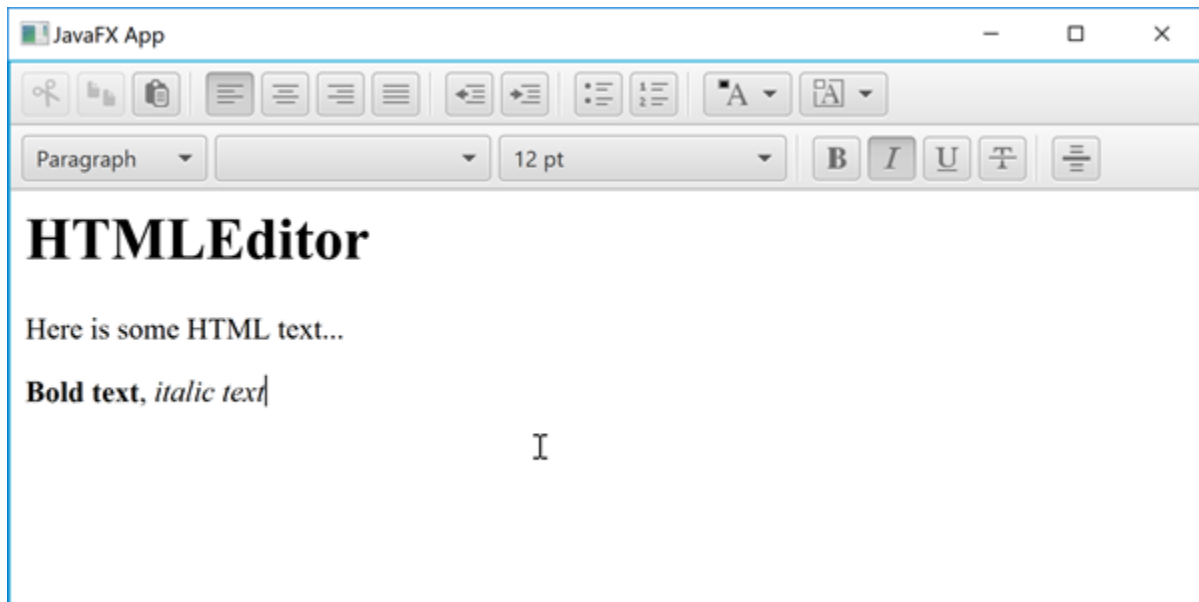
# JavaFX HTMLEditor

- [Full HTMLEditor Example](#)
- [Create an HTMLEditor](#)
- [Get HTML From HTMLEditor](#)
- [Set HTML in HTMLEditor](#)

Jakob Jenkov

Last update: 2019-08-03

The *JavaFX HTMLEditor* is an advanced HTML editor that enables the user to edit HTML easier than by writing the full HTML markup in text. The HTMLEditor contains a set of buttons that can be used to set the styles of the edited text WYSIWYG style. The JavaFX HTMLEditor is represented by the class `javafx.scene.web.HTMLEditor`. Here is a screenshot of a JavaFX HTMLEditor:



## Full HTMLEditor Example

Here is first a full `JavaFX HTMLEditor` example so you can see what using the `HTMLEditor` looks like in code:

```
import javafx.application.Application;
```

```

import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.stage.Stage;

public class HtmlEditorExample extends Application {

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

    public void start(Stage primaryStage) {

        HTMLEditor htmlEditor = new HTMLEditor();

        VBox vBox = new VBox(htmlEditor);
        Scene scene = new Scene(vBox);

        primaryStage.setScene(scene);
        primaryStage.setTitle("JavaFX App");

        primaryStage.show();
    }
}

```

## Create an HTMLEditor

Before you can use a JavaFX `HTMLEditor` in your code, you must first create an instance of it. Here is an example of creating an instance of a JavaFX `HTMLEditor`:

```
HTMLEditor htmlEditor = new HTMLEditor();
```

## Get HTML From HTMLEditor

Sooner or later you will probably want to obtain the HTML text that was edited in the `HTMLEditor` by the user. You obtain the HTML from the `HTMLEditor` via its `getHtmlText()` method. Here is an example of getting the HTML from a JavaFX `HTMLEditor` instance:

```
String htmlText = htmlEditor.getHtmlText();
```

As you can see, the HTML is returned as a standard [Java String](#).

## Set HTML in HTML editor

You can also set the HTML to be edited in a JavaFX `HTML editor` via its `setHtmlText()` method. Here is an example of setting the HTML to be edited in a JavaFX `HTML editor` instance:

```
String htmlText = "<b>Bold text</b>";  
htmlEditor.setHtmlText(htmlText);
```

Next: [JavaFX Pagination](#)

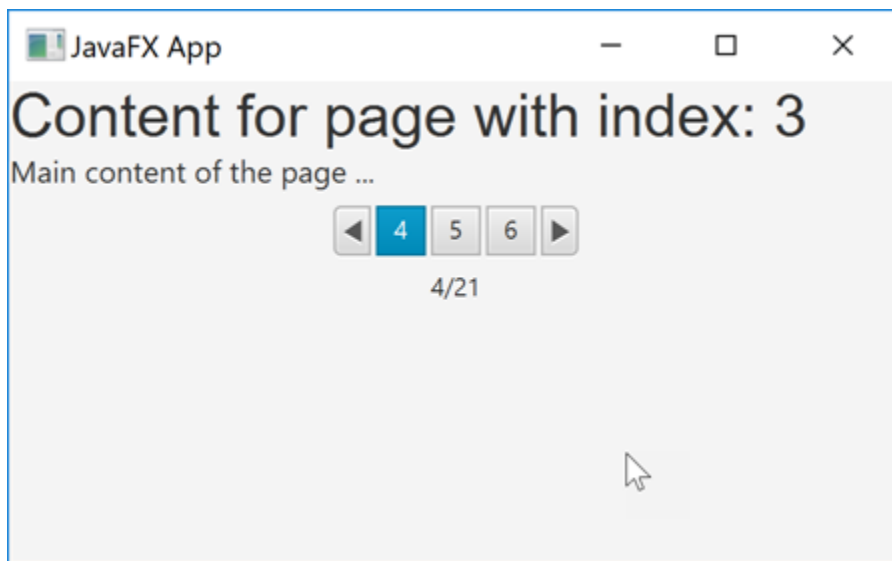
# JavaFX Pagination

- [Full Pagination Example](#)
- [Pagination Properties](#)
- [Pagination Page Factory](#)

Jakob Jenkov

Last update: 2019-08-03

The *JavaFX Pagination* control enables the user to navigate page by page through content, for instance pages of search results, articles, images or similar types of content. The JavaFX Pagination control is represented by the class `javafx.scene.control.Pagination`. Here is a screenshot of a JavaFX `Pagination` control:



## Full Pagination Example

Here is first a full Java code example of how to use a `JavaFX Pagination` control:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Pagination;
import javafx.scene.layout.VBox;
```

```

import javafx.scene.text.Font;
import javafx.stage.Stage;

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

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

        Pagination pagination = new Pagination();
        pagination.setPageCount(21);
        pagination.setCurrentPageIndex(3);
        pagination.setMaxPageIndicatorCount(3);

        pagination.setPageFactory((pageIndex) -> {
            Label label1 = new Label("Content for page with index: " +
pageIndex);
            label1.setFont(new Font("Arial", 24));

            Label label2 = new Label("Main content of the page ...");

            return new VBox(label1, label2);
        });

        VBox vBox = new VBox(pagination);
        Scene scene = new Scene(vBox, 960, 600);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

```

## Pagination Properties

The JavaFX `Pagination` control contains a set of properties that specify how the `Pagination` control is rendered. These properties are:

- Page count
- Current page index
- Max number of page indicators

The page count is the total number of pages the user can navigate between. The current page index is the page the user is looking at currently. The maximum number of page indicators is the number of shortcut buttons with

page numbers on, which the user can click on to navigate directly to that page.

The effect of all three properties are visible in the screenshot earlier in this tutorial. Here is an example of setting all three properties:

```
Pagination pagination = new Pagination();

pagination.setPageCount(21);
pagination.setCurrentPageIndex(3);
pagination.setMaxPageIndicatorCount(3);
```

## Pagination Page Factory

The JavaFX `Pagination` control needs a page factory set on it to be able to navigate properly through the paged content. The page factory is called when the user navigates to a new page. The page factory component is attached to the `Pagination` control via its `setPageFactory()` method, and must implement the interface `javafx.util.Callback` interface.

Here is first how the `CallBack` interface is defined:

```
public interface Callback<P,R> {

    public R call(P param);

}
```

In the `setPageFactory()` method the two type parameters `P` and `R` are set to `Integer` (`P`) and `Node` (`R`). That means, that the page factory must implement the `Callback<Integer, Node>` interface. Here is an example of an implementation of `Callback<Integer, Node>`:

```
public static class MyPageFactory implements Callback<Integer, Node> {
    @Override
    public Node call(Integer pageIndex) {
        return new Label("Content for page " + pageIndex);
    }
}
```

The `Integer` parameter passed to the `Callback` implementation is the index of the page the page factory should create a `Node` for. The returned `Node` should display the content for the page with the given page index.

Here is an example of setting the page factory on a `JavaFX Pagination` control:

```
pagination.setPageFactory(new MyPageFactory());
```

You can also set the page factory on a `Pagination` control using an anonymous `Callback` interface implementation, or using a [Java lambda](#) expression. Here is first an example using an anonymous `Callback` implementation:

```
pagination.setPageFactory(new Callback<Integer, Node>() {  
    @Override  
    public Node call(Integer pageIndex) {  
        return new Label("Content for page " + pageIndex);  
    }  
});
```

And here is an example of setting a `Pagination` page factory using a Java lambda expression:

```
pagination.setPageFactory((pageIndex) -> {  
    return new Label("Content for page " + pageIndex);  
});
```

And even shorter, using a shorter lambda expression syntax:

```
pagination.setPageFactory((pageIndex) -> new Label("Content for page " +  
pageIndex) );
```

Next: [JavaFX BarChart](#)

# JavaFX BarChart

- [BarChart X Axis and Y Axis](#)
- [Creating a BarChart](#)
- [BarChart Data Series](#)
- [Adding a BarChart to the Scene Graph](#)
- [Displaying Multiple Data Series in the Same BarChart](#)

Jakob Jenkov

Last update: 2016-05-26

The JavaFX BarChart component is capable of drawing a bar chart inside your JavaFX applications. This is useful in dashboard-like applications. The JavaFX BarChart component is represented by the class `javafx.scene.chart.BarChart`

## BarChart X Axis and Y Axis

A JavaFX BarChart draws a bar chart. A bar chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. For bar charts the X axis is typically a category of some kind, and the Y axis is numerical.

For instance, imagine a bar chart illustrating the number of visits to a website from desktop, phone and tablet users. The bar chart would show 3 bars. The categories on the X axis would be "Desktop", "Phone" and "Tablet". The Y axis would show how many visits each category on the X axis has, so the Y axis is numerical.

You need to define the X axis and Y axis used by a BarChart. A categorical axis is represented by the JavaFX class `javafx.scene.chart.CategoryAxis`. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

Here is an example of creating a JavaFX CategoryAxis and NumberAxis:

```
CategoryAxis xAxis = new CategoryAxis();
```



```
xAxis.setLabel("Devices");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Visits");
```

## Creating a BarChart

To create a JavaFX BarChart component you must create an instance of the BarChart class. You must pass an X axis and a Y axis instance to the BarChart constructor. Here is a JavaFX BarChart instantiation example:

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Devices");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Visits");

BarChart barChart = new BarChart(xAxis, yAxis);
```

It is possible to display multiple data series in the same bar chart. I will show how to do that later in this BarChart tutorial.

## BarChart Data Series

To get the JavaFX BarChart component to display any bars, you must provide it with a *data series*. A data series is a list of data points. Each data point contains an X value and a Y value. Here is an example of creating a data series and adding it to a BarChart component:

```
XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("2014");

dataSeries1.getData().add(new XYChart.Data("Desktop", 178));
dataSeries1.getData().add(new XYChart.Data("Phone", 65));
dataSeries1.getData().add(new XYChart.Data("Tablet", 23));

barChart.getData().add(dataSeries1);
```

First an XYChart.Series instance is created and given a name. Second, 3 XYChart.Data instances are added to

the `XYChart.Series` object. Third, the `XYChart.Series` object is added to a `BarChart` object.

## Adding a BarChart to the Scene Graph

To make a `BarChart` visible you must add it to the JavaFX scene graph. This means adding the `BarChart` to a `Scene` object, or adding the `BarChart` object to a layout component which is added to a `Scene` object.

Here is an example of adding a JavaFX `BarChart` to the JavaFX scene graph:

```
package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BarChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("BarChart Experiments");

        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("Devices");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Visits");

        BarChart barChart = new BarChart(xAxis, yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("2014");

        dataSeries1.getData().add(new XYChart.Data("Desktop", 567));
        dataSeries1.getData().add(new XYChart.Data("Phone", 65));
        dataSeries1.getData().add(new XYChart.Data("Tablet", 23));

        barChart.getData().add(dataSeries1);

        VBox vbox = new VBox(barChart);
```

```

        Scene scene = new Scene(vbox, 400, 200);

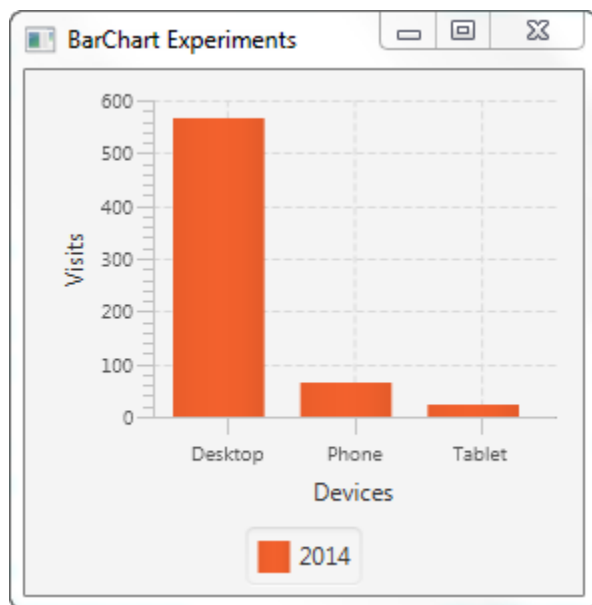
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```

The application resulting from running this code example will look similar to this:



## Displaying Multiple Data Series in the Same BarChart

As mentioned earlier in this tutorial it is possible to display multiple data series in the same `BarChart` component. You do so simply by adding multiple data series to the `BarChart` component.

When displaying multiple data series in the same `BarChart`, the data points are categorized by their X values (category). Thus, all data points across the different data series with the same X value will be displayed next to each other in the bar chart. Here is first an example of creating two data series with data points that have the same X value (category):

```

XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("2014");

dataSeries1.getData().add(new XYChart.Data("Desktop", 567));
dataSeries1.getData().add(new XYChart.Data("Phone" , 65));
dataSeries1.getData().add(new XYChart.Data("Tablet" , 23));

barChart.getData().add(dataSeries1);

XYChart.Series dataSeries2 = new XYChart.Series();
dataSeries2.setName("2015");

dataSeries2.getData().add(new XYChart.Data("Desktop", 540));
dataSeries2.getData().add(new XYChart.Data("Phone" , 120));
dataSeries2.getData().add(new XYChart.Data("Tablet" , 36));

barChart.getData().add(dataSeries2);

```

Notice how the `XYChart.Data` instances in the two data series use the same three values for X ("Desktop", "Phone" and "Tablet").

Here is a full example showing the two data series added to the `BarChart` and displayed in the scene graph:

```

package com.jenkov.javaafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BarChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("BarChart Experiments");

        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("Devices");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Visits");

        BarChart barChart = new BarChart(xAxis, yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("2014");
    }
}

```

```

        dataSeries1.getData().add(new XYChart.Data("Desktop", 567));
        dataSeries1.getData().add(new XYChart.Data("Phone" , 65));
        dataSeries1.getData().add(new XYChart.Data("Tablet" , 23));

        barChart.getData().add(dataSeries1);

        XYChart.Series dataSeries2 = new XYChart.Series();
        dataSeries2.setName("2015");

        dataSeries2.getData().add(new XYChart.Data("Desktop", 540));
        dataSeries2.getData().add(new XYChart.Data("Phone" , 120));
        dataSeries2.getData().add(new XYChart.Data("Tablet" , 36));

        barChart.getData().add(dataSeries2);

        VBox vbox = new VBox(barChart);

        Scene scene = new Scene(vbox, 400, 200);

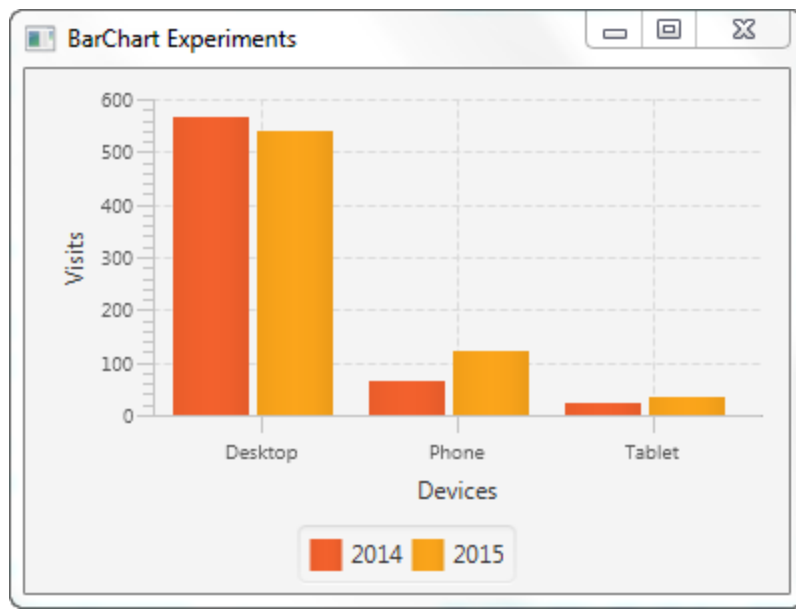
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```

The application resulting from running this example would look similar to this:



Notice how data from the two different data series are mixed. The data points with the X value "Desktop" are displayed next to each other, and the same is true for the data points with the X values "Phone" and "Tablet".

Next: [JavaFX StackedBarChart](#)

# JavaFX StackedBarChart

- [StackedBarChart X Axis and Y Axis](#)
  - [Setting Categories on the X Axis](#)
- [Creating a StackedBarChart](#)
- [StackedBarChart Data Series](#)
- [Adding a StackedBarChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-27

The JavaFX `StackedBarChart` component is capable of drawing stacked bar charts inside your JavaFX applications. This can be useful in dashboard-like applications. The JavaFX `StackedBarChart` component is represented by the `javafx.scene.chart.StackedBarChart` class.

The `StackedBarChart` component is similar in function to the [JavaFX BarChart](#) component, but there are a few important differences. I will emphasize these differences when I get to them.

## StackedBarChart X Axis and Y Axis

A JavaFX `StackedBarChart` draws a stacked bar chart. A stacked bar chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. For stacked bar charts the X axis is typically a category of some kind, and the Y axis is numerical.

For instance, imagine a stacked bar chart illustrating the number of visits to a website from desktop, phone and tablet users in the years 2014 and 2015. The stacked bar chart would show 2 bars. One bar with the stacked values for the visits from desktop, phone and tablet in 2014, and another bar with the values stacked for 2015.

The categories on the X axis would be "Desktop", "Phone" and "Tablet". The Y axis would show how many visits each category on the X axis has, so the Y axis is numerical.

You need to define the X axis and Y axis used by a `StackedBarChart`. A categorical axis is represented by the JavaFX class `javafx.scene.chart.CategoryAxis`. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

Here is an example of creating a JavaFX `CategoryAxis` and `NumberAxis`:

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Devices");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Visits");
```

## Setting Categories on the X Axis

The `StackedBarChart` has one difference from the `BarChart` with regards to the configuration of the X axis.

The `StackedBarChart` requires that you set the categories directly on the `CategoryAxis` used as X axis. The following example creates an X axis with the categories explicitly set on the X axis:

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Devices");
xAxis.getCategories().addAll("Desktop", "Phone", "Tablet");
```

## Creating a StackedBarChart

To create a JavaFX `StackedBarChart` component you must create an instance of the `StackedBarChart` class. You must pass an X axis and a Y axis instance to the `StackedBarChart` constructor. Here is a JavaFX `StackedBarChart` instantiation example:

```
CategoryAxis xAxis = new CategoryAxis();
xAxis.setLabel("Devices");
xAxis.getCategories().addAll("Desktop", "Phone", "Tablet");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Visits");
```



```
StackedBarChart stackedBarChart = new StackedBarChart(xAxis, yAxis);
```

## StackedBarChart Data Series

To get the JavaFX `StackedBarChart` component to display any bars, you must provide it with at least one *data series*. A data series is a list of data points. Each data point contains an X value and a Y value.

The `StackedBarChart` handles data series differently than the `BarChart` component. The `StackedBarChart` stacks the bars from the different data series on top of each other, instead of displaying them next to each other. The `StackedBarChart` stacks all values with the same X category into the same bar. That means that you might have to think a little differently when organizing your data than when using a `BarChart`.

To show the visits from desktop, phone and tablet from the same year stacked together in the same bar, you must create a data series per device, and use the year as category. Here is how that looks:

```
StackedBarChart    stackedBarChart = new StackedBarChart(xAxis, yAxis);

XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("Desktop");

dataSeries1.getData().add(new XYChart.Data("2014", 567));
dataSeries1.getData().add(new XYChart.Data("2015", 540));

stackedBarChart.getData().add(dataSeries1);

XYChart.Series dataSeries2 = new XYChart.Series();
dataSeries2.setName("Phone");

dataSeries2.getData().add(new XYChart.Data("2014" , 65));
dataSeries2.getData().add(new XYChart.Data("2015" , 120));

stackedBarChart.getData().add(dataSeries2);

XYChart.Series dataSeries3 = new XYChart.Series();
dataSeries3.setName("Tablet");

dataSeries3.getData().add(new XYChart.Data("2014" , 23));
dataSeries3.getData().add(new XYChart.Data("2015" , 36));

stackedBarChart.getData().add(dataSeries3);
```

Notice how one data series is created for each device type (desktop, phone, tablet), and that the data is categorized by year.

## Adding a StackedBarChart to the Scene Graph

In order to make a `StackedBarChart` visible it must be added to the JavaFX scene graph. That means adding the `StackedBarChart` to a `Scene` object, or adding the `StackedBarChart` to a layout component which is added to a `Scene` object.

Here is an example of adding a JavaFX `StackedBarChart` to the JavaFX scene graph:

```
package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class StackedBarChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("StackedBarChart Experiments");

        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("Devices");
        xAxis.getCategories().addAll("Desktop", "Phone", "Tablet");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Visits");

        StackedBarChart stackedBarChart = new StackedBarChart(xAxis,
yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("Desktop");

        dataSeries1.getData().add(new XYChart.Data("2014", 567));
        dataSeries1.getData().add(new XYChart.Data("2015", 540));

        stackedBarChart.getData().add(dataSeries1);

        XYChart.Series dataSeries2 = new XYChart.Series();
        dataSeries2.setName("Phone");

        dataSeries2.getData().add(new XYChart.Data("2014", 65));
        dataSeries2.getData().add(new XYChart.Data("2015", 120));
```

```

        stackedBarChart.getData().add(dataSeries2);

        XYChart.Series dataSeries3 = new XYChart.Series();
        dataSeries3.setName("Tablet");

        dataSeries3.getData().add(new XYChart.Data("2014" , 23));
        dataSeries3.getData().add(new XYChart.Data("2015" , 36));

        stackedBarChart.getData().add(dataSeries3);

        VBox vbox = new VBox(stackedBarChart);

        Scene scene = new Scene(vbox, 400, 200);

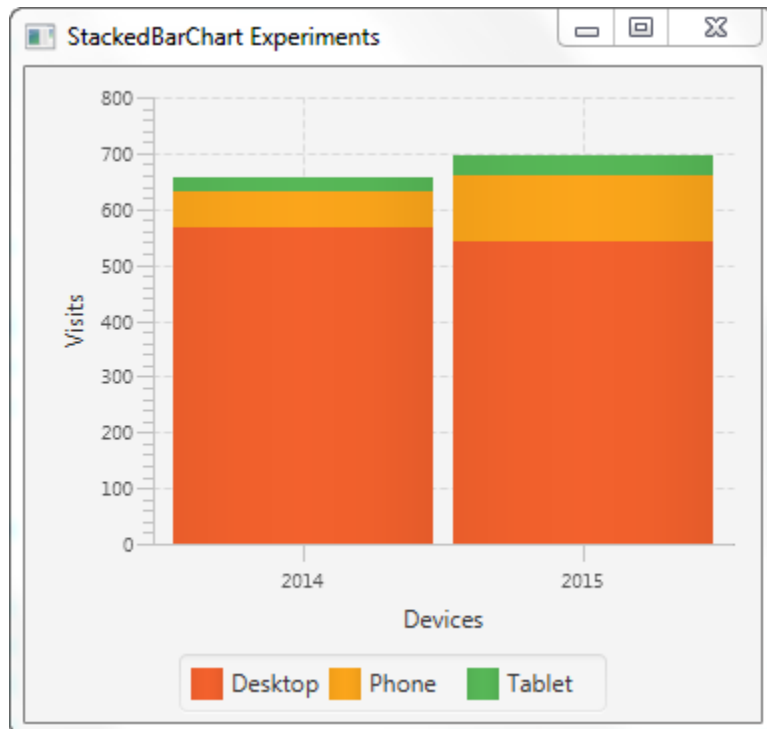
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```

The application resulting from running this code would look similar to this:



Next: [JavaFX ScatterChart](#)

# JavaFX ScatterChart

- [ScatterChart X Axis and Y Axis](#)
- [Creating a ScatterChart](#)
- [ScatterChart Data Series](#)
- [Adding a ScatterChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-28

The JavaFX ScatterChart component can draw scatter charts inside your JavaFX applications. The JavaFX ScatterChart component is represented by the class `javafx.scene.chart.ScatterChart`.

## ScatterChart X Axis and Y Axis

A JavaFX ScatterChart draws a scatter chart. A scatter chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. You can use both categorical and numerical axes, but in this example I will just use two numerical axes for the scatter chart. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

You need to define the X axis and Y axis used by a ScatterChart. Here is an example of creating two JavaFX NumberAxis instances:

```
NumberAxis xAxis = new NumberAxis();  
xAxis.setLabel("No of employees");  
  
NumberAxis yAxis = new NumberAxis();  
yAxis.setLabel("Revenue per employee");
```

## Creating a ScatterChart

You create a JavaFX ScatterChart component by creating an instance of the ScatterChart class. You need to pass an X axis and Y axis to the ScatterChart constructor. Here is a JavaFX ScatterChart instantiation example:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");

ScatterChart scatterChart = new ScatterChart(xAxis, yAxis);
```

## ScatterChart Data Series

To get a JavaFX `ScatterChart` component to display any dots, you must provide it with a *data series*. A data series is a list of data points. Each data point contains an X value and a Y value. Here is an example of creating a data series and adding it to a `ScatterChart` component:

```
XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("2014");

dataSeries1.getData().add(new XYChart.Data( 1, 567));
dataSeries1.getData().add(new XYChart.Data( 5, 612));
dataSeries1.getData().add(new XYChart.Data(10, 800));
dataSeries1.getData().add(new XYChart.Data(20, 780));
dataSeries1.getData().add(new XYChart.Data(40, 810));
dataSeries1.getData().add(new XYChart.Data(80, 850));

scatterChart.getData().add(dataSeries1);
```

First an `XYChart.Series` instance is created and given a name. Second, 6 `XYChart.Data` instances are added to the `XYChart.Series` object. Third, the `XYChart.Series` object is added to a `ScatterChart` object.

It is possible to add multiple data series to the `ScatterChart`. Just repeat the above code for additional data series.

## Adding a ScatterChart to the Scene Graph

To make a `ScatterChart` visible you must add it to the JavaFX scene graph. This means adding the `ScatterChart` to a `Scene` object or add the `ScatterChart` to a layout component which is added to a `Scene` object.

Here is an example that adds a ScatterChart to the JavaFX scene graph:

```
package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.ScatterChart;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ScatterChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("BarChart Experiments");

        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("No of employees");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Revenue per employee");

        ScatterChart scatterChart = new ScatterChart(xAxis, yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("2014");

        dataSeries1.getData().add(new XYChart.Data( 1, 567));
        dataSeries1.getData().add(new XYChart.Data( 5, 612));
        dataSeries1.getData().add(new XYChart.Data(10, 800));
        dataSeries1.getData().add(new XYChart.Data(20, 780));
        dataSeries1.getData().add(new XYChart.Data(40, 810));
        dataSeries1.getData().add(new XYChart.Data(80, 850));

        scatterChart.getData().add(dataSeries1);

        VBox vbox = new VBox(scatterChart);

        Scene scene = new Scene(vbox, 400, 200);

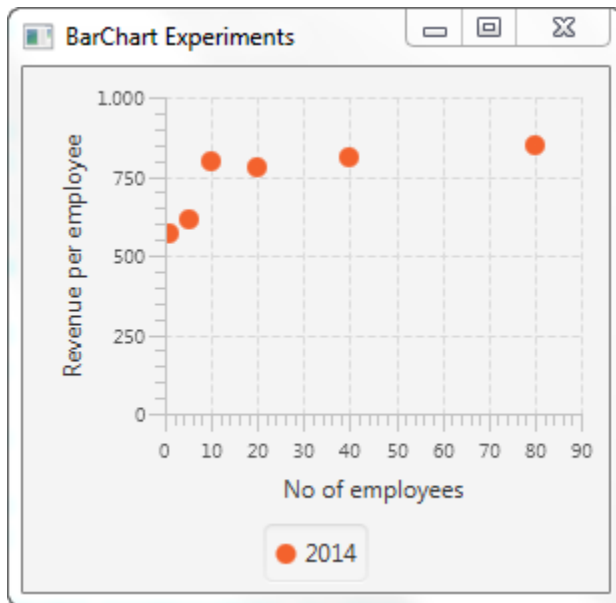
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```
}
```

The application resulting from running this application would look similar to this:



Next: [JavaFX LineChart](#)



# JavaFX LineChart

- [LineChart X Axis and Y Axis](#)
- [Creating a LineChart](#)
- [LineChart Data Series](#)
- [Adding a LineChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-28

The JavaFX LineChart can draw line charts inside your JavaFX applications. The JavaFX LineChart component is represented by the class `javafx.scene.chart.LineChart`.

## LineChart X Axis and Y Axis

A JavaFX LineChart draws a line chart. A line chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. Line charts typically have two numerical axes. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

You need to define the X axis and Y axis used by a `LineChart`. Here is an example of creating two JavaFX `NumberAxis` instances:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");
```

## Creating a LineChart

You create a JavaFX LineChart component by creating an instance of the `LineChart` class. You need to pass an X axis and Y axis to the `LineChart` constructor. Here is a JavaFX `LineChart` instantiation example:

```
NumberAxis xAxis = new NumberAxis();
```

```
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");

LineChart lineChart = new LineChart(xAxis, yAxis);
```

## LineChart Data Series

To get a JavaFX `LineChart` component to display any lines, you must provide it with a *data series*. A data series is a list of data points. Each data point contains an X value and a Y value. Here is an example of creating a data series and adding it to a `LineChart` component:

```
XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("2014");

dataSeries1.getData().add(new XYChart.Data( 1, 567));
dataSeries1.getData().add(new XYChart.Data( 5, 612));
dataSeries1.getData().add(new XYChart.Data(10, 800));
dataSeries1.getData().add(new XYChart.Data(20, 780));
dataSeries1.getData().add(new XYChart.Data(40, 810));
dataSeries1.getData().add(new XYChart.Data(80, 850));

lineChart.getData().add(dataSeries1);
```

First an `XYChart.Series` instance is created and given a name. Second, 6 `XYChart.Data` instances are added to the `XYChart.Series` object. Third, the `XYChart.Series` object is added to a `LineChart` object.

It is possible to add multiple data series to the `LineChart`. Just repeat the above code for additional data series.

## Adding a LineChart to the Scene Graph

To make a `LineChart` visible you must add it to the JavaFX scene graph. This means adding the `LineChart` to a `Scene` object or add the `LineChart` to a layout component which is added to a `Scene` object.

Here is an example that adds a `LineChart` to the JavaFX scene graph:

```

package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class LineChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("LineChart Experiments");

        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("No of employees");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Revenue per employee");

        LineChart lineChart = new LineChart(xAxis, yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("2014");

        dataSeries1.getData().add(new XYChart.Data( 1, 567));
        dataSeries1.getData().add(new XYChart.Data( 5, 612));
        dataSeries1.getData().add(new XYChart.Data(10, 800));
        dataSeries1.getData().add(new XYChart.Data(20, 780));
        dataSeries1.getData().add(new XYChart.Data(40, 810));
        dataSeries1.getData().add(new XYChart.Data(80, 850));

        lineChart.getData().add(dataSeries1);

        VBox vbox = new VBox(lineChart);

        Scene scene = new Scene(vbox, 400, 200);

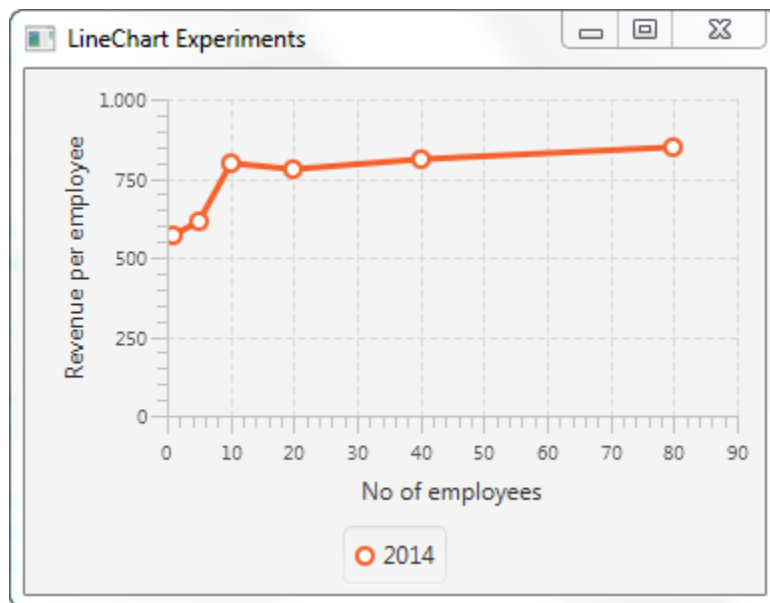
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```

The application resulting from running this application would look similar to this:



Next: [JavaFX AreaChart](#)

# JavaFX AreaChart

- [AreaChart X Axis and Y Axis](#)
- [Creating an AreaChart](#)
- [AreaChart Data Series](#)
- [Adding an AreaChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-29

The JavaFX AreaChart can draw area charts inside your JavaFX applications. An area chart is a line chart where the area below the lines are painted with color. The JavaFX AreaChart component is represented by the class `javafx.scene.chart.AreaChart`.

## AreaChart X Axis and Y Axis

A JavaFX AreaChart draws an area chart. An area chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. Area charts typically have two numerical axes. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

You need to define the X axis and Y axis used by a AreaChart. Here is an example of creating two JavaFX NumberAxis instances:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");
```

## Creating an AreaChart

You create a JavaFX AreaChart component by creating an instance of the AreaChart class. You need to pass an X axis and Y axis to the AreaChart constructor. Here is a JavaFX AreaChart instantiation example:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");

AreaChart areaChart = new AreaChart(xAxis, yAxis);
```

## AreaChart Data Series

To get a JavaFX `AreaChart` component to display anything, you must provide it with a *data series*. A data series is a list of data points. Each data point contains an X value and a Y value. Here is an example of creating a data series and adding it to a `AreaChart` component:

```
XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("2014");

dataSeries1.getData().add(new XYChart.Data( 1, 567));
dataSeries1.getData().add(new XYChart.Data( 5, 612));
dataSeries1.getData().add(new XYChart.Data(10, 800));
dataSeries1.getData().add(new XYChart.Data(20, 780));
dataSeries1.getData().add(new XYChart.Data(40, 810));
dataSeries1.getData().add(new XYChart.Data(80, 850));

areaChart.getData().add(dataSeries1);
```

First an `XYChart.Series` instance is created and given a name. Second, 6 `XYChart.Data` instances are added to the `XYChart.Series` object. Third, the `XYChart.Series` object is added to a `AreaChart` object.

It is possible to add multiple data series to the `AreaChart`. Just repeat the above code for additional data series.

## Adding an AreaChart to the Scene Graph

To make a JavaFX `AreaChart` visible you must add it to the JavaFX scene graph. This means adding the `AreaChart` to a `Scene` object or add the `AreaChart` to a layout component which is added to a `Scene` object.

Here is an example that adds an `AreaChart` to the JavaFX scene graph:

```

package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.AreaChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class AreaChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("AreaChart Experiments");

        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("No of employees");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Revenue per employee");

        AreaChart areaChart = new AreaChart(xAxis, yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("2014");

        dataSeries1.getData().add(new XYChart.Data( 1, 567));
        dataSeries1.getData().add(new XYChart.Data( 5, 612));
        dataSeries1.getData().add(new XYChart.Data(10, 800));
        dataSeries1.getData().add(new XYChart.Data(20, 780));
        dataSeries1.getData().add(new XYChart.Data(40, 810));
        dataSeries1.getData().add(new XYChart.Data(80, 850));

        areaChart.getData().add(dataSeries1);

        VBox vbox = new VBox(areaChart);

        Scene scene = new Scene(vbox, 400, 200);

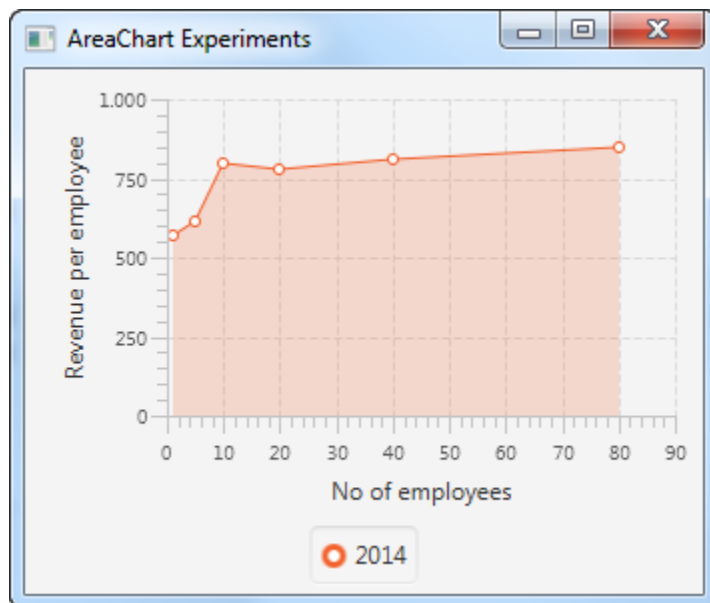
        primaryStage.setScene(scene);
        primaryStage.setHeight(300);
        primaryStage.setWidth(1200);

        primaryStage.show();
    }

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

```

The application resulting from running this application would look similar to this:



Next: [JavaFX StackedAreaChart](#)



# JavaFX StackedAreaChart

- [StackedAreaChart X Axis and Y Axis](#)
- [Creating a StackedAreaChart](#)
- [StackedAreaChart Data Series](#)
- [Adding a StackedAreaChart to the Scene Graph](#)

Jakob Jenkov

Last update: 2016-05-29

A JavaFX StackedAreaChart component is capable of drawing stacked area charts inside your JavaFX applications. A stacked area chart is similar to an area chart with multiple data series, except a stacked area chart displays the data series (and thus areas) stacked on top of each other, where the normal area chart would overlap them.

The JavaFX StackedAreaChart component is represented by the class `java.scene.chart.StackedAreaChart`.

## StackedAreaChart X Axis and Y Axis

A JavaFX StackedAreaChart draws a stacked area chart. A stacked area chart is a two-dimensional graph, meaning the graph has an X axis and a Y axis. Area charts typically have two numerical axes. A numerical axis is represented by the JavaFX class `javafx.scene.chart.NumberAxis`.

You need to define the X axis and Y axis used by a StackedAreaChart. Here is an example of creating two JavaFX NumberAxis instances:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("Last 7 Days");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Visits");
```

## Creating a StackedAreaChart

You create a JavaFX `StackedAreaChart` component by creating an instance of the `StackedAreaChart` class. You need to pass an X axis and Y axis to the `StackedAreaChart` constructor. Here is a JavaFX `StackedAreaChart` instantiation example:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("No of employees");

NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Revenue per employee");

StackedAreaChart stackedAreaChart = new StackedAreaChart(xAxis, yAxis);
```

## StackedAreaChart Data Series

To get a JavaFX `StackedAreaChart` component to display any dots, you must provide it with a *data series*. A data series is a list of data points. Each data point contains an X value and a Y value.

To see any stacked areas, you must add two or more data series to the `StackedAreaChart` component.

The `StackedAreaChart` displays the values from the data series stacked on top of each other.

Here is an example of creating two data series and adding them to a `StackedAreaChart` component:

```
XYChart.Series dataSeries1 = new XYChart.Series();
dataSeries1.setName("Desktop");

dataSeries1.getData().add(new XYChart.Data( 0, 567));
dataSeries1.getData().add(new XYChart.Data( 1, 612));
dataSeries1.getData().add(new XYChart.Data( 2, 800));
dataSeries1.getData().add(new XYChart.Data( 3, 780));
dataSeries1.getData().add(new XYChart.Data( 4, 650));
dataSeries1.getData().add(new XYChart.Data( 5, 610));
dataSeries1.getData().add(new XYChart.Data( 6, 590));

stackedAreaChart.getData().add(dataSeries1);

XYChart.Series dataSeries2 = new XYChart.Series();
dataSeries2.setName("Mobile");

dataSeries2.getData().add(new XYChart.Data( 0, 101));
dataSeries2.getData().add(new XYChart.Data( 1, 110));
```

```
dataSeries2.getData().add(new XYChart.Data( 2, 140));
dataSeries2.getData().add(new XYChart.Data( 3, 132));
dataSeries2.getData().add(new XYChart.Data( 4, 115));
dataSeries2.getData().add(new XYChart.Data( 5, 109));
dataSeries2.getData().add(new XYChart.Data( 6, 105));

stackedAreaChart.getData().add(dataSeries2);
```

The data in these data series represents the data for visits by users on desktop and mobile devices 7 days back.

## Adding a StackedAreaChart to the Scene Graph

To make a JavaFX `StackedAreaChart` visible you must add it to the JavaFX scene graph. This means adding the `StackedAreaChart` to a `Scene` object or add the `AreaChart` to a layout component which is added to a `Scene` object.

Here is an example that adds a `StackedAreaChart` to the JavaFX scene graph:

```
package com.jenkov.javafx.charts;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.StackedAreaChart;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class StackedAreaChartExperiments extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("StackedAreaChart Experiments");

        NumberAxis xAxis = new NumberAxis();
        xAxis.setLabel("7 Day Interval");

        NumberAxis yAxis = new NumberAxis();
        yAxis.setLabel("Visits");

        StackedAreaChart stackedAreaChart = new StackedAreaChart(xAxis,
yAxis);

        XYChart.Series dataSeries1 = new XYChart.Series();
        dataSeries1.setName("Desktop");
```

```

dataSeries1.getData().add(new XYChart.Data( 0, 567));
dataSeries1.getData().add(new XYChart.Data( 1, 612));
dataSeries1.getData().add(new XYChart.Data( 2, 800));
dataSeries1.getData().add(new XYChart.Data( 3, 780));
dataSeries1.getData().add(new XYChart.Data( 4, 650));
dataSeries1.getData().add(new XYChart.Data( 5, 610));
dataSeries1.getData().add(new XYChart.Data( 6, 590));

stackedAreaChart.getData().add(dataSeries1);

XYChart.Series dataSeries2 = new XYChart.Series();
dataSeries2.setName("Mobile");

dataSeries2.getData().add(new XYChart.Data( 0, 101));
dataSeries2.getData().add(new XYChart.Data( 1, 110));
dataSeries2.getData().add(new XYChart.Data( 2, 140));
dataSeries2.getData().add(new XYChart.Data( 3, 132));
dataSeries2.getData().add(new XYChart.Data( 4, 115));
dataSeries2.getData().add(new XYChart.Data( 5, 109));
dataSeries2.getData().add(new XYChart.Data( 6, 105));

stackedAreaChart.getData().add(dataSeries2);

VBox vbox = new VBox(stackedAreaChart);

Scene scene = new Scene(vbox, 400, 200);

primaryStage.setScene(scene);
primaryStage.setHeight(300);
primaryStage.setWidth(1200);

primaryStage.show();
}

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

```

The application resulting from running this example would look similar to this:

