Building Desktop Applications in Aussom with JavaFX

A Beginner's Guide

by Austin Lehman

Copyright

Building Desktop Applications in Aussom with JavaFX: A Beginner's Guide

Copyright © 2026 Austin Lehman. All rights reserved.

This book is provided for your personal use. You may download and keep one copy of this book for your own personal, non-commercial use.

No part of this book may be reproduced, distributed, republished, resold, or transmitted in any form or by any means — electronic, mechanical, photocopying, recording, or otherwise — without the prior written permission of the copyright holder.

The information in this book is provided "as is," without warranty of any kind. The author shall not be liable for any loss or damage arising from its use.

The example code accompanying this book is distributed separately under the **Apache License, Version 2.0**, and you are free to use, modify, and share it under that license's terms.

For permission requests, contact: austin@rosevillecode.com

Revision 1 (2026)

Contents

Introduction 6

Preface 7

I Getting Started 9

Chapter 1 - Hello, JavaFX 10

Chapter 2 - The Application Lifecycle 16

Chapter 3 - The Scene Graph 21

II Layout 27

Chapter 4 - Thinking in Layouts 28

Chapter 5 - The Core Layout Panes 33

Chapter 6 - Splitting, Scrolling, and Grouping 43

III Controls and User Input 50

Chapter 7 - Labels and Buttons 51

Chapter 8 - Handling Events 59

Chapter 9 - Text Input 67

Chapter 10 - Selection Controls 72

Chapter 11 - Lists, Tables, and Trees 79

Chapter 12 - Menus, Toolbars, and Tooltips 86

Chapter 13 - Dialogs and File Choosers 92

IV Text, Color, and Style 97

Chapter 14 - Text and Typography 98

Chapter 15 - Color and Paint 102

Chapter 16 - Styling with CSS 106

V Graphics and Media 110

Chapter 17 - 2D Shapes 111

Chapter 18 - Drawing on a Canvas 116

Chapter 19 - Images 120

Chapter 20 - Audio and Video 123

Chapter 21 - Embedding Web Content 126

VI Visual Effects and Animation 131

Chapter 22 - Effects 132

Chapter 23 - Transforms 137

Chapter 24 - Animation 140

VII Properties and Reactive UI 143

Chapter 25 - Observable Properties 144

Chapter 26 - Binding the UI to Data 147

VIII 3D Graphics 151

Chapter 27 - The 3D Scene 152

Chapter 28 - 3D Shapes and Meshes 156

Chapter 29 - Materials and Lighting 162

IX Charts 168

Chapter 30 - Charting Data 169

X Rich Text Editing **180**

Chapter 31 - Rich Text Controls 181

XI Extended Controls **186**

Chapter 32 - Beyond the Core Controls 187

XII Shipping Your Application **192**

Chapter 33 - Packaging and Distribution 193

Chapter 34 - A Complete Application 197

XIII Game Development with FXGL **201**

Chapter 35 - What FXGL Is 202

Appendix A - The `fx` Module Reference at a Glance 205

Appendix B - Mapping JavaFX Classes to Aussom Wrappers 207

Appendix C - Troubleshooting and Lessons Learned 212

Introduction

I started on this book in an effort to provide a true Aussom and JavaFX beginner guide, one that covers all the core JavaFX concepts and provides examples and the API references to get you started. This book is just that, a straightforward guide for kickstarting your JavaFX development in Aussom. This book is one of a number of books for the Aussom Programming Language that I plan to write to provide anyone an easy path to getting started.

I want to say a quick thank you to all the people who helped to get Aussom and JavaFX to where it is today, and to those whose libraries we directly or indirectly use in Aussom and the Aussom JavaFX bindings. There are too many to mention each by name here, but there are countless open-source libraries written by hardworking individuals on their own time that help to make this all possible. Many thanks to them and please keep them in mind when you read this book and write programs in Aussom.

Finally, I want to say thank you to you! You are the one who is making the time to learn Aussom and JavaFX, and it's you who make all of this worthwhile. Thank you for your patience and if you find anything incorrect or have any suggestions please reach out, I would love to hear from you.

Warm Regards,

Austin Lehman

austin@rosevillecode.com

Preface

Welcome! You are about to learn how to build **desktop applications** in the **Aussom** programming language using **JavaFX**.

A desktop application is a program with a real window on your screen, the kind you click and type in, with buttons, text boxes, menus, and pictures. Word processors, music players, and chat apps are all desktop applications. By the end of this book, you will be able to build your own.

About This Book

This book is a hands-on guide. You will not just read about ideas, you will write real programs and watch them run. We start at the very beginning, with a window that says “Hello”, and build up one step at a time. Each chapter adds a new piece, so by the end you will understand how a complete desktop application fits together.

You do not need to know JavaFX. You do not need to have built an app before. We explain every new word the first time it shows up, and we walk through what each line of code does and why.

Who This Book Is For

This book is for **beginners**. It is written for people who are new to programming, including:

- Students and self-taught learners just getting started.
- Developers who know a little code but have never built a desktop app.
- Curious people who are not developers yet, but want to be.
- Experienced developers who are new to JavaFX.

If you are already an experienced developer who just needs to look up a class or a method, you may prefer the **API reference documentation** at aussom-lang.com, which lists every type and function in detail. This book takes the slower, friendlier path meant for learning.

How to Read This Book

Read the chapters **in order**. Each one builds on the chapter before it, so skipping around can leave gaps.

The best way to learn is to **type the examples yourself and run them**. Reading code is useful, but running it, breaking it, and fixing it is how the ideas stick. Do not worry if something does not fully

click on the first read. Keep going, run the example, and it will make more sense in practice.

Conventions Used

A few simple conventions appear throughout the book:

- Code you write in Aussom looks like this:

```
app = fx.fxApp("Hello JavaFX!", 220, 120);
```

- Commands you type into a terminal look like this, and the program's output (if any) follows:

```
aussom hellojavafx.aus
```

- File names, class names, and short pieces of code inside a sentence are shown in `fixed-width` text, for example the `fx.fxApp` function or the `hellojavafx.aus` file.
- Screenshots show what you should see on your own screen when you run an example.

We speak directly to **you**, the reader, because you are the one building these apps.

Getting the Example Code

Every example in this book lives in a public code repository so you can read, run, and change it:

- **github.com/rsv-code/aussom-with-javafx-book-examples**

Each chapter has its own numbered directory in that repository. For example, Chapter 1's example is in `01-hello-javafx`.

To keep the book readable, we usually show only the important parts of a program in the text and link to the full, runnable version in that repository. You are encouraged to download it and run it as you read.

Let's begin.

Part I

Getting Started

Chapter 1 - Hello, JavaFX

In this chapter you will build your very first desktop application: a small window that shows a friendly greeting. It is a tiny program, but by the time you understand every line of it, you will already know the main pieces that every JavaFX app is made of.

Let's start with the big picture, then build the app, run it, and take it apart to see how it works.

What JavaFX Is and Why Aussom Uses It

When you use a program by clicking buttons, typing in boxes, and dragging windows around, you are using its **GUI**. GUI stands for *graphical user interface*, the visual part of a program that you see and interact with. (The "G" is pronounced like the "g" in "good": *GOO-ee*.)

Building a GUI from scratch is a huge job. You would have to draw every button, track every mouse click, and arrange everything on the screen yourself. To save everyone that work, programmers use a **toolkit**: a ready-made collection of windows, buttons, text labels, and other parts you can reuse.

JavaFX is one of these toolkits. It is a mature, widely used toolkit for building desktop GUIs, and it runs on the Java platform. It already knows how to draw a window, show text, handle clicks, and much more.

JavaFX has been around for a while. Sun Microsystems first released it in 2008, and Oracle took over the project when it acquired Sun in 2010. For years JavaFX came bundled inside the Java Development Kit, but in 2018 Oracle moved it into a separate open-source project called **OpenJFX**, maintained today in large part by a company named **Gluon**. Because it is free, open source, and built on top of Java, it runs almost everywhere Java does: Windows, macOS, and Linux on the desktop, small boards like the Raspberry Pi, and even mobile phones running Android and iOS.

Aussom runs on that same Java platform, so it can use JavaFX too. But instead of making you write Java, Aussom wraps JavaFX in its own friendly classes inside a module called `fx`. That means **you get the power of a proven UI toolkit while writing simple Aussom code.** Throughout this book, anything starting with `fx.` is Aussom's doorway into JavaFX.

Installing the Aussom CLI

To run the programs in this book, you need the **Aussom CLI** installed on your computer. CLI stands for *command line interface*. It is the `aussom` program that you run by typing commands into a **terminal** (the text window where you type commands instead of clicking).

Install the Aussom CLI by following the instructions at aussom-lang.com. The installer puts a program named `aussom` on your system so you can run it from any folder.

Once it is installed, open a terminal and check that it works by asking for its version:

```
aussom --version
```

You should see something like this:

```
Aussom-Lang Version 1.2.0
Copyright 2026 Austin Lehman
```

The exact version number may be higher when you read this; that is fine. If you see a version line, you are ready to go. If instead you see an error like "command not found", the install did not finish correctly. Revisit the install instructions on the website before moving on.

Your First Window

Here is the complete program. Create a file named `hellojavafx.aus` and type this into it:

```
include fx;
include fx.StackPane;
include fx.Label;

/**
 * A first JavaFX application in Aussom. It opens a window and
 * shows a short greeting, then waits until you close the window.
 */
class HelloJavaFX {
    /**
     * The program's entry point. Aussom runs this method when the
     * application starts.
     * @p args is a list of command line arguments (unused here).
     */
    public main(args) {
        // Create the application window: title, width, and height.
        app = fx.fxApp("Hello JavaFX!", 220, 120);

        // Create a layout to hold our content, then add a label to it.
        layout = new StackPane();
        layout.add(new Label("Hello, Aussom FX!"));

        // Put the layout inside the window.
        app.setLayout(layout);

        // Show the window and wait here until it is closed.
        app.show(true);

        // Cleanly shut JavaFX down once the window closes.
```

```
        fx.shutdown();
    }
}
```

Now run it from the terminal, in the same folder where you saved the file:

```
aussom hellojavafx.aus
```

A small window opens and looks like this:

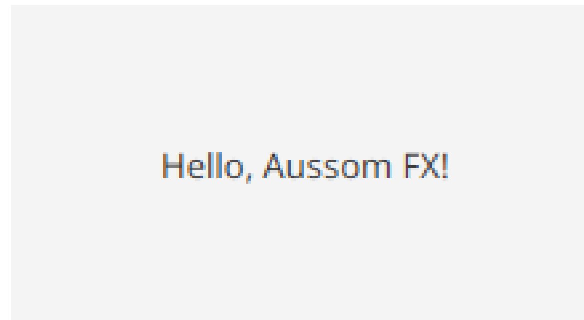


Figure 1: The Hello, JavaFX window showing the greeting "Hello, Aussom FX!"

That's it, you just built and ran a real desktop application! Close the window to end the program. You can find the full, ready-to-run example here: `01-hello-javafx`.

If nothing happened or you got an error, double-check that you typed the file exactly as shown and that you ran the command in the same folder as the file.

Running and Stopping the App

Three lines control the **life** of the application, from the moment it appears to the moment it shuts down. Understanding them now will save you confusion later, because every app in this book follows the same pattern.

```
app = fx.fxApp("Hello JavaFX!", 220, 120);   // 1. create the window
app.show(true);                               // 2. show it and wait
fx.shutdown();                                // 3. shut down when done
```

1. **Create the window.** `fx.fxApp(...)` builds the application window but does not show it yet. Think of it as setting up a stage before the curtain rises.
2. **Show it and wait.** `app.show(true)` makes the window appear on screen. The `true` means "wait here until the user closes the window." Your program pauses on this line, keeping the window open, until you close it. (This pausing is called **blocking**: the line blocks, or holds, the program in place.)
3. **Shut down.** Once you close the window, `app.show(true)` finishes and the program continues to the next line. `fx.shutdown()` then tells JavaFX to clean up and fully stop. Always end a JavaFX app this way so it closes neatly.

Anatomy of the Hello World Program

Now let's walk through the whole program, piece by piece, so nothing is a mystery.

Bringing in the parts we need

```
include fx;
include fx.StackPane;
include fx.Label;
```

The `include` keyword pulls in code that someone else already wrote so you can use it. The first line includes the whole `fx` module (Aussom's JavaFX layer). The next two lines include the two specific JavaFX parts we use in this app: a `StackPane` and a `Label`. Including a class is what lets you create one with new later on.

The class and its starting point

```
class HelloJavaFX {
    public main(args) {
        ...
    }
}
```

Aussom programs are organized into **classes**. A class is a named container for related code. Inside it, `main` is a special **method** (a named block of code) that Aussom runs first when your app starts. So `main` is the *entry point*, where everything begins. The `args` in parentheses is a list of any extra words you type after the file name on the command line. We do not use it here, but it has to be there.

Creating the window

```
app = fx.fxApp("Hello JavaFX!", 220, 120);
```

This creates the application window and stores it in a variable named app. The three values inside the parentheses are the window's **title** (the text in its title bar), its **width**, and its **height** in pixels. So this makes a window 220 pixels wide and 120 pixels tall, titled "Hello JavaFX!".

Creating a place to put content

```
layout = new StackPane();
```

A window cannot hold text directly. It needs a **layout** first: a container whose job is to hold your content and decide where each piece goes. We use a `StackPane`, a simple layout that centers whatever you put inside it. The new keyword means "make a fresh one."

Creating the greeting and adding it

```
layout.add(new Label("Hello, Aussom FX!"));
```

A `Label` is a piece of text on screen. We make a new one that says "Hello, Aussom FX!" and immediately add it to the layout. Now the layout holds our label.

Putting the layout into the window

```
app.setLayout(layout);
```

This places the layout, and everything inside it, into the window. Now the window has content to show.

Showing it and shutting down

```
app.show(true);
fx.shutdown();
```

These are the two lifecycle lines we covered above: show the window and wait, then shut down cleanly after it closes.

How the pieces fit together

Notice that each piece lives *inside* another, like a set of nested boxes: the label goes inside the layout, the layout goes inside the window. JavaFX arranges all the visual parts of your app into a tree like this, called the **scene graph** (you can think of it as the "family tree" of everything on screen). The diagram below shows the tree for our little app, along with the Aussom code that creates each part:

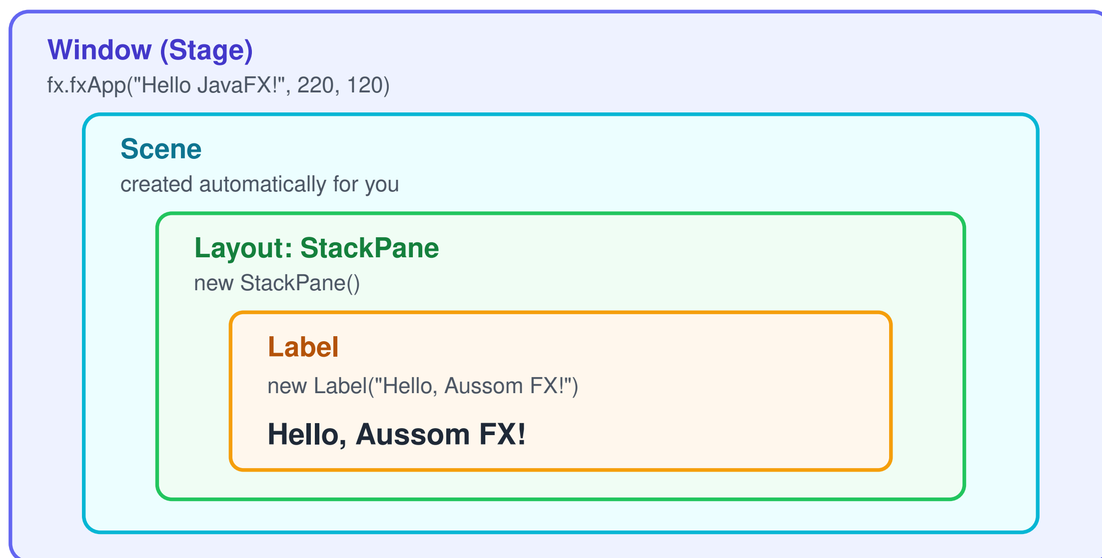


Figure 2: A diagram of nested boxes: the Window contains a Scene, which contains a StackPane layout, which contains the Label "Hello, Aussom FX!"

There is one box in the diagram we did not write ourselves: the **Scene**. A Scene is the surface that holds all the visual content of a window. Aussom creates it for you automatically when you set the layout, so you usually do not have to think about it. We will meet the scene graph again in Chapter 3.

Recap

You covered a lot for a first chapter. Here is what you now know:

- A **GUI** is the visual part of a program, and **JavaFX** is the toolkit Aussom uses to build one, through its `fx` module.
- The **Aussom CLI** runs your programs; `aussom --version` confirms it is installed, and `aussom hellojavafx.aus` runs an app.
- Every app **creates a window** with `fx.fxApp(...)`, **shows it** with `app.show(true)`, and **shuts down** with `fx.shutdown()`.
- Content lives inside a **layout** (like `StackPane`), which lives inside the window, forming the **scene graph**.

In the next chapter, we will look more closely at this lifecycle, the special thread JavaFX uses to draw your app, and how to run code at the right time.

Chapter 2 - The Application Lifecycle

In Chapter 1 you built a window with three lines that controlled its life: create it, show it, shut it down. In this chapter we slow down and look at that life more closely. You will learn the proper names for the parts of a window, meet the special "thread" that JavaFX uses to draw everything, and learn how to run your own code at just the right moments.

These ideas are the foundation for everything that follows, so it is worth taking your time here.

The Stage, the Scene, and the Layout

JavaFX borrows its names from the theater. A play happens on a **stage**, and on that stage you set a **scene**. JavaFX uses the same words:

- The **Stage** is the window itself: the frame, the title bar, and the close button. When you call `fx.fxApp(...)`, you are creating a Stage.
- The **Scene** is the surface inside the window that holds your content. JavaFX creates it for you automatically.
- The **Layout** is a container that sits in the scene and arranges your content, like the `StackPane` from Chapter 1.

So your content lives in a layout, the layout lives in the scene, and the scene lives in the stage (the window). You already drew this as nested boxes in Chapter 1; now you know the real names for each box.

The JavaFX UI Thread

To understand the next part, you need one new idea: a **thread**.

A thread is a single line of work that the computer follows, one step after another. A program can have several threads running at the same time, like several workers each doing their own task at once.

JavaFX has one very important rule about threads:

> **All drawing and all changes to the user interface happen on one special thread, called the UI thread** (you may also see it called the *JavaFX Application Thread*).

Why does this matter? Because if you change something on the screen, a label's text, say, from a *different* thread, JavaFX can become confused, glitch, or even crash. Keeping all UI changes on one thread keeps everything orderly.

Here is the catch: your `main` method does **not** run on the UI thread. It runs on the thread that launched your program. So you cannot assume it is safe to change the screen directly from `main`. You need a way to hand your work over to the UI thread, and that is exactly what the next function does.

Running Work Later with runLater

`fx.runLater(...)` takes one of your methods and schedules it to run **on the UI thread**, as soon as JavaFX is able to. The name says it well: "run this a little later, in the right place."

```
fx.runLater(::onUiThread);
```

The `::onUiThread` part means "the method named `onUiThread`." (The `::` is how you hand a method to another function instead of calling it yourself.) Whenever you need to change the user interface from code that is not already on the UI thread, wrap that work in `fx.runLater(...)` and you are safe.

Checking the UI Thread with isUiThread

How can you tell which thread your code is running on? Ask:

```
fx.isUiThread()
```

It returns `true` if the code calling it is on the UI thread, and `false` otherwise. You will not use it every day, but it is a great way to *see* the rule in action. In the example below, we call it from two places: from `main` (where it reports `false`) and from inside a `runLater` method (where it reports `true`).

Running Code at the Right Moment

Often you want to run code exactly when something happens to the window. JavaFX lets you register your own methods for two key moments:

```
app.setOnShow(::onShown);       // runs when the window appears
app.setOnClosing(::onClosing);  // runs as the window is closing
```

`setOnShow` is handy for work that should happen once the window is visible. `setOnClosing` is your chance to do cleanup, like saving the user's work, right before the window goes away. Both of these methods run on the UI thread.

Clean Shutdown

When the user closes the window, three things happen in order:

1. Your `onClosing` method runs, if you registered one.
2. The `app.show(true)` line finishes, and your program continues to the next line.
3. `fx.shutdown()` tells JavaFX to stop completely and tidy up.

Always end a JavaFX program with `fx.shutdown()`. Without it, parts of JavaFX may keep running in the background even after your window is gone.

This whole journey, from creation to shutdown, is the **application lifecycle**:

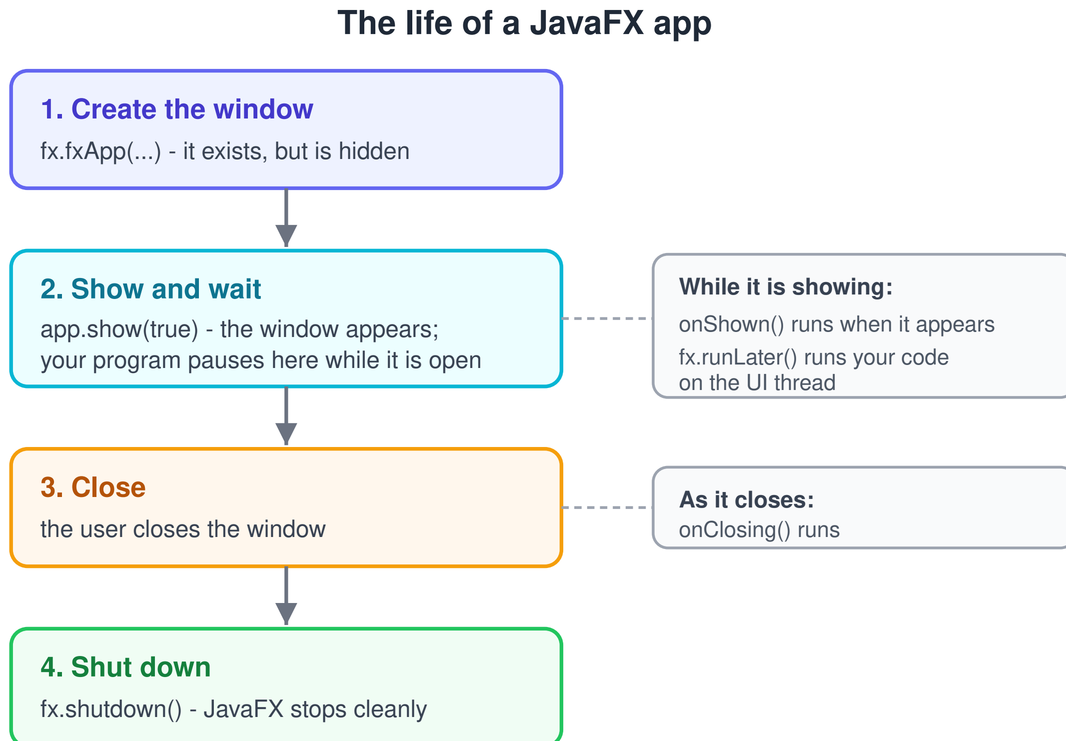


Figure 3: A flow diagram of the application lifecycle: create the window, show and wait, close, shut down, with notes that onShown and runLater run while showing and onClosing runs as it closes

The Full Example

Here is a small program that shows every idea from this chapter working together:

```
include fx;
include fx.StackPane;
include fx.Label;

class Lifecycle {
    private app = null;
    private label = null;

    public main(args) {
        this.app = fx.fxApp("Application Lifecycle", 380, 140);
```

```
        this.label = new Label("Starting up ...");
        layout = new StackPane();
        layout.add(this.label);
        this.app.setLayout(layout);

        // main() runs on the launcher thread, not the UI thread.
        c.info("Is main() on the UI thread? " + fx.isUiThread());

        // Ask JavaFX to run our method on the UI thread.
        fx.runLater(::onUiThread);

        // Run our code when the window is shown and when it closes.
        this.app.setOnShow(::onShown);
        this.app.setOnClosing(::onClosing);

        this.app.show(true);
        fx.shutdown();
    }

    public onUiThread() {
        c.info("Is onUiThread() on the UI thread? " + fx.isUiThread());
        this.label.setText("Hello from the UI thread!");
    }

    public onShown() {
        c.info("The window is now showing.");
    }

    public onClosing() {
        c.info("The window is closing. Goodbye!");
    }
}
```

A few things to notice:

- The two lines at the top, `private app = null;` and `private label = null;`, create variables that belong to the whole object instead of to a single method. That way every method (`main`, `onUiThread`, `onShown`, `onClosing`) can share them. `null` just means “nothing yet”; we fill them in inside `main`.
- We then write `this.app` and `this.label` to use those shared variables. The `this.` prefix means “this object’s own variable.”
- `c.info(...)` prints a line of text to the terminal. The `c` is Aussom’s built-in console, and `.info(...)` writes one message to it. We use it here to watch the lifecycle happen; it does not change the window at all.
- The label starts out saying “Starting up …”, but `onUiThread` changes it with `this.label.setText(...)`. Because that change happens on the UI thread (thanks to `runLater`), it is safe.

Run it:

```
aussom lifecycle.aus
```

The window opens and shows the updated greeting:

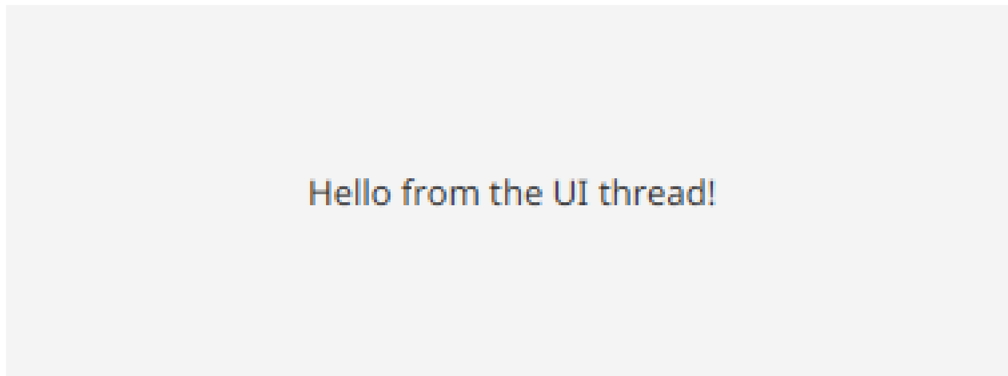


Figure 4: The Application Lifecycle window showing the text "Hello from the UI thread!"

And in your terminal, you will see the lifecycle play out, including proof of which thread each method ran on:

```
Is main() on the UI thread? false
Is onUiThread() on the UI thread? true
The window is now showing.
```

(The "The window is closing. Goodbye!" line appears when you close the window.)

Look closely at the first two lines. The very same question, "are we on the UI thread?", gives a different answer depending on *where* it is asked: `false` in `main`, but `true` inside the method we handed to `runLater`. That is the rule from this chapter, proven in front of you.

The full example is here: `02-application-lifecycle`.

Recap

- A JavaFX window has three nested parts: the **Stage** (the window), the **Scene** (its surface), and a **Layout** (which arranges your content).
- JavaFX does all its drawing on one **UI thread**, and you must only change the user interface from that thread.
- Your `main` method is **not** on the UI thread. Use **`fx.runLater(...)`** to hand UI work to the UI thread, and **`fx.isUiThread()`** to check where you are.
- **`setOnShow`** and **`setOnClosing`** let you run code when the window appears and when it closes.
- Always finish with **`fx.shutdown()`** for a clean exit.

Next, we will look at the **scene graph**: the tree of visual pieces that makes up everything you see in a JavaFX window.

Chapter 3 - The Scene Graph

You have already seen that content nests inside a layout, which sits in a scene, which lives in a window. That whole arrangement of visual pieces has a name: the **scene graph**. In this chapter we look at it directly. You will learn what a "node" is, how nodes form a tree of parents and children, how Aussom wraps the underlying JavaFX objects, and how to read and change nodes while your program is running.

Nodes, Parents, and the Tree

In JavaFX, every single visual piece is called a **node**. A label is a node. A button is a node. Even a layout is a node. If you can see it (or it holds things you can see), it is a node.

Nodes connect together into a **tree**. A node that contains other nodes is a **parent**, and the nodes inside it are its **children**. A layout is a parent: its whole job is to hold child nodes. In this chapter we use a **VBox** (short for *vertical box*), a layout that arranges its children in a vertical list, one above the next. In Chapter 1 we used a `StackPane`, which instead stacks its children on top of one another and centers them; a `VBox` is a better fit here because we want to see several labels lined up. (We cover all the layouts in Chapter 5; for now, just picture a `VBox` as a stack of shelves.) A `Label`, by contrast, is usually a **leaf**: a node with no children of its own.

This tree of nodes *is* the scene graph. Here is the tree for the example we will build in this chapter, a `VBox` parent holding three `Label` children:

Notice that all three labels are part of the tree, even the one marked "hidden." Being in the tree and being *visible* are two different things, as you will see shortly.

The FxObj Wrapper and the .obj Handle

Now is a good time to explain something we glossed over in Chapter 1.

When you write `new Label("Hi")`, Aussom does not hand you a raw JavaFX object. It hands you a friendly **wrapper**: an Aussom object that holds the real JavaFX object inside it and gives you simple methods like `setText`, `setOpacity`, and `setVisible`. Every `fx` node is a wrapper like this. They all share a common base called `FxObj`, and most build on a richer base called `Node` that provides the behavior every visual node has in common.

Inside each wrapper, the real JavaFX object is stored in a member named `obj`. So `label.obj` is the actual JavaFX label underneath Aussom's wrapper. You will rarely need it: `fx` methods accept wrappers directly and unwrap them for you (that is why, back in Chapter 1, we could write

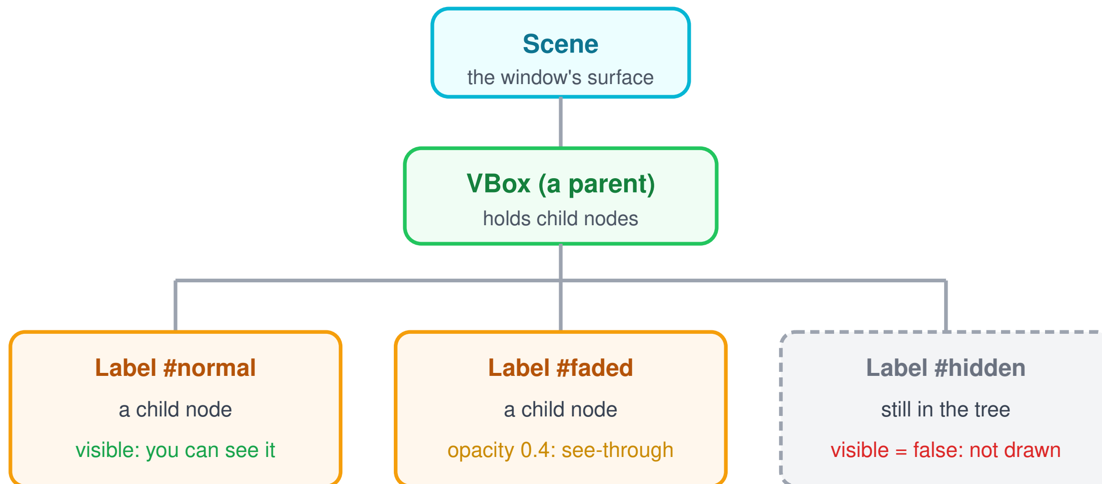


Figure 5: A tree diagram: the Scene contains a VBox parent, which has three Label children with ids normal, faded, and hidden

`app.setLayout(layout)` instead of `app.setLayout(layout.obj)`). Just know that `.obj` exists for the occasional advanced case when some code needs the raw JavaFX object.

For everyday work, the rule is simple: **use the wrapper and its friendly methods.**

Identity, Visibility, and Opacity

Because every node builds on `Node`, they all share a useful set of properties. Here are three you will reach for constantly.

Identity (an id). You can give a node a name tag with `setId`, and read it back with `getId`:

```
label.setId("greeting");
label.getId();              // "greeting"
```

An id lets you find a node later by searching for `#greeting`. The `#` is just a way of saying "the node whose id is this," so `#greeting` means "the node with the id `greeting`." You will see how to search with it using `lookup` near the end of this chapter.

Visibility. You can show or hide a node:

```
label.setVisible(false);    // hide it
label.isVisible();          // false
```

A hidden node is not drawn, but it is **still in the tree** and still reserves its space in the layout. It is there; you just cannot see it.

Opacity. Opacity controls how see-through a node is, from `0.0` (completely transparent) to `1.0` (completely solid):

```
label.setOpacity(0.4);      // 40% visible, faded
label.getOpacity();         // 0.4
```

There is one more shared property worth mentioning: a node can change the **mouse cursor** that appears when you hover over it (for example, a pointing hand over a link). Setting the cursor nicely involves styling, which gets its own chapter later, so we will leave it there for now, just know it is a node property too.

Reading the Scene Graph at Runtime

The scene graph is not just something you build once and forget. Your program can explore it while it runs. Three methods do most of the work.

`lookup` finds a node by id. Starting from any parent, `lookup` searches the whole tree beneath it for a node whose id matches:

```
found = box.lookup("#hidden");
```

If a matching node exists, `lookup` hands it back; if nothing matches, it hands back `null` (Aussom's word for "nothing"). So checking `found != null` is just asking "did we actually find a node?" This even finds hidden nodes, because they are still in the tree.

`getParent` walks up the tree. It returns the node that contains a given node:

```
label.getParent();          // the VBox that holds the label
```

`getScene` finds the scene. Once a node is showing, this returns the scene it belongs to:

```
label.getScene();           // the scene, after the window is shown
```

Together, these let your code answer questions like "where is this node?" and "what is around it?" without you having to remember every reference yourself.

The Full Example

This program builds the three-label tree from the diagram, hides one label, fades another, and then reads facts back out of the scene graph once the window appears:

```
include fx;
include fx.VBox;
include fx.Label;

class SceneGraph {
    private app = null;
    private box = null;
    private normal = null;
    private faded = null;
```

```
    private hidden = null;

    public main(args) {
        this.app = fx.fxApp("The Scene Graph", 420, 180);

        this.normal = new Label("1. A normal label (you can see me).");
        this.normal.setId("normal");

        this.faded = new Label("2. A faded label (opacity 0.4).");
        this.faded.setId("faded");
        this.faded.setOpacity(0.4);

        this.hidden = new Label("3. A hidden label (you cannot see me).");
        this.hidden.setId("hidden");
        this.hidden.setVisible(false);

        this.box = new VBox();
        this.box.setId("box");
        this.box.setSpacing(8);
        this.box.setAlignment(fxpos.CENTER);
        this.box.add([this.normal, this.faded, this.hidden]);
        this.app.setLayout(this.box);

        this.app.setOnShow(::readSceneGraph);
        this.app.show(true);
        fx.shutdown();
    }

    public readSceneGraph() {
        c.info("Reading the scene graph:");

        found = this.box.lookup("#hidden");
        c.info("  found the hidden label by id? " + (found != null));
        c.info("  is the hidden label visible? " + this.hidden.isVisible());
        c.info("  the faded label's opacity is " + this.faded.getOpacity());

        c.info("  the normal label has a parent? " + (this.normal.getParent()
↪   != null));
       c.info("  the normal label has a scene? " + (this.normal.getScene() !=
↪   null));
    }
}
```

A couple of new pieces appear here. `this.box.add([...])` adds several children at once by passing them in a list (the square brackets). And `setSpacing` and `setAlignment` are `VBox` settings that put a little gap between the labels and center them. (`fxpos.CENTER` is one of a set of named alignment positions; we explain `fxpos` in Chapter 4.) We cover layouts fully in Chapter 5, so do not

worry about the details yet.

Run it:

```
aussom scenegraph.aus
```

The window appears like this:

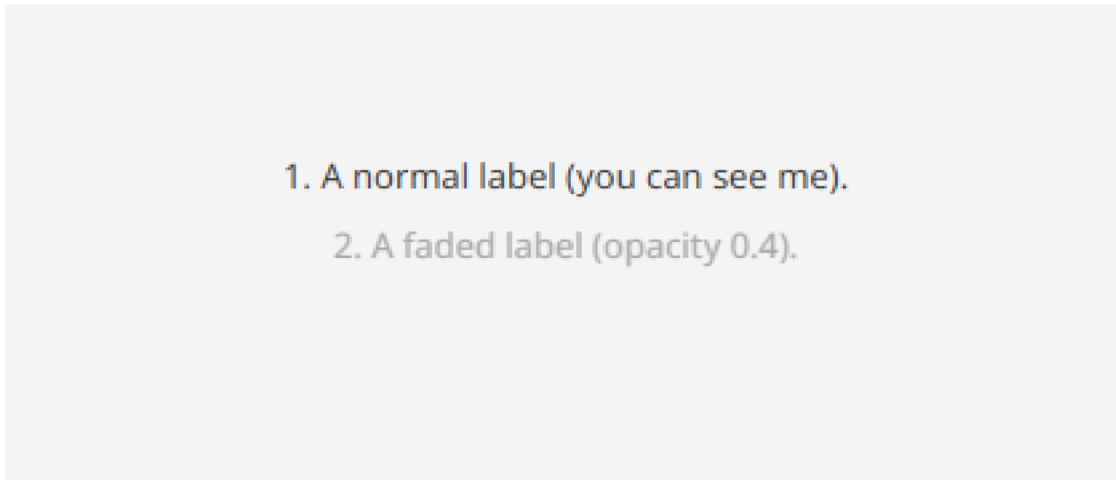


Figure 6: The Scene Graph window showing a solid first label, a faded second label, and no third label because it is hidden

Look at what you see and what you do not:

- **Label 1** is solid and easy to read.
- **Label 2** is faded, because we set its opacity to `0.4`.
- **Label 3** is nowhere on screen, because we made it invisible, yet notice the empty space below label 2 where it still sits in the layout.

Meanwhile, the terminal shows your program reading the tree back out:

```
Reading the scene graph:
  found the hidden label by id? true
  is the hidden label visible? false
  the faded label's opacity is 0.4
  the normal label has a parent? true
  the normal label has a scene? true
```

The first line is the key insight of this chapter: `lookup` **found** the hidden label, proving it is still in the tree, even though `isVisible` is `false` and you cannot see it on screen.

The full example is here: `03-scene-graph`.

Recap

- Every visual piece is a **node**, and nodes form a **tree** of **parents** and **children** called the **scene graph**.

- Aussom gives you each node as a friendly **wrapper** (built on `FxObj` and `Node`); the raw JavaFX object lives in `.obj`, which you rarely need.
- Every node shares properties like **id** (`setId/getId`), **visibility** (`setVisible/isVisible`), and **opacity** (`setOpacity/getOpacity`).
- A hidden node is **still in the tree** and still takes up its space; it is just not drawn.
- You can explore the tree at runtime with **`lookup`**, **`getParent`**, and **`getScene`**.

With windows, the lifecycle, and the scene graph behind you, you have the foundation to start arranging content with intention. In the next chapter, we dig into layouts: how JavaFX decides where each node goes.

Part II

Layout

Chapter 4 - Thinking in Layouts

So far you have used layouts without thinking about them much. You created a `StackPane` or a `VBox`, dropped some nodes in, and they appeared. That is on purpose: JavaFX is good at placing things for you. But to build real windows, you need to understand *how* that placement works, so you can pick the right layout and bend it to your needs.

This is a "thinking" chapter. There is one small example, but the goal is the ideas behind it. They apply to every layout you will meet in the rest of Part II.

How Panes Arrange Their Children

Here is the most important mindset shift in this whole part of the book:

> You do **not** place nodes at exact positions yourself. You put nodes into a **pane**, and the pane decides where each one goes.

A **pane** is just a layout container, a node whose job is to hold and position other nodes (its children). Each *kind* of pane follows its own strategy for arranging children. Give the very same three children to three different panes and you get three different results:

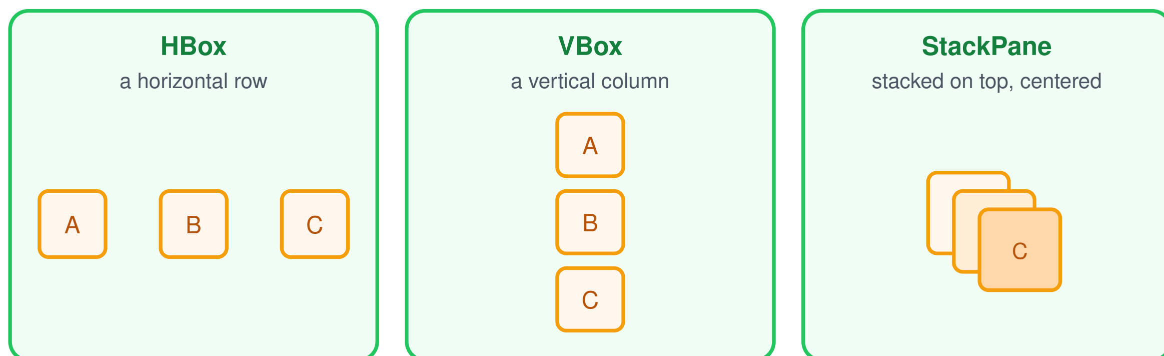


Figure 7: Three panels showing the same children A, B, C arranged by an HBox as a row, by a VBox as a column, and by a StackPane stacked on top of each other

- An **HBox** lays its children out in a horizontal **row**.
- A **VBox** lays them out in a vertical **column** (you saw this in Chapter 3).
- A **StackPane** stacks them on top of each other and centers them (you saw this in Chapter 1).

Because you choose the pane, you choose the arrangement. Here is a small program that puts three labels into an `HBox`:

```
include fx;
include fx.HBox;
include fx.Label;

class Layouts {
    private app = null;
    private box = null;

    public main(args) {
        this.app = fx.fxApp("Thinking in Layouts", 440, 160);

        one = new Label("One");
        two = new Label("Two");
        three = new Label("Three");

        // An HBox arranges its children in a horizontal row.
        this.box = new HBox();
        this.box.setSpacing(20);          // a 20 pixel gap between
                                          // children
        this.box.setAlignment(fxpos.CENTER);  // center the row in the
                                              // window
        this.box.add([one, two, three]);
        this.app.setLayout(this.box);

        this.app.show(true);
        fx.shutdown();
    }
}
```

Run it:

```
aussom layouts.aus
```

The three labels sit in a tidy row, with a gap between each, centered in the window. If you changed `HBox` to `VBox`, the very same labels would stack into a column instead, exactly like the labels you saw in Chapter 3. That is the whole idea: **same children, different pane, different arrangement.** The full example is here: `04-thinking-in-layouts`.

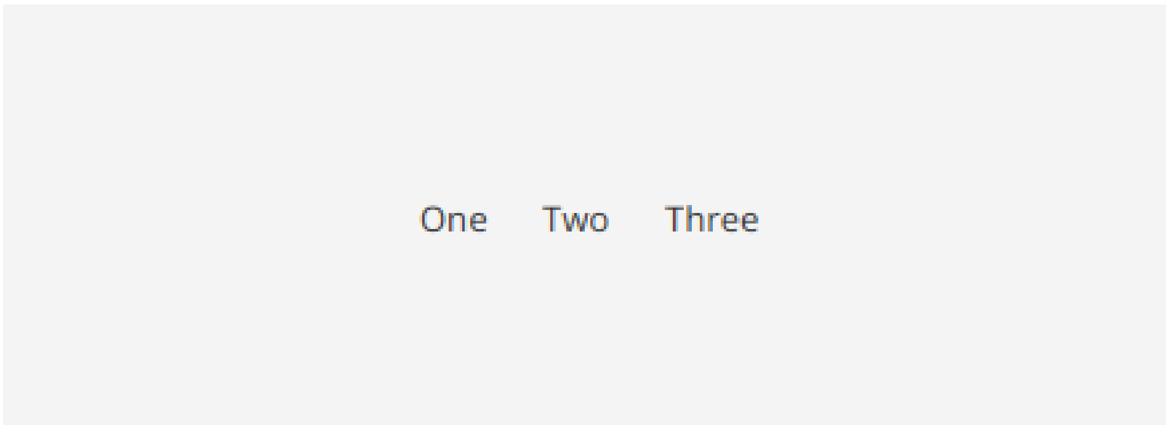


Figure 8: The Thinking in Layouts window showing the labels One, Two, and Three in a centered horizontal row

Sizing: Preferred, Minimum, and Maximum

Look again at the screenshot. The window is 440 pixels wide, but the row of labels only takes up a small strip in the middle. Why didn't the labels stretch to fill the window?

Because every node has a **preferred size**: the size it would *like* to be. For a label, the preferred size is just big enough to fit its text. The `HBox` asked each label how big it wanted to be, made it that size, and no bigger. The leftover space stayed empty (and the alignment, which we will cover in a moment, centered the row within it).

Along with a preferred size, a node has a **minimum size** (never shrink smaller than this) and a **maximum size** (never grow larger than this). Panes use all three when they decide how to share space, which matters most when the window is resized and there is suddenly more or less room to go around.

You can set these yourself with plain numbers (pixels):

```
label.setPrefWidth(200);    // the width it would like to be
label.setMinWidth(80);      // never narrower than this
label.setMaxWidth(300);     // never wider than this
label.setPrefSize(200, 40); // preferred width and height at once
```

Most of the time you will not set sizes by hand at all, you will let nodes use their natural preferred size and let the pane do the work. But it helps to know these knobs exist for when you need them.

Padding, Spacing, and Alignment

Three more settings let you fine-tune how a pane treats its children. Two of them appear in the example above.

Spacing is the gap *between* children. In the example, `setSpacing(20)` put 20 pixels of empty space between each label so they were not crammed together.

Alignment decides *where* the children sit inside the pane when there is extra room. We used `setAlignment(fxpos.CENTER)` to center the row.

`fxpos` is an **enum**: a fixed set of named values, here the alignment positions that JavaFX understands. It comes with the `fx` module, so `include fx;` is all you need to use it. Writing `fxpos.CENTER` is clearer than the bare text `"CENTER"`, and it is safer too: if you misspell it as `fxpos.CENTRE`, Aussom tells you right away, instead of the mistake slipping through and failing later. Common positions include `fxpos.CENTER`, `fxpos.TOP_LEFT`, `fxpos.CENTER_LEFT`, and `fxpos.BOTTOM_RIGHT`. Try changing `fxpos.CENTER` to `fxpos.TOP_LEFT` and the row will jump to the top-left corner. (The plain strings like `"CENTER"` still work, but the enum is the tidier, safer choice, and the one we use from here on.)

Padding is a cushion of empty space *inside* the pane's edges, between the edge of the pane and its children. (You may also see padding called **insets**; *padding* is the more common name, so that is what we will use.) Padding and spacing are easy to mix up, so here is the difference side by side:

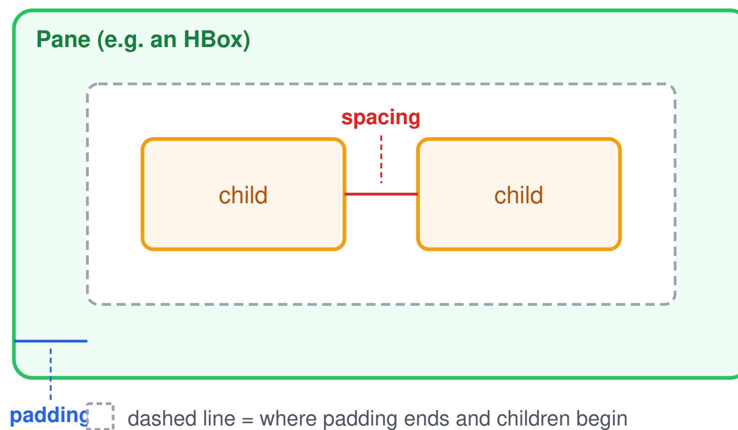


Figure 9: A diagram of a pane with a dashed inner boundary: padding is the cushion between the pane edge and the children, spacing is the gap between two children

- **Padding** pushes *all* the children inward, away from the pane's outer edge.
- **Spacing** only opens gaps *between* the children.

Setting padding needs an **`Insets`** value (a set of four edge cushions), but the quickest way is a one-line inline style, styling gets its full chapter later:

```
box.setStyle("-fx-padding: 10;");         // 10 pixels of cushion on all
                                          // four sides
box.setStyle("-fx-padding: 10 20 10 20;"); // or per-edge: top right
                                            // bottom left
```

For now, the important thing is the concept: padding is the cushion around the group, spacing is the gaps within it.

Recap

- You do not position nodes by coordinates. You put them in a **pane**, and the pane arranges them. The **kind** of pane decides the arrangement: **HBox** (row), **VBox** (column), **StackPane** (stacked), and more to come.
- Every node has a **preferred**, **minimum**, and **maximum** size. A node uses its preferred size by default; panes use all three to share space, especially when the window is resized.
- **Spacing** is the gap between children, **alignment** is where the children sit in the pane, and **padding** (also called **insets**) is the cushion inside the pane's edges.
- On top of these shared settings, each kind of pane adds options of its own: filling, wrapping, anchoring, spanning, and more. We meet them pane by pane in the next chapter.

Now that you understand how panes think, the next chapter tours the core layout panes one by one, so you can reach for the right one in any situation.

Chapter 5 - The Core Layout Panes

Chapter 4 explained *how* panes think: you place nodes in a pane, and the pane arranges them. This chapter introduces the core panes themselves, one by one. Each has its own personality and its own best use. By the end you will know which one to reach for in almost any situation.

Every example in this chapter lives here: `05-core-layout-panes`.

A note on the colors. To make it easy to see exactly where each pane places a node, the examples give every node a different background color. They do this with a small helper method, `cell`, that builds a colored label:

```
public cell(text, color) {
    lbl = new Label(text);
    lbl.setStyle("-fx-background-color: " + color + "; -fx-padding: 8 12;");
    return lbl;
}
```

So whenever you see `this.cell("Top", "#cfe2ff")` below, it just makes a label with that text on that background color. The coloring uses a small touch of styling, which gets its own chapter later (Chapter 16); here it is only to make the layout visible.

StackPane - Layered Stacking

You met the `StackPane` back in Chapter 1. It stacks its children on top of one another and centers them, like a deck of cards squared up in the middle of the table.

```
layout = new StackPane();
layout.add(new Label("Hello, Aussom FX!"));
layout.setAlignment(fxpos.BOTTOM_CENTER);   // where the stack sits;
                                            // default is CENTER
```

With a single child, a `StackPane` simply centers it, which is why it was perfect for our first "Hello" window. With several children, each new one is drawn on top of the last, so it is also the pane you use when you want to **layer** things, for example a caption on top of an image.

By default a `StackPane` centers everything; **`setAlignment`** takes an `fxpos` value (like `fxpos.BOTTOM_CENTER` above) to move the whole stack to another spot, handy for pinning that caption to the bottom of the image.

HBox and VBox - Rows and Columns

These two are the workhorses of everyday layout. An **HBox** arranges its children in a horizontal **row** (you used one in Chapter 4); a **VBox** arranges them in a vertical **column** (you used one in Chapter 3).

```
row = new HBox();
row.setSpacing(20);              // pixels of gap between children
row.setAlignment(fxpos.CENTER);  // an fxpos value: where the group sits
                                 // (Chapter 4)
row.add([one, two, three]);

column = new VBox();
column.setSpacing(8);            // pixels between children
column.setFillWidth(true);       // stretch every child to the column's
                                 // width
column.add([first, second, third]);
```

Both share the same handy settings: **`setSpacing`** (the gap between children, in pixels) and **`setAlignment`** (an `fxpos` value for where the group sits). Each can also stretch its children to fill the *other* direction with a `true/false` switch: **`setFillHeight`** on an `HBox` makes every item as tall as the row, and **`setFillWidth`** on a `VBox` makes every item as wide as the column, a tidy way to give a stack of buttons one even width. Reach for an `HBox` or a `VBox` whenever you need a simple line of items, like a toolbar (`HBox`) or a stacked form (`VBox`).

BorderPane - Five Regions

A **BorderPane** divides the window into five regions: **top**, **bottom**, **left**, **right**, and **center**. You place one node in each region you want to use, and you can leave any of them empty.

The four edge regions size themselves to fit their content, and the **center takes all the space that is left over**. That makes a BorderPane the natural skeleton for a whole application window: a menu or toolbar on top, a status bar on the bottom, and your main content in the center.

```
include fx;
include fx.BorderPane;
include fx.Label;

class BorderPaneDemo {
    public main(args) {
        app = fx.fxApp("BorderPane", 360, 220);

        bp = new BorderPane();
        bp.setTop(this.cell("Top", "#cfe2ff"));
        bp.setLeft(this.cell("Left", "#d1e7dd"));
        bp.setCenter(this.cell("Center", "#fff3cd"));
        bp.setRight(this.cell("Right", "#f8d7da"));
        bp.setBottom(this.cell("Bottom", "#e2d9f3"));
```

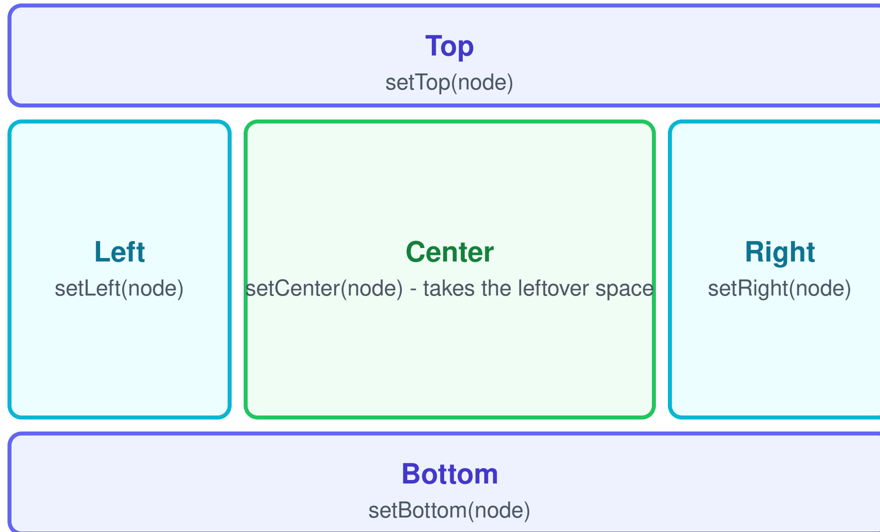


Figure 10: A diagram of a BorderPane split into Top and Bottom bars, Left and Right side panels, and a large Center area, each labeled with its setter method

```
        app.setLayout(bp);
        app.show(true);
        fx.shutdown();
    }
}
```

Any region can hold a whole layout of its own, not just a single label. That is how real windows are built: a toolbar (an HBox) in the top, a form (a GridPane) in the center, a status bar in the bottom. Leave any region out and the others simply take up the space.

FlowPane and TilePane - Wrapping Grids

Sometimes you have a bunch of items and you just want them to fill the space, wrapping onto new lines as needed. Two panes do this.

A **FlowPane** lays its children out one after another and **wraps** to the next line when it runs out of width, exactly like words wrapping in a paragraph.

```
flow = new FlowPane();
flow.setHgap(8);   // horizontal gap between items
flow.setVgap(8);   // vertical gap between wrapped rows
flow.add([
```

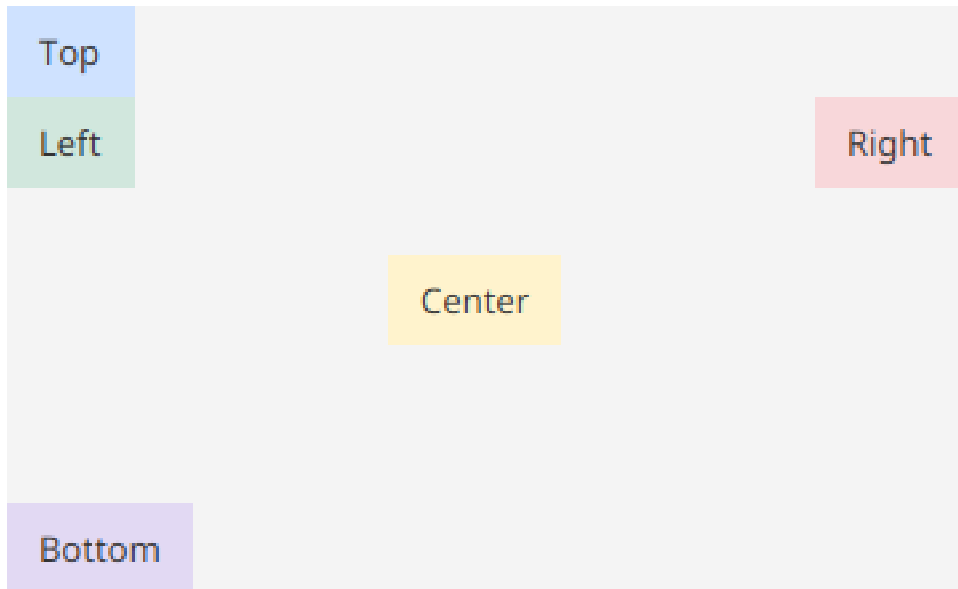


Figure 11: The BorderPane window showing labels in the top, left, center, right, and bottom regions

```
    this.cell("Apple", "#cfe2ff"), this.cell("Banana", "#d1e7dd"),
    this.cell("Cherry", "#fff3cd")   // ...and so on
]);
```

Figure 12: The FlowPane window showing seven fruit labels that wrap onto a second line

A **TilePane** does the same wrapping, but it gives **every item the same size cell** (a "tile"), so the result lines up into a neat grid even when the items themselves are different sizes.

```
tiles = new TilePane();
tiles.setHgap(8);             // pixels between tiles across
tiles.setVgap(8);             // pixels between tile rows
tiles.setPrefColumns(3);      // a count: aim for three tiles per row
tiles.setPrefTileWidth(80);   // pixels: make every tile this wide
tiles.add([
```

```
    this.cell("One", "#cfe2ff"), this.cell("Two", "#d1e7dd"),
    this.cell("Three", "#fff3cd")   // ...and so on
]);
```

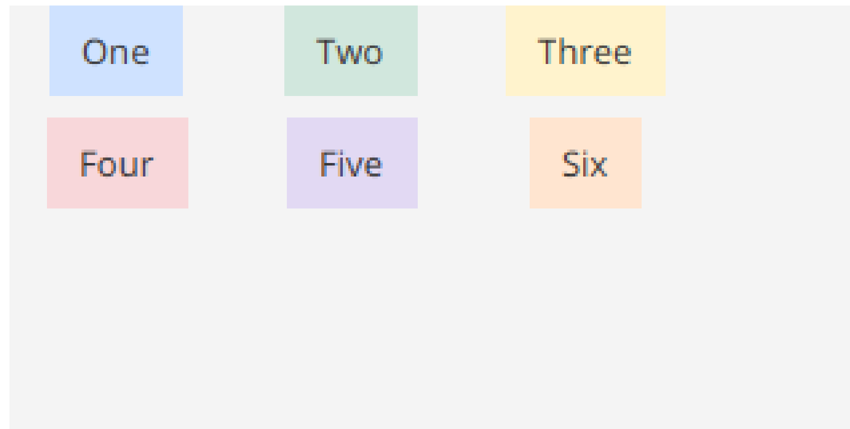


Figure 13: The TilePane window showing six labels in a tidy three-column, two-row grid

Notice in the picture how the items line up into clean columns: that even alignment is the difference between a `TilePane` (uniform tiles) and a `FlowPane` (tightly packed). Use a FlowPane for free-flowing content like tags, and a TilePane when you want an even, grid-like look.

Both panes can do more than wrap left to right:

- A **FlowPane** can flow **vertically** instead with **`setOrientation`** (an `fxorient` value, `fxorient.HORIZONTAL` or `fxorient.VERTICAL`), and **`setAlignment`** (an `fxpos`) chooses how its items line up.
- A **TilePane** can pin the number of columns or rows (**`setPrefColumns`** / **`setPrefRows`**, each a count), set the exact tile size (`setPrefTileWidth` / `setPrefTileHeight`, in pixels), and place each item within its tile with **`setTileAlignment`** (an `fxpos`).

As always, the API reference lists the full set of options.

AnchorPane - Pinning to Edges

An **AnchorPane** lets you **pin** a child a fixed distance from one or more of the pane's edges. Pin a child to the top-right and it stays glued to that corner even as the window resizes. Pin a child to *both* the left and right edges and it stretches to keep both distances, growing and shrinking with the window.

Unlike the other panes, you add children with add and then set their anchors separately. Each anchor method takes the child and a distance in pixels: `setTopAnchor`, `setBottomAnchor`, `setLeftAnchor`, and `setRightAnchor`.

```
include fx;
include fx.AnchorPane;
include fx.Label;
```

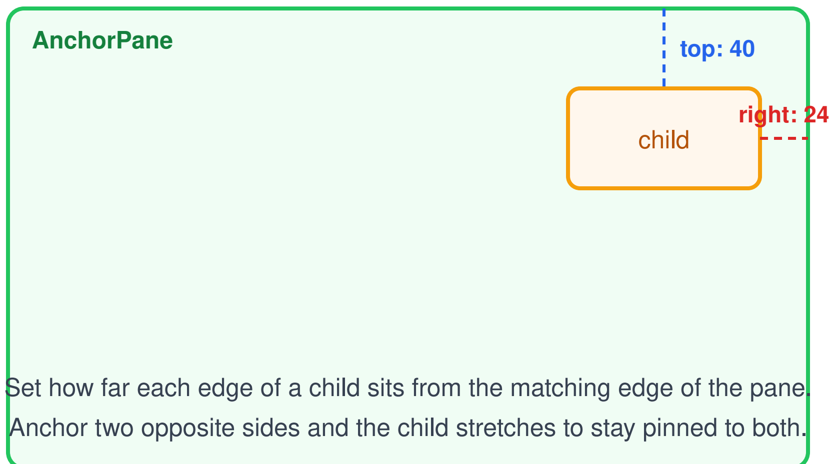


Figure 14: A diagram of an AnchorPane with a child pinned 40 pixels from the top and 24 pixels from the right edge

```
class AnchorPaneDemo {
    public main(args) {
        app = fx.fxApp("AnchorPane", 380, 240);

        ap = new AnchorPane();
        topLeft = this.cell("Top-Left", "#cfe2ff");
        bottomRight = this.cell("Bottom-Right", "#f8d7da");
        banner = this.cell("Stretched (left + right)", "#e2d9f3");
        ap.add([topLeft, bottomRight, banner]);

        // pin a label 10 pixels from the top and left edges
        ap.setTopAnchor(topLeft, 10.0);
        ap.setLeftAnchor(topLeft, 10.0);

        // pin another to the bottom-right corner
        ap.setBottomAnchor(bottomRight, 10.0);
        ap.setRightAnchor(bottomRight, 10.0);

        // anchor BOTH left and right, and this one stretches across
        ap.setTopAnchor(banner, 100.0);
        ap.setLeftAnchor(banner, 10.0);
        ap.setRightAnchor(banner, 10.0);

        app.setLayout(ap);
        app.show(true);
```

```
        fx.shutdown();
    }
    // cell(text, color) is the colored-label helper from the start of
    // the chapter
}
```

The full example pins a label in each corner as well; here is the result:

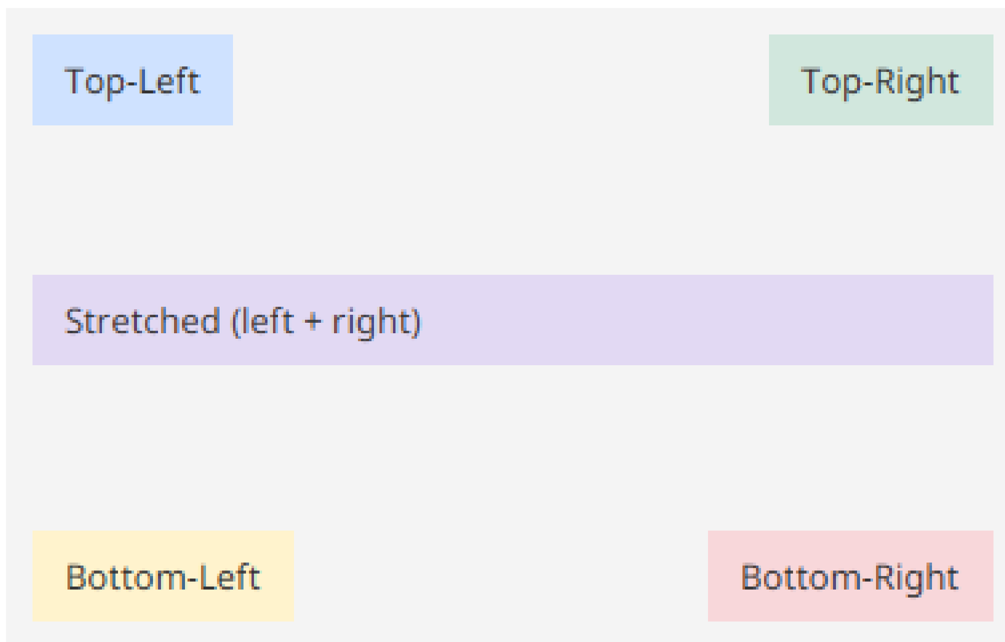


Figure 15: The AnchorPane window with labels pinned to all four corners and a fifth label stretched across the middle from the left edge to the right edge

Look at the middle label: because it is anchored to *both* the left and right edges, it stretches to span the whole width. The corner labels, anchored to only two edges each, simply stay pinned where they are. AnchorPane is the tool you reach for when you need a node stuck to a particular spot, or stretched between edges, no matter how the window is resized. (To remove a child's anchors, call `clearConstraints(child)`.)

GridPane - Rows and Columns of Cells

A **GridPane** arranges nodes into a grid of **rows** and **columns**. You add each node by saying which **column** and **row** it goes in. Both counts start at `0`, columns increase to the right, and rows increase downward.

```
include fx;
include fx.GridPane;
include fx.Label;

class GridPaneDemo {
    public main(args) {
```

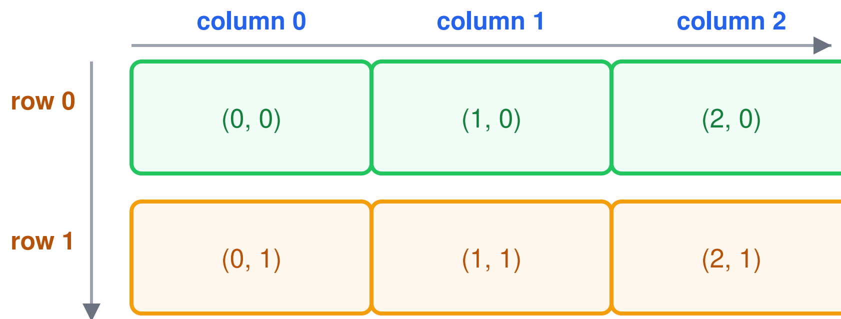


Figure 16: A diagram showing a grid with column 0, 1, 2 across the top and row 0, 1 down the side, each cell labeled with its (column, row) pair

```
        app = fx.fxApp("GridPane", 320, 160);

        grid = new GridPane();
        grid.setHgap(8);   // gap between columns
        grid.setVgap(8);   // gap between rows

        // add(node, column, row) - both are zero-based.
        grid.add(this.cell("Cell 0,0", "#cfe2ff"), 0, 0);
        grid.add(this.cell("Cell 1,0", "#d1e7dd"), 1, 0);
        grid.add(this.cell("Cell 0,1", "#fff3cd"), 0, 1);
        grid.add(this.cell("Cell 1,1", "#f8d7da"), 1, 1);

        app.setLayout(grid);
        app.show(true);
        fx.shutdown();
    }
}
```

The colored cells make the two-by-two grid easy to see:

A GridPane is the right choice whenever your content is naturally tabular: forms with labels beside fields, calculators, calendars, and the like.

GridPane can do much more than place one node per cell. The most useful extra is letting a node **span** several columns or rows, which is how a form's heading stretches across the whole width:

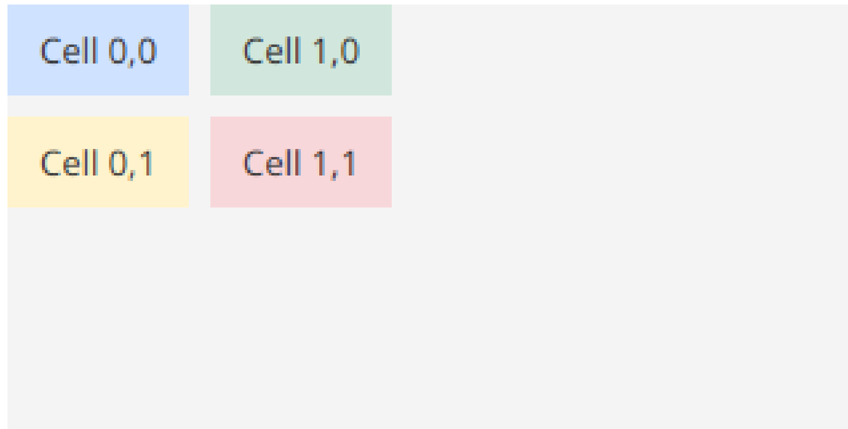


Figure 17: The GridPane window showing a two-by-two grid with a differently colored label in each cell

```
grid.add(header, 0, 0);
grid.setColumnSpan(header, 2);   // stretch it across two columns
                                 // (setRowSpan for rows)
```

Two more worth knowing by name:

- **`setGridLinesVisible(true)`** draws the cell outlines while you design a grid.
- Column and row **constraints** fine-tune how individual columns and rows size and align.

When you need them, the `GridPane` page in the API reference documentation at aussom-lang.com lists every option.

Choosing a Pane

That is a lot of panes. Here is a quick guide to picking one:

If you want to...	Use a...
center one thing, or stack layers	`StackPane`
line items up in a row or a column	`HBox`/`VBox`
build a whole-window frame (top/side/center)	`BorderPane`
let many items wrap onto new lines	`FlowPane`
do the same, but with even, grid-like tiles	`TilePane`
pin a node a fixed distance from the edges	`AnchorPane`
place items in tidy rows and columns (forms)	`GridPane`

You can also **nest** panes inside one another: a `BorderPane` whose top is an `HBox` toolbar and whose center is a `GridPane` form, for instance. Because every pane is itself a node, combining them is how real, complex windows are built.

Finally, keep in mind that each pane has more settings than we have room to show here, such as extra alignment, sizing, and spanning options. When you need finer control, look the pane up in the

API reference documentation at aussom-lang.com; it lists everything that pane can do.

Recap

- **StackPane** stacks and centers; **HBox**/**VBox** make rows and columns.
- **BorderPane** frames a window into top, bottom, left, right, and center, with the center taking the leftover space.
- **FlowPane** wraps items like text; **TilePane** wraps them into uniform tiles.
- **AnchorPane** pins children a fixed distance from the edges.
- **GridPane** places nodes in rows and columns with `add(node, column, row)`.
- Panes **nest**, and that is how you build real layouts.

You now have the full toolbox for arranging content. In the next chapter, we look at panes that add structure in a different way, by splitting, scrolling, and grouping your content.

Chapter 6 - Splitting, Scrolling, and Grouping

The panes in Chapter 5 arrange content. The controls in this chapter help you fit and organize content, especially when there is more of it than the window can comfortably show at once. You will learn how to scroll content, split a window into resizable areas, collapse sections to save room, switch between tabbed views, and break content into numbered pages. We start, though, with the simplest container of all.

Every example in this chapter lives here: `06-splitting-scrolling-grouping`.

Group - A Bare Container

A **Group** is a container that does **no layout of its own**. Unlike the panes in the rest of this book, it never repositions or stretches its children to make them fit. Each child stays exactly where you put it and the size you made it, and the Group simply sizes itself to wrap snugly around whatever it holds. It has no background or border; on its own, it is invisible.

So why would you want a container that does nothing to arrange its contents? Because a Group's real job is different: it lets you treat several separate nodes as a **single unit**. Anything you apply to the Group is applied to **all of its children together**, in one step. Move the Group and every child moves with it. Rotate it, scale it, fade it, or apply a visual effect, and all the children respond as one, keeping their positions relative to each other. The Group's size is just the combined area of its children, so it hugs them exactly.

That makes a Group the right tool for two main situations:

- **Manual, fixed-position layout.** When you want to place nodes at exact coordinates yourself instead of letting a pane arrange them, a Group holds them at those positions and keeps the whole arrangement together.
- **Treating a drawing as one object.** When you build something out of several pieces, most often drawn shapes (covered in Chapter 17), a Group lets you move, rotate, scale, or fade the entire drawing with a single setting, instead of adjusting every piece by hand.

```
g = new Group();
g.add([nodeA, nodeB]);
```

Because it leaves positions alone, a Group is not the tool for laying out buttons and labels, the layout panes are far better at that. Reach for a Group when your nodes already know where they belong

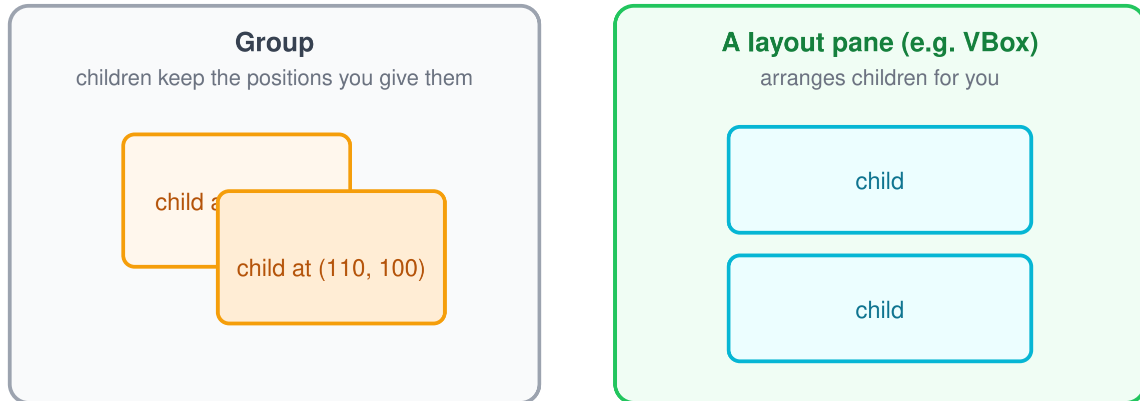


Figure 18: A diagram comparing a Group, whose two children overlap at the positions they were given, with a VBox that arranges its children into a tidy column

and you want to handle them all as one.

ScrollPane - Scrollable Content

When content is bigger than the space you have, a **ScrollPane** adds scroll bars so the user can move around it. You put your content in with `setContent`.

```
scroller = new ScrollPane();
scroller.setContent(column);    // the node to scroll
scroller.setFitToWidth(true);   // stretch content to the pane's width
```

Here a tall column of twelve labels is placed in a short window, so a vertical scroll bar appears automatically:

`setFitToWidth(true)` tells the content to fill the pane's width so it does not also scroll sideways; there is a matching `setFitToHeight(true)` for the other direction. A ScrollPane can do more, too:

- Decide when each scroll bar shows with **`setHBarPolicy`** / **`setVBarPolicy`**, each taking an `fxscrollbar` value: `fxscrollbar.ALWAYS`, `fxscrollbar.NEVER`, or `fxscrollbar.AS_NEEDED` (the default, shown only when the content overflows).
- Let the user drag the content itself to move around with **`setPannable(true)`**, handy for panning a large image or map.

SplitPane - Resizable Dividers

A **SplitPane** shows two or more areas separated by a **divider** that the user can drag to give more room to one side or the other. You add an item for each area.

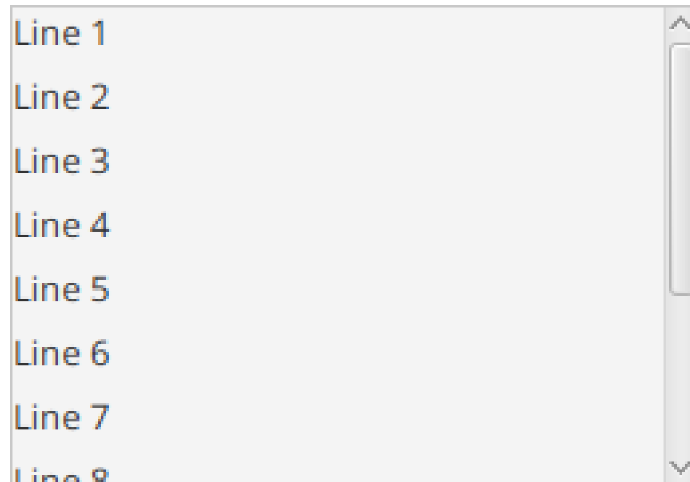


Figure 19: A small ScrollPane window showing the first several of twelve lines, with a vertical scroll bar on the right

```
split = new SplitPane();
split.add([left, right]);            // each item becomes one area
split.setDividerPositions([0.4]);    // divider at 40% across
```

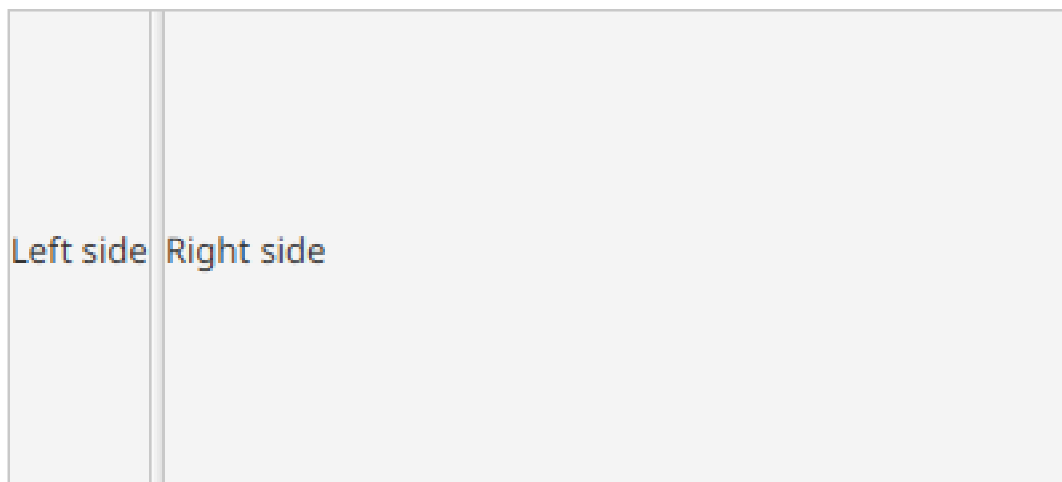


Figure 20: A SplitPane window with a draggable divider separating a Left side area from a Right side area

`setDividerPositions` takes a list of values from `0.0` to `1.0`, one per divider, saying how far across each divider starts. By default a SplitPane lays its areas out left to right; call `setOrientation(fxorient.VERTICAL)` to stack them top to bottom instead. `fxorient` is the orientation counterpart to the `fxpos` enum from Chapter 4: it has just two values, `fxorient.HORIZONTAL` and `fxorient.VERTICAL`, and using it is clearer and safer than passing the raw string `"VERTICAL"`. Like `fxpos`, it comes with the `fx` module.

A SplitPane is not limited to two areas. Add three or more items and you get several draggable dividers between them, each with its own starting position.

TitledPane and Accordion - Collapsible Sections

A **TitledPane** is a single section with a title bar. Clicking the title bar collapses or expands the content below it, which is handy for hiding details until the user wants them.

```
panel = new TitledPane("Details", content);   // title, content node
panel.setExpanded(true);                       // start open
```

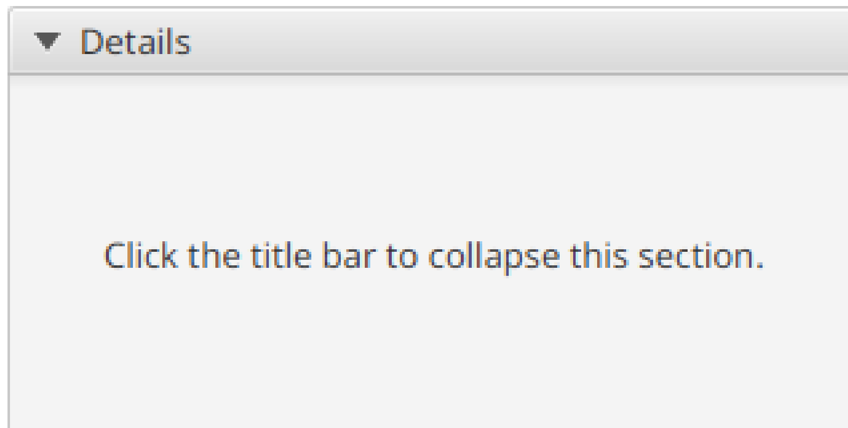


Figure 21: A TitledPane titled Details, expanded to show a line of content beneath its title bar

You can also:

- Stop a TitledPane from being collapsed at all (`setCollapsible(false)`).
- Turn off the small slide animation that plays when it opens and closes (`setAnimated(false)`).

An **Accordion** groups several TitledPanes together and keeps only **one open at a time**: expanding one section automatically collapses the others. It is a tidy way to offer several sections in a small space.

```
one = new TitledPane("Section One", new Label("Content of section one."));
two = new TitledPane("Section Two", new Label("Content of section two."));
three = new TitledPane("Section Three", new Label("Content of section
↪   three."));

acc = new Accordion();
acc.add([one, two, three]);
acc.setExpandedPane(one);   // start with the first one open
```

Calling `setExpandedPane` is optional; leave it out and the accordion starts with every section collapsed.

TabPane and Tab - Tabbed Views

A **TabPane** shows several **tabs**, each holding its own content. Only the selected tab's content is visible, so tabs are a great way to pack several screens' worth of content into one window. Each tab is a Tab object with a title and a content node.

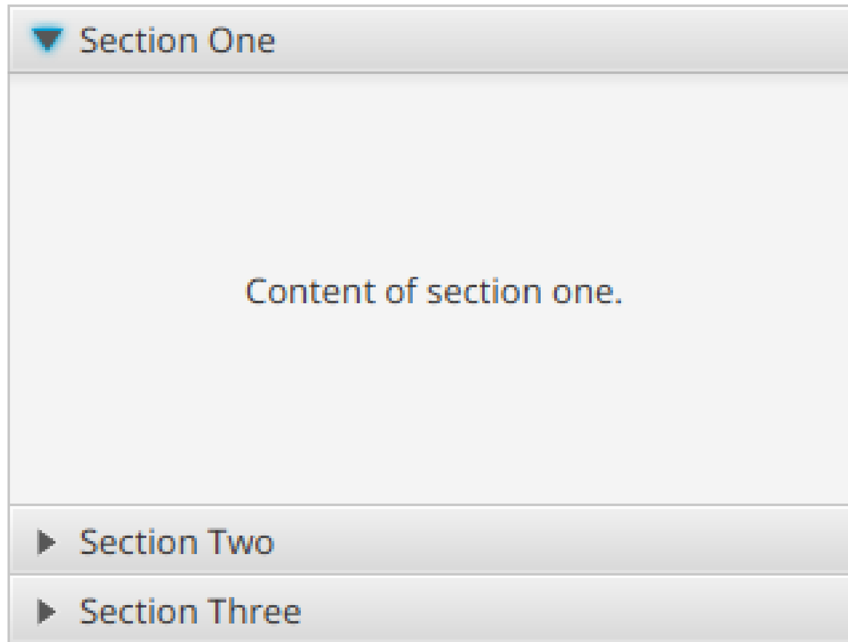


Figure 22: An Accordion with Section One expanded to show its content, and Sections Two and Three collapsed below it

```
home = new Tab("Home");
home.setContent(new Label("Welcome to the Home tab."));

profile = new Tab("Profile");
profile.setContent(new Label("This is the Profile tab."));

tabs = new TabPane();
tabs.add([home, profile]);
tabs.setTabClosingPolicy(fxtabclosingpolicy.UNAVAILABLE);   // no close
                                                            // buttons
```

By default each tab has a small close button. Setting the closing policy to `fxtabclosingpolicy.UNAVAILABLE` removes those buttons so the user cannot close the tabs, which is usually what you want for an app's main sections (the other choices are `fxtabclosingpolicy.SELECTED_TAB` and `fxtabclosingpolicy.ALL_TABS`).

There is a lot more you can do here:

- The whole tab strip can sit on any side of the window with **`setSide`** (an `fxside` value: `fxside.TOP`, `fxside.BOTTOM`, `fxside.LEFT`, or `fxside.RIGHT`).
- Run a method whenever the user switches tabs with **`onChange`**.
- Let the user drag tabs to reorder them with **`setTabDragPolicy`** (an `fxtabdragpolicy` value: `fxtabdragpolicy.REORDER` to allow it, or `fxtabdragpolicy.FIXED` to lock the order).
- Each Tab can carry an icon (**`setGraphic`**), show a tooltip (**`setTooltip`**), or be greyed out (**`setDisable(true)`**).

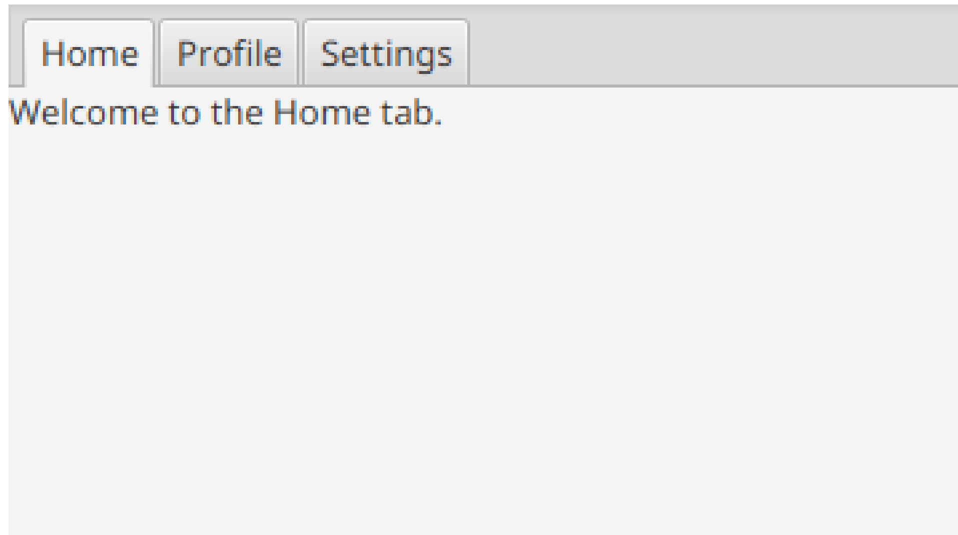


Figure 23: A TabPane with Home, Profile, and Settings tabs; the Home tab is selected and shows its content

Pagination - Paged Content

A **Pagination** control breaks content into numbered pages with a row of page buttons, like the results of a search. You tell it how many pages there are and give it a **page factory**: a method that builds the content for whichever page the user clicks.

```
pages = new Pagination(5);        // five pages
pages.setPageFactory(::makePage); // how to build each page
```

The page factory is one of your own methods. JavaFX calls it with the page's index (counting from `0`) whenever it needs to show a page, and your method returns the node to display:

```
public makePage(index) {
    label = new Label("This is page " + (index + 1));
    return label.obj;   // the factory hands the node straight to JavaFX
}
```

Two small things to notice. The index counts from `0`, so we add `1` to show a friendly page number that matches the buttons. And we return `label.obj` rather than the wrapper: this is one of those rare moments (mentioned back in Chapter 3) where the underlying JavaFX node is needed directly, because JavaFX calls your factory and expects a real node back.

If a page count gets large:

- Limit how many page buttons show at once with **`setMaxPageIndicatorCount`** (a count).
- Jump straight to a page from your own code with **`setCurrentPageIndex`** (a zero-based index).

Recap

Figure 24: A Pagination control showing the content of page 1 with page buttons 1 through 5 beneath it

If you want to…	Use a…
place nodes by hand and treat them as one unit	`Group`
scroll content that is too big for the space	`ScrollPane`
split an area with a draggable divider	`SplitPane`
collapse one section to save room	`TitledPane`
offer several sections, one open at a time	`Accordion`
switch between several views with tabs	`TabPane/Tab`
break content into numbered pages	`Pagination`

- A **Group** does no layout; it lets you place nodes by hand and treat them as a single unit, moving, rotating, or fading them all together.
- **ScrollPane**, **SplitPane**, **TitledPane**, **Accordion**, **TabPane**, and **Pagination** are all ways to organize more content than the window can show at once.
- Like all panes, these can be **nested** and combined with the layouts from Chapter 5 to build a real application window.
- Each control here has more options than we have shown. When you need finer control, its page in the API reference documentation at aussom-lang.com lists everything it can do.

That completes our tour of layout. From here on, we turn from *arranging* nodes to the nodes themselves, starting in the next chapter with the controls users actually click and read: labels and buttons.

Part III

Controls and User Input

Chapter 7 - Labels and Buttons

Parts I and II were about *arranging* nodes. Now we turn to the nodes users actually read and operate: **controls**. A control is a ready-made, interactive piece of a user interface, a button to click, a box to tick, a link to follow. JavaFX gives you a rich set of them, and this chapter covers the most common: the humble label and the whole family of buttons.

Every example in this chapter lives here: `07-labels-and-buttons`.

Label and the Labeled Base

You have used a `Label` since Chapter 1: it simply shows a piece of text. What is new here is *why* so many controls feel alike. Behind the scenes, labels, buttons, check boxes, and more all share a common base class called **`Labeled`**, which gives each of them a text label. Here is the family we will meet in this chapter:

Because they all descend from `Labeled`, they all set their text the same way, in the constructor or with `setText`, whether it is a label, a button, or a check box. Learn it once and it works everywhere.

A `Label` has a couple of tricks of its own. The handiest is **wrapping**: by default a long line of text is cut off at the edge, but `setWrapText(true)` lets it flow onto several lines instead.

```
title = new Label("A Label shows a piece of text.");

long = new Label("This is a longer label. With wrap turned on, its text " +
    "flows onto several lines instead of running off the edge.");
long.setWrapText(true);   // let the text wrap
```

Beyond wrapping, the `Labeled` base gives all of these controls a few shared text settings:

- **`setTextAlignment`** lines up several lines of text (an `fxtextalign` value: `fxtextalign.LEFT`, `CENTER`, `RIGHT`, or `JUSTIFY`).
- **`setTextOverrun`** trims text that does not fit with an ellipsis (an `fxoverrun` value like `fxoverrun.ELLIPSIS`), and **`setEllipsisString`** changes the "…" it uses.
- **`setGraphic`** shows a small picture beside the words, and **`setContentDisplay`** (an `fxcontentdisplay` value like `fxcontentdisplay.LEFT`) picks which side it sits on. We use graphics once we reach images in Chapter 19.

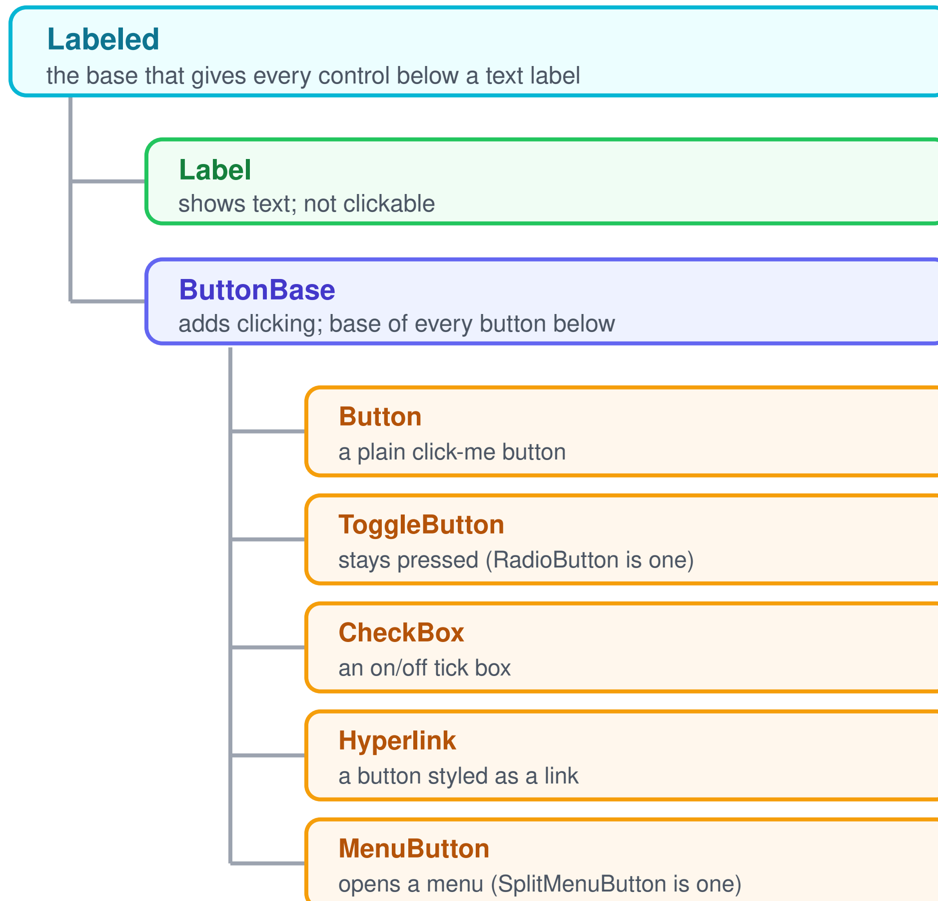


Figure 25: A tree diagram: Labeled is the base, with Label under it (shows text, not clickable) and ButtonBase (adds clicking), whose children are Button, ToggleButton, CheckBox, Hyperlink, and MenuButton

}

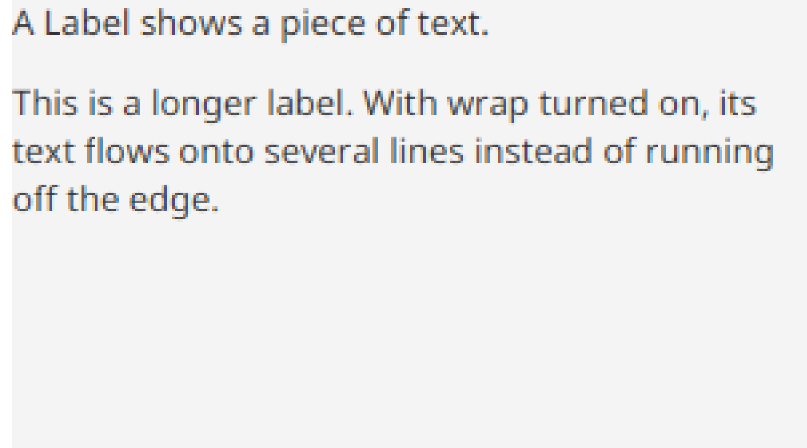


Figure 26: A window with a short title label and a longer label whose text has wrapped onto three lines

Button and ButtonBase

A **`Button`** is the classic control you click to make something happen. Every button in the family shares a base called **`ButtonBase`**, which provides the actual clicking behavior. The most important thing that base gives you is **`onClick`**: you hand it one of your own methods, and JavaFX calls that method every time the button is pressed.

```
class ButtonsDemo {
    private message = null;

    public main(args) {
        app = fx.fxApp("Button", 300, 140);

        this.message = new Label("You have not clicked yet.");

        btn = new Button("Click Me");
        btn.onClick(::onClick);   // run onClick() when pressed

        col = new VBox();
        col.setSpacing(12);
        col.setAlignment(fxpos.CENTER);
        col.add([btn, this.message]);

        app.setLayout(col);
        app.show(true);
        fx.shutdown();
    }

    public onClick(event) {
        this.message.setText("You clicked the button!");
    }
```

```
}
```

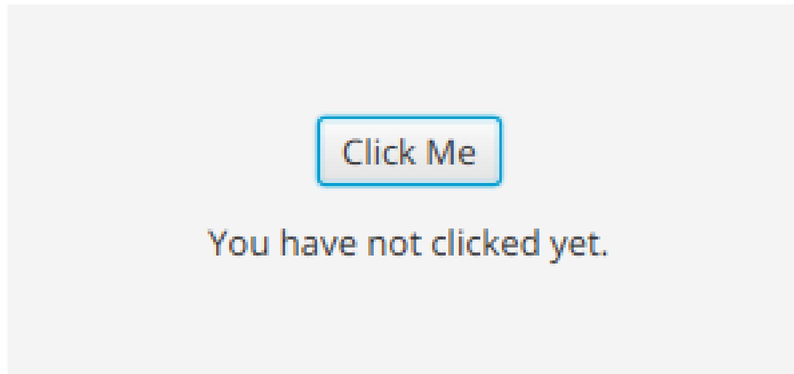


Figure 27: A window with a Click Me button above a label reading "You have not clicked yet."

Run it, click the button, and the label changes to "You clicked the button!".

Two things about the handler. First, `::onClick` means "the method named `onClick`", the same way you passed methods to `runLater` back in Chapter 2. You are handing the method to the button, not calling it yourself. Second, notice that `onClick(event)` takes one parameter. JavaFX passes your method an **event** that describes what happened. We do not need it here, so we simply ignore it, but it must be there. Events get their own full chapter next (Chapter 8).

A button can do a little more, too. You can mark one button as the window's **default** (it fires when the user presses Enter) or its **cancel** button (it fires on Escape), which is exactly how "OK" and "Cancel" buttons behave in a dialog:

```
ok = new Button("OK");
ok.setDefaultButton(true);      // also fires when the user presses
                                // Enter

cancel = new Button("Cancel");
cancel.setCancelButton(true);   // also fires when the user presses
                                // Escape
```

ToggleButton, RadioButton, and ToggleGroup

Some buttons **remember a state**. A **`ToggleButton`** stays pressed after you click it, and pops back up when you click again, like a light switch. You can read or set whether it is pressed with `isSelected` and `setSelected`.

A **`RadioButton`** is a toggle drawn as a small round option. On its own it is not very useful; radio buttons shine when you gather several into a **`ToggleGroup`**. A ToggleGroup enforces a simple rule: **only one option in the group can be selected at a time.** Selecting one automatically clears the others, which is exactly what you want when the user must pick a single choice.

```
group = new ToggleGroup();
```

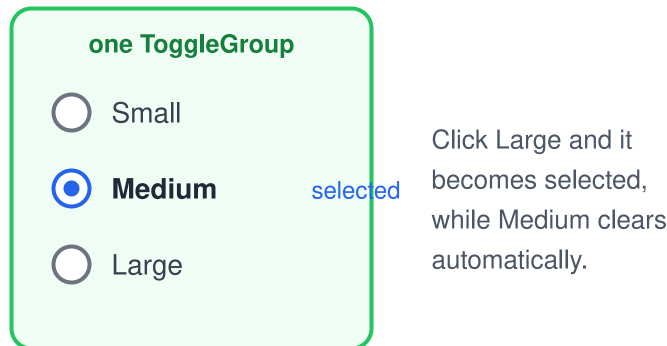


Figure 28: A diagram of three radio buttons, Small, Medium, and Large, in one ToggleGroup, with Medium selected; selecting another would clear Medium

```
small = new RadioButton("Small");
medium = new RadioButton("Medium");
large = new RadioButton("Large");
small.setToggleGroup(group);
medium.setToggleGroup(group);
large.setToggleGroup(group);
medium.setSelected(true);   // start with one chosen
```

Once the buttons are grouped, you can ask the group which toggle is currently selected, or run a method whenever the choice changes.

CheckBox and Hyperlink

A **`CheckBox`** is the familiar tick box for turning an option on or off. Read its state with `isSelected`, or set it with `setSelected`. A check box can also show a third, “in between” state called **indeterminate**, drawn as a dash. It is handy for a “select all” box when only some items are chosen. You turn that on with `setAllowIndeterminate(true)`.

A **`Hyperlink`** looks like a web link, underlined and colored, but under the hood it is just another button. You give it an action with `onClick`, exactly like a `Button`, and it quietly remembers whether it has been “visited.”

```
accept = new CheckBox("Accept the terms");
accept.setSelected(true);            // checked

subscribe = new CheckBox("Subscribe to the newsletter");   // unchecked

partial = new CheckBox("Select all");
```

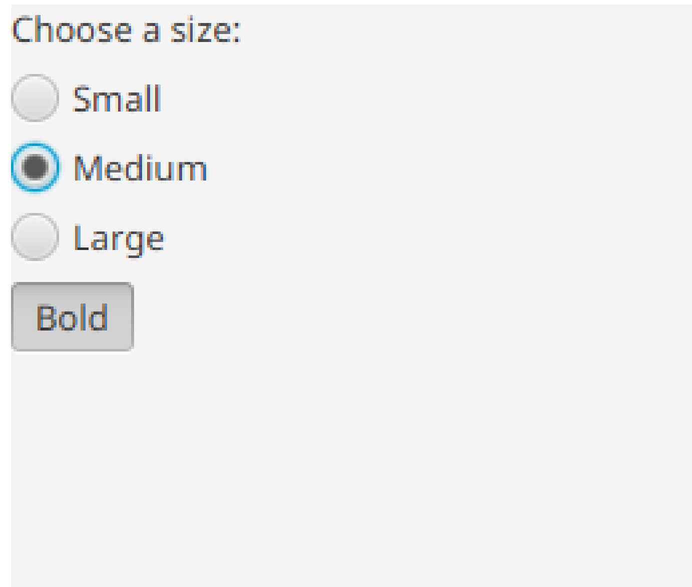


Figure 29: A window showing Small, Medium, and Large radio buttons with Medium selected, plus a pressed Bold toggle button

```
partial.setAllowIndeterminate(true);
partial.setIndeterminate(true);      // the in-between state

link = new Hyperlink("Visit aussom-lang.com");
```

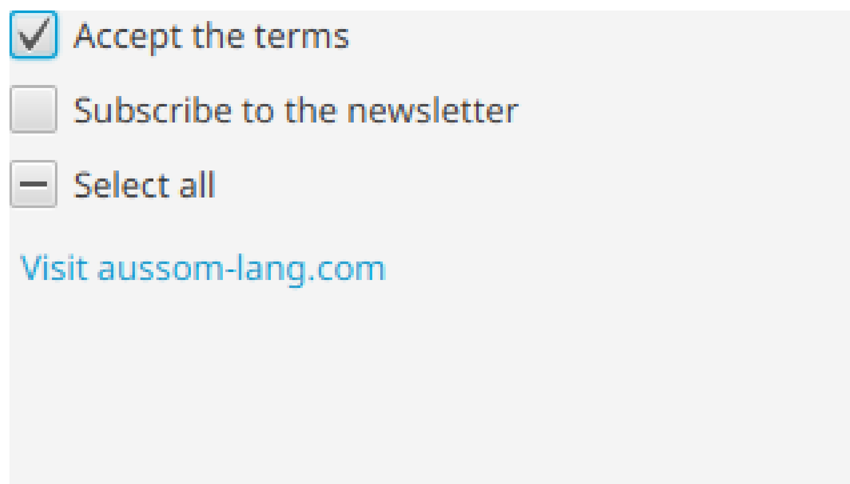


Figure 30: A window with three check boxes, one checked, one empty, one showing the in-between dash, and a blue hyperlink below them

MenuButton and SplitMenuButton

Sometimes one button should offer several related choices. A **MenuButton** does that: clicking it drops down a small menu. You fill it with **MenuItem** objects, and each item gets its own `onClick`, just like a button.

A **SplitMenuButton** is a close cousin. It is split into two parts: the main part acts like a normal button (it runs an action when clicked), while a small arrow beside it opens the menu. Think of a "Save" button whose arrow also offers "Save As...".

```
cut = new MenuItem("Cut");
copy = new MenuItem("Copy");
paste = new MenuItem("Paste");
edit = new MenuButton("Edit");
edit.add([cut, copy, paste]);

save = new SplitMenuButton("Save");
save.add([new MenuItem("Save"), new MenuItem("Save As...")]);
```

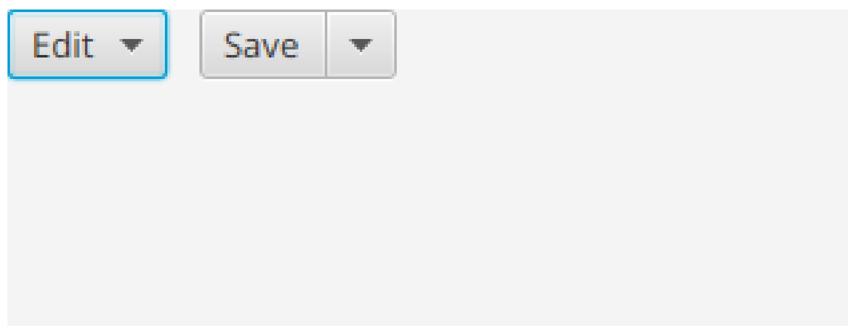


Figure 31: A window with an Edit menu button and a Save split menu button; the split button shows a separate dropdown arrow

Look closely at the picture: the `Edit` button is one solid piece (the whole thing opens the menu), while `Save` is visibly split, with its own arrow on the right. You can also choose which side the menu pops out on with **`setPopupSide`** (an `fxside` value), and run code when it opens or closes.

Recap

Control	What it is for
`Label`	show a piece of (optionally wrapping) text
`Button`	run an action when clicked
`ToggleButton`	a button that stays pressed or unpressed
`RadioButton` + `ToggleGroup`	pick exactly one option from a set
`CheckBox`	turn an option on, off, or "in between"
`Hyperlink`	a link-styled button
`MenuButton` / `SplitMenuButton`	offer a drop-down menu of choices

- Labels and buttons all share the **`Labeled`** base, so they set their text the same way; the clickable ones also share **`ButtonBase`** and its **`onClick`**.
- Wire a click by handing a method to `onClick`; the method takes an **event** parameter it can ignore for now.
- A **`ToggleGroup`** keeps a set of options to a single selection.
- Each control has more options than we have shown; the API reference documentation at aussom-lang.com lists them all.

You can now put clickable controls on screen, but so far our buttons only do one simple thing. In the next chapter we look properly at **events**: how to respond to clicks and other actions, read the event details, and connect your controls to real behavior.

Chapter 8 - Handling Events

The controls from Chapter 7 are only useful if they *do* something. Making them do something is what this chapter is about. When the user clicks a button, moves the mouse, or presses a key, JavaFX sends your program an **event**, and you write code to respond. You already met the simplest case, `onClick`, in Chapter 7. Now we look at events properly.

Every example in this chapter lives here: `08-handling-events`.

What Is an Event?

An **event** is a message JavaFX creates the moment something happens: a click, a key press, the mouse moving. You do not check for these things yourself in a loop. Instead, you register a **handler**, one of your own methods, and JavaFX calls it for you when the event occurs, handing it an **event object** that describes what happened.

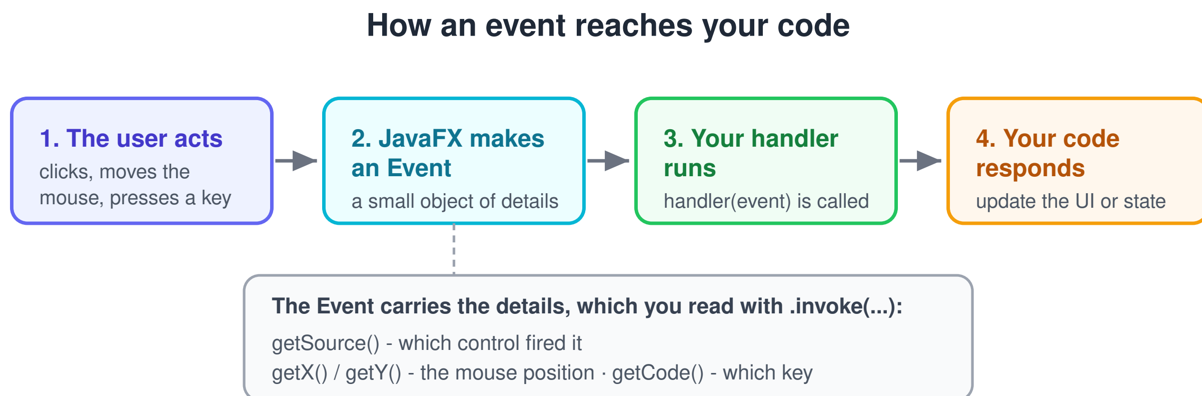


Figure 32: A flow diagram: the user acts, JavaFX makes an Event carrying details, JavaFX calls your handler(event), and your code responds by updating the UI

That is the whole cycle: **you say what to run, JavaFX decides when to run it.**

Handlers already run on the UI thread

Back in Chapter 2 you learned that JavaFX draws on a single **UI thread**, and that code which is *not* on that thread must use `fx.runLater` before it touches the interface. It is worth being clear about

how that applies here, because it is a common point of confusion in JavaFX:

> **Every event handler already runs on the UI thread.** JavaFX calls your `onClick`, mouse, and key handlers there for you.

So inside a handler you can change labels, read fields, and update the interface **directly, with no `fx.runLater` needed** (every example in this chapter does exactly that). You only reach for `fx.runLater` when the code doing the UI work is somewhere else, such as a background task, or `main` itself as you saw in Chapter 2.

Action Events: Wiring a Button to Code

The most common event is a control's **action**, the main thing it does: a button being clicked, a menu item being chosen. You handle it with `onClick`, giving it one of your methods. Here is a small counter whose two buttons change a number:

```
class CounterDemo {
    private count = 0;
    private display = null;

    public main(args) {
        app = fx.fxApp("Counter", 260, 150);

        this.display = new Label("Count: 0");

        minus = new Button("-");
        minus.onClick(::onMinus);   // run onMinus() when clicked
        plus = new Button("+");
        plus.onClick(::onPlus);     // run onPlus() when clicked

        buttons = new HBox();
        buttons.setSpacing(10);
        buttons.setAlignment(fxpos.CENTER);
        buttons.add([minus, plus]);

        col = new VBox();
        col.setSpacing(14);
        col.setAlignment(fxpos.CENTER);
        col.add([this.display, buttons]);

        app.setLayout(col);
        app.show(true);
        fx.shutdown();
    }

    public onPlus(event) {
        this.count = this.count + 1;
        this.showCount();
```

```
    }

    public onMinus(event) {
        this.count = this.count - 1;
        this.showCount();
    }

    public showCount() {
        this.display.setText("Count: " + this.count);
    }
}
```

Figure 33: The Counter window showing "Count: 3" above minus and plus buttons

Run it and click + a few times; the count climbs. Two ideas make this work. First, the running total lives in `this.count`, a variable that belongs to the whole object, so it survives between clicks. Second, each handler takes an `event` parameter (the action event), which we simply do not use here. It has to be there, but a lot of the time you can ignore it, exactly as we do in this counter.

The **action** is a control's headline event, but it is not the only one. The same `onClick` style works for menu items, and a text field fires its action when the user presses Enter (Chapter 9). And because every control is also a node, it can respond to all the mouse and keyboard events we are about to see, plus a **context-menu request** (a right-click on the control). So a button can react not only to being clicked, but to being hovered, right-clicked, and more.

Reading the Event

Sometimes you *do* want the details, and that is what the event object is for. The event JavaFX hands you is a Java object, and you read information from it by calling its methods with `.invoke("methodName")`.

The handiest detail for an action is **which control fired it**, available from `getSource`. That lets one handler serve several buttons and still tell them apart:

```
public onColor(event) {
    button = event.invoke("getSource");           // the button that was
```

```
                                              // clicked
    name = button.invoke("getText");          // its label text
    this.status.setText("You chose " + name);
}
```

Reading details this way, with `.invoke(...)`, is how you get at anything the event carries.

One Handler for Many Controls

Reading `getSource` really pays off when several controls should behave alike. You could give each button its own method, but it is often cleaner to write the handler *once* and attach the same one to all of them. `fx.actionEventHandler` turns one of your methods into a single reusable handler, and `setOnAction` puts it on a control:

```
this.handler = fx.actionEventHandler(::onChoose);

red = new Button("Red");
red.setOnAction(this.handler);     // the same handler ...
green = new Button("Green");
green.setOnAction(this.handler);   // ... on every button
```

Inside the handler, `getSource` tells you which button was clicked, so one method serves them all:

```
public onChoose(event) {
    name = event.invoke("getSource").invoke("getText");
    this.status.setText("You chose " + name + ".");
}
```

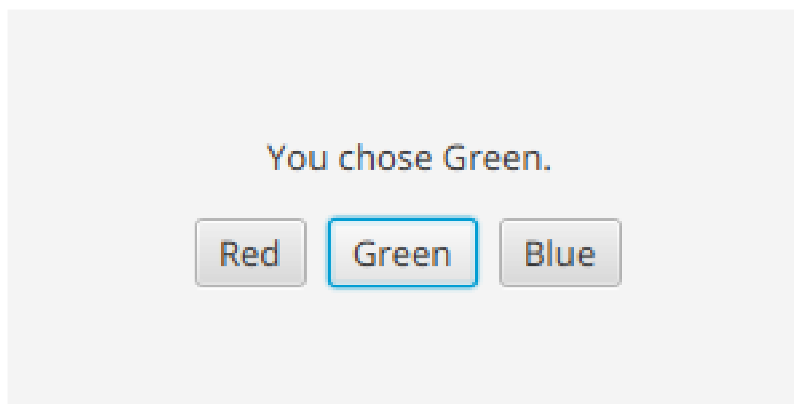


Figure 34: The Chooser window showing “You chose Green.” above Red, Green, and Blue buttons

As it happens, `onClick`, which you have used all along, is really a shortcut for this: it wraps your method in a handler and attaches it to one button. `actionEventHandler` just lets you build the handler yourself and reuse it.

Reading the event comes into its own with the mouse and the keyboard, which we turn to next.

Mouse Events

Clicking is not only for buttons. **Any node** can react to the mouse. You register a handler with `setOnMouseClicked`, and the event this time is a mouse event that knows *where* the click happened, through `getX` and `getY`.

```
pad = new StackPane();
pad.add(this.status);
// A background color makes the whole pad receive clicks. (Styling: Chapter
↪   16.)
pad.setStyle("-fx-background-color: #eef2ff;");
pad.setOnMouseClicked(::onClick);
```

```
public onClick(event) {
    x = event.invoke("getX");
    y = event.invoke("getY");
    this.status.setText("You clicked at (" + x + ", " + y + ").");
}
```

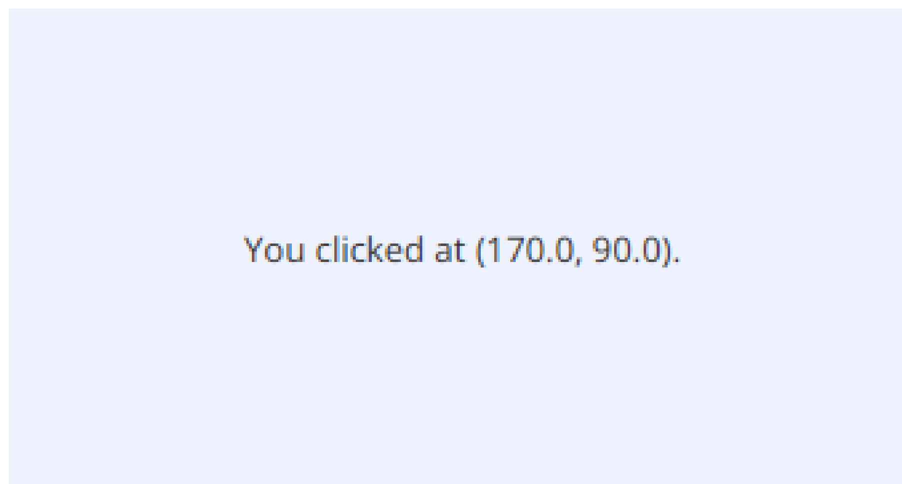


Figure 35: A window filled with a light blue pad showing "You clicked at (170.0, 90.0)."

The mouse event carries more than the position, and you read each part the same way, with `.invoke(...)`:

- **`getX`** / **`getY`** - the click position within the node (there are also `getSceneX` / `getScreenX` for window- and screen-relative coordinates).
- **`getButton`** - which button was used (primary, secondary, or middle).
- **`getClickCount`** - 1 for a single click, 2 for a double-click.
- **`isShiftDown`**, **`isControlDown`**, **`isAltDown`** - whether a modifier key was held at the time.

`setOnMouseClicked` is just one of a whole family of mouse handlers. You register them the same way, and each hands your method a mouse event. Without going into the details of each, here are the common ones, so you know they are available:

- **`setOnMousePressed`** and **`setOnMouseReleased`** - the separate "down" and "up" halves of a

click.
- **setOnMouseEntered** and **setOnMouseExited** - the pointer moving onto or off a node (perfect for hover effects).
- **setOnMouseMoved** - the pointer moving across a node.
- **setOnMouseDragged** - the pointer moving while a button is held down.
- **setOnScroll** - the scroll wheel turning.
- **setOnDragDetected** - the start of a drag gesture.

Keyboard Events

Reacting to the keyboard works the same way, with `setOnKeyPressed`. The event is a key event, and you can find out which key it was from its **key code**:

```
public onKey(event) {
    name = event.invoke("getCode").invoke("getName");
    this.status.setText("You pressed: " + name);
}
```

Like the mouse event, a key event also reports the modifier keys through **isShiftDown**, **isControlDown**, and **isAltDown**, so you can detect combinations like Ctrl+S.

There is one extra wrinkle with the keyboard. Unlike the mouse, which reports clicks wherever the pointer happens to be, key presses have to go *somewhere* specific. JavaFX sends every key event to a single node called the one with **keyboard focus**, the node that is currently "listening" to the keyboard (you have seen focus as the faint outline around a button you have tabbed to).

The catch is that **plain layout containers, like our StackPane, cannot hold focus by default.** They are meant to arrange other nodes, not to be typed into, so JavaFX skips right over them. If we do nothing, our pad never becomes the focused node, and its key handler never runs. Two calls fix that:

- `setFocusTraversable(true)` makes the pad **eligible** to hold focus. Without it, the pad is simply not a candidate, and the next step would do nothing.
- `requestFocus()` then **actually gives** the pad the focus. We do this in an onShow handler because a node can only take focus once its window is on screen; asking earlier has no effect.

```
this.pad.setFocusTraversable(true);   // allow the pad to hold focus
this.pad.setOnKeyPressed(::onKey);
app.setLayout(this.pad);
app.setOnShow(::focusPad);            // grab focus once shown
```

```
public focusPad() {
    this.pad.requestFocus();
}
```

You only ever need this two-step dance for **layout containers** (panes and other grouping nodes). The interactive **controls**, buttons, text fields, check boxes, and the rest, are focusable out of the box: they take focus automatically the moment the user clicks them or tabs to them, so their key handlers just work with no extra setup. In other words, the setup above is the exception, needed only because we chose to listen for keys on a bare pane rather than on a control. It is worth knowing,

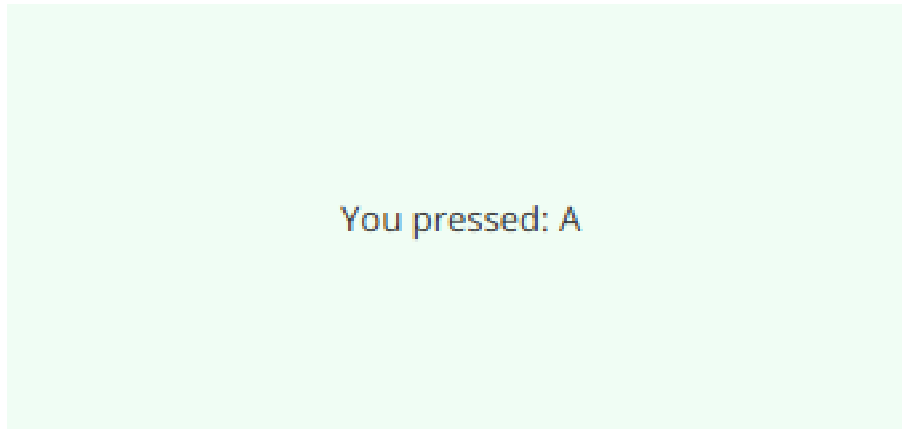


Figure 36: A window filled with a light green pad showing "You pressed: A"

though, because a key handler that "silently does nothing" on a pane is a classic head-scratcher, and now you know the cause.

As with the mouse, `setOnKeyPressed` has a couple of common companions worth knowing about:

- **`setOnKeyReleased`** - the key coming back up.
- **`setOnKeyTyped`** - a *character* being entered. This is the one to use when you care about the letter typed rather than the physical key, since it combines keys for you (Shift plus a arrives as A).

Bridging to Java Interfaces

Every handler you have written, whether through `onClick`, `actionEventHandler`, or `setOnMouseClicked`, ends up doing the same thing behind the scenes: it hands JavaFX a small object that implements a Java "listener" interface and wraps your Aussom method. Aussom builds that object for you, which is why you have not had to think about it.

Once in a long while you will meet a Java library that asks you to implement some interface *it* defines, one Aussom has not wrapped. For those advanced cases, `fx.getClosure` is the general escape hatch. You give it the interface's name and one of your methods, and it hands back an object that satisfies the interface by calling your method:

```
handler = fx.getClosure("javafx.event.EventHandler", ::onAction);
someControl.setOnAction(handler);
```

You will rarely reach for this as a beginner, the ready-made wrappers cover the everyday cases, but it is reassuring to know the door is there when you need to touch a corner of Java that the `fx` module has not already smoothed over.

The Wider World of Events

Between clicks, the mouse, and the keyboard, you can already respond to most of what a user does. But JavaFX has many more events, and they all follow the same pattern, register a handler, receive

an event object. Here are the ones you will reach for most:

To react to...	Register with...	Read from the event...
a button or menu action	`onClick`	`getSource`
a mouse click	`setOnMouseClicked`	`getX`, `getY`
the pointer entering a node	`setOnMouseEntered`	`getX`, `getY`
a key press	`setOnKeyPressed`	`getCode`

Beyond these there are events for dragging, scrolling, touch gestures, and more. Each node lists its full set in the API reference documentation at aussom-lang.com.

Recap

- An **event** is JavaFX's message that something happened; you respond by registering a **handler** method that JavaFX calls with an **event object**.
- Handlers already run on the **UI thread**, so you can update the interface directly inside them, with no `fx.runLater` needed.
- Handle a control's action with **`onClick`**; the handler takes an `event` parameter it can ignore when it does not need the details.
- Keep changing data (like a counter) in object variables (`this.count`) so it survives between events.
- Read details from the event with **`.invoke(...)`**: `getSource` for the control, `getX`/`getY` for the mouse, `getCode` for a key.
- Share one handler across several controls by building it with **`fx.actionEventHandler`** and attaching it with `setOnAction`; for advanced Java integration, **`fx.getClosure`** bridges to any listener interface.
- Handle the mouse with **`setOnMouseClicked`** and the keyboard with **`setOnKeyPressed`**; remember that key events need a **focused** node.

Your controls can now respond to the user. Next we look at the controls built specifically for **text input**, where reading and reacting to what the user types is the whole point.

Chapter 9 - Text Input

The buttons and boxes in Chapter 7 let the user *choose* things. This chapter is about controls the user *types into*: a box for a name, a hidden box for a password, a large area for a paragraph, and a small number box with up and down arrows. Reading what the user typed is the whole point, so the event ideas from Chapter 8 come right back into play.

Every example in this chapter lives here: `09-text-input`.

Most of these controls are cousins. `TextField`, `PasswordField`, and `TextArea` all share a common base, `TextInputControl`, which is where they get their text from, so once you learn to read one, you can read them all:

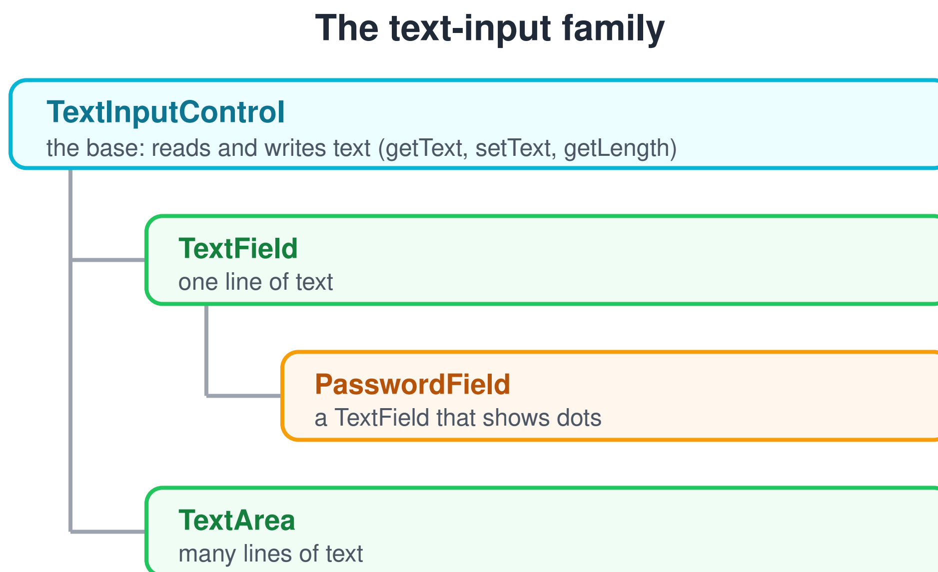


Figure 37: A tree diagram: TextInputControl is the base, with TextField (one line) under it, Password-Field (a TextField that shows dots) under that, and TextArea (many lines) also under the base

TextField and PasswordField

A **`TextField`** is a single-line box for typing text. You usually create an empty one and, if you like, give it **prompt text**, the faint hint that shows while the box is empty:

```
nameField = new TextField();
nameField.setPromptText("Username");   // faint hint text
```

Whenever you want to know what the user typed, call `getText`.

A **`PasswordField`** is a `TextField` that hides what is typed behind dots, for passwords and other secrets. It works exactly like a text field in every other way, and, importantly, **your code can still read the real text with `getText`**; only the *display* is hidden.

Here the two come together in a small sign-in form. Clicking **Sign In** reads both fields:

```
public onSignIn(event) {
    name = this.nameField.getText();
    if (name == "") {
        this.status.setText("Please enter a username.");
    } else {
        // getText reads the password too, even though it is hidden.
        length = this.passwordField.getLength();
        this.status.setText("Welcome, " + name +
            "! Your password has " + length + " characters.");
    }
}
```

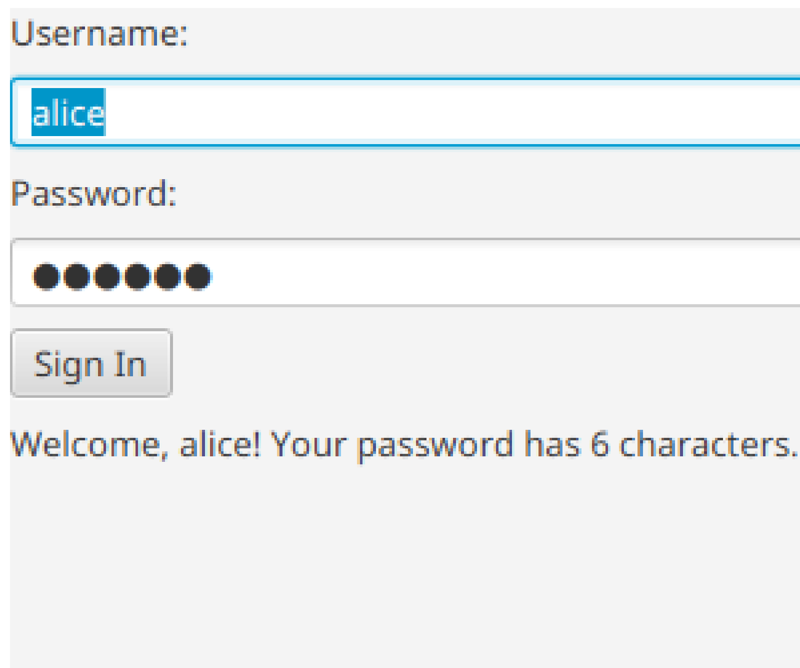


Figure 38: A sign-in form with the username “alice”, a password shown as dots, a Sign In button, and the message “Welcome, alice! Your password has 6 characters.”

Notice the status line proves the point: the password shows as dots on screen, but `getLength` (and `getText`) happily report what is really in there. A text field can also run a method the instant the user presses **Enter**, using `setOnAction`, a nice touch for a search box or a login form:

```
searchField.setOnAction(::onSearch);   // run onSearch() when the user
                                       // presses Enter
```

TextArea and the TextInputControl Base

When one line is not enough, a **`TextArea`** gives you a resizable, multi-line box, the kind of thing you would write a message or a note in. Two settings make it comfortable:

```
editor = new TextArea("Some starting text.");
editor.setWrapText(true);      // wrap long lines instead of scrolling
                               // sideways
editor.setPrefRowCount(5);     // start about five rows tall
```

Because `TextField` and `TextArea` share the `TextInputControl` base, everything you learned about reading a text field works here unchanged. `getText` returns all the text, and `getLength` returns how many characters it holds:

```
public onCount(event) {
    this.status.setText(this.editor.getLength() + " characters.");
}
```

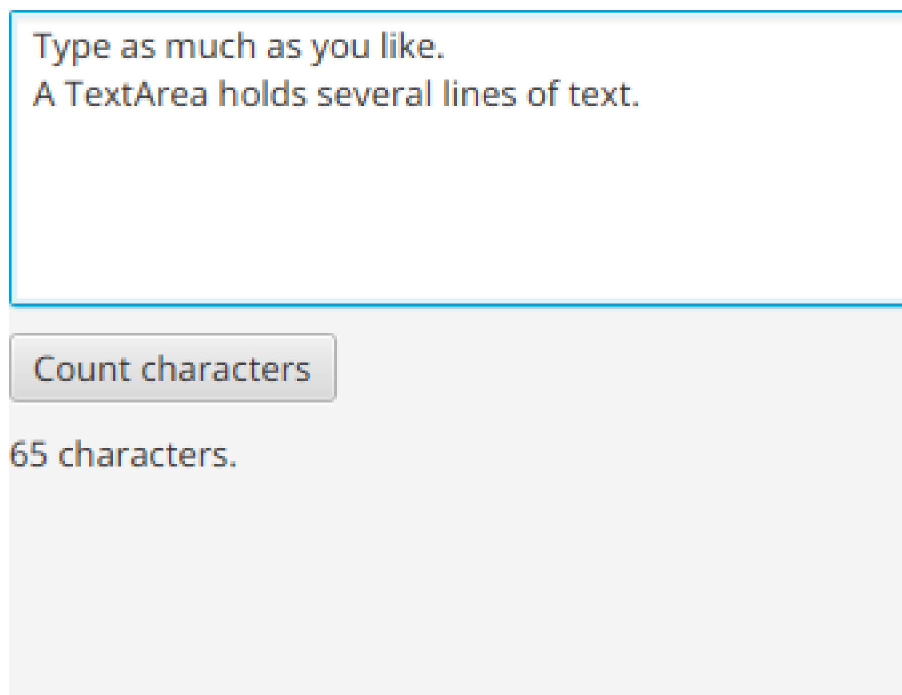


Figure 39: A multi-line text area with two lines of text, a Count characters button, and the message “65 characters.”

That shared base carries more than reading. Here are some of the other common things every text control inherits from it, so you know they are on hand:

- **`setText`** and **`clear`** - set the text yourself, or empty the box.

- **`appendText`** - add to the end of whatever is there.
- **`setEditable(false)`** - make the box read-only.
- **`selectAll`**, **`cut`**, **`copy`**, and **`paste`** - work with the selection and the clipboard.
- **`onChange`** - run a method **every time the text changes**, so you can react as the user types (a live character count, for instance).

We put that last one, `onChange`, to work in the next control.

Spinner - Stepped Numeric Input

A **`Spinner`** is a small box for a number, with up and down arrows to nudge the value. You create one by giving it a range and a starting value:

```
this.spinner = new Spinner(0.0, 10.0, 3.0);   // min, max, starting
                                              // value
this.spinner.onChange(::onChange);
```

Reading its value is a matter of `getValue`. But a spinner is a great place to react *live*, and that is what `onChange` does: it runs a method every time the value changes. This kind of method is called a **change listener**, and JavaFX hands it three things, the value that changed, its old value, and its new value:

```
public onChange(property, oldValue, newValue) {
    this.status.setText("Value: " + newValue);
}
```

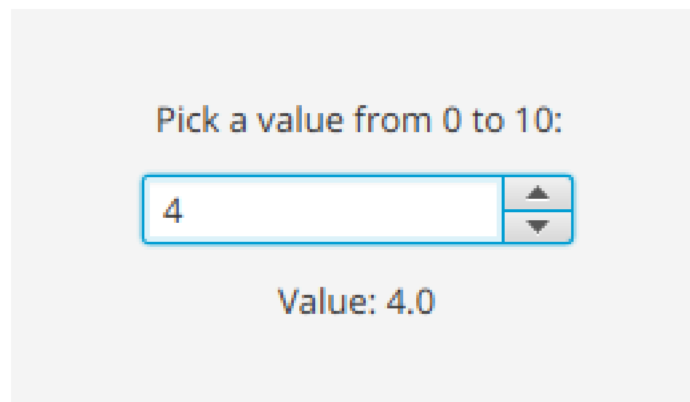


Figure 40: A spinner showing the number 4 with up and down arrows, above the label "Value: 4.0"

Two small notes. A `Spinner` built this way holds a **decimal** number, which is why the label reads `4.0` even though the box shows 4. And because the change listener receives the old and new values for you, you rarely need to call `getValue` from inside it, though it is there when you want it.

A spinner has a few more common tricks worth knowing: **`increment`** and **`decrement`** nudge its value from your own code, each taking an optional number of steps (`increment(2)` jumps by two; handy for a "reset" button), and **`setEditable(true)`** lets the user type a value straight into the box instead of only using the arrows.

Recap

- **`TextField`** is a one-line text box; **`PasswordField`** is the same but hides what is typed (your code can still read it). Both use **`getText`** to read, and **`setPromptText`** for a hint.
- **`TextArea`** is a multi-line box; `setWrapText` and `setPrefRowCount` shape it.
- All three share the **`TextInputControl`** base, so `getText`, `getLength`, `setText`, and `onChange` work the same on each.
- A **`Spinner`** holds a number in a range; read it with `getValue`, or react live with an **`onChange`** change listener that receives the old and new values.
- Text controls have many more options (selection, editability, formatting); the API reference documentation at aussom-lang.com has the full list.

You can now gather typed text and numbers from the user. In the next chapter we return to *choosing* rather than *typing*, with the controls for picking from a set of options: sliders, combo boxes, date pickers, and more.

Chapter 10 - Selection Controls

Chapters 7 and 9 covered controls the user clicks and types into. This chapter is about controls for **choosing a value**: picking one item from a list, dragging to a number, or popping up a color palette or a calendar. We also meet the two controls that *show* progress rather than take input. Here is the whole cast, grouped by what each one does:

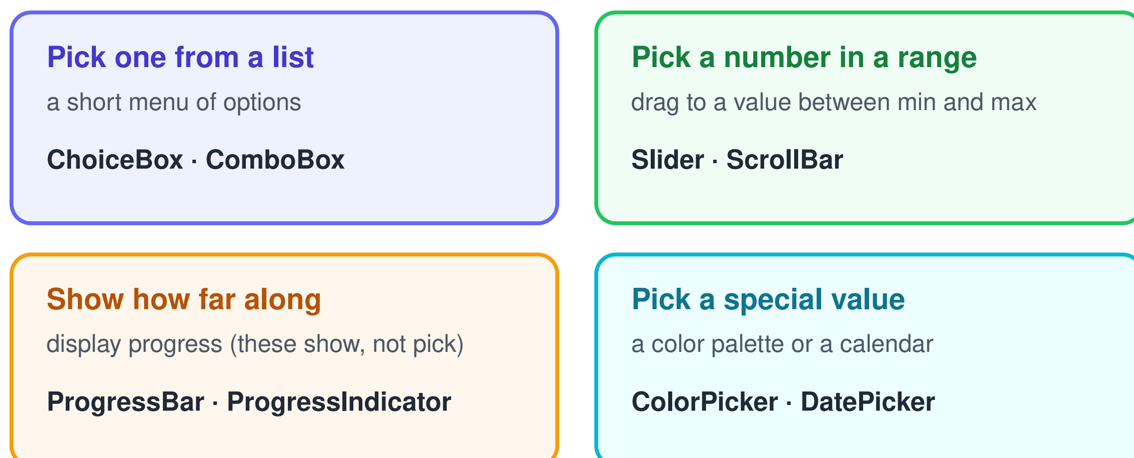


Figure 41: A grid of four groups: pick one from a list (ChoiceBox, ComboBox), pick a number in a range (Slider, ScrollBar), show how far along (ProgressBar, ProgressIndicator), and pick a special value (ColorPicker, DatePicker)

Every example in this chapter lives here: `10-selection-controls`.

ChoiceBox and ComboBox

Both of these are **drop-downs**: a small box that opens a list so the user can pick one item. A **`ChoiceBox`** is the simplest. You give it a list of items with `setItems`, optionally a starting choice with `setValue`, and read the current pick with `getValue`:

```
choice = new ChoiceBox(null);
choice.setItems(["Red", "Green", "Blue"]);   // the options
```

```
choice.setValue("Green");                        // the starting choice
choice.onChange(::onChange);                     // run onChange when it
                                                 // changes
```

The handler here is a **change listener**, the same shape you met with the spinner in Chapter 9: JavaFX hands it the property that changed, its old value, and its new value.

```
public onChange(property, oldValue, newValue) {
    this.status.setText("Color: " + this.choice.getValue() +
        ", Fruit: " + this.combo.getValue());
}
```

A **`ComboBox`** is a richer drop-down. It works the same way, but it can show more rows at once and can even be made **editable**, so the user may type a value that is not in the list. The example uses one for a fruit picker:

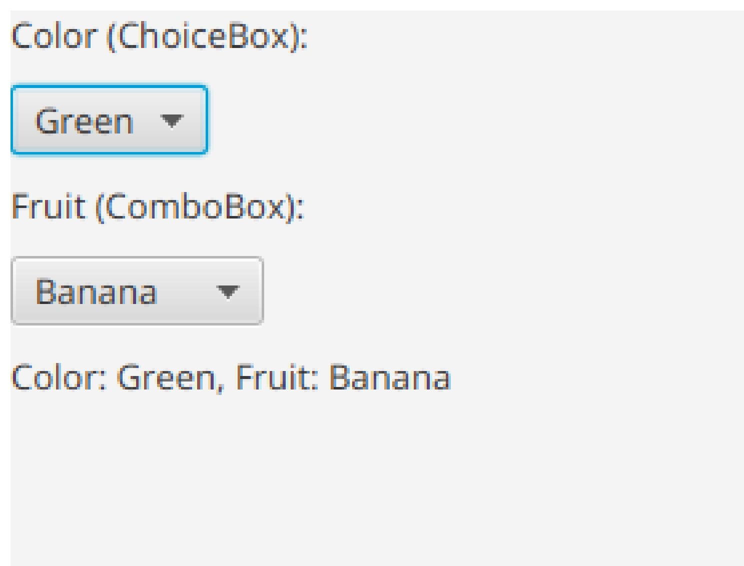


Figure 42: A window with a ChoiceBox set to Green, a ComboBox set to Banana, and a status line reading “Color: Green, Fruit: Banana”

Beyond the basics, these drop-downs offer a few more common options:

- **`setValue`** - choose an item from your own code.
- **`getItems`** - reach the list to add or remove options later.
- **`setEditable(true)`** (ComboBox) - let the user type a value, not just pick.
- **`setVisibleRowCount`** (ComboBox) - how many rows show before scrolling.

Slider and ScrollBar

A **`Slider`** lets the user drag a thumb along a track to choose a number in a range. You build it with a minimum, a maximum, and a starting value, and read it with `getValue`. A change listener updates a label live as the thumb moves:

```
slider = new Slider(0.0, 100.0, 40.0);   // min, max, starting value
slider.setShowTickMarks(true);
slider.setShowTickLabels(true);
slider.setMajorTickUnit(25.0);           // a labelled tick every 25
slider.onChange(::onSlide);
```

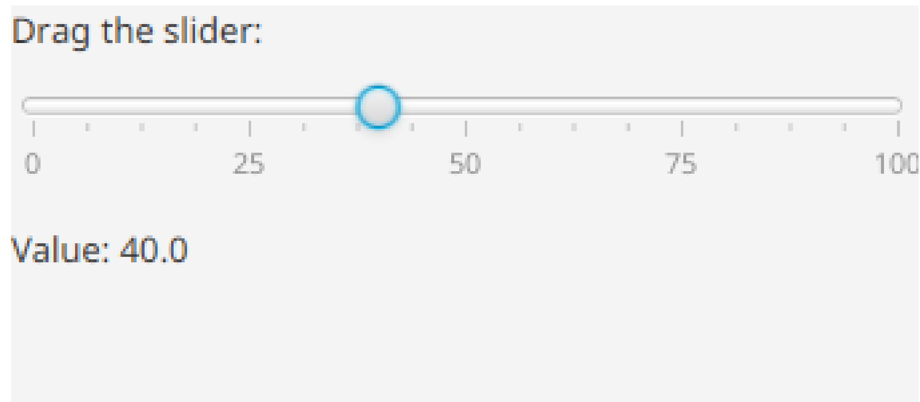


Figure 43: A slider set to 40 with tick labels 0, 25, 50, 75, 100, above a line reading “Value: 40.0”

A slider has several appearance options worth knowing:

- **`setShowTickMarks`** and **`setShowTickLabels`** - show the little marks and their numbers.
- **`setMajorTickUnit`** - the spacing between labelled ticks.
- **`setSnapToTicks(true)`** - make the thumb settle exactly on ticks.
- **`setOrientation(fxorient.VERTICAL)`** - stand the slider up on its end.

A **`ScrollBar`** is a close relative: it also drags a thumb to pick a value in a range, with the same `setMin`, `setMax`, `getValue`, and `onChange`. You will rarely create one yourself, though, because scroll bars appear automatically inside a `ScrollPane` (Chapter 6); reach for a `Slider` when you want the user to choose a number directly.

ProgressBar and ProgressIndicator

These two are the odd ones out: they do not take input at all. Instead they **show** how far along a task is, a download, a file copy, anything with a sense of “percent done.” A **`ProgressBar`** is a horizontal bar; a **`ProgressIndicator`** is a round dial. Both take a progress value from `0.0` (nothing) to `1.0` (done):

```
bar = new ProgressBar();
bar.setProgress(0.65);           // 65% complete

spinner = new ProgressIndicator();
spinner.setProgress(-1.0);       // indeterminate: a spinning "working"
                                 // dial
```

The special value `-1.0` puts the control in **indeterminate** mode, the endless spinner you show when you know a task is running but not how long it will take. As a real task makes progress, you would

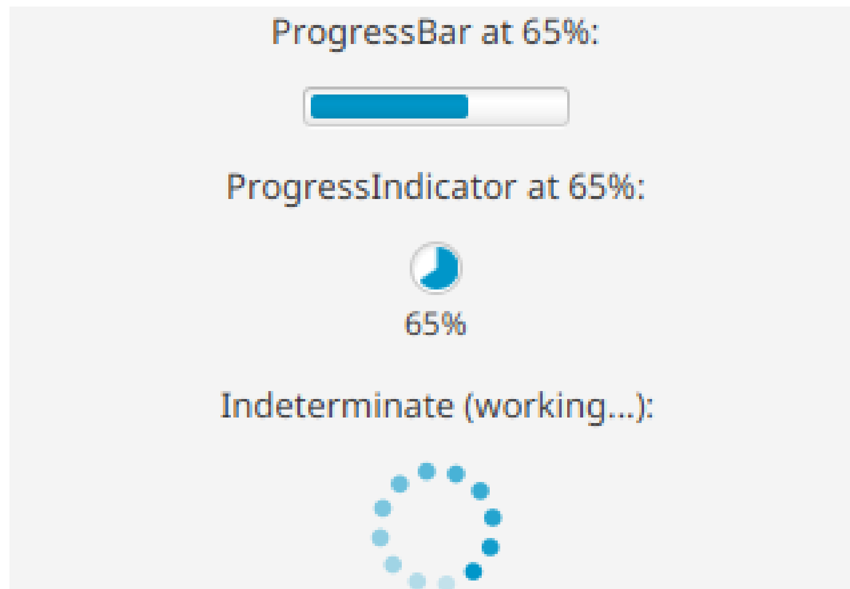


Figure 44: A ProgressBar at 65%, a round ProgressIndicator at 65%, and an indeterminate spinning indicator

call `setProgress` with rising values. If that work runs on a background thread, remember to push each update back to the UI thread with `fx.runLater`, the function you first met in Chapter 2 (which shows exactly how to use it).

ColorPicker and DatePicker

The last two are **specialized pickers** that pop up a whole mini-interface. A **`ColorPicker`** opens a color palette; a **`DatePicker`** opens a calendar.

A `ColorPicker` needs a starting color, which you make from a hex string with `Color`. Reading its value with `getValue` gives you back a tidy hex string:

```
colorPicker = new ColorPicker(new Color("#3366cc"));
colorPicker.onChange(::onColor);

public onColor(event) {
    // getValue() returns a hex string like "#3366CC"
    this.status.setText("Color: " + this.colorPicker.getValue());
}
```

A `DatePicker` opens a calendar, and it speaks Aussom's own **`Date`** type (more on that just below). On its own it starts empty, with a prompt inviting the user to pick:

```
datePicker = new DatePicker();
datePicker.setPromptText("Pick a date");
datePicker.onChange(::onDate);
```

There is one difference worth flagging. For these two pickers, `onChange` is the simpler, one-

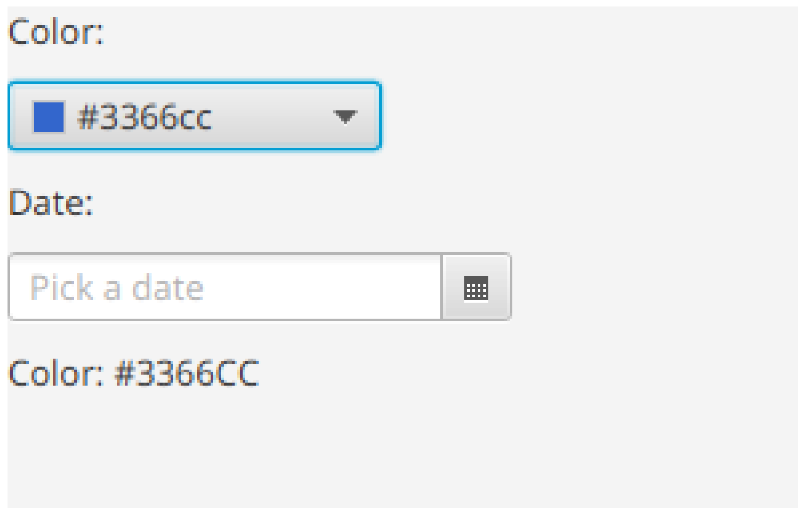


Figure 45: A ColorPicker showing a blue swatch and #3366cc, a DatePicker prompting "Pick a date", and a status line reading "Color: #3366CC"

argument kind, it runs *when the user makes a pick*, just like a button's `onClick`, rather than the three-argument change listener the drop-downs and slider use. So its handler takes a single `event`, and you read the value with `getValue`. Both styles are common in JavaFX; the rule of thumb is simply to read the value inside the handler and not worry about which flavor it is.

A color is more than a swatch. If you wrap a color value in a `Color` (for example `new Color(colorPicker.getValue())`), you get a set of genuinely useful helpers:

- **`fx.toHexString(color)`** turns a color into a hex string like `#3366CC` (a `ColorPicker`'s `getValue` already does this for you).
- **`getRed`**, **`getGreen`**, and **`getBlue`** read the red, green, and blue amounts (each `0.0` to `1.0`); **`getHue`**, **`getSaturation`**, and **`getBrightness`** give the same color the other way, as hue/saturation/ brightness.
- **`brighter()`** and **`darker()`** hand back a lighter or darker shade (with `saturate`, `desaturate`, `grayscale`, and `invert` for other effects).
- You can build a color not just from a hex string (`new Color("#3366cc")`) but from red/green/blue or hue/saturation/brightness values (`rgb`, `hsb`).

Working with dates. A `DatePicker` gives and takes a value of Aussom's own `Date` type, the same date type built into the language, and a `Date` can do far more than hold a day. This example starts the picker on a chosen date and, each time the user picks a new one, describes it:

```
public main(args) {
    // Build a starting date from text with parse().
    start = new Date();
    start.parse("2026-07-15", "yyyy-MM-dd");

    this.picker = new DatePicker();
    this.picker.setValue(start);       // a DatePicker accepts an Aussom
                                       // Date
    this.picker.onChange(::onPick);
```

```
    // ...
}

public onPick(event) {
    this.describe(this.picker.getValue());   // getValue returns an
                                              // Aussom Date
}

public describe(date) {
    this.status.setText(date.format("EEEE, MMMM d, yyyy") +
        " (day " + date.getDayOfYear() + " of the year)");
}
```

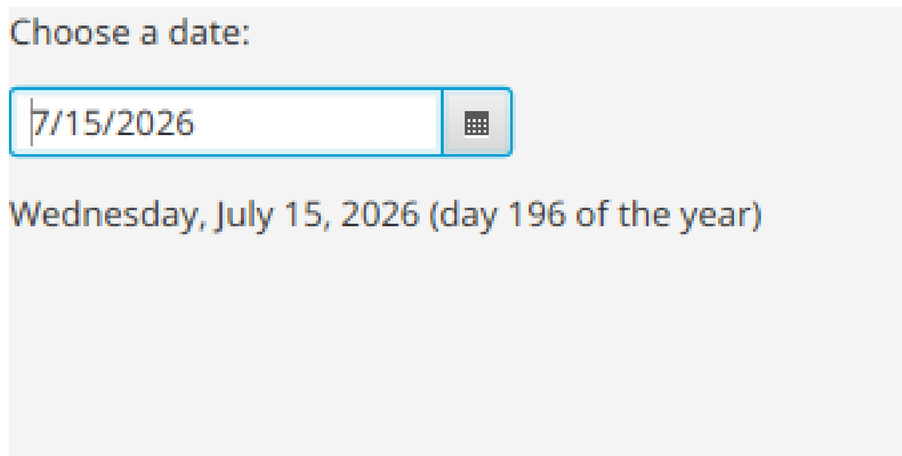


Figure 46: A DatePicker set to 7/15/2026, above the line “Wednesday, July 15, 2026 (day 196 of the year)”

A `Date` is a small toolbox in its own right. The most useful things it can do:

- **`format(pattern)`** and **`parse(text, pattern)`** turn a date into text or read one back. The pattern `"EEEE, MMMM d, yyyy"` produced the line above; the letters stand for the weekday, month, day, and year.
- **`getYear`**, **`getMonth`**, **`getDayOfMonth`** pull a date apart, and **`getMonthName`** and **`getDayOfWeekName`** give friendly names.
- **`addDays`**, **`addMonths`**, **`addYears`** (and hours, minutes, seconds) do date arithmetic; `start.addDays(30)` moves a date a month ahead.
- **`isBefore`**, **`isAfter`**, and **`between`** compare two dates or measure the gap between them (`a.between(b, "days")` counts the days from a to b).

One thing to know: `new Date()` on its own is the **epoch** (January 1, 1970), a blank starting point. Call **`now()`** for today’s date, or `parse` (as above) to set a specific one.

Recap

- **`ChoiceBox`** and **`ComboBox`** are drop-downs for picking one item from a list (`setItems`, `getValue`); a `ComboBox` can also be typed into.
- **`Slider`** drags to a number in a range (`Slider(min, max, value)`, `getValue`); **`ScrollBar`** is similar but usually comes from a `ScrollPane`.
- **`ProgressBar`** and **`ProgressIndicator`** *show* progress from `0.0` to `1.0`, or spin when set to `-1.0`.
- **`ColorPicker`** and **`DatePicker`** pop up a palette or a calendar; `getValue` gives the color (as a hex string) or an Aussom **`Date`**, which can `format` itself, break into parts, and do date math.
- Most of these report changes through **`onChange`** — a three-argument change listener for the drop-downs and slider, a one-argument action for the pickers. Either way, read the value with `getValue` inside the handler.

You can now offer the user almost every everyday way to enter a value. In the next chapter we move up to controls that present *many* values at once: lists, tables, and trees.

Chapter 11 - Lists, Tables, and Trees

The controls so far have dealt with one value at a time. This chapter is about controls that present **many** values at once and let the user pick from them: a scrollable **list**, a **table** of rows and columns, and a **tree** of items that nest inside one another. They are the workhorses for showing real data.

Every example in this chapter lives here: `11-lists-tables-trees`.

ListView - Scrollable Lists

A **`ListView`** is the simplest of the three: a scrollable column of items the user can select. You hand it a list when you create it, and read the current pick with `getSelected`.

```
list = new ListView(["Apples", "Bananas", "Cherries", "Dates"]);
list.onChange(::onSelect);

public onSelect(property, oldValue, newValue) {
    this.status.setText("You picked: " + this.list.getSelected());
}
```

The `onChange` handler is the three-argument change listener you have seen since the drop-downs in Chapter 10; it runs each time the selection changes.

A ListView has more on offer:

- **`getSelectedIndex`** reads the position rather than the item.
- **`setItems`** / **`getItems`** change the list later.
- **`setSelectionMode`** allows picking several items at once (an `fxselectionmode` value: `fxselectionmode.SINGLE` or `fxselectionmode.MULTIPLE`).
- **`setPlaceholder`** shows a message when the list is empty.
- **`setCellFactory`** renders each row as more than plain text, an icon, a checkbox, several lines, with a custom cell (a more advanced technique).
- **`setEditable`** lets the user rename items in place.

TableView - Tabular Data

A **`TableView`** shows data in rows and columns, like a spreadsheet. Setting it up takes two pieces: the **columns** and the **rows**.

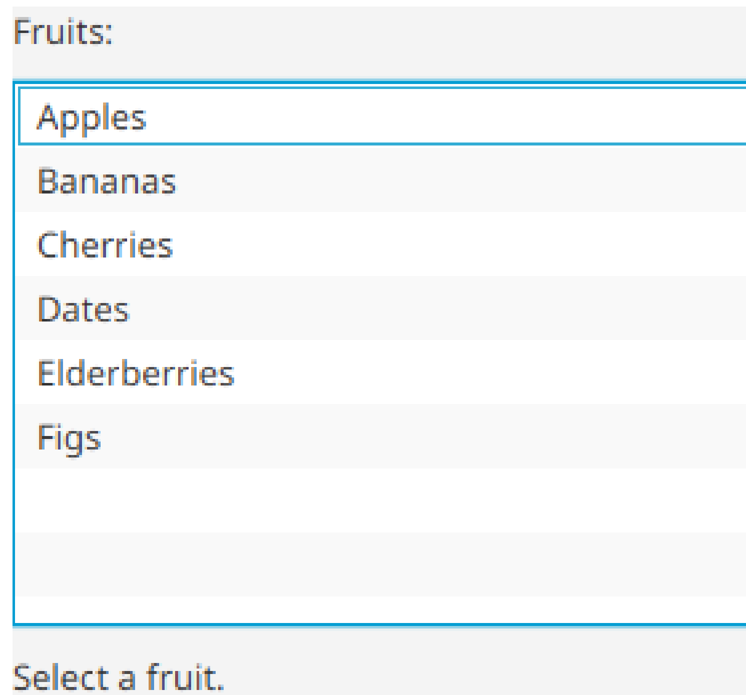


Figure 47: A ListView of fruits (Apples, Bananas, Cherries, Dates, Elderberries, Figs) with the first selected, above the line "Select a fruit."

Each column is described by two things: a **`display`** (the header text) and a **`colName`** (the key it should read from each row). Each row is a **map**, a collection of key-value pairs written in braces, whose keys match the column `colName`s:

```
table = new TableView();

table.setColumns([
    {"display": "Name", "colName": "name"},
    {"display": "Age",  "colName": "age"},
    {"display": "City", "colName": "city"}
]);

table.setItems([
    {"name": "Alice", "age": 30, "city": "Portland"},
    {"name": "Bob",   "age": 25, "city": "Denver"},
    {"name": "Carol", "age": 41, "city": "Austin"}
]);
```

The `colName` is the glue between the two: it names the header *and* the row key to pull the value from, so the value lands in the right cell.

Reading the selection gives you back the whole row map, so you pull out the fields you want by key:

```
public onSelect(property, oldValue, newValue) {
    row = this.table.getSelected();
```

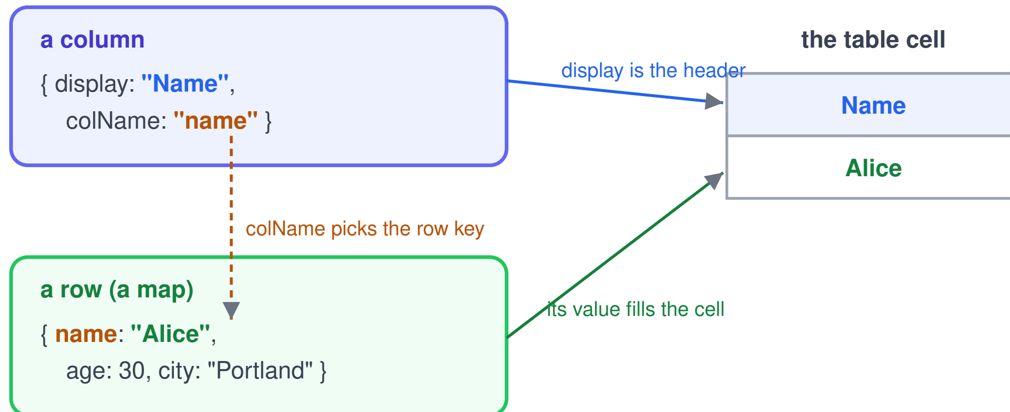


Figure 48: A diagram: a column {display “Name”, colName “name”} and a row map {name “Alice”, …}; display becomes the header, colName picks the row key, and its value fills the cell

People:

Name	Age	City
Alice	30	Portland
Bob	25	Denver
Carol	41	Austin

Select a row.

Figure 49: A TableView with Name, Age, and City columns and three rows for Alice, Bob, and Carol, above the line “Select a row.”

```
    this.status.setText("You picked: " + row["name"] + " from " +
↪   row["city"]);
}
```

Out of the box, a TableView lets the user **sort** by clicking a column header and **resize** columns by dragging their edges. You can also sort it yourself from code with **`sortBy`**, naming the column and a direction:

```
table.sortBy("Age", fxsorttype.DESCENDING);   // sort by the Age column,
                                              // high to low
```

The direction is an `fxsorttype` value (`fxsorttype.ASCENDING` or `fxsorttype.DESCENDING`), and **`clearSort`** returns the rows to their original order. Beyond that:

- **`getSelectedIndex`** reads the selected row number.
- **`setColumnResizePolicy`** controls how columns share the width.
- **`setPlaceholder`** shows a message when there is no data.
- For richer cells (images, checkboxes, colored or formatted values) or in-place editing, a TableView supports custom cell rendering and **`setEditable`**, more advanced techniques in the API reference.

TreeView and TreeItem - Hierarchical Data

A **`TreeView`** shows items that **nest** inside one another, like folders and files, or categories and their members. You build the tree out of **`TreeItem`** objects. Each `TreeItem` holds a value and can hold child items, and the whole tree hangs from a single **root**.

```
mammals = new TreeItem("Mammals");
mammals.add([new TreeItem("Dog"), new TreeItem("Cat"), new
↪   TreeItem("Horse")]);

birds = new TreeItem("Birds");
birds.add([new TreeItem("Eagle"), new TreeItem("Sparrow")]);

root = new TreeItem("Animals");
root.add([mammals, birds]);     // put the branches under the root
root.setExpanded(true);         // open it so the children show

tree = new TreeView(root);
```

Selection in a tree is a little different: `getSelected` gives you the selected `TreeItem`, not just its text, so you read the text from it with `getValue`:

```
public onSelect(property, oldValue, newValue) {
    if (newValue != null) {
        this.status.setText("You picked: " + newValue.invoke("getValue"));
    }
}
```

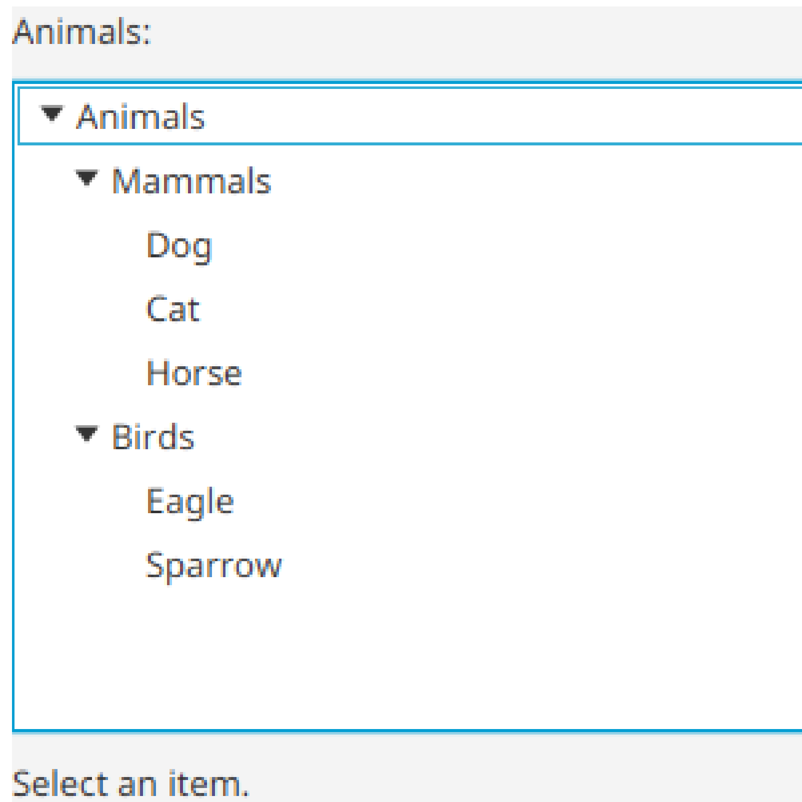


Figure 50: A TreeView with a root “Animals” containing “Mammals” (Dog, Cat, Horse) and “Birds” (Eagle, Sparrow), all expanded, above the line “Select an item.”

(newValue here is the newly selected TreeItem; .invoke("getValue") reads the text we gave it. We check for null because clicking empty space clears the selection.)

Adding icons

Real trees, like a list of folders and files, usually show a small **icon** beside each label. Any TreeItem can carry one with **setGraphic**, which takes a node to place before the text. The usual choice is an **ImageView** (a node that displays a picture) built from an **Image** (a picture loaded from a file):

```
folderImg = new Image("icons/folder.png");   // load each picture once
fileImg   = new Image("icons/file.png");

documents = new TreeItem("Documents");
documents.setGraphic(new ImageView(folderImg));

notes = new TreeItem("notes.txt");
notes.setGraphic(new ImageView(fileImg));
documents.add([notes]);
```

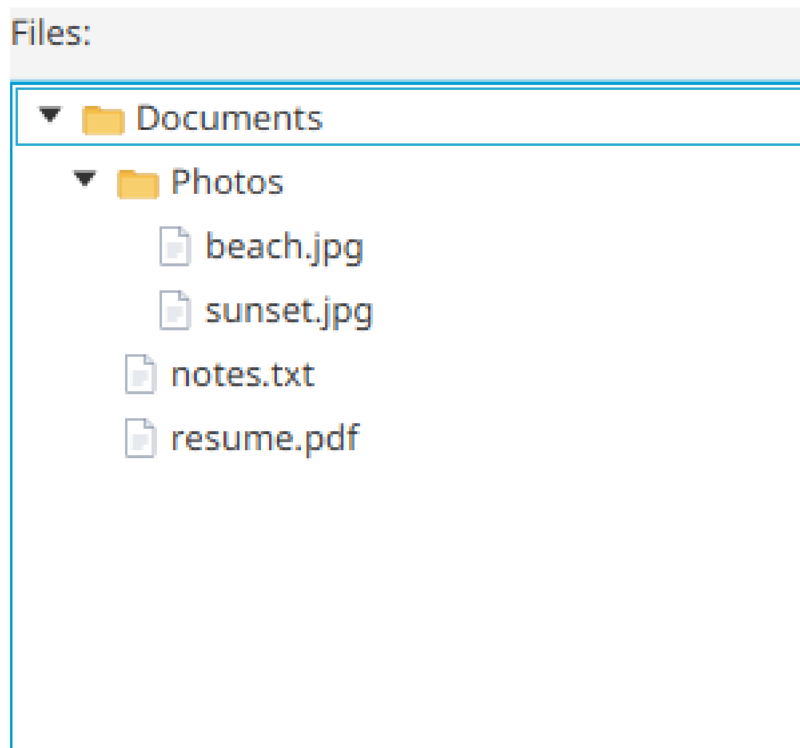


Figure 51: A TreeView of folders and files where every item shows a folder or file icon beside its name

Two things are worth knowing. Each item needs its **own** ImageView (a node can appear in only one place), but they can all share one Image, so you load each picture just once. And you size an icon to fit the row with setFitWidth and setFitHeight, as the full example does.

A tree offers plenty more:

- **`setShowRoot(false)`** hides the single root when it is just a container.
- Each `TreeItem` can report **`getChildren`**, **`getParent`**, and **`isLeaf`** (whether it has children).
- **`setOnBranchExpanded`** / **`setOnBranchCollapsed`** run a method when a branch opens or closes.

Recap

- **`ListView`** shows a scrollable list; build it from a list and read the choice with **`getSelected`**.
- **`TableView`** shows rows and columns: **columns** carry a `display` header and a `colName` key, and **rows** are **maps** keyed by those `colNames`. Selecting a row gives you its map back.
- **`TreeView`** shows nested data built from **`TreeItem`** objects under one **root**; its selection is a `TreeItem` whose text you read with `getValue`, and each item can carry an icon with **`setGraphic`**.
- All three report selection through the three-argument **`onChange`** change listener, and all three have far more options in the API reference at aussom-lang.com.

That completes our tour of the input controls. In the next chapter we turn to the commands that frame an application: menus, toolbars, and tooltips.

Chapter 12 - Menus, Toolbars, and Tooltips

The controls in the last few chapters sit *in* your window. This chapter is about the controls that *frame* an application: the menu bar along the top, the toolbar of quick buttons, the menu that pops up on a right-click, and the little hint that appears when you hover. Together they turn a screenful of controls into something that feels like a real program.

A quick note on the pictures. Menus, right-click menus, and tooltips all **pop up** in their own little windows, so a snapshot of the main window does not capture them open. The screenshots below show the parts that live in the window; the diagrams show what the pop-ups look like.

Every example in this chapter lives here: `12-menus-toolbars-tooltips`.

MenuBar, Menu, and MenuItem

The strip of words across the top of most applications, File, Edit, View, is a **`MenuBar`**. It holds one **`Menu`** per word, and each `Menu` holds the **`MenuItem`**s that drop down when you click it. A `MenuItem` runs an action with `onClick`, exactly like the menu items on the buttons back in Chapter 7.

```
newItem = new MenuItem("New");
newItem.onClick(::onNew);
openItem = new MenuItem("Open");
openItem.onClick(::onOpen);

fileMenu = new Menu("File");
fileMenu.add([newItem, openItem]);

bar = new MenuBar();
bar.add([fileMenu, viewMenu]);

app.setMenuBar(bar);     // attach the bar to the window
```

You attach the finished bar to the window with `app.setMenuBar`, which places it neatly along the top:

Like every handler you have written, a menu item's `onClick` runs on the UI thread, so it can change the interface directly.

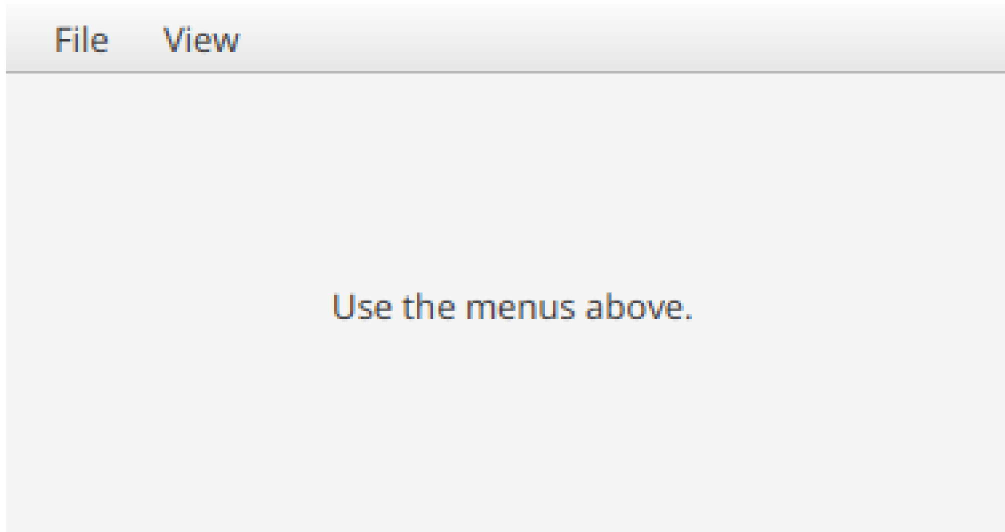


Figure 52: A window with a menu bar showing File and View, above the text "Use the menus above."

Check, Radio, and Separator Items

A menu can hold more than plain commands. Four item types cover almost everything you will want:

- A **`MenuItem`** is a plain command, like Undo or Open.
- A **`CheckMenuItem`** carries a tick you can toggle on and off (`setSelected`, `isSelected`), good for an option like "Word Wrap".
- A **`RadioMenuItem`** is like a check item, but you put several in a `ToggleGroup` so only one can be on at a time, perfect for choosing a theme.
- A **`SeparatorMenuItem`** is just a dividing line that groups related items.

```
wrap = new CheckMenuItem("Word Wrap");
wrap.setSelected(true);

group = new ToggleGroup();
light = new RadioMenuItem("Light Theme");
light.setToggleGroup(group);
light.setSelected(true);
dark = new RadioMenuItem("Dark Theme");
dark.setToggleGroup(group);

viewMenu = new Menu("View");
viewMenu.add([wrap, new SeparatorMenuItem(), light, dark]);
```

There is a fifth type, the **`CustomMenuItem`**, which lets you put any node (a slider, say) right inside a menu when the four standard kinds are not enough.

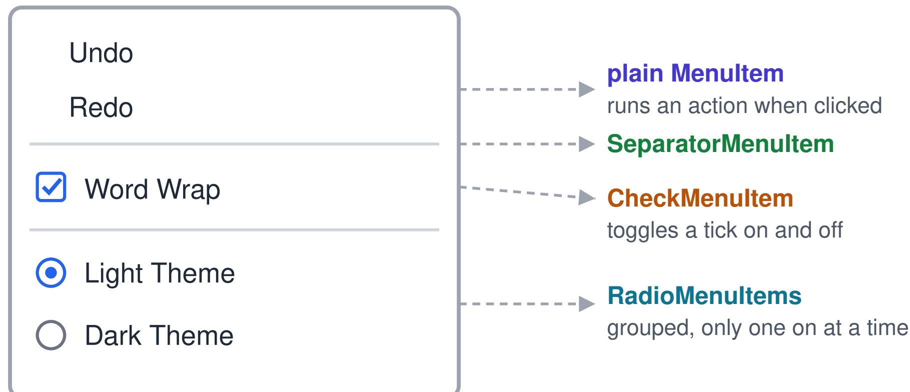


Figure 53: A diagram of an open menu showing Undo and Redo (plain items), a divider, a checked "Word Wrap" box, another divider, and "Light Theme"/"Dark Theme" radio buttons, each labelled with its item type

ContextMenu - Right-Click Menus

A **`ContextMenu`** is the menu that appears when you **right-click** something. You build it from `MenuItems` just like a regular menu, then attach it to any node, and JavaFX shows it on right-click automatically.

```
menu = new ContextMenu();
menu.add([cutItem, copyItem, pasteItem]);

button.setContextMenu(menu);   // attach it to the button
```

Attach a context menu wherever a right-click should offer choices, a list row, a canvas, a whole window.

ToolBar, ButtonBar, and Separator

A **`ToolBar`** is a strip of buttons and other controls, usually just under the menu bar, for the actions people reach for most. You add controls to it in order, and drop in a **`Separator`** to divide them into groups:

```
toolbar = new ToolBar();
toolbar.add([
    new Button("New"), new Button("Open"), new Button("Save"),
    new Separator(),   // a divider between the two groups
    new Button("Cut"), new Button("Copy"), new Button("Paste")
]);
```

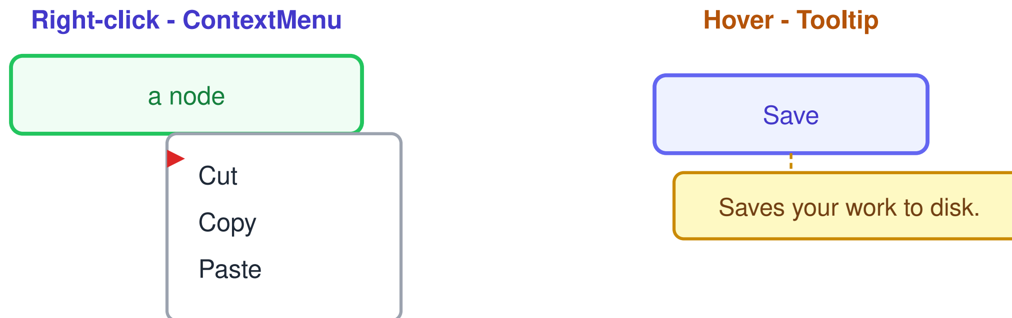


Figure 54: A diagram: a node with a right-click context menu (Cut, Copy, Paste) popped up beside it, and a button with a tooltip bubble reading “Saves your work to disk.” below it

A **ButtonBar** is a specialist for the row of buttons at the bottom of a dialog, OK, Cancel, Apply. Its trick is that it arranges those buttons in the order each operating system expects, so your dialogs feel native everywhere.

Figure 55: A window showing a ToolBar with New, Open, Save then a divider then Cut, Copy, Paste, and below it a ButtonBar with OK, Cancel, and Apply

A `Separator` on its own is simply a thin dividing line; it runs vertically inside a horizontal toolbar, as above, or horizontally to split stacked content, and you flip it with **`setOrientation`** (an `fxorient` value: `fxorient.HORIZONTAL` or `fxorient.VERTICAL`).

Adding Icons

The toolbar above is all words, but real applications lean on **icons**. Any button, a toolbar button or a plain one, can show a picture with **`setGraphic`**, and so can any **`MenuItem`**.

There are two easy ways to get an icon:

- An **image file**, exactly as Chapter 11 put icons on tree items: wrap an `Image` in an `ImageView` and pass it to `setGraphic`.
- A **glyph icon** from the built-in **Font Awesome** set, handy when you want a crisp symbol and no image files to manage.

The glyph route is a one-liner. After including the module, `fontawesome.iconView` builds an icon node from a name and a size, and `setGraphic` places it on a button:

```
include fx.fontawesome.fontawesome;

// An icon-only toolbar button.
saveTool = new Button("");
saveTool.setGraphic(fontawesome.iconView("SAVE", "16px"));

// A plain button with an icon beside its text.
saveBtn = new Button("Save");
saveBtn.setGraphic(fontawesome.iconView("SAVE", "16px"));
```

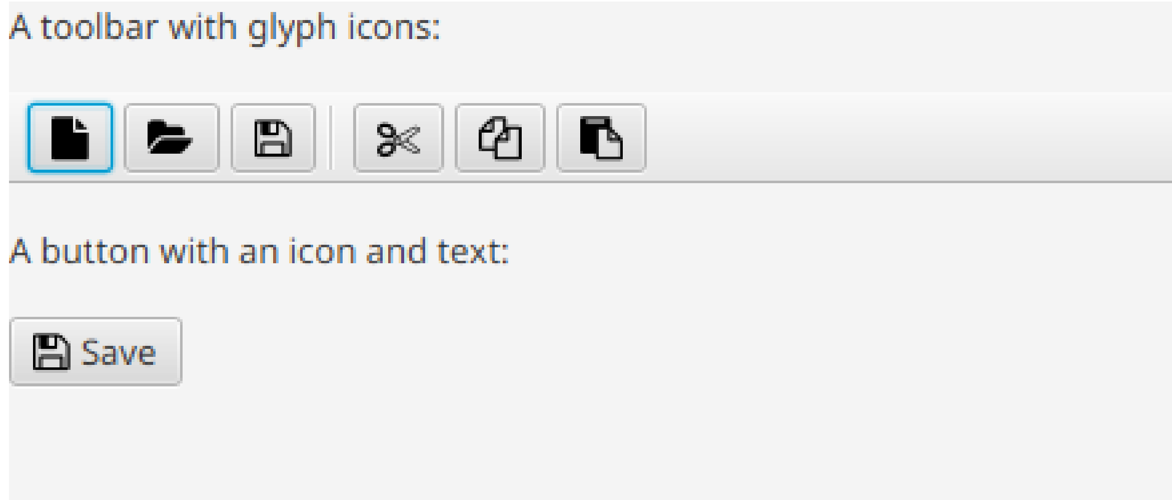


Figure 56: A toolbar of icon-only buttons (new file, open folder, save, cut, copy, paste) above a Save button showing a floppy-disk icon beside its text

A few things make glyphs pleasant to use:

- The name is one of hundreds of icons (`"SAVE"`, `"FOLDER_OPEN"`, `"TRASH"`, `"SEARCH"`, and more); `fontawesome.names()` lists them all.
- Because a glyph is drawn from a font, it stays crisp at any size, and you can recolor it (`setFill`) like text.
- The very same `setGraphic` works on a `MenuItem`, so a File > Save menu entry can carry the same little disk.

Tooltip - Hover Hints

Finally, a **`Tooltip`** is the small label that appears when the mouse rests on a control, the perfect place for a hint about what a button does. You create one with its text and attach it to a control

with `setTooltip`:

```
tip = new Tooltip("Saves your work to disk.");
button.setTooltip(tip);
```

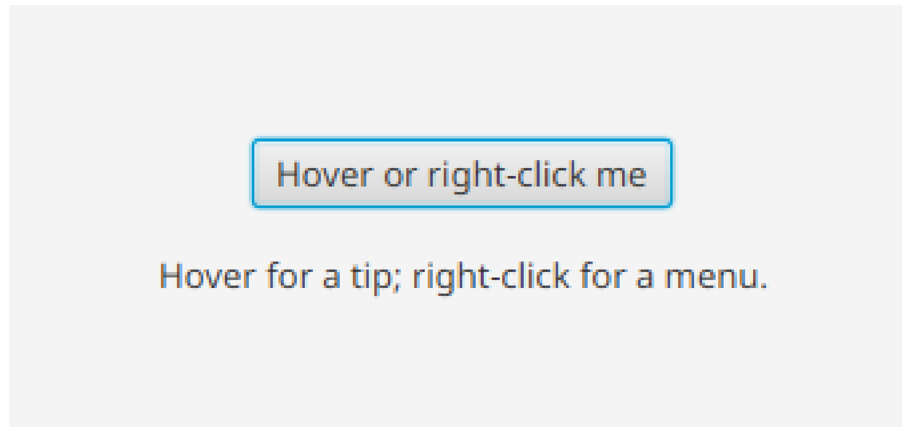


Figure 57: The button "Hover or right-click me" above the line "Hover for a tip; right-click for a menu."

That is all it takes; JavaFX shows and hides the tip as the pointer comes and goes (the tooltip half of the diagram above shows how it looks). A tooltip can be tuned further:

- **`setShowDelay`** sets how long the pointer must rest before the tip appears, in milliseconds (`setShowDelay(200.0)` for a fifth of a second).
- **`setWrapText(true)`** lets a long hint wrap onto several lines.

Recap

- A **`MenuBar`** holds **`Menu`**s, each holding **`MenuItem`**s; attach it with `app.setMenuBar`, and give items actions with `onClick`.
- Menus can also hold a **`CheckMenuItem`** (a tick), grouped **`RadioMenuItem`**s (one at a time), **`SeparatorMenuItem`** dividers, and a **`CustomMenuItem`** for any node.
- A **`ContextMenu`** attaches to a node with `setContextMenu` and pops up on right-click.
- A **`ToolBar`** is a strip of quick controls, a **`ButtonBar`** orders dialog buttons, and a **`Separator`** draws a divider.
- Any button or **`MenuItem`** can show an icon with **`setGraphic`**, from an image (`ImageView`) or a Font Awesome glyph (`fontawesome.iconView`).
- A **`Tooltip`** attaches with `setTooltip` and shows a hint on hover.

Next we reach the controls that appear *on top of* your window when you need the user's attention: dialogs and the file and folder choosers.

Chapter 13 - Dialogs and File Choosers

Sometimes an app needs to stop and talk to the user: to confirm a risky action, to ask for a name, or to let the user pick a file. For that you use a **dialog**, a small window that appears on top of your own and waits for an answer. This chapter covers the ready-made dialogs, the system file choosers, and how to build a custom dialog when the built-ins are not enough.

A note on the pictures, as in Chapter 12: most dialogs open in their **own window** (and the file choosers are provided by the **operating system**), so a snapshot of the main window cannot capture them. The diagrams show what they look like; the real screenshots are the demo's main window and the custom dialog.

Every example in this chapter lives here: `13-dialogs-file-choosers`.

Our first demo is a window with a button for each kind of dialog:

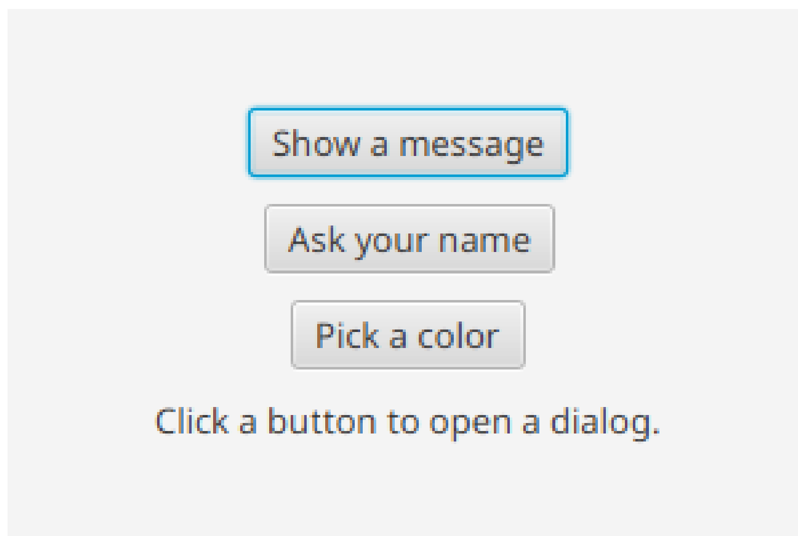


Figure 58: A window with three buttons, "Show a message", "Ask your name", and "Pick a color", above the text "Click a button to open a dialog."

Alert and the DialogPane

An **`Alert`** is the simplest dialog: a message with an icon and a button or two. You choose its kind with an **`fxalert`** value, `fxalert.INFORMATION`, `fxalert.WARNING`, `fxalert.ERROR`, or

`fxalert.CONFIRMATION`, give it text, and call **`showAndWait`**, which pauses your program until the user closes it.

```
alert = new Alert(fxalert.INFORMATION, "Your work has been saved.");
alert.setTitle("Saved");
alert.showAndWait();
```

(`fxalert` is one of Aussom's enums, like `fxpos` from the layout chapters. Each value is really just its own name as text, so a misspelling is caught right away rather than failing deep inside JavaFX.)

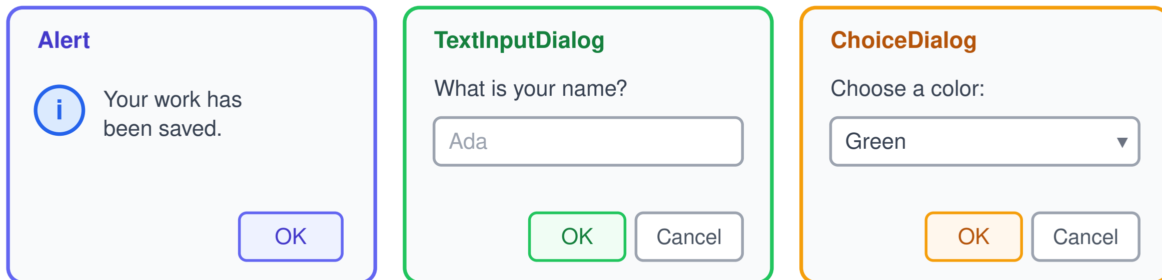


Figure 59: Three mock dialogs: an Alert with an info icon and OK, a TextInputDialog with a text field and OK/Cancel, and a ChoiceDialog with a dropdown and OK/Cancel

For an information, warning, or error alert, showing it is all you need. A `fxalert.CONFIRMATION` alert adds OK and Cancel buttons, and after `showAndWait` you can check `getResult` to see which one the user clicked.

Every dialog, the `Alert` here and the two below, shares three text pieces you can set:

- **`setTitle`** sets the window's title-bar text.
- **`setHeaderText`** sets the larger heading line inside the dialog.
- **`setContentText`** sets the main message (the `Alert` constructor's second argument fills this in for you).

Under the hood, every dialog is built on a **`DialogPane`**, the inner area that arranges the icon, the text, and the buttons. You rarely touch it directly, but it is there if you ever need to customize a standard dialog's contents.

TextInputDialog and ChoiceDialog

Two dialogs go a step further and collect an answer.

A **`TextInputDialog`** shows a single text field. After `showAndWait`, its `getResult` gives you what the user typed, or `null` if they cancelled:

```
dialog = new TextInputDialog("");
dialog.setHeaderText("What is your name?");
dialog.showAndWait();
name = dialog.getResult();
if (name != null) {
    this.status.setText("Hello, " + name + "!");
}
```

A **`ChoiceDialog`** shows a drop-down of options. You give it a default and the list of choices, and read the pick with `getSelectedItem`:

```
dialog = new ChoiceDialog("Green", ["Red", "Green", "Blue"]);
dialog.setHeaderText("Choose a color:");
dialog.showAndWait();
this.status.setText("You chose " + dialog.getSelectedItem() + ".");
```

FileChooser and DirectoryChooser

When you need a real file, a **`FileChooser`** opens the operating system's own open/save dialog, so it looks and behaves exactly the way the user expects on their platform.

```
chooser = new FileChooser("Open a File");
chooser.addFilter("Text Files", ["*.txt"]);   // optional file-type
                                              // filter
path = chooser.show(app.getAjo());             // the app is the parent
                                               // window
if (path != null) {
    this.status.setText("You opened: " + path);
}
```

There are two things to know. You pass the chooser your app window with `app.getAjo()` so the dialog knows what it belongs to, and `show` hands back the chosen **path** as a string (or `null` if the user cancels). A few more settings shape it:

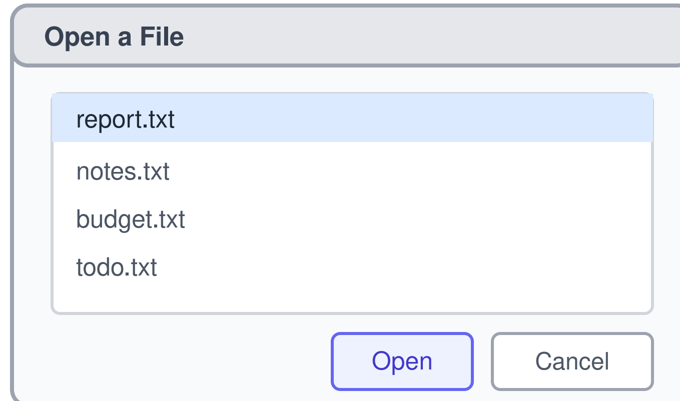
- **`addFilter(name, extensions)`** adds a file-type filter, like `addFilter("Text Files", ["*.txt"])` above; add several to offer a choice.
- **`setInitDir`** picks the folder the dialog opens in, and **`setInitFileName`** suggests a name (handy for a save dialog).
- **`show`** opens it to *open* a file, **`showSave`** to *save* one, and **`showMultiple`** lets the user pick several at once (it returns a list of paths).

A **`DirectoryChooser`** is the matching control for choosing a *folder* rather than a file.

Building a Custom Dialog with fxDialog

When none of the built-in dialogs fit, you can build your own. **`fx.fxDialog`** gives you a small window, just like `fx.fxApp`, that you fill with any layout and controls you like:

A FileChooser uses the operating system's own dialog



show() returns the chosen path, or null if the user cancels

Figure 60: A mock operating-system file dialog titled "Open a File" with a list of text files and Open/Cancel buttons, noting that show() returns the chosen path or null

```
dialog = fx.fxDialog("Confirm", 320, 150);   // title, width, height

message = new Label("Save your changes before closing?");
save = new Button("Save");
save.onClick(::onSave);
discard = new Button("Don't Save");
discard.onClick(::onDiscard);

buttons = new HBox();
buttons.setSpacing(10);
buttons.setAlignment(fxpos.CENTER);
buttons.add([discard, save]);

layout = new VBox();
layout.setSpacing(18);
layout.setAlignment(fxpos.CENTER);
layout.add([message, buttons]);

dialog.setLayout(layout);
dialog.show(true);      // show it and wait until it closes
```

Because an `fxDialog` is just a window you control, everything you have learned applies: layouts, controls, events, all of it. Each button's handler calls `dialog.close()` to dismiss the window when it is done.

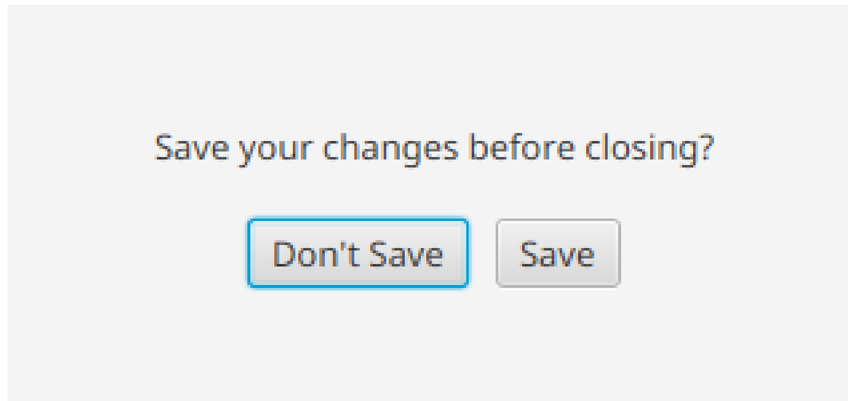


Figure 61: A custom dialog titled Confirm reading "Save your changes before closing?" with Don't Save and Save buttons

Recap

- An **`Alert`** shows a message (`fxalert.INFORMATION`, `fxalert.WARNING`, `fxalert.ERROR`) or asks a yes/no question (`fxalert.CONFIRMATION`); **`showAndWait`** pauses until the user answers.
- A **`TextInputDialog`** collects a line of text (`getResult`), and a **`ChoiceDialog`** collects a pick from a list (`getSelectedItem`).
- A **`FileChooser`** opens the system open/save dialog and returns the chosen path (or `null`); a **`DirectoryChooser`** does the same for folders.
- **`fx.fxDialog`** builds a custom dialog window you fill and control yourself.

That wraps up the controls. In the next part we shift from *what* to put on screen to *how it looks*, starting with text and typography.

Part IV

Text, Color, and Style

Chapter 14 - Text and Typography

So far every word on screen has come from a `Label`, a control built for short, plain captions. This chapter is about text as its own subject: the **`Text`** node for drawing and styling words richly, **`TextFlow`** for paragraphs that mix styles and wrap like a document, and the **`Font`** that decides how any of it looks.

This is where an app starts to feel designed rather than merely functional, and it is the natural lead-in to the next two chapters on color and CSS.

Every example in this chapter lives here: `14-text-typography`.

The Text Node

A **`Text`** is a **shape** that draws a string, in the same family as rectangles and circles. That is the key difference from a `Label`: a `Label` is a control for UI captions, while a `Text` is a graphic you can **fill** with a color, outline with a **stroke**, and drop into a drawing or a `TextFlow`.

You create one with its position and its string, then style it. (The `0.0, 0.0` is where the text sits; inside a layout pane, the pane positions it for you, so zeros are fine.)

```
title = new Text(0.0, 0.0, "Bold and blue");
// Font.font is covered below in this chapter
title.setFont(Font.font("System", fxweight.BOLD, fxposture.REGULAR, 24.0));
title.setFill(new Color("#3366cc"));   // the letters' color
title.setUnderline(true);
```

A `Text` node styles more than the font:

- **`setFill`** paints the letters (a solid color now, or a gradient in Chapter 15).
- **`setStroke`** paints an outline around the glyphs, and **`setStrokeWidth`** sets its thickness in pixels.
- **`setUnderline(true)`** and **`setStrikethrough(true)`** add a line under or through the text.
- **`setWrappingWidth`** wraps a long single `Text` at a given width in pixels.

TextFlow - Rich, Flowing Text

One `Text` node has one style. To mix styles in a single paragraph, bold here, a colored word there, you put several `Text` runs inside a **`TextFlow`**. A `TextFlow` is a container that lays its runs out end to end and **wraps** them across lines, exactly like text in a document.

Plain text

Bold and blue

Italic gray

Underlined

Struck through

Figure 62: Five lines of text: plain, large bold blue, italic gray, underlined, and struck through

```
intro   = new Text(0.0, 0.0, "JavaFX makes it easy to mix ");
bold    = new Text(0.0, 0.0, "bold");
bold.setFont(Font.font("System", fxweight.BOLD, fxposture.REGULAR, 15.0));
colored = new Text(0.0, 0.0, "colored");
colored.setFill(new Color("#cc0000"));
rest    = new Text(0.0, 0.0, " words that wrap neatly across lines.");

flow = new TextFlow();
flow.add([intro, bold, colored, rest]);   // runs flow in order
```

Because a `TextFlow` is a container, you add runs to it just like children of a layout box. It also shapes the paragraph as a whole:

- **`setTextAlignment`** aligns the lines with an **`fxtextalign`** value (`fxtextalign.LEFT`, `fxtextalign.CENTER`, `fxtextalign.RIGHT`, `fxtextalign.JUSTIFY`).
- **`setLineSpacing`** sets the gap between lines, in pixels.
- **`setTabSize`** sets how wide a tab character is, counted in spaces.

Working with Font

Every `Text` (and every `Label` and control) draws with a **`Font`**: a typeface at a chosen weight, slant, and size. `Font` is a small helper you call directly, and its fullest form takes four parts:

A flowing paragraph:

JavaFX makes it easy to mix **bold**, colored, and *italic* words in one flowing paragraph that wraps neatly across lines.

Figure 63: A paragraph mixing bold, red, and italic words that wraps across three lines

The four parts of a font

```
Font.font("Helvetica", fxweight.BOLD, fxposture.ITALIC, 18.0)
```

renders as ***Helvetica***

Figure 64: A diagram of Font.font("Helvetica", fxweight.BOLD, fxposture.ITALIC, 18.0) with each argument labelled: family, weight, posture, size

```
Font.font("System", fxweight.BOLD, fxposture.ITALIC, 18.0);  // family,
                                                             // weight,
                                                             // posture,
                                                             // size
Font.get("System", 18.0);                                    // just a
                                                             // family
                                                             // and size
```

- **`Font.get(family, size)`** is the short form when you only need a typeface and a size, no bold or italic.
- The **weight** is an **`fxweight`** value like `fxweight.NORMAL` or `fxweight.BOLD` (with in-between steps such as `fxweight.LIGHT` and `fxweight.BLACK`), and the **posture** is an **`fxposture`** value, `fxposture.REGULAR` or `fxposture.ITALIC`. These enums, like `fxpos` earlier, each evaluate to their own name, so a typo is caught right away.
- **`Font.getFamilies`** lists every typeface installed on the computer, and **`Font.getDefault`** returns the standard one.
- **`Font.load`** installs a font from a file you ship with your app, so your design looks the same everywhere.

`"System"` always works because it is the platform's default typeface; for anything else, pick a family name from `Font.getFamilies` so you know it is present.

Recap

- A **`Text`** node draws a string as a shape you can **`setFill`**, **`setStroke`**, **`setUnderline`**, and **`setStrikethrough`**.
- A **`TextFlow`** holds several `Text` runs and flows them into one wrapping, mixed-style paragraph.
- A **`Font`** is a family, weight, posture, and size; build one with **`Font.font(...)`** or the shorter **`Font.get(...)`**.

Text is styled one node at a time here. Next we look at **color and paint**, the fills that bring both text and shapes to life, and then at **CSS**, which styles a whole window at once.

Chapter 15 - Color and Paint

Color has quietly been everywhere already, a hex string handed to `setFill` or a `ColorPicker`. This chapter gives it a proper look: the **`Color`** class and the many ways to build one, and then **gradients**, paints that blend between colors for backgrounds, buttons, and glows.

The word for all of these is **paint**: anything you can fill a shape or a piece of text with. A solid `Color` is the simplest paint; a gradient is a fancier one.

Every example in this chapter lives here: `15-color-paint`.

The Color Class

A **`Color`** is one solid color. The quickest way to make one is a **hex string**, the same `#rrggbb` code used on the web, but there are others. You build the non-hex ones by starting with new `Color()` and calling the builder you want:

```
new Color("#e63946");                 // a hex string
new Color("#e63946").darker();        // a darker shade of another
                                      // color
new Color().color(0.2, 0.6, 0.86);    // red, green, blue amounts, each
                                      // 0.0-1.0
new Color().hsb(140.0, 0.6, 0.7);     // hue (0-360), saturation,
                                      // brightness
new Color().gray(0.6);                // a shade of gray, 0.0 black to
                                      // 1.0 white
new Color("#3366cc", 0.4);            // a color plus alpha
                                      // (transparency)
```

The last one shows **alpha**: a second number from `0.0` (fully see-through) to `1.0` (fully solid), so colors can let what is behind them show through. Pick whichever builder fits how you are thinking about the color, they all produce the same kind of `Color`.

A `Color` can also be taken apart and adjusted:

- **`getRed`**, **`getGreen`**, **`getBlue`**, and **`getOpacity`** read the pieces back out (each `0.0` to `1.0`).
- **`getHue`**, **`getSaturation`**, and **`getBrightness`** read the same color the hue/saturation/brightness way.
- **`brighter`** and **`darker`** return a new `Color` a step lighter or darker (the `darker()` swatch above), ready to hand straight to `setFill`; **`saturate`** and **`desaturate`** adjust how vivid it is.

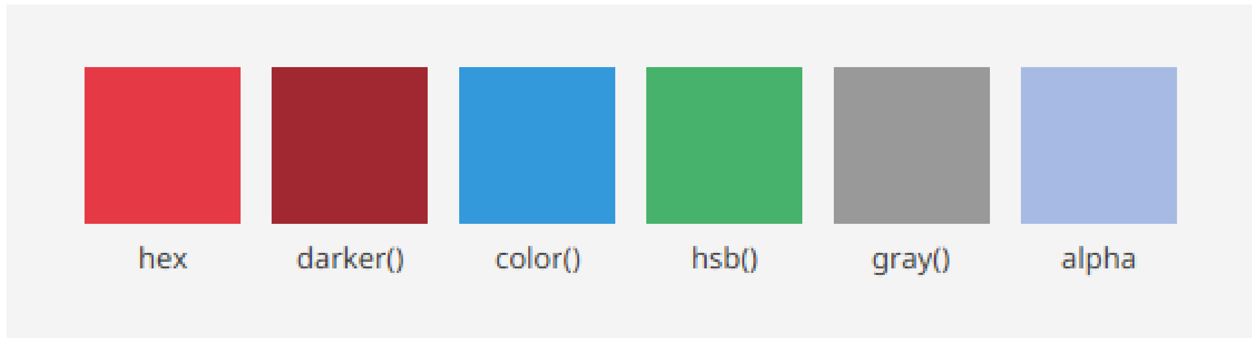


Figure 65: Six colored squares captioned hex, darker(), color(), hsb(), gray(), and alpha

Gradients: LinearGradient, RadialGradient, and Stop

A **gradient** is a paint that blends smoothly between two or more colors. Each color is pinned to a **`Stop`**, a position from `0.0` to `1.0` along the blend:

```
new Stop(0.0, new Color("#ff8800"));   // orange at the start
new Stop(1.0, new Color("#cc0000"));   // red at the end
```

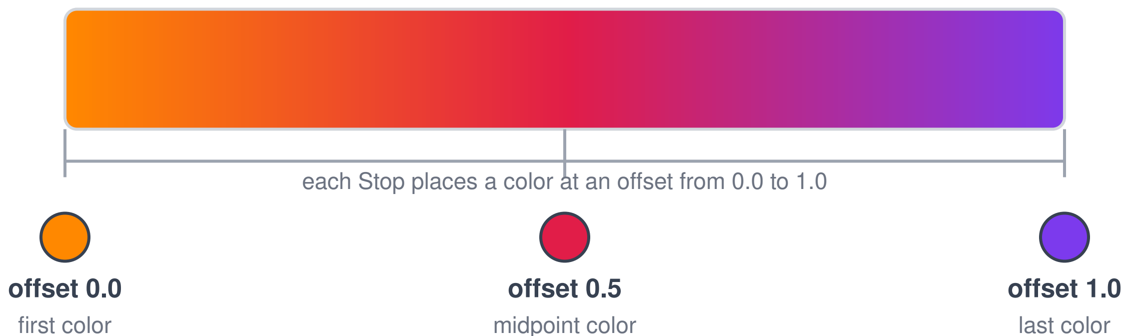


Figure 66: A gradient bar with three stops marked at offsets 0.0, 0.5, and 1.0, each placing a color along the blend

A **`LinearGradient`** blends those stops along a straight line, and a **`RadialGradient`** blends them outward from a center point, like a spotlight. You give each the line or circle to blend over, then the list of stops:

```
// Linear: from the left edge (0,0) to the right edge (1,0).
rect.setFill(new LinearGradient(0.0, 0.0, 1.0, 0.0, true,
↪   fxcyclemethod.NO_CYCLE,
    [new Stop(0.0, new Color("#ff8800")), new Stop(1.0, new
↪   Color("#cc0000"))]));

// Radial: centered at (0.5, 0.5) with radius 0.5, white fading to blue.
```

```
circle.setFill(new RadialGradient(0.0, 0.0, 0.5, 0.5, 0.5, true,
↪   fxcyclemethod.NO_CYCLE,
    [new Stop(0.0, new Color("#ffffff")), new Stop(1.0, new
↪   Color("#1d4ed8"))]));
```

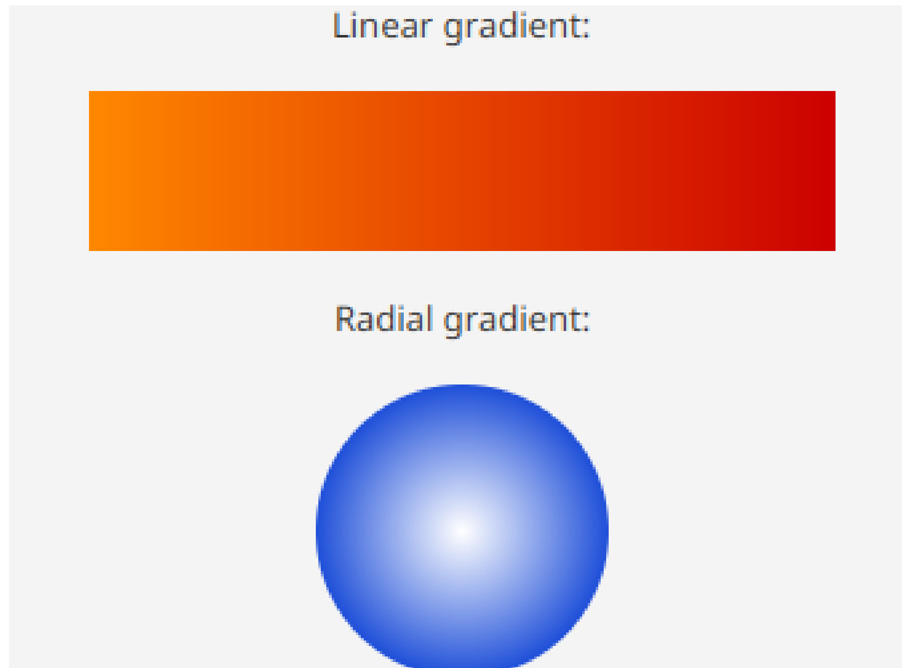


Figure 67: A horizontal orange-to-red linear gradient above a circle filled with a white-to-blue radial gradient

The two `true` and `fxcyclemethod.NO_CYCLE` arguments are the parts worth knowing:

- **`proportional`** (the `true`) measures the coordinates as fractions of the shape, so `0.0` to `1.0` spans it edge to edge whatever its size. Pass `false` to use exact pixel positions instead.
- **`cycleMethod`** is an **`fxcyclemethod`** value that decides what happens past the last stop: `fxcyclemethod.NO_CYCLE` holds the end color, while `fxcyclemethod.REFLECT` and `fxcyclemethod.REPEAT` tile the blend outward.
- You can pass **more than two stops** to blend through several colors, as the diagram above does.

A `RadialGradient` starts with two extra numbers, its **focus angle** and **focus distance**, which offset the bright spot away from the center; the `0.0, 0.0` above just keeps it centered.

Recap

- A **`Color`** is one solid paint; build it from a hex string, from **`color`** (red/green/blue), **`hsb`**, or **`gray`**, and add **alpha** for transparency.
- A **`Stop`** pins a color to an offset from `0.0` to `1.0` along a blend.
- A **`LinearGradient`** blends its stops along a line and a **`RadialGradient`** blends them outward from a center; fill any shape or text with either.

Colors and gradients style one node at a time. In the next chapter, **CSS** lets you set all of this, and much more, for a whole window from a single stylesheet.

Chapter 16 - Styling with CSS

Setting a color here and a font there works, but it scatters your app's look across the code. **CSS** gathers it in one place. It is the same styling language that dresses up web pages, and JavaFX speaks a version of it, so you can describe how a whole window looks in a single file and change the whole theme by editing it.

JavaFX's CSS property names all start with **`-fx-`** (for example `-fx-background-color`, `-fx-text-fill`, `-fx-font-size`), but the idea, a **selector** that picks nodes and a set of **properties** that style them, is exactly the web's.

Every example in this chapter lives here: `16-styling-css`.

Style Classes and Inline Styles

The quickest way to style one node is an **inline style**: a small string of CSS set right on it with **`setStyle`**. You have seen this since the layout chapters.

```
label.setStyle("-fx-text-fill: #cc0000; -fx-font-style: italic;");
```

Inline styles are perfect for a one-off, but repeating them on every button gets tedious. For that you write the CSS once in a **stylesheet** (a `.css` file) and pick nodes with **selectors**. There are four kinds worth knowing:

- A **type selector** matches nodes by kind. Every control carries a built-in **style class** from its type, a `Button` is a `.button`, a `Label` is a `.label`, so `.label { ... }` styles them all at once. The window's root node always has the class **`.root`**, handy for the page background.
- A **class selector** matches your own named style. Tag any node with **`addStyleClass("primary")`** and every tagged node picks up the `.primary { ... }` rule, the same way you reuse a class across a web page.
- An **id selector** matches one node. Give a node an id with **`setId("title")`**, then target it with `#title { ... }`.
- When two rules touch the same node, the **more specific** one wins: an id beats a class, which beats a type selector.

Two more selector tricks are worth remembering:

- **Pseudo-classes** style a node in a certain state: `.button:hover { ... }` applies while the pointer is over it, and `:pressed` while it is held down.
- A **descendant** selector like `.root .label { ... }` matches labels inside the root, letting you scope rules to part of the window.

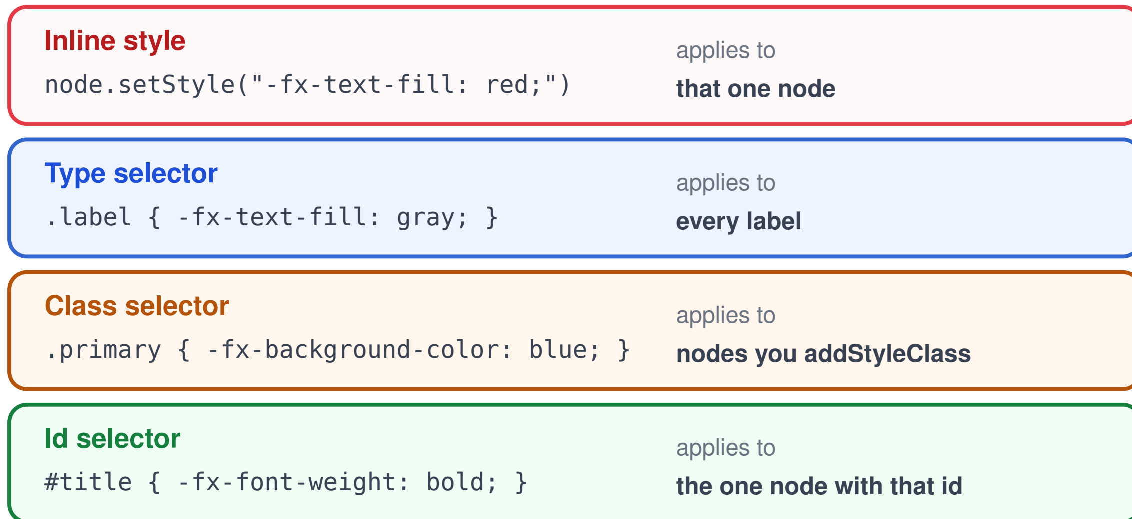


Figure 68: A diagram of four ways to apply a style: an inline style on one node, a type selector on every label, a class selector on nodes you tag, and an id selector on one node

Attaching Stylesheets

A stylesheet does nothing until you attach it to the app with **`addStyleSheet`**. Here is the stylesheet for our demo:

```
.root    { -fx-background-color: #eef1f6; -fx-padding: 18; }
.label   { -fx-text-fill: #333333; }
.card    { -fx-background-color: white; -fx-padding: 16;
↪   -fx-background-radius: 10;
           -fx-border-color: #e2e6ee; -fx-border-radius: 10; }
.primary { -fx-background-color: #3366cc; -fx-text-fill: white;
           -fx-padding: 8 18; -fx-background-radius: 6; }
.pill    { -fx-background-color: #22c55e; -fx-text-fill: white;
           -fx-padding: 6 16; -fx-background-radius: 14; }
#title   { -fx-font-size: 22px; -fx-font-weight: bold; -fx-text-fill:
↪   #3366cc; }
```

The app tags nodes with a class or an id and attaches the file; the selectors do the rest:

```
title = new Label("Styled with CSS");
title.setId("title");              // matched by "#title"
save = new Button("Save");
save.addStyleClass("primary");    // matched by ".primary"
card = new VBox();
card.addStyleClass("card");       // matched by ".card"
// ... add them to a layout ...
```

```
app.addStyleSheet("styles.css"); // attach the stylesheet
```

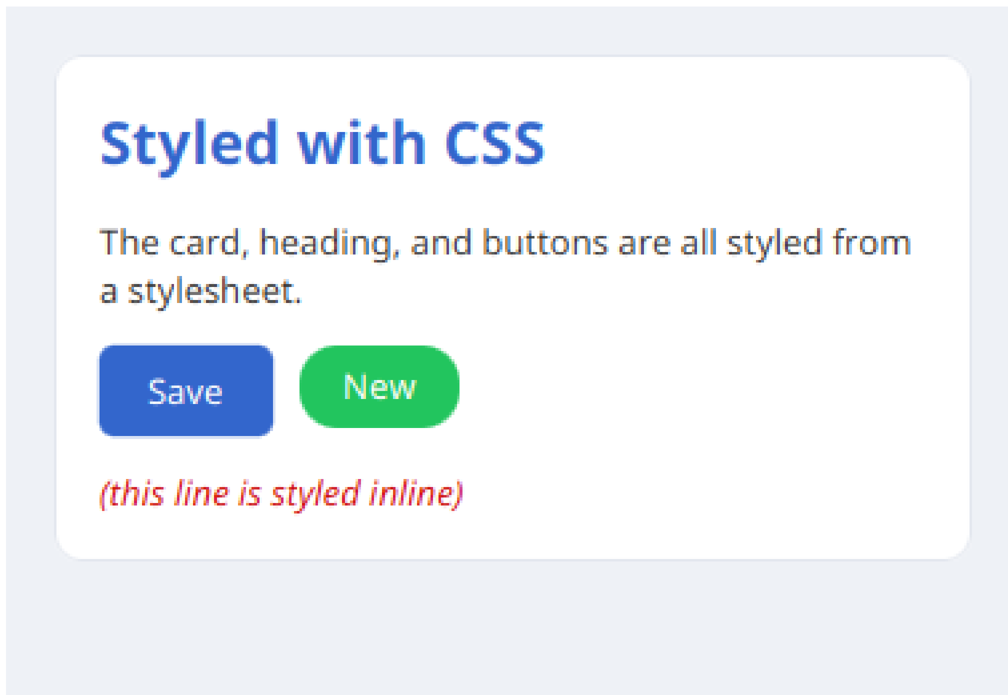


Figure 69: A styled window: a white rounded card holding a blue bold heading, gray body text, a blue Save button, a green pill New button, and a red italic inline-styled line, on a light page

Notice how little the Aussom code has to say about looks now: it builds the controls and tags a few with a class or id, and every color, size, and corner radius lives in the stylesheet. Change the file and the whole window restyles.

The property names all start with `-fx-`. A handful cover most of what you will style:

- **`-fx-background-color`** and **`-fx-background-radius`** - a fill and how round the corners are.
- **`-fx-text-fill`**, **`-fx-font-size`**, and **`-fx-font-weight`** - the text color, size, and boldness.
- **`-fx-padding`** - the inner cushion, one value for all sides (`-fx-padding: 12;`) or four for `top right bottom left` (`-fx-padding: 8 16 8 16;`).
- **`-fx-border-color`** and **`-fx-border-radius`** - an outline and its corners.

Live CSS Reloading with CSSFX

Tweaking a design usually means edit, restart, look, over and over. **CSSFX** removes the restart: it watches your stylesheet and reloads it into the running app the instant you save. Turn it on with one call:

```
include cssfx;
// ...
cssfx.startMonitor();   // watch attached stylesheets and reload on save
```

Now edit `styles.css` while the app is open, change `.primary` to green, save, and the button recolors immediately, no restart. It is a development helper you leave out of the shipped app, but while you are designing it makes styling feel instant.

Recap

- An **inline style** (`setStyle`) styles one node; a **stylesheet** styles many from one file.
- Selectors pick nodes by kind (**type selectors** like `.label`), by a named **class** (`addStyleClass`, matched by `.name`), by the root (**`.root`**), or by **id** (`#name` after `setId`); a more specific selector wins.
- **`addStyleSheet`** attaches a stylesheet to the app.
- **CSSFX** (`cssfx.startMonitor`) live-reloads your stylesheet as you edit it.

This closes our part on text, color, and style. Next we move into graphics and media: drawing on a canvas, shapes, images, and animation.

Part V

Graphics and Media

Chapter 17 - 2D Shapes

We reached shapes briefly in the color chapter, filling a rectangle and a circle with gradients. This chapter is the full set: the **Shape** family that lets you draw rectangles, circles, ellipses, arcs, lines, many-sided polygons, and even freeform paths, all as nodes you place, color, and outline.

This is the start of **Part V, Graphics and Media**. Shapes are the foundation: once you can draw and style them, canvases, effects, and animation all build on the same ideas.

Every example in this chapter lives here: `17-2d-shapes`.

The Shape Base Class

Every shape is a **node**, just like a control, so it goes into a scene the same way. What they all share comes from their common base, **Shape**:

- **setFill** paints the inside (a `Color` or a gradient from Chapter 15).
- **setStroke** paints the outline, and **setStrokeWidth** sets its thickness.

Shapes are placed by **coordinates**, measured in pixels from the top-left corner, with x growing rightward and y growing **downward**.

The Pane container

To let shapes sit at exact coordinates, we put them in a **Pane**. A Pane is the simplest container of all: unlike the `VBox` and `HBox` from the layout chapters, which arrange their children for you, a Pane does **no** arranging, each child stays exactly where you place it. (In fact `VBox`, `HBox`, and the other layout panes are all built *on top of* Pane, each adding its own way of arranging.) That makes a plain Pane the natural home for a drawing or any free-form layout.

You use it in three steps: create it, add children, and position each one yourself, shapes by the coordinates in their constructor, other nodes with **setLayoutX** and **setLayoutY**:

```
pane = new Pane();
pane.add([rect, circle, line]);   // each sits where its own coordinates
                                  // put it
```

Like any container, a Pane also grows to fill the space it is given and can hold any number of children.

Beyond fill and stroke, `Shape` offers finer control:

- **setStrokeType** puts the outline inside, centered on, or outside the edge.

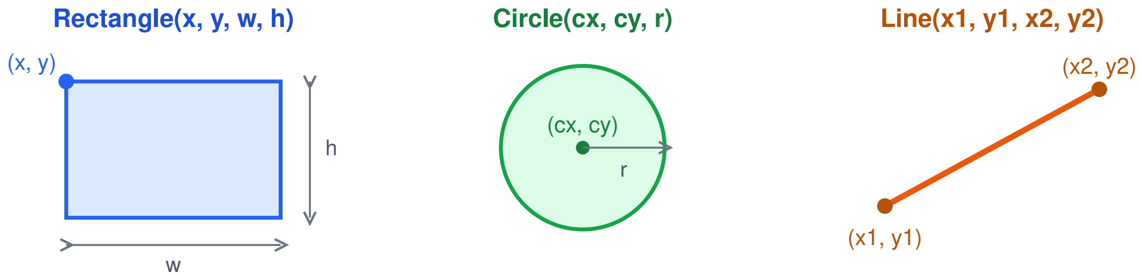


Figure 70: A diagram showing a Rectangle defined by its top-left x,y and width and height; a Circle by its center and radius; and a Line by its two end points

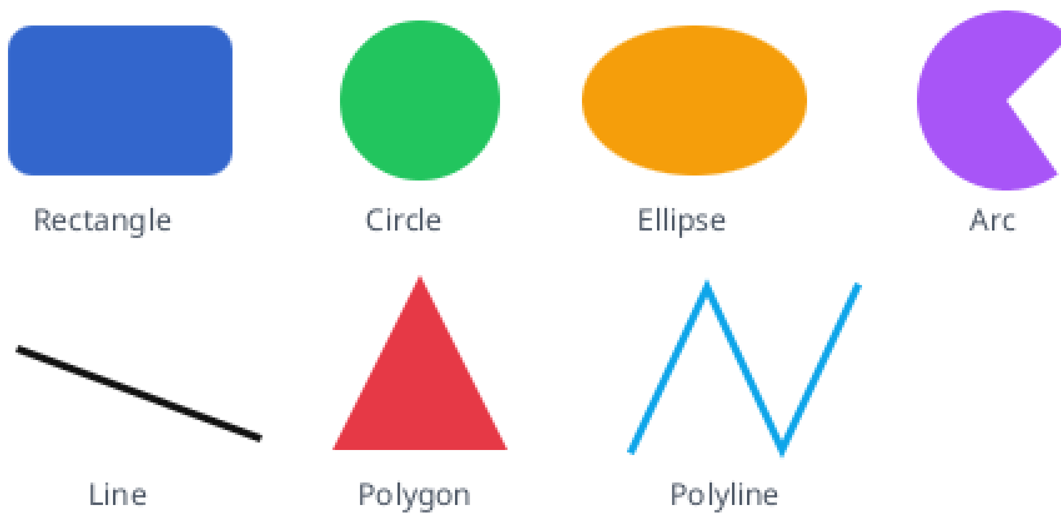


Figure 71: A window of shapes: a rounded rectangle, circle, ellipse, and arc across the top, and a line, triangle, and zig-zag across the bottom, each captioned

- **`setStrokeLineCap`** shapes the stroke's ends (an `fxlinecap` value: `fxlinecap.BUTT`, `fxlinecap.ROUND`, or `fxlinecap.SQUARE`), and **`setStrokeLineJoin`** shapes its corners (an `fxlinejoin` value: `fxlinejoin.MITER`, `fxlinejoin.BEVEL`, or `fxlinejoin.ROUND`).
- **`setStrokeDashOffset`** (with dash settings) draws dashed outlines.
- **`union`**, **`subtract`**, and **`intersect`** combine two shapes into a new one.

Rectangle, Circle, Ellipse, and Arc

These four cover most rounded and boxed shapes. Each constructor takes the numbers that define it, and you style it with the Shape methods above:

```
rect = new Rectangle(20.0, 20.0, 90.0, 60.0);   // x, y, width, height
rect.setArcWidth(20.0);                          // round the corners
rect.setArcHeight(20.0);
rect.setFill(new Color("#3366cc"));

circle  = new Circle(185.0, 50.0, 32.0);              // centerX,
                                                      // centerY, radius
ellipse = new Ellipse(295.0, 50.0, 45.0, 30.0);       // center,
                                                      // radiusX,
                                                      // radiusY
arc     = new Arc(420.0, 50.0, 36.0, 36.0, 45.0, 260.0);  // +
                                                          // startAngle,
                                                          // length
arc.setType(fxarctype.ROUND);
```

Two are worth a note:

- A **`Rectangle`** becomes a rounded rectangle once you set **`setArcWidth`** and **`setArcHeight`** (the corner radii).
- An **`Arc`** is a slice of an ellipse from `startAngle` through `length` degrees; **`setType`** picks how it closes, using an **`fxarctype`** value: `fxarctype.OPEN`, `fxarctype.CHORD` (a straight chord), or `fxarctype.ROUND` (a pie slice, as above). Like the book's other enums, each value is just its own name, so a typo is caught at once.

Line, Polygon, and Polyline

The next three are built from points. A **`Line`** is just two end points and has a stroke but no inside to fill. A **`Polygon`** and a **`Polyline`** take a flat list of `x, y` numbers through **`addPoints`**:

```
line = new Line(25.0, 150.0, 120.0, 185.0);     // startX, startY, endX,
                                                // endY
line.setStroke(new Color("#111111"));

triangle = new Polygon();
// three x, y pairs
triangle.addPoints([185.0, 120.0, 220.0, 190.0, 150.0, 190.0]);
triangle.setFill(new Color("#e63946"));
```

```
zigzag = new Polyline();
zigzag.addPoints([270.0, 190.0, 300.0, 125.0, 330.0, 190.0, 360.0, 125.0]);
zigzag.setStroke(new Color("#0ea5e9"));
```

The difference is the ending: a **`Polygon`** joins the last point back to the first into a closed, fillable shape (the triangle), while a **`Polyline`** leaves the path open (the zig-zag). Since an open polyline has no real inside, give it a transparent fill (`new Color("#000000", 0.0)`) so only its stroke shows.

SVGPath - Path Data

For anything the ready-made shapes cannot express, **`SVGPath`** draws from a string of path commands, the same mini-language SVG files use. You set it with **`setContent`**:

```
heart = new SVGPath();
heart.setContent("M 120 70 C 120 40 170 40 170 85 C 170 125 120 150 120 180 " +
                 "C 120 150 70 125 70 85 C 70 40 120 40 120 70 Z");
heart.setFill(new Color("#ec4899"));
```

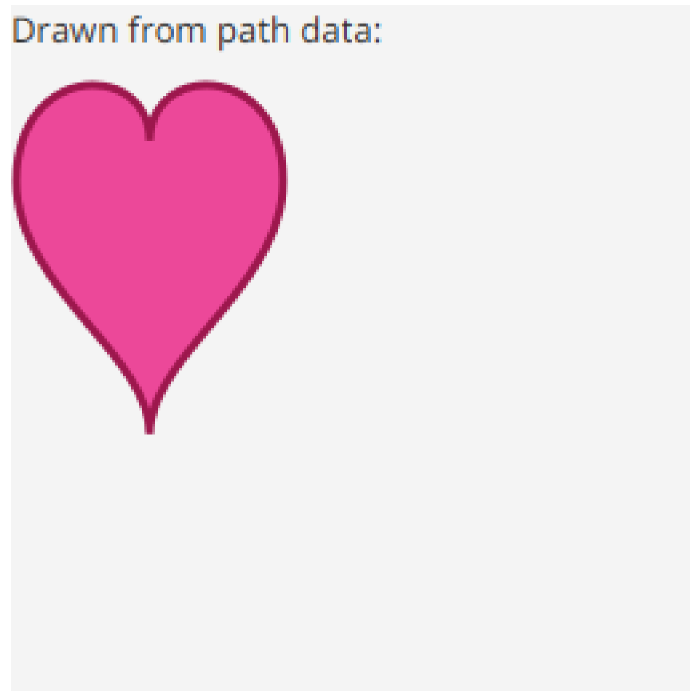


Figure 72: A pink heart with a dark outline, drawn from SVG path data

The letters are the commands: **`M`** moves the pen to a point, **`L`** draws a straight line, **`C`** draws a curve, and **`Z`** closes the shape back to the start. It is the most powerful shape, an icon, a logo, any outline at all, but also the most detailed to write by hand, so it is usually copied from a drawing tool. For a path that overlaps itself, **`setFillRule`** (an `fxfillrule` value: `fxfillrule.EVEN_ODD` or `fxfillrule.NON_ZERO`) decides which regions count as “inside” and get filled.

Recap

- Every shape extends **`Shape`**, so all share **`setFill`**, **`setStroke`**, and **`setStrokeWidth`**, and all are nodes you place by coordinate.
- **`Rectangle`**, **`Circle`**, **`Ellipse`**, and **`Arc`** are defined by their size numbers; a rectangle rounds with `setArcWidth/setArcHeight`.
- **`Line`**, **`Polygon`**, and **`Polyline`** are built from points; a polygon closes and fills, a polyline stays open.
- **`SVGPath`** draws any outline from a string of path commands.

Shapes are permanent nodes the scene keeps track of. In the next chapter we meet the **`Canvas`**, where you paint pixels directly instead.

Chapter 18 - Drawing on a Canvas

The shapes in the last chapter are **nodes**: the scene remembers each one and redraws it for you, so you can move or restyle any shape at any time. That is perfect for a handful of pieces, but not for a drawing made of hundreds of marks, a chart, a game frame, a paint program. For that JavaFX gives you a **`Canvas`**: one node you paint on directly, pixel by pixel.

Every example in this chapter lives here: `18-canvas`.

The Canvas Node

A **`Canvas`** is a node that is simply a rectangle of pixels of a given size:

```
canvas = new Canvas(320.0, 260.0);    // width, height in pixels
```

The difference from shapes is worth pinning down, because it shapes how you think:

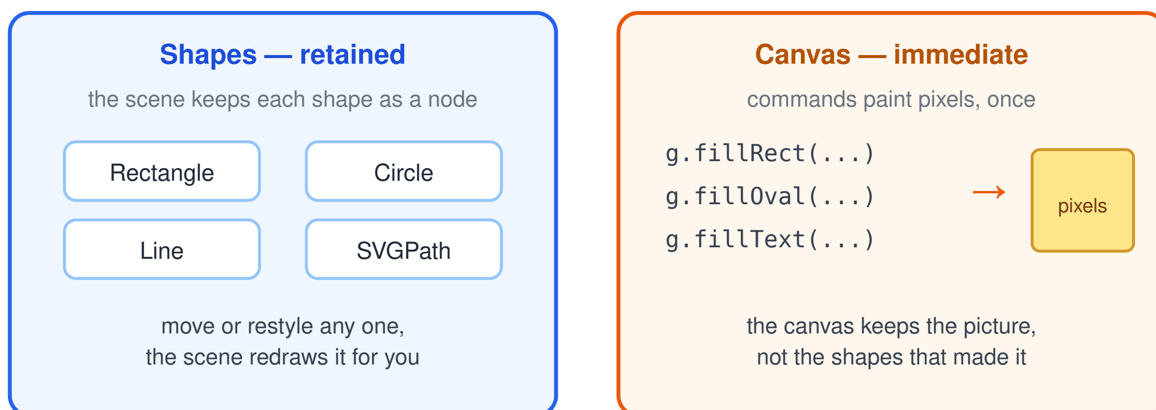


Figure 73: A diagram contrasting retained-mode shapes, where the scene keeps each shape as a node, with immediate-mode canvas drawing, where commands paint pixels the canvas keeps as a picture

- **Shapes are *retained*.** The scene holds each one as an object. Ask a circle to move and the scene repaints it; you never redraw anything yourself.

- **A canvas is *immediate*.** You issue drawing commands that paint pixels **right then**, and the canvas keeps the resulting picture, not the commands. There are no individual shapes afterward to move, just pixels.

Reach for shapes when you have a few things to place and adjust, and a canvas when you have a lot to draw or need to repaint quickly.

Immediate-Mode Drawing with GraphicsContext

You do not paint the canvas directly; you ask it for its **`GraphicsContext`**, the "pen", and give the pen commands:

```
g = canvas.getGraphicsContext2D();
```

Drawing has two halves. First you set the pen's **state**, its fill color, stroke color, line width, or font, and then you draw; the state sticks until you change it. Commands paint in order, so later ones cover earlier ones (draw the sky, *then* the house on top):

```
g.setFill(new Color("#bfdbfe"));
g.fillRect(0.0, 0.0, 320.0, 180.0);       // sky

g.setFill(new Color("#fde047"));
g.fillOval(238.0, 20.0, 60.0, 60.0);      // sun

g.setFill(new Color("#b91c1c"));          // roof: a triangle path
g.beginPath();
g.moveTo(70.0, 120.0);
g.lineTo(140.0, 72.0);
g.lineTo(210.0, 120.0);
g.closePath();
g.fill();

g.setFill(new Color("#1f2937"));
g.setFont(Font.font("System", fxweight.BOLD, fxposture.REGULAR, 16.0));
g.fillText("Home", 232.0, 232.0);         // text
```

The `GraphicsContext` has a command for every kind of mark:

- **`fillRect(x, y, w, h)`** / **`strokeRect(x, y, w, h)`** draw rectangles, and **`fillOval(x, y, w, h)`** / **`strokeOval(x, y, w, h)`** draw ovals that fill that box; **`strokeLine(x1, y1, x2, y2)`** draws a straight line.
- **`beginPath`**, **`moveTo(x, y)`**, **`lineTo(x, y)`**, **`closePath`**, then **`fill`** or **`stroke`**, draw any custom outline (the roof above).
- **`fillText(text, x, y)`** / **`strokeText(text, x, y)`** draw text, using the font from **`setFont`**.
- **`drawImage(image, x, y)`** paints a picture (from Chapter 19) onto the canvas.
- **`clearRect(x, y, w, h)`** erases a region back to transparent, the key to redrawing a canvas or animating it.

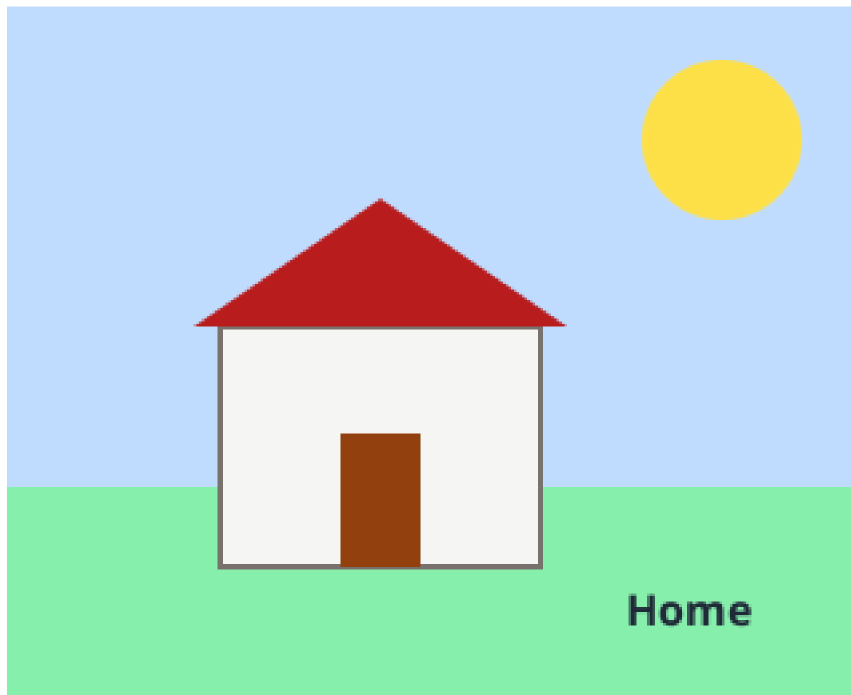


Figure 74: A canvas drawing of a house with a red roof and a door, a yellow sun, blue sky, green grass, and the word Home

The pen's state has a few more knobs beyond fill, stroke, line width, and font: **`setLineCap`** / **`setLineJoin`** (an `fxlinecap` / `fxlinejoin` value) shape a stroke's ends and corners, **`setGlobalAlpha`** (`0.0`-`1.0`) makes everything you draw partly transparent, and **`setTextAlign`** (an `fxtextalign` value) sets how `fillText` lines up around its point.

Recap

- A **`Canvas`** is one node that holds a rectangle of pixels you paint yourself.
- Shapes are **retained** (the scene keeps and redraws them); a canvas is **immediate** (you paint pixels the canvas then keeps as a flat picture).
- Get the **`GraphicsContext`** with **`getGraphicsContext2D`**, set its state (**`setFill`**, **`setStroke`**, **`setLineWidth`**, **`setFont`**), then draw with commands like **`fillRect`**, **`fillOval`**, paths, and **`fillText`**.

We have drawn shapes and painted pixels. The next chapter brings in ready-made pictures: loading and displaying **images**.

Chapter 19 - Images

Pictures have slipped into the book already, the folder and file icons on the tree in Chapter 11 were images. This short chapter tells the full story, which comes in two clean halves: an **`Image`** holds a picture's data, and an **`ImageView`** is the node that actually shows it on screen.

Every example in this chapter lives here: `19-images`.

Loading an Image

An **`Image`** loads a picture, a PNG, JPG, GIF, and holds its pixels in memory. You create one from a file path:

```
photo = new Image("photo.png");
```

The important thing to understand is that an `Image` **does not appear on screen by itself**, it is just the data. You load it once and hold onto it; showing it is the next step's job. A few useful things it can tell you:

- **`getWidth`** and **`getHeight`** report the picture's size in pixels.
- **`isError`** reports whether loading failed (a missing or unreadable file), so you can check before using it.
- A path that starts with `http://` or `https://` loads the picture from the web instead of a file.

Displaying with ImageView

An **`ImageView`** is the node that puts an `Image` on screen. You hand it the image and add it to a layout like any other node:

```
full = new ImageView(photo);      // shows photo at its natural size
```

By default it shows the picture at its natural size, but you can resize it. Set a target width or height, and turn on **`setPreserveRatio`** so the picture scales evenly instead of stretching:

```
thumb = new ImageView(photo);
thumb.setFitWidth(120.0);
thumb.setPreserveRatio(true);     // keep the proportions
```

Notice that both views share the **one** `photo`. An `Image` holds the data; an `ImageView` is just a window onto it, so you can point several views at the same image and size each differently (each

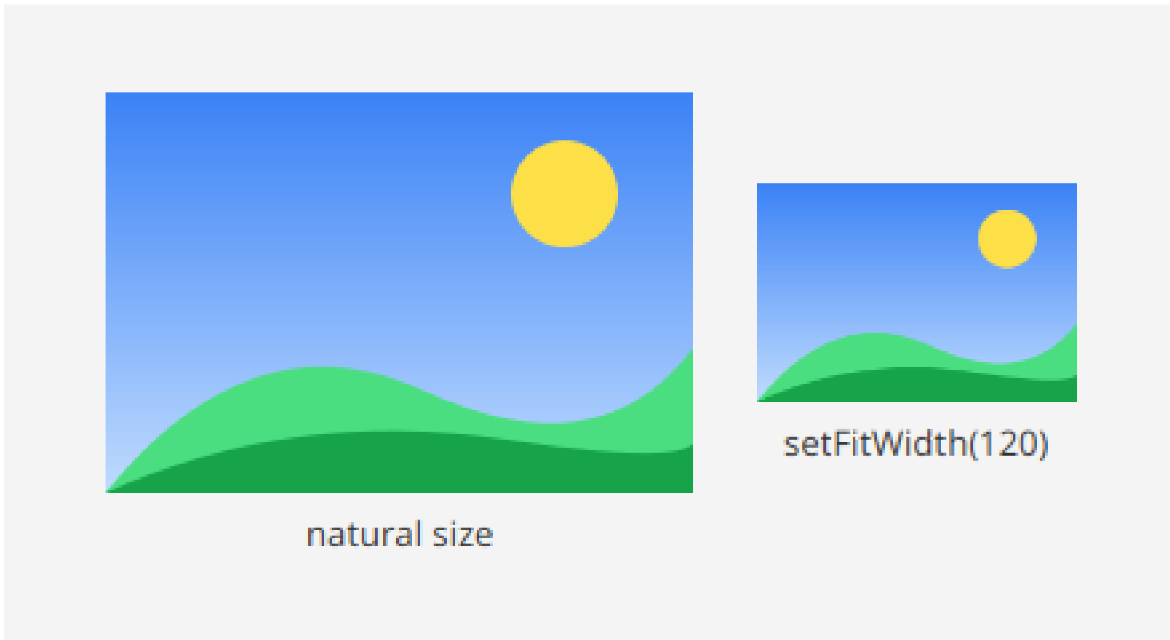


Figure 75: One landscape photo shown twice: at its natural size on the left, and scaled down with setFitWidth on the right

node needs its own `ImageView`, as the tree icons did):

An `ImageView` has a few more knobs:

- **`setFitWidth`** and **`setFitHeight`** set the display size in pixels; with **`setPreserveRatio(true)`** on, setting just one scales the other to match.
- **`setSmooth(true)`** trades a little speed for smoother-looking scaling.
- **`setViewport`** shows only a sub-rectangle of the image, handy for sprite sheets or cropping.

(To paint an image straight onto a `Canvas` instead of placing a node, use the `GraphicsContext`'s **`drawImage`** from Chapter 18.)

Recap

- An **`Image`** loads and holds a picture's data with **`new Image(path)`**; on its own it is data, not something on screen.
- An **`ImageView`** is the node that displays an `Image`; size it with **`setFitWidth`** / **`setFitHeight`** and **`setPreserveRatio`**.
- One `Image` can feed **many** `ImageViews`, each its own size.

That rounds out drawing and pictures. Next we add motion and polish: the visual **effects** and **animation** that bring an interface to life.

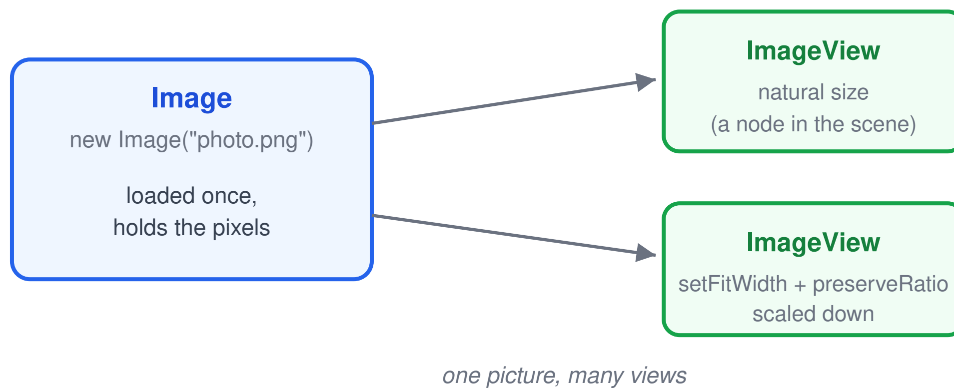


Figure 76: A diagram of one Image feeding two ImageView nodes, one at natural size and one scaled down with setFitWidth and preserveRatio

Chapter 20 - Audio and Video

JavaFX can play sound and video right inside your window. It takes three pieces working together, and once you see how they fit, both audio and video follow the same pattern.

This chapter stays practical rather than exhaustive, media is a big topic, but by the end you can load a clip, show it, and drive it with real controls.

Every example in this chapter lives here: `20-audio-video`.

Media, MediaPlayer, and MediaView

The three pieces each have one job:

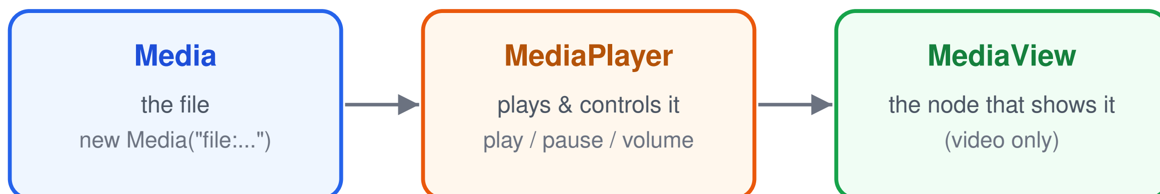


Figure 77: A diagram of the media pipeline: a Media (the file) feeds a MediaPlayer (plays and controls it) which feeds a MediaView (the node that shows the video)

- A **`Media`** is the clip itself, the file. You create it from a **URI**, not a plain path, so a local file needs a `file:` in front, and the helper `file.getAbsPath` turns a relative path into the absolute one the URI needs. Once loaded, it can report its **`getDuration`**, its size (**`getWidth`** / **`getHeight`** for video), and whether it failed with **`isError`**.
- A **`MediaPlayer`** wraps a `Media` and actually plays it, this is the object you press play, pause, and set the volume on.
- A **`MediaView`** is the **node** that shows the video on screen, and you size it like an `ImageView` with **`setFitWidth`** / **`setFitHeight`** and **`setPreserveRatio`**. Audio needs only the first two pieces; video adds a `MediaView`.

```
media  = new Media("file:" + file.getAbsPath("media/sample.mp4"));
player = new MediaPlayer(media);
player.setAutoPlay(true);        // start as soon as it is ready

view = new MediaView(player);    // the node that displays the video
view.setFitWidth(360.0);
```

One important detail: like the WebView in the next chapter, the media pieces must be created on the **UI thread**, so the example builds them inside `fx.runLater` rather than directly in `main`.

Figure 78: A media player window: a video frame reading “Aussom Media”, Play/Pause/Stop buttons, a volume slider, and a “Playing” status

Playing Audio

Audio is the same process, only simpler. Sound has nothing to show, so you drop the **`MediaView`** entirely and keep just the `Media` and the `MediaPlayer`. And since the `MediaView` was the piece that required the UI thread, plain audio can be built right in `main`, with no `fx.runLater`:

```
media  = new Media("file:" + file.getAbsPath("media/sample.mp3"));
player = new MediaPlayer(media);
player.setAutoPlay(true);
// no MediaView; the same play, pause, stop, and volume controls drive it
```

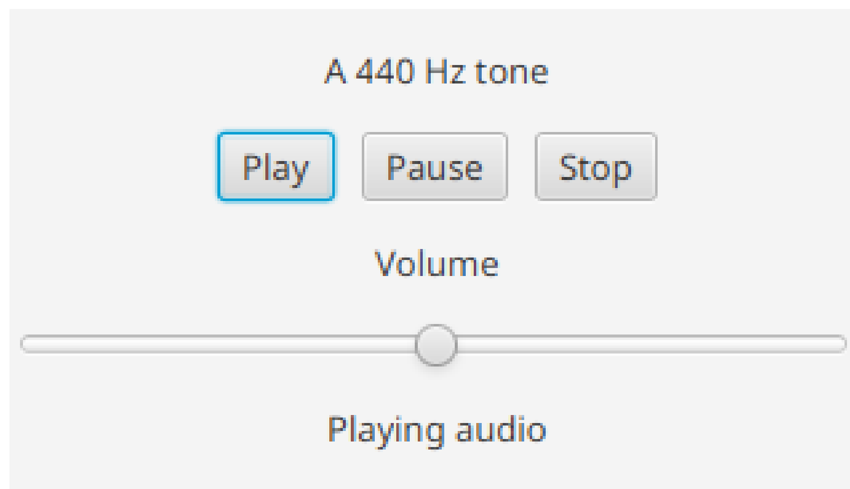


Figure 79: An audio player window with Play, Pause, and Stop buttons, a volume slider, and a "Playing audio" status, and no video area

Everything in the next section is identical whether you are playing a song or a movie, only the video case needs a `MediaView` to see it.

Playback Controls

Everything you would expect of a player lives on the **`MediaPlayer`**:

- **`play`**, **`pause`**, and **`stop`**, pause keeps the current position, while stop rewinds to the beginning.
- **`seek(ms)`** jumps to a moment, given in milliseconds.
- **`setVolume`** (from `0.0` to `1.0`) and **`setMute`** control the sound.
- **`setAutoPlay`** starts the clip as soon as it is ready, and **`setCycleCount`** repeats it (`-1` loops forever).
- **`setOnReady`** and **`setOnEndOfMedia`** run a method of yours when the clip is ready to play or reaches the end.
- **`getStatus`** reports what the player is doing right now (playing, paused, and so on).

That is the whole shape of it: a `Media` to load, a `MediaPlayer` to run it, a `MediaView` if there is a picture, and these controls to drive it. The next chapter embeds something even larger in your window, a whole web browser.

Chapter 21 - Embedding Web Content

Sometimes the easiest way to show something, a help page, a chart, a login form, already exists as a web page. JavaFX lets you drop a **whole web browser** into your window with a **`WebView`**. It renders HTML and CSS, runs JavaScript, and, most usefully, can trade information back and forth with your Aussom code.

This chapter covers the essentials, not the whole browser. We will load a page, run scripts, pass data between Aussom and JavaScript, and lock things down safely.

Every example in this chapter lives here: `21-web-content`.

WebView and WebEngine

Two objects make up the browser. A **`WebView`** is the **node** you place in the window, the visible browser. Behind it sits a **`WebEngine`**, the part that actually loads pages and runs scripts. You reach the engine with `getEngine`:

```
webView = new WebView();
engine  = webView.getEngine();
```

One rule to remember: a `WebView` must be created on the **UI thread**. Building one directly in `main` fails, so the examples create it inside `fx.runLater`, which runs its work on the UI thread for you.

The `WebView` node has a few display settings of its own: **`setZoom`** (a factor, `1.0` is normal and `1.5` zooms everything in), **`setFontScale`** (scales just the text, for accessibility), and **`setPrefSize(width, height)`** (its preferred size in the layout).

Loading Pages and Running Scripts

The engine loads content two ways. **`load`** takes a web address, and **`loadContent`** takes a string of HTML you build yourself:

```
engine.load("https://example.com");              // a live web page
engine.loadContent("<h1>Hello from the Web</h1>");   // your own HTML
```

Our example builds a small styled page with `loadContent`, everything you see below, the heading, the card, the button, is HTML and CSS rendered by the `WebView`:

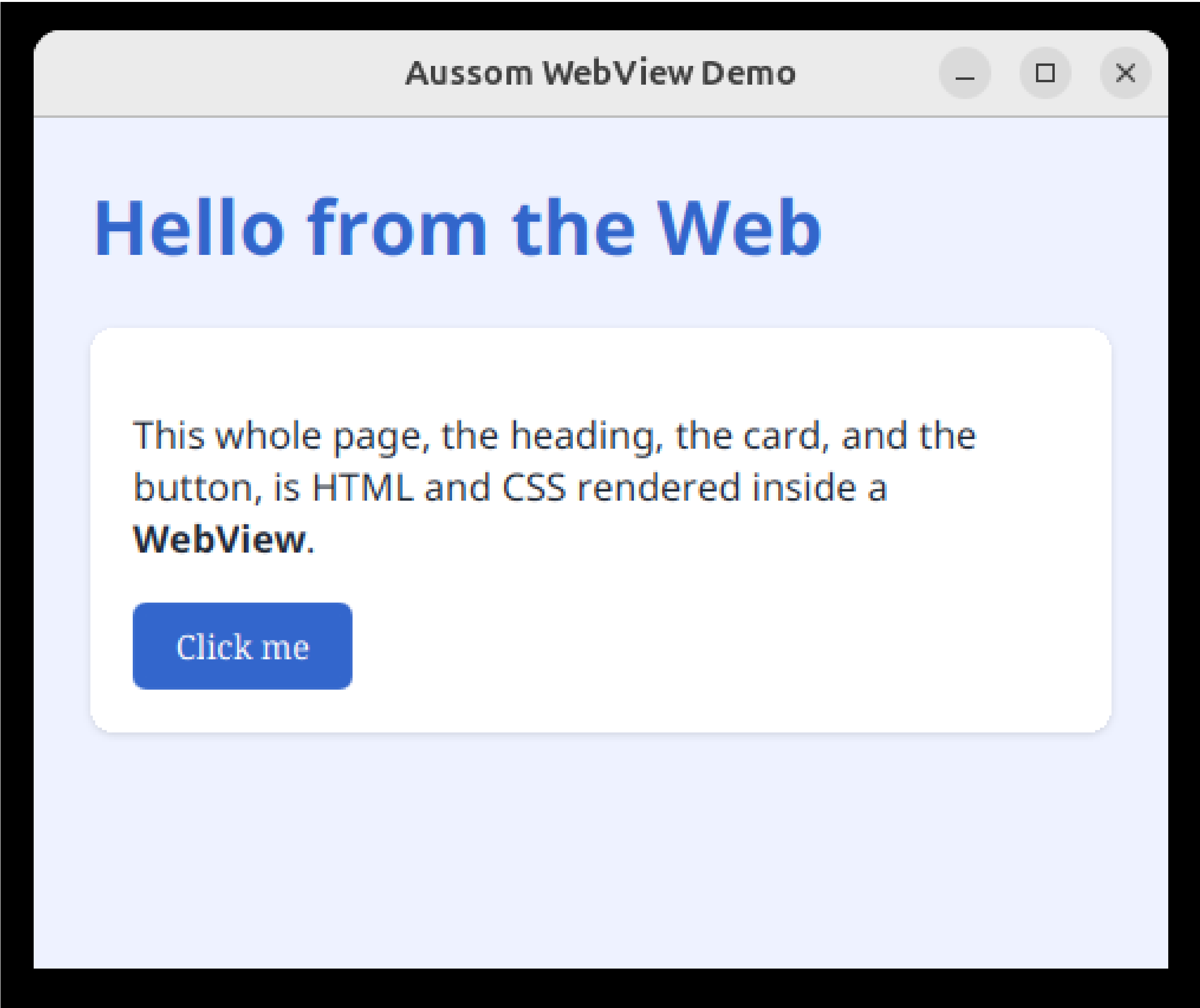


Figure 80: A window showing an HTML page rendered by a WebView: a blue “Hello from the Web” heading, a white card with text, and a blue “Click me” button

To run JavaScript on the loaded page, call **`executeScript`**. It runs the script and hands back the result:

```
sum = engine.executeScript("3 + 4");      // sum is 7
```

Talking to JavaScript

This is where a `WebView` becomes more than a picture: Aussom and the page's JavaScript pass data both ways, and both directions go through `executeScript`. The `interop.aus` example puts both to work, click its button and Aussom reads the name you typed, then writes a greeting back into the page.

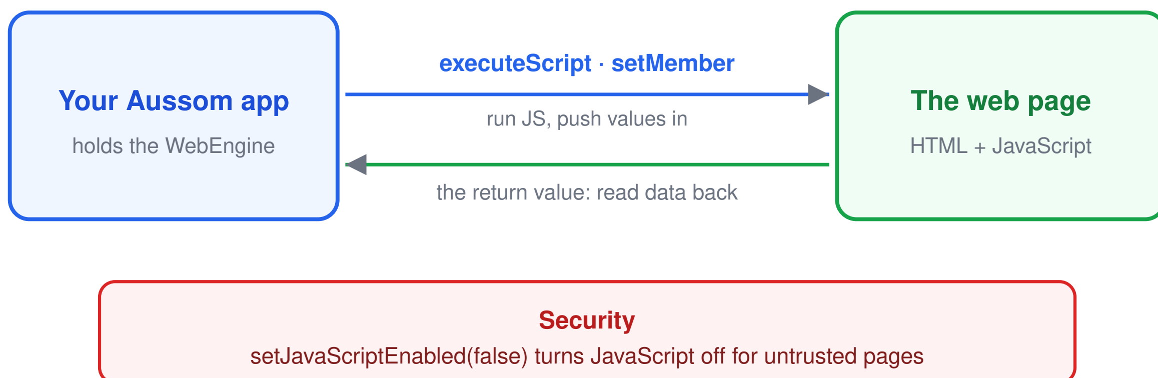


Figure 81: A diagram: the Aussom app and the web page exchange data through the WebEngine; executeScript and setMember run JS and push values in, and the return value reads data back; a note that setJavaScriptEnabled(false) turns JS off for untrusted pages

Aussom to JavaScript

`executeScript` runs a snippet of JavaScript and hands back whatever it evaluates to, with the common types converted for you:

- a JavaScript **number** becomes an Aussom `int` or `double`,
- a **string** becomes a `string`, a **boolean** becomes a `bool`, and JavaScript **null** becomes `null`.

To send a value the *other* way, into the page, set it on the page's **`window`** object. `executeScript("window")` returns that object, and **`setMember`** puts a named value on it that the page's own scripts can then read:

```
window = engine.executeScript("window");
window.invoke("setMember", "who", "Ada");
engine.executeScript("document.body.innerHTML = 'Hello ' + who");
```

JavaScript to Aussom

The page cannot call your Aussom code directly (see the limits below), so instead Aussom **reads** from the page whenever it needs to, a form field, a variable, or the result of a function:

```
// read what the user typed into an <input id="name">
name = engine.executeScript("document.getElementById('name').value");
```

That is the whole "JavaScript to Aussom" story: run a script that produces the value you want, and it comes straight back to Aussom, usually when a button in *your* app is clicked.

What converts, and the limits

A few things are worth knowing before you lean on this:

- **Objects and arrays do not convert.** A JavaScript array or object comes back as a `JSObject` (an `AussomJavaObject`), not an Aussom list or map. Read into it with `.invoke("getSlot", index)` for an array element and `.invoke("getMember", name)` for an object's property, or simply have the script return the one value you need.
- **JavaScript cannot call Aussom methods.** You *can* hand the page an Aussom object with `setMember`, but its methods are not callable from JavaScript, so there is no true "the page drives the app" callback. Have Aussom read from the page instead.
- **Scripts run on the UI thread.** `executeScript` is synchronous and runs on the UI thread, so a long-running script freezes the window, keep them quick.

Security

A `WebView` runs **real** web content, real HTML, real JavaScript, so treat a page you did not write with the same caution as any website:

- **`setJavaScriptEnabled(false)`** turns JavaScript off entirely. If you only need to display trusted, static content, this is the safest setting.
- **`setOnAlert`**, **`setConfirmHandler`**, and **`setPromptHandler`** let you catch the pop-up dialogs a page's JavaScript tries to open (`alert`, `confirm`, `prompt`), so they appear the way you want, or not at all.
- Prefer loading content you **control**. Anything you push into the page with `setMember` can be read by every script on it, so never hand a page you do not trust a password, token, or other secret.

Recap

- A **`WebView`** is a browser node; its **`WebEngine`** (from `getEngine`) loads pages with **`load`** or **`loadContent`** and runs scripts with **`executeScript`**. Create a `WebView` on the UI thread.
- Aussom and the page exchange data through the engine: `executeScript` runs JS and returns results (numbers, strings, booleans; objects come back as a `JSObject`), **`setMember`** on the page's `window` passes values in, and Aussom **reads** from the page for the other direction, JavaScript cannot call Aussom methods.

- **`setJavaScriptEnabled`** and the dialog handlers keep untrusted content in check, load pages you trust.

This closes Part V. Next we add the finishing polish: visual **effects** and **animation**.

Part VI

Visual Effects and Animation

Chapter 22 - Effects

We reach **Part VI, Visual Effects and Animation**, the polish that makes an interface feel alive. This chapter is about **effects**: an effect is a small object you attach to a node that changes how it is **drawn**, a blur, a shadow, a glow, without changing the node itself. Turn the effect off and the node looks exactly as it did before.

Every example in this chapter lives here: `22-effects`.

Applying an Effect

Every node has a **`setEffect`** method. You build the effect you want, set its options, and hand it to the node:

```
blur = new GaussianBlur();
blur.setRadius(6.0);
card.setEffect(blur);   // attach the effect to the node
```

That is the whole pattern, and it is the same for every effect below. Here is the same gradient card with each one applied:

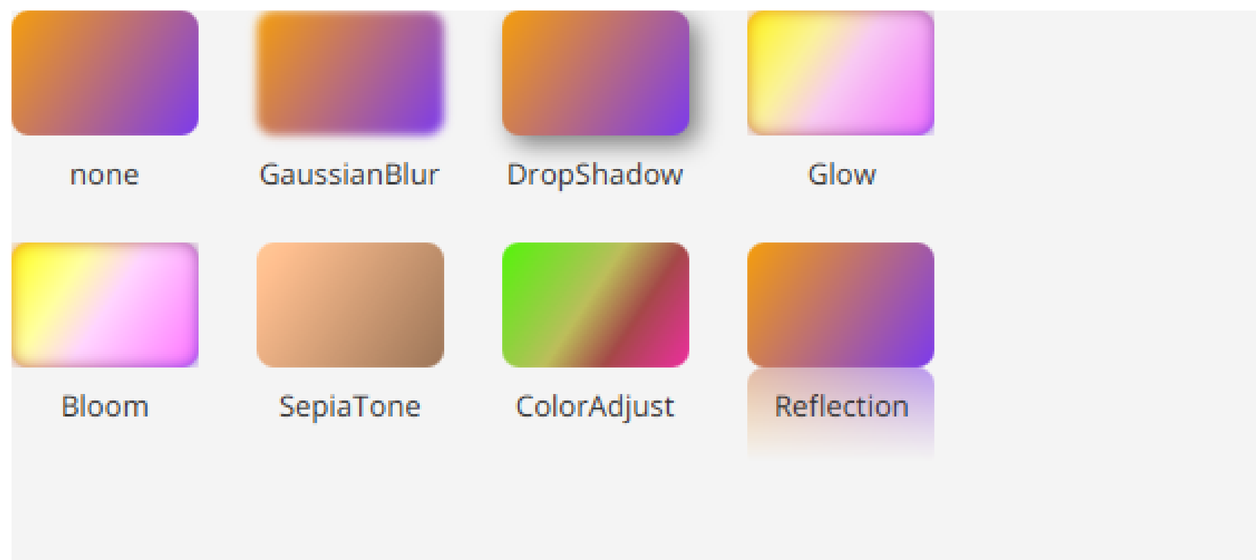


Figure 82: A grid of the same card shown with no effect, GaussianBlur, DropShadow, Glow, Bloom, SepiaTone, ColorAdjust, and Reflection

Blurs

Blurs soften a node. There are two. **`GaussianBlur`** is the smooth, natural one, and its single setting is the radius, the softness, in pixels:

```
blur = new GaussianBlur();
blur.setRadius(6.0);      // 0 to about 63; bigger is blurrier
```

`BoxBlur` is a lighter, boxier blur. You set its blur size in each direction and how many passes to make (more passes look smoother):

```
box = new BoxBlur();
box.setWidth(8.0);        // blur width in pixels
box.setHeight(8.0);       // blur height in pixels
box.setIterations(3);     // 1 to 3; more is smoother
```

Shadows

Shadows give a node depth. A **`DropShadow`** casts a shadow *behind* the node, here soft, half-transparent black, thrown down and to the right, which is exactly the tile in the gallery:

```
shadow = new DropShadow();
shadow.setRadius(14.0);                       // softness of the shadow,
                                              // in pixels
shadow.setColor(new Color("#000000", 0.5));  // color (here 50%-opacity
                                              // black)
shadow.setOffsetX(4.0);                       // how far right it falls
shadow.setOffsetY(4.0);                       // how far down it falls
```

It also has `setSpread` (`0.0`-`1.0`, how solid the shadow's core is) and `setBlurType` for the blur algorithm, but the four above cover almost every use.

An **`InnerShadow`** is the same idea turned inward, a shadow *inside* the shape for a carved-in or pressed look, with the same radius, color, and offset settings (plus `setChoke`, the inner cousin of spread):

```
inner = new InnerShadow();
inner.setRadius(10.0);
inner.setColor(new Color("#000000", 0.6));
```

Glow, Bloom, Reflection, SepiaTone, and ColorAdjust

The rest change light and color. The first three are one-setting effects, and each value runs from `0.0` to `1.0`:

```
glow = new Glow();
glow.setLevel(0.9);         // how strongly it glows
```

```
bloom = new Bloom();
bloom.setThreshold(0.2);    // how bright a pixel must be before it
                            // blooms

reflect = new Reflection(); // a fading mirror image below the node
                            // (default look)
```

`Reflection` also has `setFraction` (how much of the node reflects) and `setTopOpacity` / `setBottomOpacity` to fade the mirror, if you want to shape it.

`SepiaTone` gives an old-photograph brown wash, again a single `setLevel` from `0.0` to `1.0`:

```
sepia = new SepiaTone();
sepia.setLevel(0.8);
```

`ColorAdjust` recolors the whole node. Each of its four settings runs from `-1.0` to `1.0`, where `0.0` leaves that aspect unchanged:

```
adjust = new ColorAdjust();
adjust.setHue(0.35);        // shift the hue
adjust.setSaturation(0.2);  // more or less vivid
adjust.setBrightness(0.0);  // lighter or darker
adjust.setContrast(0.0);    // more or less contrast
```

Chaining Effects on a Node

A node can wear only one effect at a time, but effects **chain**: each has a **`setInput`**, and whatever you set as the input is drawn *first*, then this effect is applied on top. So to both glow *and* cast a shadow, feed a `Glow` into a `DropShadow`:

```
glow = new Glow();
glow.setLevel(0.9);

shadow = new DropShadow();
shadow.setRadius(14.0);
shadow.setInput(glow);    // glow first, then the shadow
card.setEffect(shadow);   // attach the end of the chain to the node
```

Notice the two ends of the chain: **`setInput`** links one effect to the next, and **`setEffect`** attaches the last one to the node. Chain as many as you like.

Recap

- An **effect** attaches to any node with **`setEffect`** and changes how it is drawn, not what it is.
- **Blurs** (`GaussianBlur`, `BoxBlur`) soften; **shadows** (`DropShadow`, `InnerShadow`) add depth.
- **`Glow`**, **`Bloom`**, **`Reflection`**, **`SepiaTone`**, and **`ColorAdjust`** change light and color.
- Effects **chain** with **`setInput`**, drawn input-first, so you can combine several on one node.

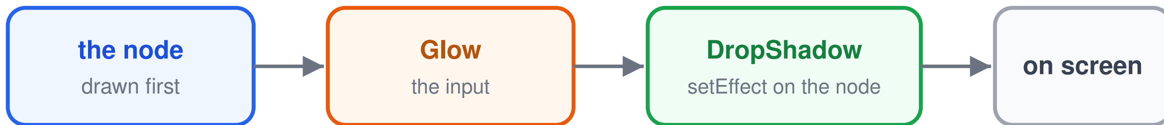


Figure 83: A diagram: the node is drawn, then Glow (the input), then DropShadow (setEffect), then to screen

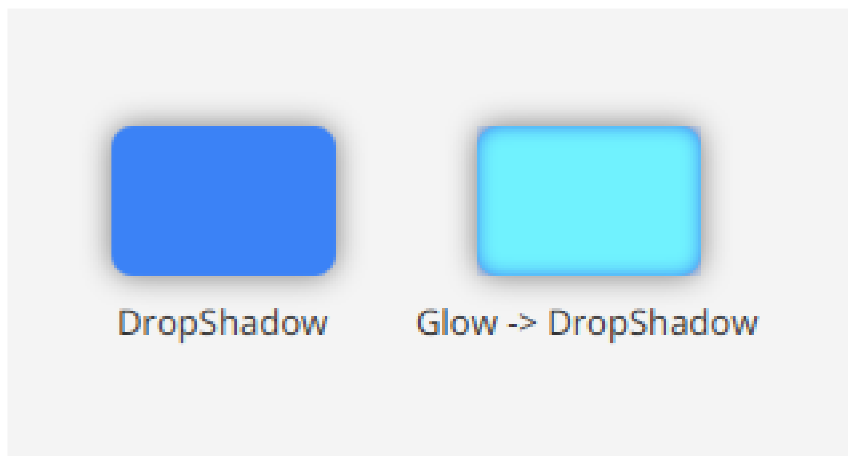


Figure 84: Two blue cards: one with just a DropShadow, and one with a Glow chained into a Drop-Shadow, so it both glows and casts a shadow

Effects change how a node looks in place. The next chapter moves it, with **transforms**.

Chapter 23 - Transforms

Effects changed how a node *looks*; a **transform** changes where and how it *sits*, moving it, turning it, or resizing it. The neat part is that a transform is purely visual: it shifts the picture without changing the node's real position or size in the layout, so the space it occupies stays put while the node itself slides, spins, or grows.

Every example in this chapter lives here: `23-transforms`.

The Transform Base

Every node has quick methods for the three transforms you reach for most:

- **`setTranslateX`** and **`setTranslateY`** move the node by a number of pixels.
- **`setRotate`** turns it, in degrees, clockwise around its center.
- **`setScaleX`** and **`setScaleY`** resize it, `1.0` is normal size, above grows, below shrinks.

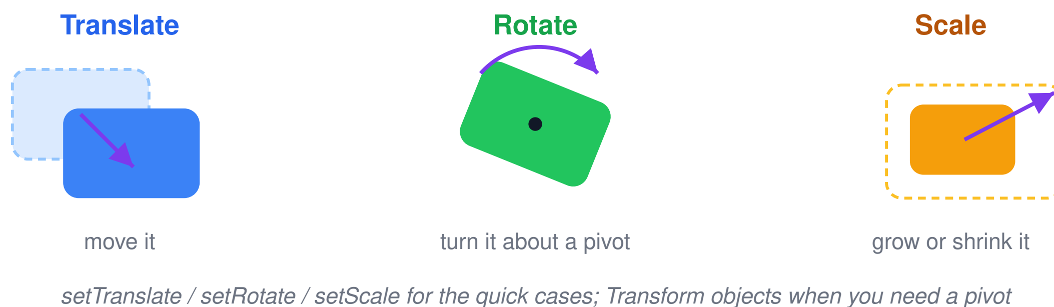


Figure 85: A diagram of the three transforms: translate moves a node, rotate turns it about a pivot, and scale grows or shrinks it

```
moved = this.card();
moved.setTranslateX(18.0);      // shift right and down
moved.setTranslateY(12.0);

turned = this.card();
turned.setRotate(25.0);         // turn 25 degrees
```

```
grown = this.card();
grown.setScaleX(1.3);          // 1.3x the size
grown.setScaleY(1.3);
```

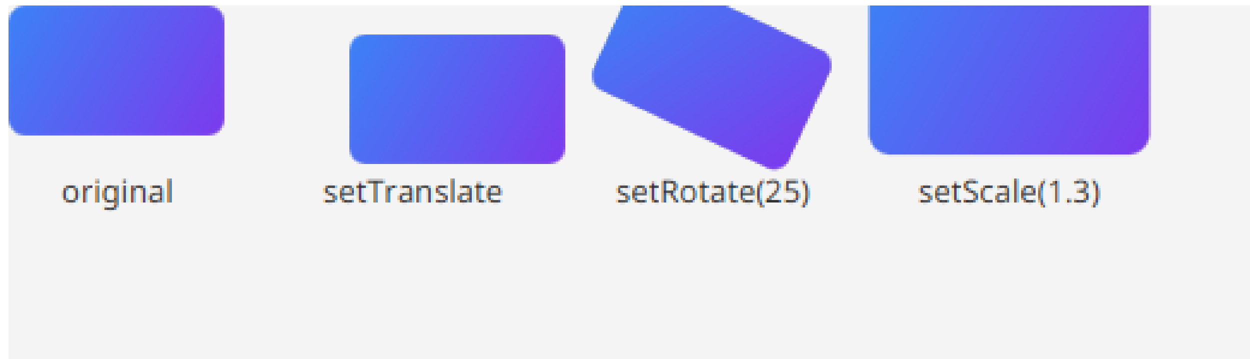


Figure 86: The same card four times: original, translated, rotated 25 degrees, and scaled to 1.3x

These methods cover the everyday cases, and they are all you need most of the time.

Translate, Rotate, Scale, and Affine

Underneath, each of those is a **`Transform`** object, and you can use those objects directly when you need more control. They give you two things the quick methods do not.

First, a **pivot**. `setRotate` and `setScale` always work around the node's center, but a `Rotate` object lets you choose the point to turn around:

```
// turn 45 degrees about the node's top-left corner (0, 0)
spin = new Rotate(45.0, 0.0, 0.0);
node.getTransforms().invoke("add", spin.obj);
```

Second, you can **stack** several. A node has a list of transforms (its `getTransforms`), and they combine in order, so you can rotate *and* scale *and* shift with separate, reusable objects. The full set is:

- **`Translate(x, y)`**, **`Rotate(angle, pivotX, pivotY)`**, and **`Scale(x, y, pivotX, pivotY)`**, the object forms of the three quick methods, where the pivot is the point rotation or scaling turns about.
- **`Shear`** slants a node, pushing its top and bottom in opposite directions.
- **`Affine`** is the raw math behind all of them, a matrix you can set directly when you need a transform the others cannot express (an advanced tool).

Recap

- A **transform** changes how a node is drawn, moved, turned, or resized, without changing the space it takes in the layout.

- The quick methods, **`setTranslateX`**/**`setTranslateY`**, **`setRotate`**, and **`setScaleX`**/**`setScaleY`**, handle the common cases.
- **`Transform`** objects (**`Translate`**, **`Rotate`**, **`Scale`**, **`Shear`**, **`Affine`**) add a **pivot** and can be **stacked** through a node's `getTransforms` list.

A transform sets a node to a new position or size all at once. In the next chapter, we make that change happen *gradually*, with **animation**.

Chapter 24 - Animation

A transform sets a node to a new position or size all at once. **Animation** makes that change happen **gradually**, over time, so a panel slides in, a button fades, an icon spins. This is the chapter that makes an interface feel smooth and alive.

There are two tools. A **`Timeline`** runs *your* code at set moments, and a **`Transition`** animates one of a node's properties *for* you. We will use both.

Every example in this chapter lives here: `24-animation`.

Keyframes and the Timeline

A **`Timeline`** is a schedule. You add **keyframes**, each a time paired with a callback, and when you play the timeline, your callback runs at that time. With a cycle count it repeats:

```
timeline = new Timeline();
timeline.addKeyFrame(1000.0, ::onTick);   // run onTick every 1000 ms
timeline.setCycleCount(-1);               // repeat forever (-1)
timeline.play();
```

Use a `Timeline` when *you* want to change things on a beat, a clock that ticks each second, a game loop, stepping a drawing frame by frame. For the everyday case of smoothly changing a single property, though, the transitions below do the work for you.

Transitions: Fade, Translate, Rotate, Scale

A **`Transition`** animates **one property** of a node smoothly from a start value to an end value over a span of time. You build each one with its **duration in milliseconds** (`new FadeTransition(1500.0)`), then give it a node and its from/to values, and play it. There is one for each common property:

- **`FadeTransition`** changes opacity, **`setFromValue`** / **`setToValue`** from `0.0` (invisible) to `1.0` (solid).
- **`TranslateTransition`** moves it, **`setFromX`** / **`setToX`** (and the Y pair) in pixels.
- **`RotateTransition`** turns it, **`setFromAngle`** / **`setToAngle`** in degrees.
- **`ScaleTransition`** resizes it, **`setFromX`** / **`setToX`** (and the Y pair) as size factors, where `1.0` is normal size.

```
move = new TranslateTransition(1500.0);   // over 1.5 seconds
move.setNode(card);
```

```
move.setFromX(0.0);
move.setToX(220.0);
move.play();
```

Figure 87: A filmstrip of three frames: a card that slides left to right while fading from solid to faint

Every animation, transitions and timelines alike, shares a set of controls from their common base:

- **`play`**, **`pause`**, and **`stop`** drive it.
- **`setCycleCount`** repeats it (`-1` loops forever), and **`setAutoReverse`** makes each repeat run backward.
- **`setDelay`** waits before starting, and **`setOnFinished`** runs a method of yours when it ends.

Composing with ParallelTransition and SequentialTransition

Single transitions are the pieces; you combine them with two containers:

- A **`ParallelTransition`** runs its children **all at once** (the example fades *and* moves the card together).
- A **`SequentialTransition`** runs them **one after another**, in order.

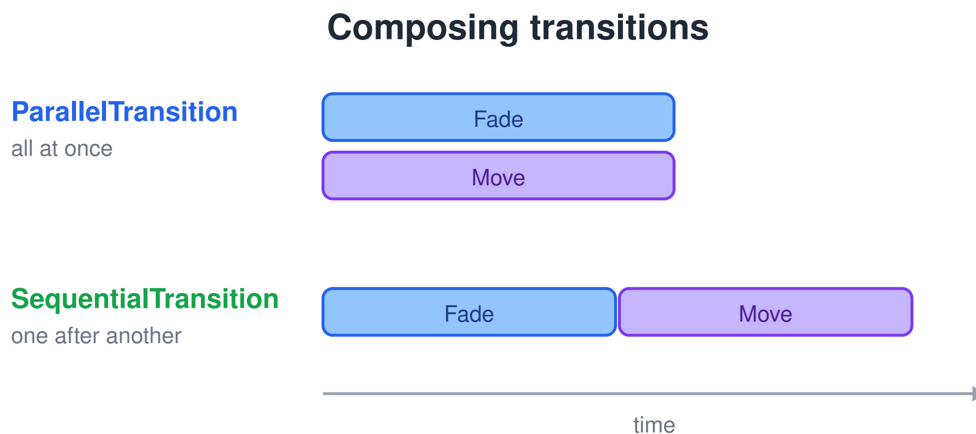


Figure 88: A diagram: a ParallelTransition stacks Fade and Move over the same time, while a SequentialTransition places Fade then Move back to back along a time line

```
anim = new ParallelTransition();
anim.add([move, fade]);        // both play together
anim.setCycleCount(-1);
```

```
anim.setAutoReverse(true);
anim.play();
```

Both take a list of animations with add and then behave like one animation you can play, loop, and reverse.

PauseTransition and Timing

A **`PauseTransition`** is an animation that does nothing but **wait** for a set time. On its own it is useful with `setOnFinished` to run something after a delay, and inside a `SequentialTransition` it becomes a pause between steps:

```
steps = new SequentialTransition();
steps.add([fadeIn, new PauseTransition(800.0), fadeOut]);   // in, hold,
                                                            // out
steps.play();
```

All timing is in **milliseconds**, so `1000.0` is one second. Between that, `setDelay`, `setCycleCount`, and a `PauseTransition`, you can arrange animations to happen exactly when you want.

Recap

- A **`Timeline`** runs your callbacks at set **keyframe** times; a **`Transition`** smoothly animates one property of a node for you.
- The four transitions are **`FadeTransition`**, **`TranslateTransition`**, **`RotateTransition`**, and **`ScaleTransition`**; all share **`play`**, **`setCycleCount`**, **`setAutoReverse`**, and friends.
- **`ParallelTransition`** runs animations together, **`SequentialTransition`** runs them in turn, and **`PauseTransition`** waits.

This closes Part VI. Next we turn to what powers reactive interfaces underneath: **observable properties** and **binding**.

Part VII

Properties and Reactive UI

Chapter 25 - Observable Properties

We reach **Part VII, Properties and Reactive UI**. So far, when you wanted to react to a change you wired up an **event** (Chapter 8), and when you wanted to update the screen you called a setter like `setText` yourself. JavaFX has a second, quieter model running underneath all of it: the **observable property**. A property is a value that can be *watched*, when it changes, it tells whoever is listening. That idea is the foundation for **binding**, which the next chapter builds on to keep a whole interface in sync with your data.

Every example in this chapter lives here: `25-observable-properties`.

The Property Model

A **property** is a small box that holds a value *and* can announce when that value changes. You read it and write it with plain methods:

```
count = new IntegerProperty(0);   // an int property, starting at 0
count.set(4);                     // write the value
now = count.get();                // read it back: 4
```

There are two pairs of accessors. **`get`** / **`set`** are typed to the property (here they take and return an `int`); **`getValue`** / **`setValue`** are the generic form that every property shares. Use whichever reads better; they do the same thing.

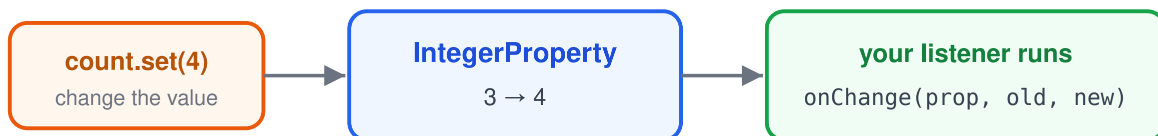


Figure 89: A diagram: count.set(4) changes an IntegerProperty from 3 to 4, which runs your listener onChange(prop, old, new)

The magic is that last part of the diagram: when the value changes, the property **notifies** anyone watching. You never poll it in a loop, it tells you.

Typed Properties

There is a property type for every kind of value, so the value stays correctly typed. Each constructor takes a starting value:

- **`BooleanProperty`** holds a `true/false` (`new BooleanProperty(false)`).
- **`StringProperty`** holds text (`new StringProperty("")`).
- **`IntegerProperty`**, **`LongProperty`**, **`FloatProperty`**, and **`DoubleProperty`** hold numbers (`new DoubleProperty(0.0)`).
- **`NumberProperty`** is the shared base of the four numeric ones, useful when a value could be any number.
- **`ObjectProperty`** holds anything else, a `Color`, a `Date`, one of your own objects (`new ObjectProperty(someValue)`).

They all work the same way, with `get/set` and the listeners below; only the type of value they carry differs.

Listening for Changes

To be told when a property changes, hand it one of your methods with **`addListener`**. JavaFX calls that method every time the value changes, passing three things, the property, its **old** value, and its **new** value:

```
count.addListener(::onCountChange);

public onCountChange(property, oldValue, newValue) {
    this.display.setText("Count: " + newValue);
}
```

In the example, two buttons only ever call `count.set(...)`; they never touch the label. The listener does that, reacting to the change. The property is the single **source of truth**, and the display simply follows it:

A few more things about listeners:

- **`addInvalidationListener`** is a lighter cousin: it fires when the value *becomes* invalid (may have changed) and hands your method only the property, not the old and new values, useful when you just need "something changed" and will read the value yourself.
- **`removeListener`** stops a listener you added earlier.
- A listener runs on whatever thread changed the property. In a UI that is normally an event handler on the UI thread, so updating controls from a listener is fine, the same UI-thread rule from Chapter 8 applies if you ever change a property from a background task.

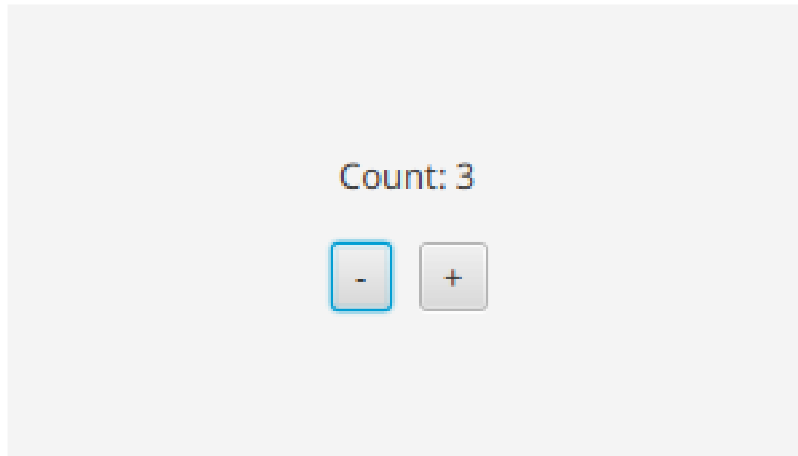


Figure 90: The Observable Properties window showing "Count: 3" above minus and plus buttons

Recap

- A **property** is a value that can be watched; read and write it with **`get`** / **`set`** (or **`getValue`** / **`setValue`**).
- There is a typed property for each kind of value: **`BooleanProperty`**, **`StringProperty`**, the numeric ones (**`IntegerProperty`**, **`LongProperty`**, **`FloatProperty`**, **`DoubleProperty`**, sharing **`NumberProperty`**), and **`ObjectProperty`** for anything else.
- **`addListener`** runs your method on every change, with the property and its old and new values; **`addInvalidationListener`** is the lighter "something changed" version, and **`removeListener`** detaches one.

A property that notifies its listeners is powerful on its own, but its real payoff comes when you let JavaFX wire the notifications for you. That is **binding**, and it is next.

Chapter 26 - Binding the UI to Data

Chapter 25 gave us properties that announce their own changes. **Binding** is the payoff: instead of writing a listener that copies one value to another, you simply *declare* that two values should stay equal, and JavaFX keeps them that way for you. Change one, and the other follows, with no event handler in sight. This is how you tie an interface to your data and let the two stay in step on their own.

Every example in this chapter lives here: `26-binding`.

One-Way and Two-Way Binding

There are two kinds of binding, and the difference is simply which way the values flow.

A **one-way** bind makes one property *follow* another. You call **`bind`** on the follower and pass it the source:

```
copy.bind(source);   // copy now always equals source
```

From then on, `copy` mirrors `source` automatically. While it is bound, `copy` is **read-only**, you cannot set it yourself (it belongs to `source` now); call **`unbind`** to release it. One-way binding is exactly what a *display* wants: a label that shows a value but never changes it.

A **two-way** bind links two properties so **either** can change and the other keeps up. You call **`bindBidirectional`**:

```
first.bindBidirectional(second);   // change either one; both stay equal
```

Now editing `first` updates `second`, and editing `second` updates `first`. Both stay writable, which is what an *editable* control needs. **`unbindBidirectional`** breaks the link.

Keeping Controls in Sync

Controls expose their important values as properties too, through accessors like **`textProperty`** (text controls and labels), **`valueProperty`** (a slider or spinner), **`selectedProperty`** (a check box or toggle), and **`progressProperty`** (a progress bar). Bind those to a property of your own and the control and your data move together.

The example uses one `StringProperty` as the single source of truth, ties **two** text fields to it two-way, and a label to it one-way:

Figure 91: A diagram: one central name StringProperty, two TextFields bound to it two-way, and a Label bound one-way, all showing the same value

```
name = new StringProperty("");     // the one value everything shares

fieldA = new TextField("");
fieldB = new TextField("");
name.bindBidirectional(fieldA.textProperty());   // field <-> value
name.bindBidirectional(fieldB.textProperty());   // field <-> value

mirror = new Label("");
mirror.textProperty().invoke("bind", name.obj);  // label follows the
                                                 // value

name.set("Ada");                   // seed it once; every bound control
                                   // follows
```

Type into either field and watch the other field **and** the label change with it, though no method wires them together. They are all bound to `name`, so they can only ever show the same thing:

Two details are worth pointing out in that code:

- **`name.bindBidirectional(fieldA.textProperty())`** is called on *your* property (the friendly wrapper), and handed the field's property from `textProperty`. This is the clean way to tie a control to your data two-way.
- **`mirror.textProperty().invoke("bind", name.obj)`** goes the other direction: a control's `textProperty` hands back the raw JavaFX property, so to make the *label* follow the value you call its `bind` directly with `.invoke`, passing the data property's underlying object (`name.obj`).

That single seed, `name.set("Ada")`, then flows out to all three controls at once, which is the whole point of binding: **one source of truth, and a UI that keeps itself in sync.**

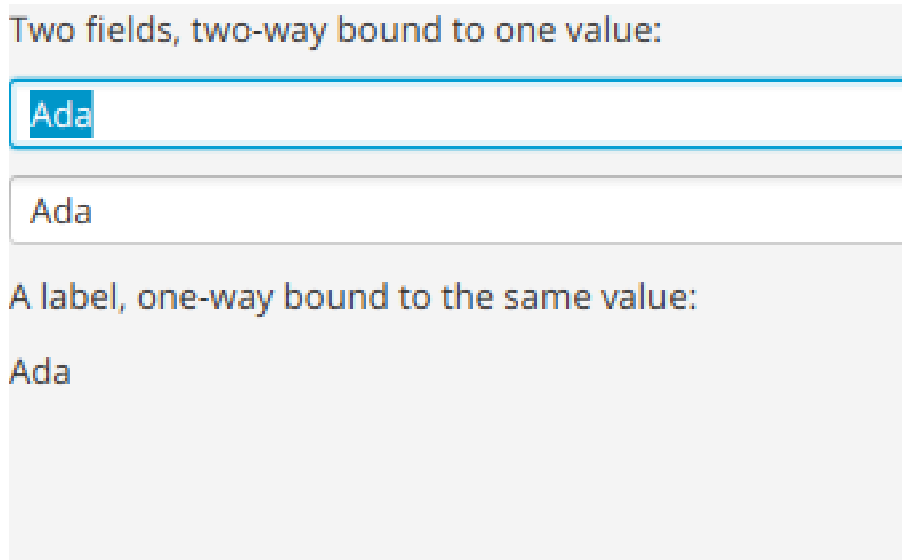


Figure 92: The Binding window: two text fields both reading "Ada" and a label reading "Ada", all bound to one value

More Things You Can Bind

Text fields and labels are only the start. Almost every value a control exposes is a property, so almost anything can be bound. A few common pairings, to give you a feel for what is out there:

- **A check box to a flag** - bind `checkBox.selectedProperty()` two-way to a `BooleanProperty` so the box and your on/off setting always agree.
- **A slider to a number** - bind `slider.valueProperty()` two-way to a `DoubleProperty`, so dragging the slider and setting the value from code both work.
- **A progress bar to a number** - bind `progressBar.progressProperty()` one-way to a `DoubleProperty` (`0.0-1.0`) that a task updates as it runs.
- **A label to any value** - bind `label.textProperty()` one-way to a `StringProperty` to display it, the display half of what we just built.
- **Enable or disable from a flag** - bind a control's `disableProperty()` one-way to a `BooleanProperty`, to grey out a button or field while something is busy.
- **Show or hide from a flag** - bind a section's `visibleProperty()` one-way to a `BooleanProperty`, to reveal part of the window only when it is needed.
- **One control to another** - bind `label.textProperty()` to a field's `textProperty()` for a live echo, or a child's `prefWidthProperty()` to its parent's `widthProperty()` so it tracks the parent's size.

The pattern never changes: find the property (its accessor is usually the value's name plus `Property`), then `bind` it one-way or `bindBidirectional` it two-way.

Recap

- **Binding** declares that values stay equal, so JavaFX keeps them in step with no listener of your own.

- **`bind`** is one-way: the follower mirrors the source and becomes read-only until you **`unbind`** it, ideal for a display.
- **`bindBidirectional`** is two-way: either side can change and both stay equal, ideal for an editable control; **`unbindBidirectional`** breaks it.
- Reach a control's value as a property with accessors like **`textProperty`**, **`valueProperty`**, **`selectedProperty`**, and **`progressProperty`**, then bind it to a property of your own.
- Bind on **your** property (`name.bindBidirectional(control.textProperty())`) for the clean two-way case; call **`.invoke("bind", data.obj)`** on a control's property to make the control follow your data one-way.

With properties and binding, you can keep an interface and its data locked together with almost no wiring code. That closes Part VII. Next, in Part VIII, we leave the flat screen behind and step into **3D graphics**.

Part VIII

3D Graphics

Chapter 27 - The 3D Scene

Welcome to **Part VIII, 3D Graphics**. Everything in the book so far has been **flat**, shapes, text, and controls painted on a 2D surface. JavaFX can also draw in real **3D**: solid boxes and spheres, lit and shaded, seen through a movable **camera**. This chapter builds the stage, the **SubScene** that holds a 3D world and the **camera** that looks into it. The next two chapters fill that stage with shapes, materials, and light.

Every example in this chapter lives here: `27-the-3d-scene`.

SubScene and Depth

A 3D scene does not replace your window. It sits **inside** it as a **SubScene**, a rectangular 3D viewport you drop into an ordinary 2D layout like any other node. That means a window can be part normal controls and part 3D, or all 3D, whatever you put in the layout.

Three pieces build the stage:

- a **Group** that acts as the **root** of the 3D world (it holds the shapes and lights, just as a layout pane holds 2D nodes),
- a **SubScene** that wraps the group, gives it a size, and turns on a **depth buffer**,
- and a **camera** (next section) that decides how the scene is viewed.

Here is the stage, with three cubes placed at different depths:

```
root = new Group();                    // the 3D root: shapes and lights go
                                       // here
root.add([nearCube, midCube, farCube, lights]);

sub = new SubScene(root, 460.0, 340.0, true, fxantialiasing.BALANCED);
sub.setFill(new Color("#0f172a"));  // the 3D background color
```

The `SubScene` constructor takes the **root**, a **width** and **height** in pixels, a **depth buffer** flag, and an **antialiasing** setting:

- The **depth buffer** (`true`) is what makes 3D look right: it lets nearer surfaces correctly cover the ones behind them. Leave it on for any real 3D.
- **Antialiasing** smooths the jagged edges of angled surfaces. Pass an `fxantialiasing` value: `fxantialiasing.BALANCED` for smooth edges or `fxantialiasing.DISABLED` for none.

One thing to know before you place anything: the **3D coordinate system**. x runs right, but y runs **down** (the same as 2D), and z runs **into the screen**, so a *larger* z is *farther away*. You move a shape

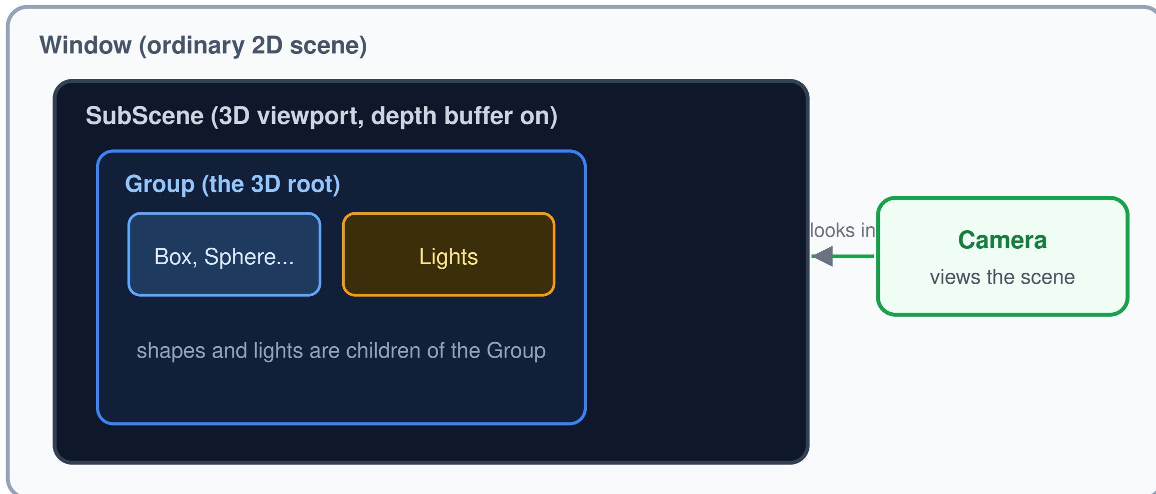


Figure 93: A diagram: a Window holds a SubScene (a dark 3D viewport), which holds a Group root containing shapes and lights, with a Camera looking in

with `setTranslateX`, `setTranslateY`, and `setTranslateZ`, the same node methods from Chapter 23, now with depth in play.

Cameras

Set up the stage and run it, and you will see **nothing**. A 3D scene is invisible until a **camera** looks into it. You attach one with **`setCamera`**:

```
sub.setCamera(cam);
```

There are two kinds, and the difference is how they handle distance.

A **`PerspectiveCamera`** sees like your eye or a real camera does: things farther away look **smaller**. It is what you want for almost any realistic 3D. You build it, back it away from the scene along z, and can tune its lens:

```
cam = new PerspectiveCamera(true);   // true: the eye sits at the
                                     // camera's own origin
cam.setTranslateZ(-560.0);           // pull the camera back so the
                                     // shapes fit in view
cam.setFieldOfView(38.0);            // lens angle in degrees; smaller
                                     // looks more zoomed in
cam.setNearClip(0.1);                // nearest visible distance
cam.setFarClip(4000.0);              // farthest visible distance;
                                     // anything beyond is cut off
```

`setNearClip` and `setFarClip` come from the shared **`Camera`** base and set the **range** of depth the camera can see; anything nearer or farther is clipped away. With the perspective camera in place, the three cubes clearly recede into the distance:

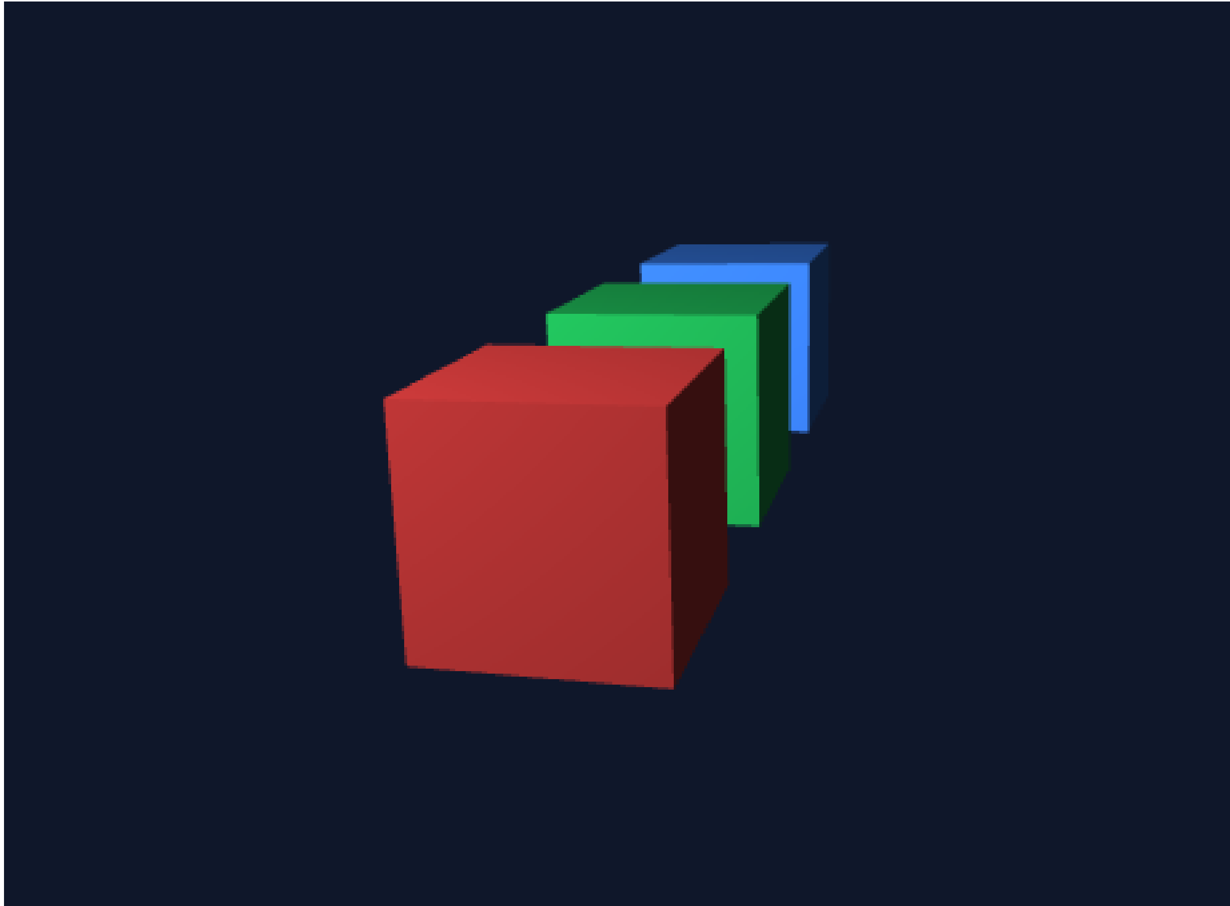


Figure 94: The 3D scene: a large red cube in front, a smaller green cube behind it, and a smaller blue cube farthest back, on a dark background

A **`ParallelCamera`** is the other option. It is **orthographic**: distance does **not** change size, so parallel lines stay parallel. That is exactly wrong for a lifelike view but exactly right for technical or diagram-like drawing, and for 2.5D games. One quirk: it places world `(0, 0)` at the viewport's **top-left** corner (like 2D coordinates), so you shift it by half the size to center your scene.

The same three cubes make the difference obvious, the perspective camera shrinks the far ones, the parallel camera keeps them all the same size:

Recap

- A 3D world lives in a **`SubScene`**, a 3D viewport you place inside an ordinary 2D window. Its root is a **`Group`** that holds the shapes and lights.
- The `SubScene` constructor takes the root, a width and height, a **depth buffer** (keep it `true` so near surfaces cover far ones), and an **antialiasing** setting (`fxantialiasing.BALANCED` or `fxantialiasing.DISABLED`).

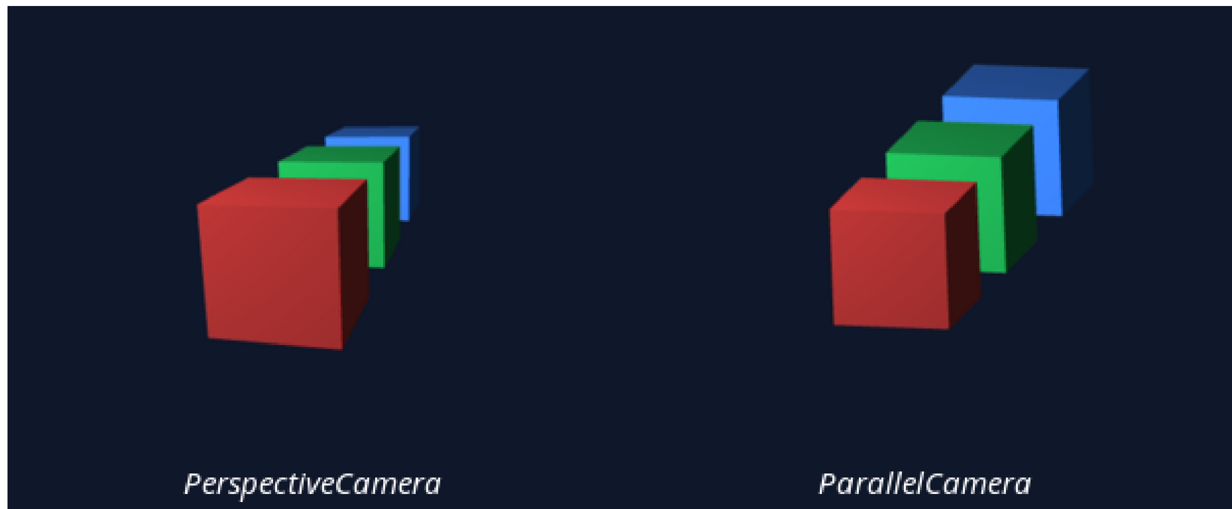


Figure 95: Side by side: with a PerspectiveCamera the cubes shrink into the distance; with a ParallelCamera all three cubes are the same size

- In 3D, x is right, y is **down**, and z goes **into the screen** (bigger z is farther); move shapes with `setTranslateX` / `setTranslateY` / `setTranslateZ`.
- Nothing shows until you **`setCamera`**. A **`PerspectiveCamera`** makes distant things smaller (`setFieldOfView`, `setNearClip`, `setFarClip`); a **`ParallelCamera`** keeps sizes flat for a technical look.

The stage is set and the camera is rolling. In the next chapter we fill the scene with **shapes**, the built-in box, sphere, and cylinder, and even custom geometry you build point by point.

Chapter 28 - 3D Shapes and Meshes

Chapter 27 built the stage: a `SubScene` with a camera looking into it. Now we fill it with **things to look at**. JavaFX gives you three ready-made solids for the common cases, a shared base that controls how any solid is drawn, and a way to build **any** shape you like out of triangles. By the end you will have a box, a sphere, a cylinder, and a pyramid you built yourself, all in one scene.

Every example in this chapter lives here: `28-3d-shapes`.

Built-in Solids: Box, Sphere, Cylinder

Three solids come ready to use. You create each with the numbers that define its size, then give it a **material** for color (materials are the next chapter; for now, a `PhongMaterial` with a diffuse color is all you need):

```
box = new Box(95.0, 95.0, 95.0);       // width, height, depth

sphere = new Sphere(55.0);             // radius (an optional 2nd arg
                                       // sets smoothness)

cylinder = new Cylinder(45.0, 105.0);  // radius, height
```

`Sphere` and `Cylinder` take an optional last argument, the number of **divisions**, which is how many flat segments approximate their curve (more divisions means a smoother curve but more work to draw); it defaults to 64, which looks smooth. You place each solid in space with `setTranslateX` / `setTranslateY` / `setTranslateZ`, just like the cubes in Chapter 27.

Here are all three, lined up with the custom pyramid we build further down:

The Shape3D Base, CullFace, and DrawMode

Every 3D shape, the three solids **and** the custom meshes below, shares a base called **`Shape3D`**. It carries the settings they all understand:

- **`setMaterial`** gives the surface its color and shine (Chapter 29).
- **`setDrawMode`** switches between a **solid** surface and a **wireframe**. Pass an `fxdrawmode` value: `fxdrawmode.FILL` for solid (the default) or `fxdrawmode.LINE` to see just the edges of the triangles the shape is made of.

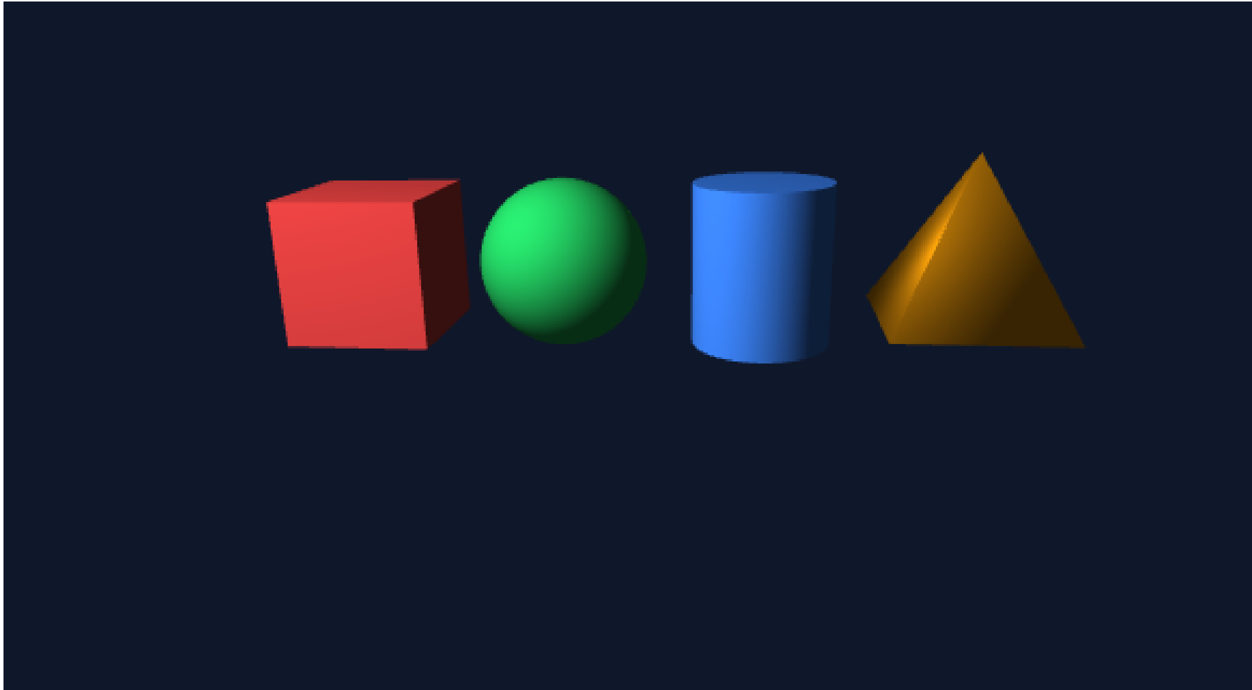


Figure 96: Four 3D shapes in a row: a red box, a green sphere, a blue cylinder, and an orange pyramid, all lit and shaded

- **`setCullFace`** chooses which side of each triangle to skip drawing, a speed trick. Pass an `fxcullface` value: `fxcullface.BACK` (the default) to skip the faces pointing away from you, `fxcullface.FRONT` to skip the ones facing you, or `fxcullface.NONE` to draw both sides.

Drawing a sphere solid and then as a wireframe shows what `fxdrawmode.LINE` reveals, the grid of triangles underneath every smooth surface:

Custom Geometry: Building a Mesh

When the three solids are not enough, you build your own shape from scratch. Every 3D surface, even a sphere, is really a lot of tiny **triangles**, and a **`TriangleMesh`** lets you specify them yourself. It holds two lists:

- **points** - the corners (vertices) of your shape, each an `x, y, z` position.
- **faces** - the triangles, each naming three points by their position in the list.

A **`MeshView`** then wraps the mesh so it can be shown, and because a `MeshView` is a `Shape3D`, everything above (material, draw mode, cull face) works on it too.

This builds the orange pyramid, five points (an apex plus four base corners) joined into six triangles (four sides and two for the square base):

```
mesh = new TriangleMesh();
mesh.addPoints([
      0.0, -70.0,   0.0,     // point 0: the apex (y is up when
                             // negative)
```

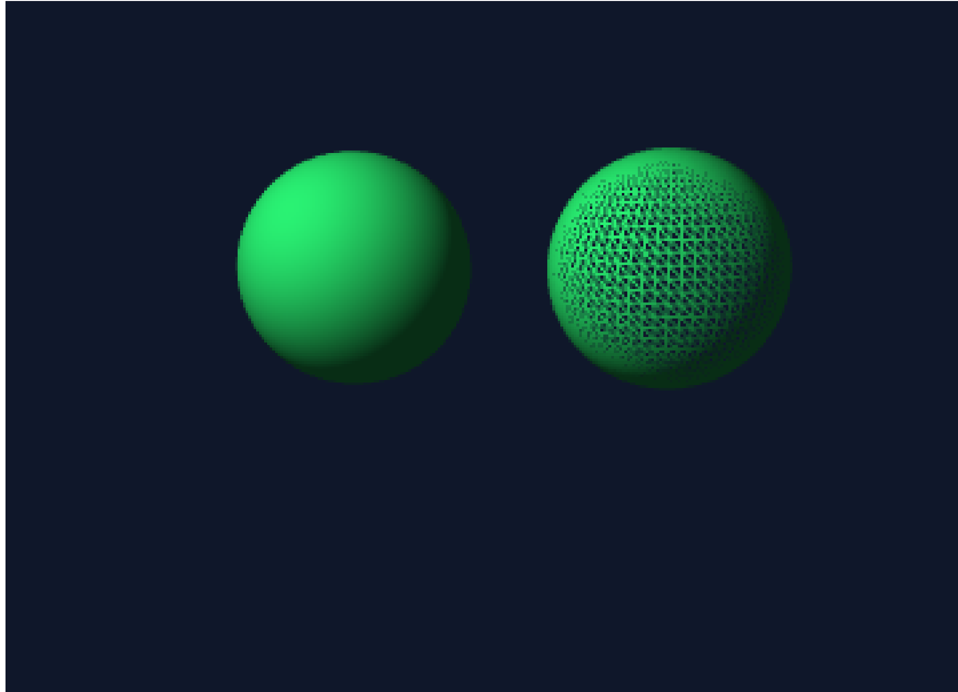


Figure 97: Two green spheres: the left one solid, the right one drawn as a wireframe of small triangles

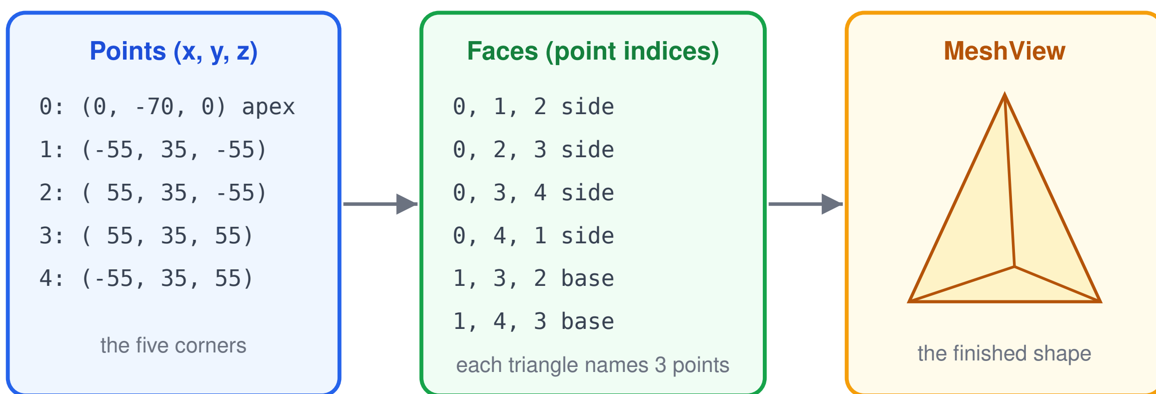


Figure 98: A diagram: a list of five points feeds a list of faces (triangles of point indices), which assemble into a pyramid MeshView

```
    -55.0,  35.0, -55.0,     // points 1-4: the four base corners
     55.0,  35.0, -55.0,
     55.0,  35.0,  55.0,
    -55.0,  35.0,  55.0
]);
mesh.addTexCoords([0.0, 0.0]);   // one dummy point; our color comes
                                 // from the material
mesh.addFaces([
    0,0, 1,0, 2,0,    // four side triangles: apex + two base corners
    0,0, 2,0, 3,0,
    0,0, 3,0, 4,0,
    0,0, 4,0, 1,0,
    1,0, 3,0, 2,0,    // the square base as two triangles
    1,0, 4,0, 3,0
]);

view = new MeshView(mesh);
view.setCullFace(fxcullface.NONE);   // draw every face, whichever way
                                     // it points
```

Two things to notice. Each number in **addFaces** comes in **pairs**, a point index and a texture-coordinate index, because a mesh's **VertexFormat** is `fxvertexformat.POINT_TEXCOORD` by default (every vertex ties a point to a spot on a texture image). Since we color the pyramid with a plain material and no image, one dummy texture point and index `0` everywhere is fine. And we set **setCullFace(fxcullface.NONE)** so the pyramid shows no matter which way each triangle happens to face, handy while you are still getting a hand-built shape's triangles pointing the right way.

For finer control, `setVertexFormat` can switch a mesh to `fxvertexformat.POINT_NORMAL_TEXCOORD`, which adds a **normal** (a direction) to every vertex so you can steer exactly how light bounces off it, an advanced option most shapes do not need.

Loading a Model from a File

Placing points by hand is fine for a pyramid, but a real model, a character, a car, a teapot, has thousands of them. Those are built in tools like Blender and saved to a **model file**, and you load one with **Model3D** instead of writing geometry yourself. It reads the common **.obj** (Wavefront) format, along with Maya **.ma** and JavaFX **.fxml** files.

Loading is two lines, and **getRoot** hands the whole model back as a `Group` you drop into the scene like any other node:

```
model = new Model3D();
model.load("teapot.obj");     // read the file (must be done after the
                              // app has started)
shape = model.getRoot();      // the model as a Group, ready to add to
                              // the scene
```

The importer gives the model a plain gray material. To recolor it, reach its meshes with **getMeshViews** and set your own material on each, the same **setMaterial** the built-in solids use:

```
paint = new PhongMaterial();
paint.setDiffuseColor(new Color("#3b82f6"));
for (mesh : model.getMeshViews()) {
    mesh.setMaterial(paint);
}
```

One thing to expect: a loaded model arrives in **its own size and orientation**, set by whoever made it, not yours. This teapot's file is small and built **Y-up**, while JavaFX points y **down**, so the example scales it up, recenters it, and flips it upright (with the transform methods from Chapter 23) before it looks right. That fiddling is normal, and once done you have a real model in your scene:

Figure 99: A blue Utah Teapot, loaded from a .obj file, lit and shaded in the 3D scene

That is the famous **Utah Teapot**, the reference model 3D graphics has used since 1975, here loaded straight from a file. `Model3D` has a few more helpers worth knowing: **getMeshNames** and **getMeshView** reach a single named piece, **supportedFormats** lists the file types it reads, and **getTimeline** returns a model's built-in animation when it has one.

> **Load after the toolkit is running.** `Model3D` builds live JavaFX nodes as it reads the file, so load the model *after* your `fx.fxApp` exists (as the example does), the same rule the

`WebView` and media chapters followed.

Points in Space with Point3D

You have already met **`Point3D`** in passing, it was the rotation axis in Chapter 27 (`setRotationAxis(new Point3D(1.0, 1.0, 0.0))`). It is simply a value that bundles three numbers, an `x`, a `y`, and a `z`:

```
p = new Point3D(1.0, 0.0, 0.0);   // a point, or a direction, in 3D
                                  // space
```

Beyond holding a position, a `Point3D` does a little vector math: **`getX`**, **`getY`**, and **`getZ`** read its parts, **`distance`** measures the gap to another point, and **`add`** combines two of them. You reach for it whenever an API wants a point or a direction, an axis to spin around, the spot to place something, the heading a light points.

Recap

- Three solids come built in: **`Box`** (`width, height, depth`), **`Sphere`** (`radius`), and **`Cylinder`** (`radius, height`); the curved two take an optional **divisions** count for smoothness.
- All 3D shapes share **`Shape3D`**: **`setMaterial`** for color (next chapter), **`setDrawMode`** (`fxdrawmode.FILL` or `fxdrawmode.LINE` for wireframe), and **`setCullFace`** (`fxcullface.BACK` / `fxcullface.FRONT` / `fxcullface.NONE`).
- Build any other shape from a **`TriangleMesh`** of **points** (`addPoints`) and **faces** (`addFaces`), then show it with a **`MeshView`**; its **`VertexFormat`** decides what each face vertex lists.
- Load a ready-made model file (`.obj`, `.ma`, `.fxml`) with **`Model3D`**: **`load`** reads it and **`getRoot`** returns it as a `Group` for the scene; expect to scale, center, and orient an imported model to fit.
- **`Point3D`** bundles an `x, y, z` and does simple vector math; it is what APIs want wherever a point or a direction is needed.

Our shapes so far are flat, solid colors. In the next chapter we make them look **real**, with **materials** that add shine and texture, and **lights** that give the whole scene depth and mood.

Chapter 29 - Materials and Lighting

The shapes in the last chapter were solid blocks of flat color. What makes 3D look **real** is the interplay of two things: the **material** on a surface (is it matte or glossy? what color? does it have a texture?) and the **light** falling on it (where is it coming from? what color?). This chapter covers both, and with them your scenes go from flat toys to something that looks lit and solid.

Every example in this chapter lives here: `29-materials-lighting`.

Material and PhongMaterial

A **`Material`** describes how a surface *looks*. In practice you use one kind, a **`PhongMaterial`** (named after the Phong lighting model it follows). You have already used its simplest setting, a base color:

```
mat = new PhongMaterial();
mat.setDiffuseColor(new Color("#3b82f6"));   // the base ("diffuse")
                                             // color of the surface
shape.setMaterial(mat);
```

The **diffuse** color is the plain color the surface shows. To make it look **shiny**, add a **specular** highlight, the bright spot you see on a glossy ball:

```
mat.setSpecularColor(new Color("#ffffff"));  // color of the shiny
                                             // highlight
mat.setSpecularPower(32.0);                  // higher = smaller,
                                             // sharper highlight
```

`setSpecularPower` controls how *tight* that highlight is: a low value like 32 gives a soft, broad sheen, while `100` and up gives a small, glassy pinpoint. The three spheres below share one diffuse color but climb from matte (no specular) to glossy to near-mirror, and the highlight shrinks and sharpens as the power rises:

Textures: Painting Images onto Shapes

A material can carry more than colors, it can carry **images**. The most useful is a **diffuse map**: hand **`setDiffuseMap`** a picture and it is painted across the whole surface like a photographic skin, wrapping around the geometry using the shape's built-in texture coordinates. Wrap a satellite photo of Earth around a sphere and you have a globe:

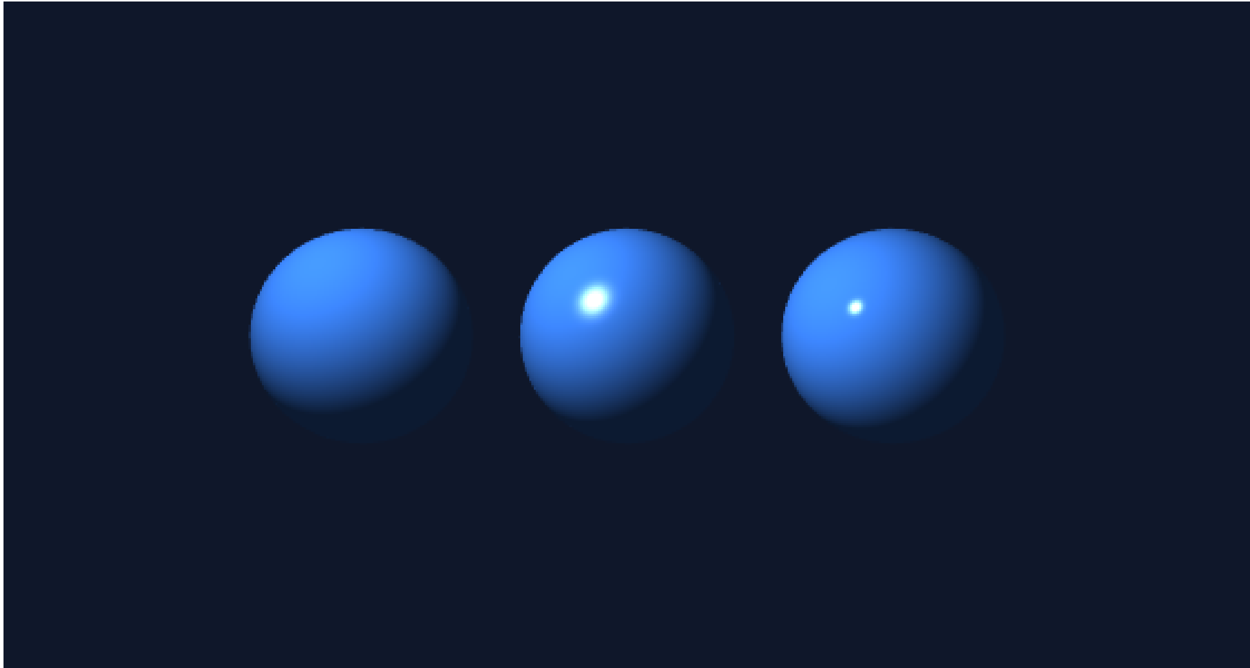


Figure 100: Three blue spheres: a matte one with no highlight, a glossy one with a soft highlight, and a mirror-like one with a tiny sharp highlight

```
globe = new Sphere(85.0);
mat = new PhongMaterial();
mat.setDiffuseMap(new Image("earth.jpg"));   // paint the photo onto the
                                             // sphere
globe.setMaterial(mat);
```

That is one `new Image` (the same `Image` type from Chapter 19) and one `setDiffuseMap`. The identical call textures any built-in shape, a `Box` or a `Cylinder` just as easily as this sphere, since all three carry the texture coordinates that tell the picture how to wrap.

Faking surface detail with a bump map

A **bump map** does not change a shape's color, it changes how light *hits* it, so a perfectly flat surface can look bumpy without adding a single real triangle. You hand **`setBumpMap`** a special image called a **normal map** (it looks mostly blue, and encodes which way each tiny patch of surface tilts), and the lighting does the rest. Here two cubes share one plain silver material, and the right one adds a bump map:

```
metal = new PhongMaterial();
metal.setDiffuseColor(new Color("#b8bcc4"));   // plain silver
metal.setBumpMap(new Image("normal.jpg"));     // fake fine bumps and
                                               // dents
cube.setMaterial(metal);
```

The left cube is glassy-smooth; the right looks hammered, yet both are the same six-sided box.

Figure 101: A photographic Earth globe: a sphere wrapped in a satellite image of the planet

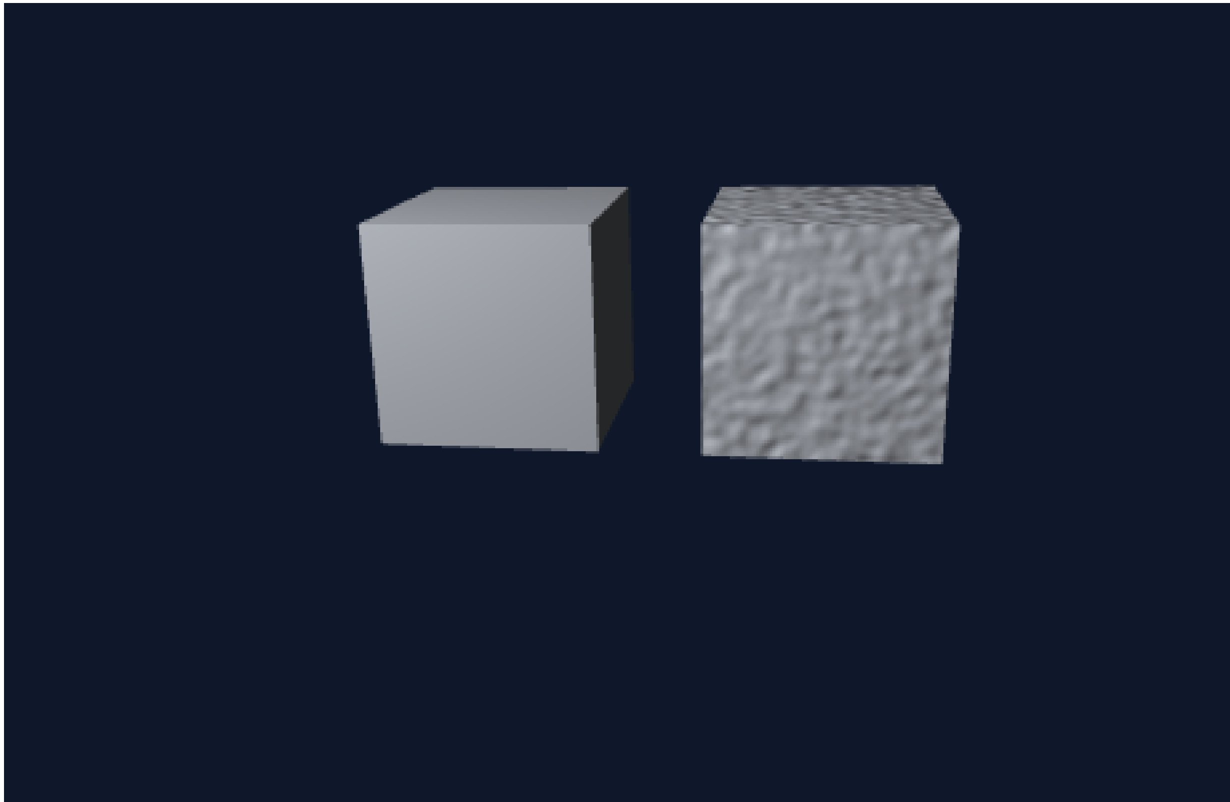


Figure 102: Two silver cubes: the left one glassy-smooth, the right one covered in fine bumps from a bump map

This is how games make a flat wall look like rough brick or pitted metal cheaply, real geometry is expensive, a bump map is nearly free.

A `PhongMaterial` has one more image slot, **`setSelfIlluminationMap`**, for parts that should **glow on their own** (the lit windows on a spaceship, say) no matter how dark the scene's lighting is.

LightBase, AmbientLight, and PointLight

A material has nothing to show until light hits it. A scene with **no** light is pure black. Lights are nodes you add to the scene like shapes; they all share a base called **`LightBase`**, which gives each a **`setColor`** and a **`setLightOn`** to switch it off. There are two kinds, and they behave very differently:

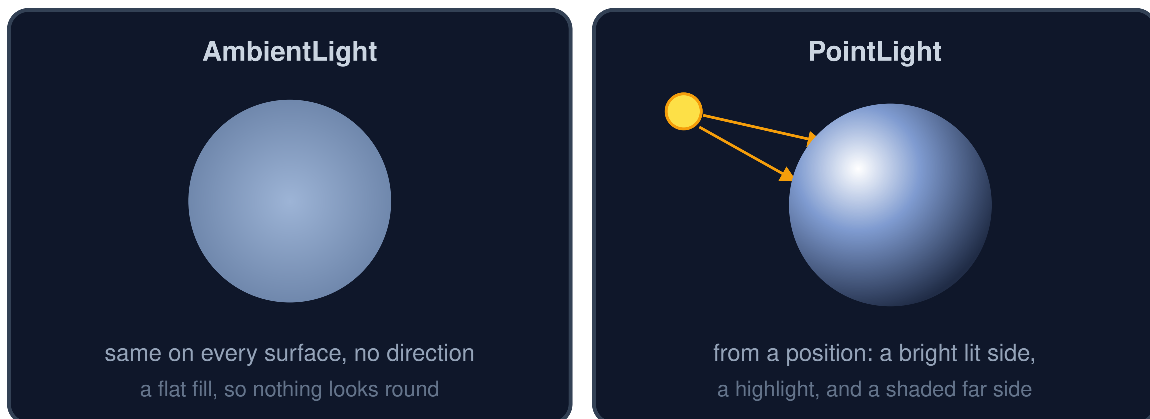


Figure 103: A diagram: under AmbientLight a sphere looks like a flat disc; under a PointLight it has a bright lit side, a highlight, and a shaded far side

An **`AmbientLight`** lights every surface **equally, from no particular direction**. Because it hits all sides the same, it flattens everything, a sphere under ambient light alone looks like a flat disc. Its real job is a gentle **fill** so shadowed areas are not pitch black:

```
amb = new AmbientLight(new Color("#333333"));   // a soft, even fill
```

A **`PointLight`** is a lamp at a **position**. Light spreads out from that point, so surfaces facing it are bright, surfaces turned away fall into shadow, and glossy materials get their highlight, exactly the cues that make a shape look round and solid. You place it with the translate methods and can color it:

```
lamp = new PointLight(new Color("#ffffff"));
lamp.setTranslateX(-240.0);   // position the lamp up and to the left
lamp.setTranslateY(-260.0);
lamp.setTranslateZ(-320.0);
```

Because a point light has both a place and a color, two of them in different colors light a shape from

different sides. Here one pale sphere sits between a red lamp on the left and a blue lamp on the right:

Figure 104: A pale sphere lit red on its left side and blue on its right, blending to purple in the middle, on a dark background

That single image is the whole point of the chapter: the **material** (a plain pale surface) and the **lights** (red from one side, blue from the other) together decide what you actually see.

Recap

- A **`PhongMaterial`** gives a surface its look: **`setDiffuseColor`** for the base color, and **`setSpecularColor`** with **`setSpecularPower`** for a shiny highlight (higher power is a smaller, sharper highlight).
- A material can also carry **images**: **`setDiffuseMap`** paints a photo across the shape (a world map makes a globe), **`setBumpMap`** fakes surface relief from a normal map, and **`setSelfIlluminationMap`** makes parts glow on their own.
- Lights share **`LightBase`** (**`setColor`**, **`setLightOn`**). An **`AmbientLight`** fills the whole scene evenly (a flat fill, no direction); a **`PointLight`** is a lamp at a position that creates a lit side, a highlight, and shadow, the cues that make 3D look solid.

That completes **Part VIII**. You can now build a 3D world, a `SubScene` and camera, fill it with built-in and custom shapes, and dress it in materials and light. Next, in Part IX, we come back to the flat screen to turn data into pictures with **charts**.

Part IX

Charts

Chapter 30 - Charting Data

Welcome to **Part IX, Charts**. A column of numbers is hard to read; a picture of those numbers tells a story at a glance. JavaFX has a family of ready-made **chart** controls, bar, line, area, scatter, bubble, and pie, and the good news is that they are all built the same way. Learn one and you have learned them all.

Every example in this chapter lives here: `30-charts`.

Chart Building Blocks

Almost every chart (all but the pie) is an **X/Y chart**, and it is made of three parts:

- two **axes**, the lines up the side and along the bottom that give the numbers a scale,
- one or more **series**, each a named group of data points,
- and the **chart** itself, which draws the series against the axes.

There are two kinds of axis. A **`CategoryAxis`** holds text **labels** (months, names, products), while a **`NumberAxis`** holds a range of **numbers**. A bar chart of monthly sales uses a `CategoryAxis` along the bottom (the months) and a `NumberAxis` up the side (the amounts).

The data goes in a **`ChartSeries`**: you give it a name (which shows in the legend) and add points to it with **`addPoint`**:

```
series = new ChartSeries("2024");   // the name shows in the legend
series.addPoint("Jan", 120);        // an (x, y) point: label, value
series.addPoint("Feb", 95);
```

Put together, every X/Y chart follows the same three-step recipe: **make the axes, make the chart from them, then add series.** The chart types below differ only in the first word, `BarChart`, `LineChart`, `ScatterChart`, everything else is identical.

Bar Charts

A **`BarChart`** draws each value as a column. You build it from a category axis and a number axis, then add series; with more than one series, the bars sit **grouped** side by side:

```
chart = new BarChart(new CategoryAxis(), new NumberAxis());
chart.setTitle("Monthly Sales");

```

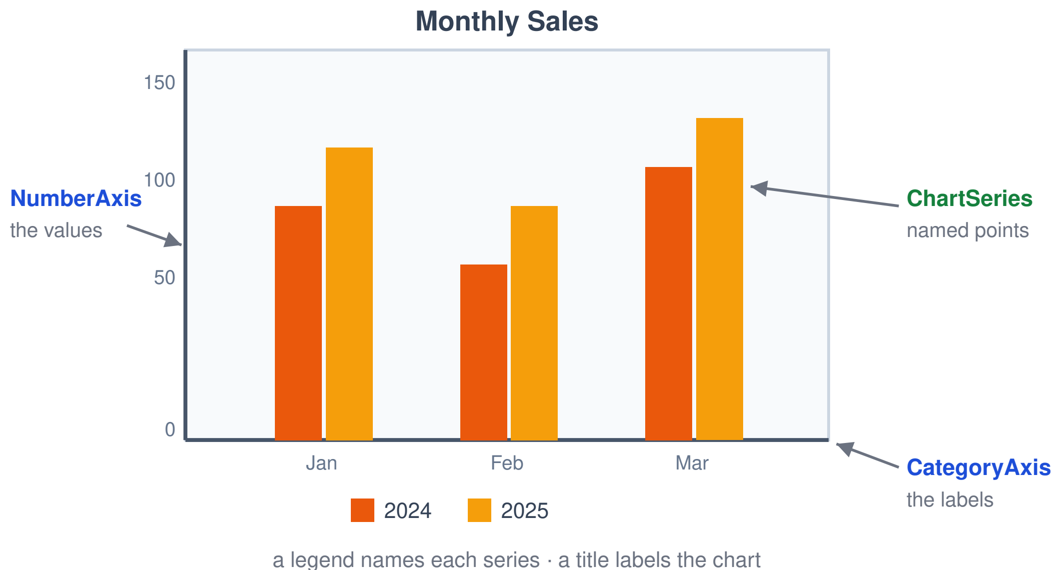


Figure 105: A labeled bar chart showing which part is the NumberAxis, the CategoryAxis, a ChartSeries, the legend, and the title

```
y2024 = new ChartSeries("2024");
y2024.addPoint("Jan", 120);
y2024.addPoint("Feb", 95);
y2024.addPoint("Mar", 140);
chart.addSeries(y2024);

y2025 = new ChartSeries("2025");
y2025.addPoint("Jan", 150);
// ... more points ...
chart.addSeries(y2025);
```

A **`StackedBarChart`** is built exactly the same way, but instead of grouping the series side by side it **stacks** them into a single bar per label, so you can see both the parts and the total at once:

Line and Area Charts

Swap `BarChart` for **`LineChart`** and the same data becomes connected lines, the natural choice for showing a **trend over time**:

```
chart = new LineChart(new CategoryAxis(), new NumberAxis());
chart.setTitle("Average Temperature");

denver = new ChartSeries("Denver");
```

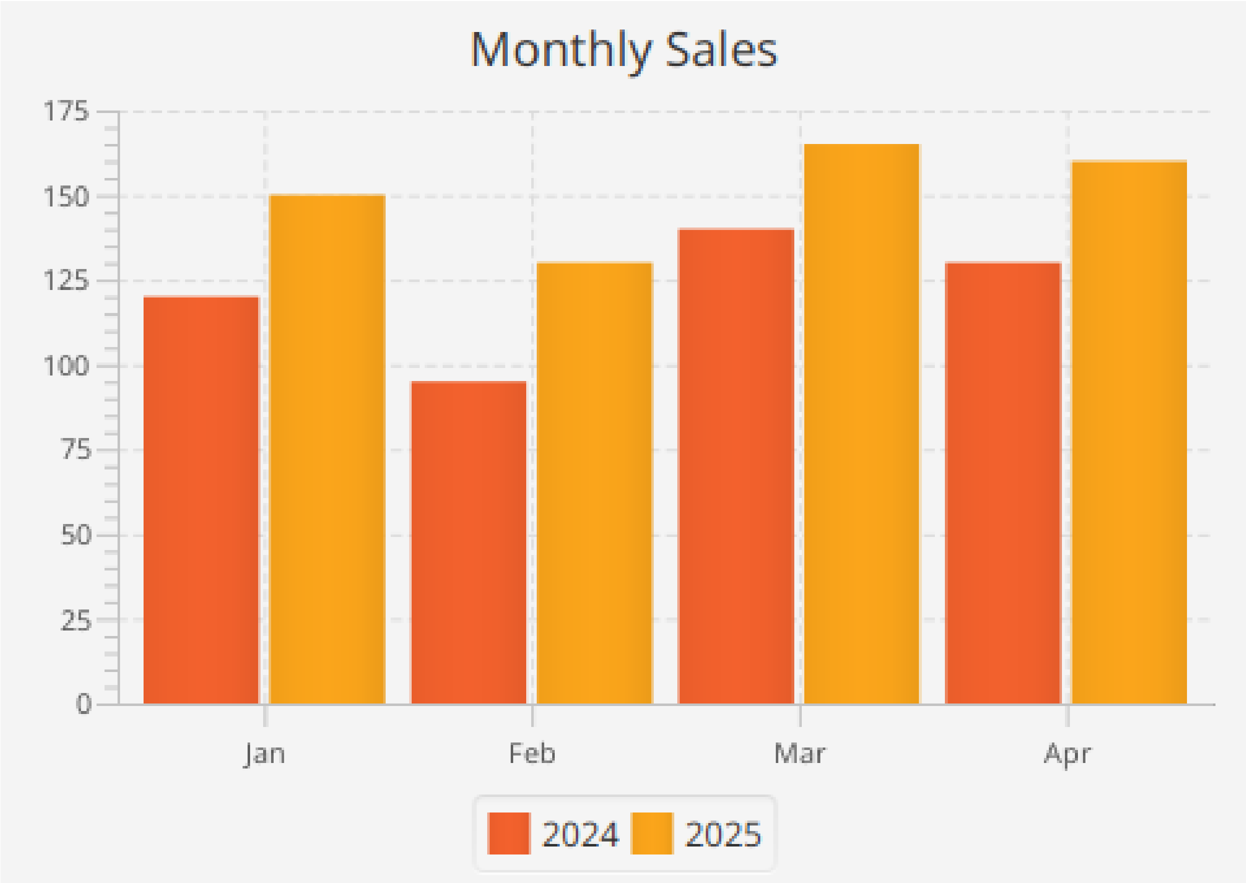


Figure 106: A grouped bar chart titled “Monthly Sales” with 2024 and 2025 bars side by side for each month

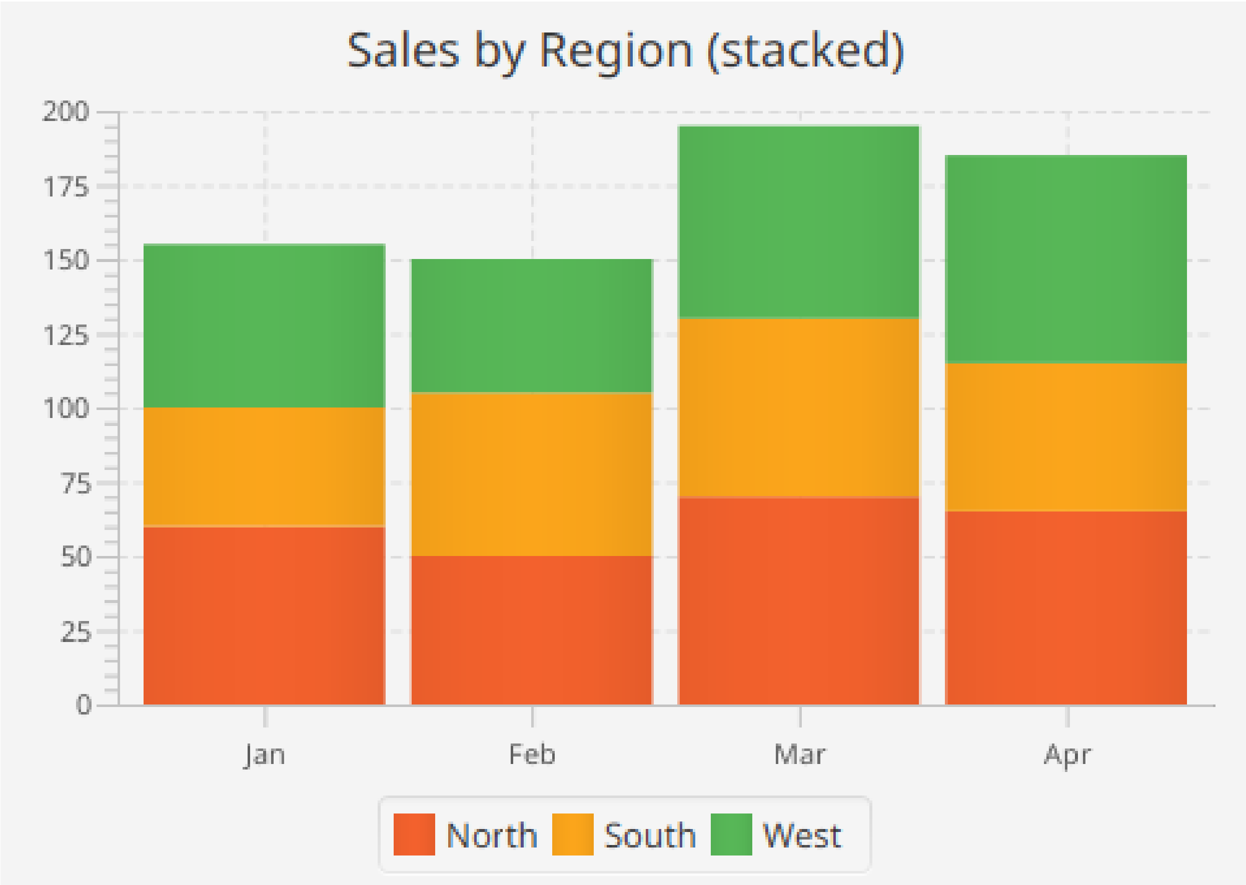


Figure 107: A stacked bar chart where North, South, and West values stack into one bar per month

```
denver.addPoint("Jan", 30);
denver.addPoint("Feb", 45);
denver.addPoint("Mar", 68);
chart.addSeries(denver);
```

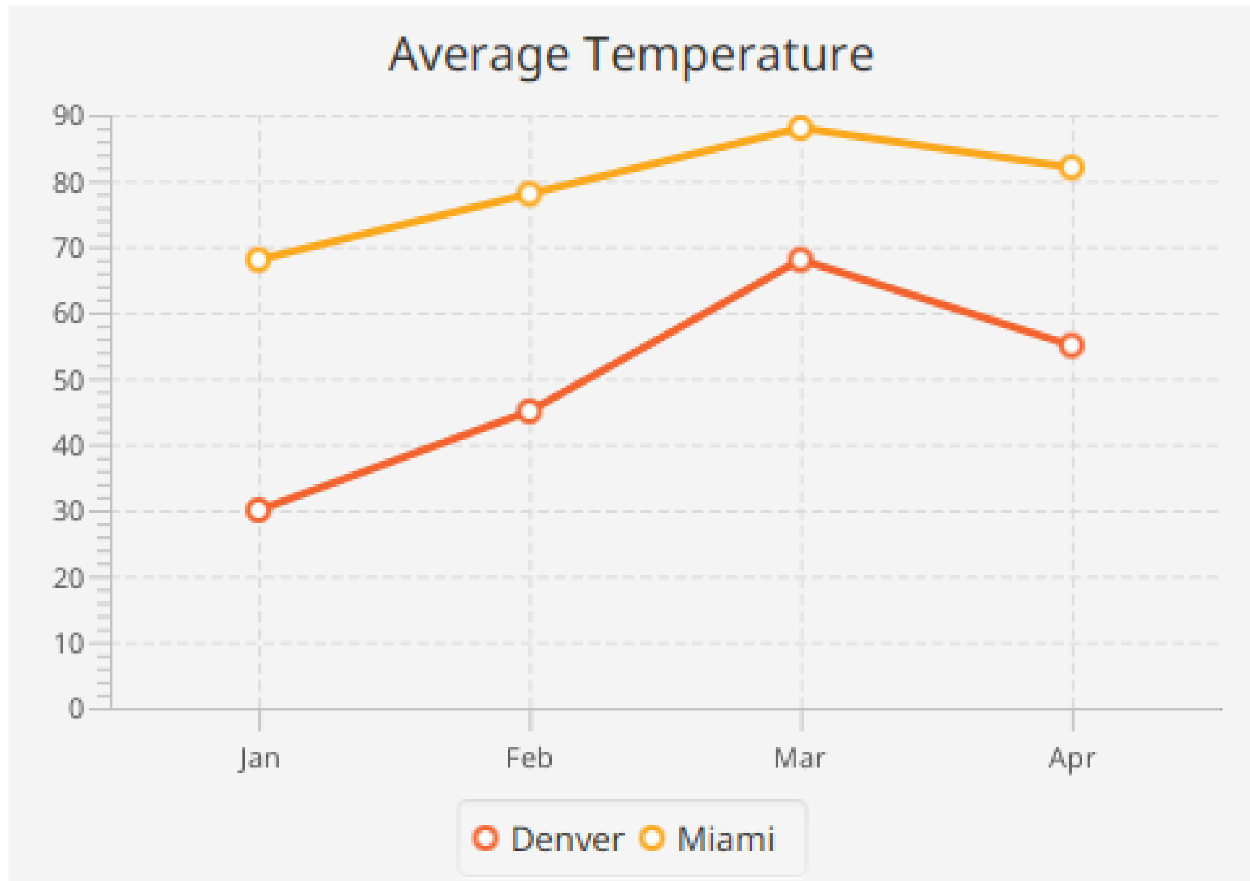


Figure 108: A line chart of average temperature with a line each for Denver and Miami across the months

An **`AreaChart`** is a line chart with the space **under** each line filled in, which emphasizes volume, a running total, traffic, rainfall:

And a **`StackedAreaChart`** stacks those filled areas the way a stacked bar chart stacks its bars, for showing how several parts add up to a whole over time.

Time Series: Dates on the X Axis

Charting values **over time**, daily sales, hourly temperatures, is one of the most common needs, and it takes nothing new. JavaFX has no special "date axis"; a time series is just an ordinary line or bar chart whose x-axis labels happen to be **dates**. The `"Average Temperature"` chart above already worked this way, its months are simply date labels.

The trick is to turn each date into a label string, and the `Date` type from Chapter 10 does exactly that with **`format`**. Use a **`CategoryAxis`** for the x-axis, format each date, and add it as the point's

Figure 109: An area chart of website visits with the area under the line shaded

label:

```
chart = new LineChart(new CategoryAxis(), new NumberAxis());
series = new ChartSeries("July");

day = new Date();
day.parse("2026-07-01", "yyyy-MM-dd");
for (visits : [120, 145, 130, 170, 160]) {
    series.addPoint(day.format("MMM d"), visits);   // label like "Jul
                                                    // 1"
    day.addDays(1);
}
```

That produces a clean daily chart (`Jul 1`, `Jul 2`, ...), and the same idea covers **date-times**: format with a pattern that includes the time (`"MMM d HH:mm"`) for an hourly or minute-by-minute series. One thing to know: a `CategoryAxis` spaces every label **evenly**, so it is perfect for regular steps (each day, each hour) but does not leave wider gaps for missing dates. For readings at **irregular** moments where the true spacing matters, plot the dates as **numbers** (each date's millisecond value) on a `NumberAxis` and hand its **`setTickLabelFormatter`** a function that turns those numbers back into date labels, a more advanced route most time series do not need.

Scatter and Bubble Charts

A **`ScatterChart`** plots points with **no lines** joining them, ideal for spotting a pattern in raw measurements. Both of its axes are usually a `NumberAxis`, because each point is a pair of numbers rather than a labeled category:

```
chart = new ScatterChart(new NumberAxis(), new NumberAxis());
chart.setTitle("Height vs Weight");

groupA = new ChartSeries("Group A");
groupA.addPoint(62, 120);
groupA.addPoint(65, 135);
groupA.addPoint(68, 150);
chart.addSeries(groupA);
```

A **`BubbleChart`** is a scatter chart with one extra trick: each point also has a **size**, so it can show a *third* number, a budget, a population, as the size of the bubble. Just add that third value to `addPoint` after the x and y:

```
chart = new BubbleChart(new NumberAxis(), new NumberAxis());
teams = new ChartSeries("Teams");
teams.addPoint(45.0, 55.0, 6.0);   // x, y, size
teams.addPoint(75.0, 70.0, 8.0);
chart.addSeries(teams);
```

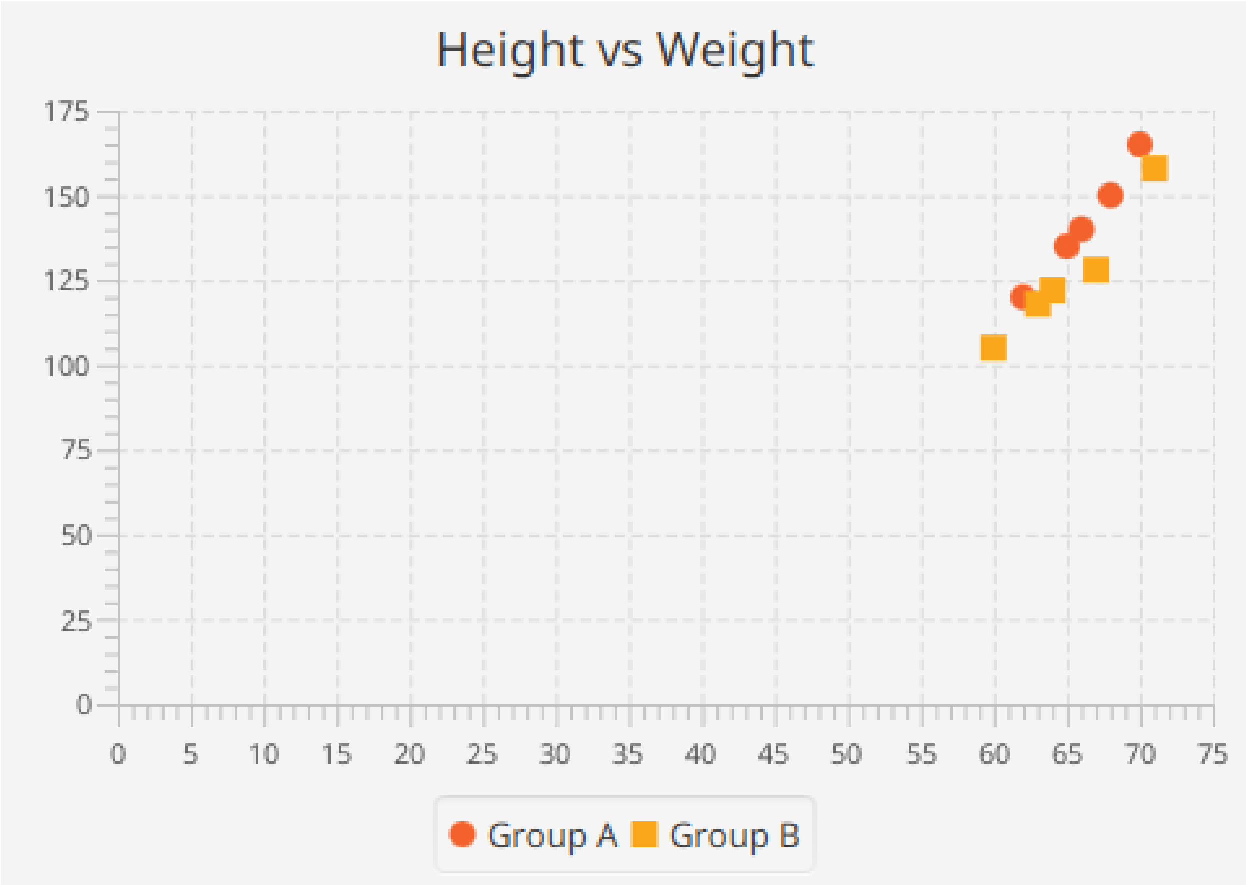


Figure 110: A scatter chart of height versus weight with two groups of points in different colors

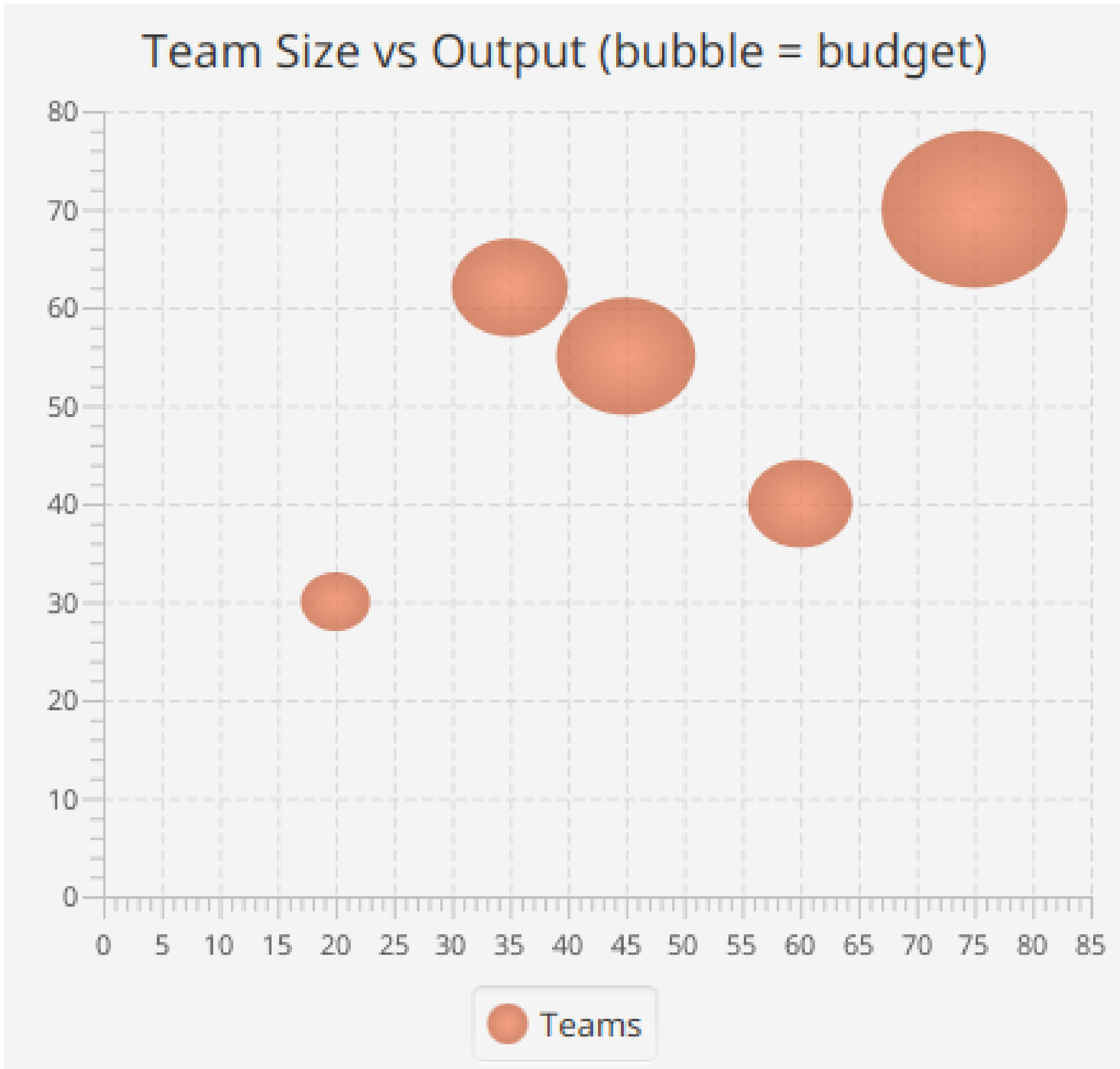


Figure 111: A bubble chart where each team is a bubble whose size represents its budget

PieChart

A **PieChart** is the odd one out: it has no axes and no series, just a set of **slices**. Each **PieSlice** is a label and a value, and JavaFX turns the values into proportional wedges automatically (you do not compute percentages yourself):

```
chart = new PieChart();
chart.setTitle("Browser Share");
chart.addSlice(new PieSlice("Chrome", 64.0));    // label, value
chart.addSlice(new PieSlice("Safari", 19.0));
chart.addSlice(new PieSlice("Edge", 5.0));
```

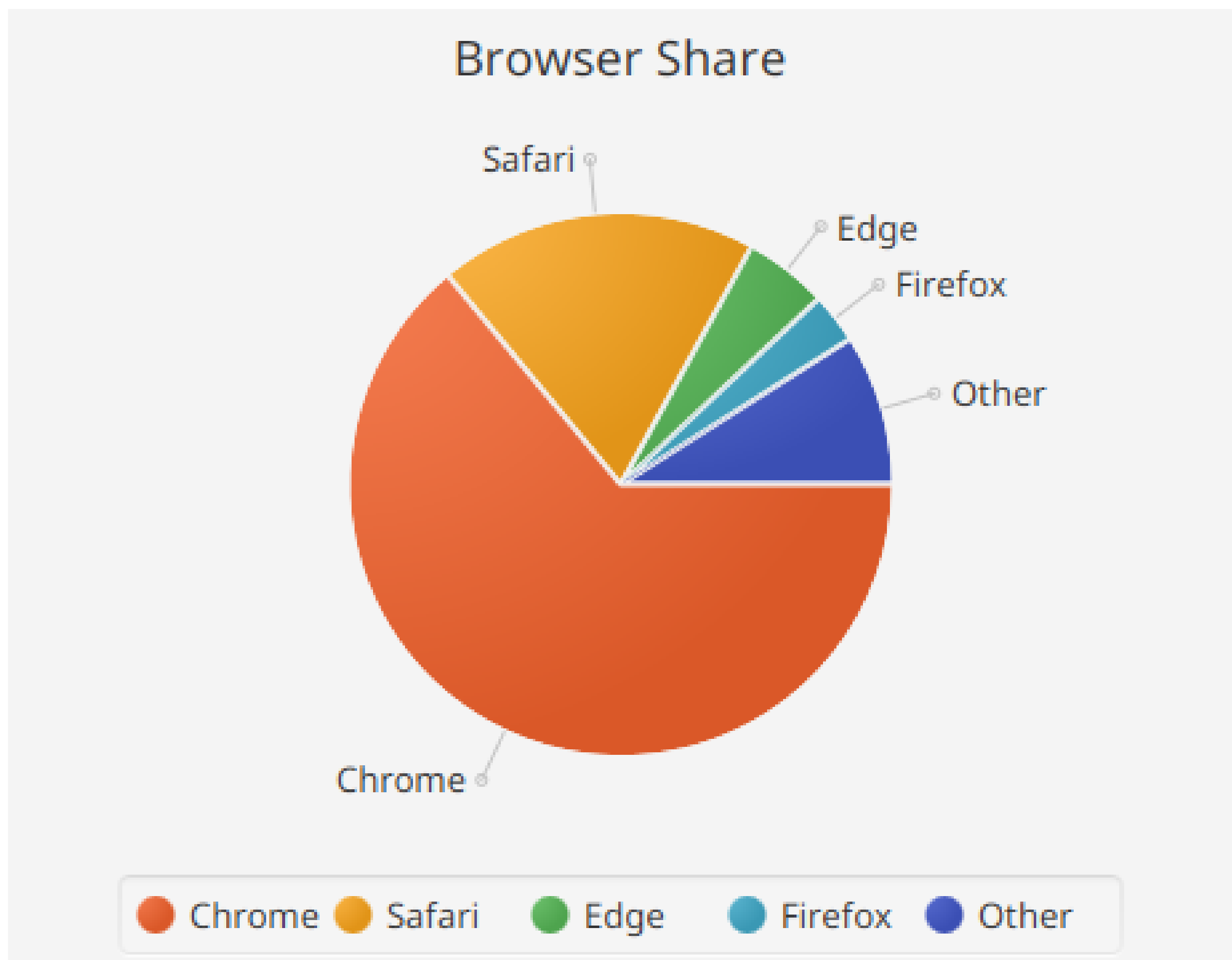


Figure 112: A pie chart of browser share with wedges for Chrome, Safari, Edge, Firefox, and Other

Titles, Legends, and Other Touches

Every chart shares a base (**Chart**) with a handful of common settings:

- **setTitle** sets the heading above the chart (used in every example above).

- **`setLegendVisible(false)`** hides the legend, and **`setLegendSide`** moves it to a different edge with an `fxside` value (`fxside.TOP`, `fxside.BOTTOM`, `fxside.LEFT`, `fxside.RIGHT`).
- **`setAnimated(false)`** turns off the animation charts play when their data changes.

The X/Y charts add a few more, such as **`setVerticalGridLinesVisible`** and **`setHorizontalGridLinesVisible`** to show or hide the faint grid behind the data.

Recap

- Every X/Y chart is built the same way: make two **axes**, make the chart from them, then **`addSeries`**. A **`CategoryAxis`** holds text labels; a **`NumberAxis`** holds numbers.
- A **`ChartSeries`** is a named group of points added with **`addPoint(x, y)`**; the name appears in the legend.
- Pick the type by its name: **`BarChart`** / **`StackedBarChart`** for columns, **`LineChart`** / **`AreaChart`** / **`StackedAreaChart`** for trends, **`ScatterChart`** / **`BubbleChart`** for raw points (a bubble adds a size).
- A **`PieChart`** is different: no axes, just **`PieSlice`** (label, value) wedges added with **`addSlice`**.
- For a **time series**, there is no special date axis: format each date into a label with the `Date` type's **`format`** and use a **`CategoryAxis`**, exactly like the months in the temperature chart.
- The shared **`Chart`** base gives every chart **`setTitle`**, **`setLegendVisible`**, **`setLegendSide`**, and **`setAnimated`**.

You can now turn data into a picture. In the next chapter we switch gears entirely to **rich text editing**, controls that go beyond plain text boxes to syntax highlighting and styled, formatted documents.

Part X

Rich Text Editing

Chapter 31 - Rich Text Controls

The `TextArea` from Chapter 9 shows text in one uniform font, which is all most apps need. But sometimes you want **more**: a code editor that colors keywords as you type, a box that mixes bold and colored words in one line, or a full document editor with a formatting toolbar. **Part X** is about those three controls.

Every example in this chapter lives here: `31-rich-text`.

CodeArea, Syntax-Aware Editing

A **`CodeArea`** is a text editor built for code. Out of the box it uses a monospaced font and can show **line numbers**; its special power is that you can **color the text** based on what it means, keywords one color, strings another. That coloring is called **syntax highlighting**.

You turn it on with **`setSyntaxHighlighter`**, handing it one of your own methods. JavaFX runs that method every time the text changes, and inside it you paint ranges of characters with **style classes** using `setStyleClass(from, to, className)`:

```
editor = new CodeArea();
editor.setLineNumbers(true);
editor.setSyntaxHighlighter(::onEdit);   // run onEdit on every change
editor.appendText(code);
```

The highlighter itself does two things: it wipes the old colors with `clearStyle`, then finds each keyword and paints it. One wrinkle: you must not restyle text *during* a text change, so the handler defers the work a moment with `fx.runLater` (the function from Chapter 2):

```
public onEdit(text) {
    fx.runLater(::rehighlight);      // apply colors just after the edit
                                     // settles
}

public rehighlight() {
    text = this.editor.getText();
    this.editor.clearStyle(0, this.editor.getLength());   // clear old
                                                          // colors
    for (kw : ["class", "public", "private", "return", "if", "else", "new"]) {
        for (pos : this.findAll(text, kw)) {              // findAll: a
                                                          // small
```

```
                                                          // helper
            this.editor.setStyleClass(pos, pos + kw.length(), "keyword");
        }
    }
    // ... and the same for strings, tagged "string" ...
}
```

Those class names, `"keyword"` and `"string"`, get their actual colors from a stylesheet, exactly like the CSS from Chapter 16:

```
.keyword { -fx-fill: #0000cc; -fx-font-weight: bold; }
.string  { -fx-fill: #067d17; }
```

Attach the stylesheet with `app.addStyleSheet(...)` and the editor lights up:

```
1   class Counter {
2       private count = 0;
3
4       public increment() {
5           this.count = this.count + 1;
6           return this.count;
7       }
8
9       public label() {
10          if (this.count == 0) {
11              return "empty";
12          } else {
13              return "count: " + this.count;
14          }
15      }
16  }
```

Figure 113: A CodeArea showing an Aussom class with keywords in bold blue, strings in green, and line numbers down the side

The `findAll` helper (finding every spot a keyword appears) lives in the example; it is ordinary text searching. The point is the shape: **on each edit, scan the text and tag ranges with style classes**, and the CodeArea keeps your code colored live as you type.

StyleClassedTextArea and InlineCssTextArea

A `CodeArea` is really a specialized **`StyleClassedTextArea`**, a text area where any range of characters can be tagged with a **style class**. That is the first of two ways to style ranges of text, and the two differ only in *where the styling lives*:

- A **`StyleClassedTextArea`** tags ranges with named **style classes** (`setStyleClass(from, to, "warning")`) whose looks come from a **stylesheet**, best when many ranges share the same style.
- An **`InlineCssTextArea`** tags ranges with a **string of CSS** right on the spot (`setStyle(from, to, "-fx-fill: red;")`), best for one-off styling with no stylesheet to manage.

Here is an `InlineCssTextArea` mixing several styles in a single box. The helper appends a piece of text, then styles exactly the range it just added:

```
area = new InlineCssTextArea();

start = area.getLength();
area.appendText("can be bold");
// style that range
area.setStyle(start, area.getLength(), "-fx-font-weight: bold;");
```

Rich text **can be bold**, colored, big and blue, or *italic*, all in one box.

Figure 114: One text box mixing plain, bold, red, large blue, and italic words on a single line

The CSS you use here is the JavaFX text kind: **`-fx-fill`** for color, **`-fx-font-weight`** for bold, **`-fx-font-size`** for size, **`-fx-font-style`** for italic, and so on.

HtmlEditor, WYSIWYG HTML

The last control is the heaviest. An **`HtmlEditor`** is a complete **what-you-see-is-what-you-get** editor: a toolbar of formatting buttons (bold, lists, alignment, fonts) above an area where the user types styled text, just like a mini word processor. Under the hood it stores everything as HTML.

You set its starting content with **`setHtmlText`** and read back whatever the user wrote with **`getHtmlText`** (which returns an HTML string you can save):

```
editor = new HtmlEditor();
editor.setHtmlText("<h2>Meeting Notes</h2><p>Discuss the
↪   <b>roadmap</b>.</p>");
```

Like a `WebView` (Chapter 21), an `HtmlEditor` must be built on the **UI thread**, so the example creates it inside `fx.runLater`. The result is a full rich-text editor, toolbar and all:

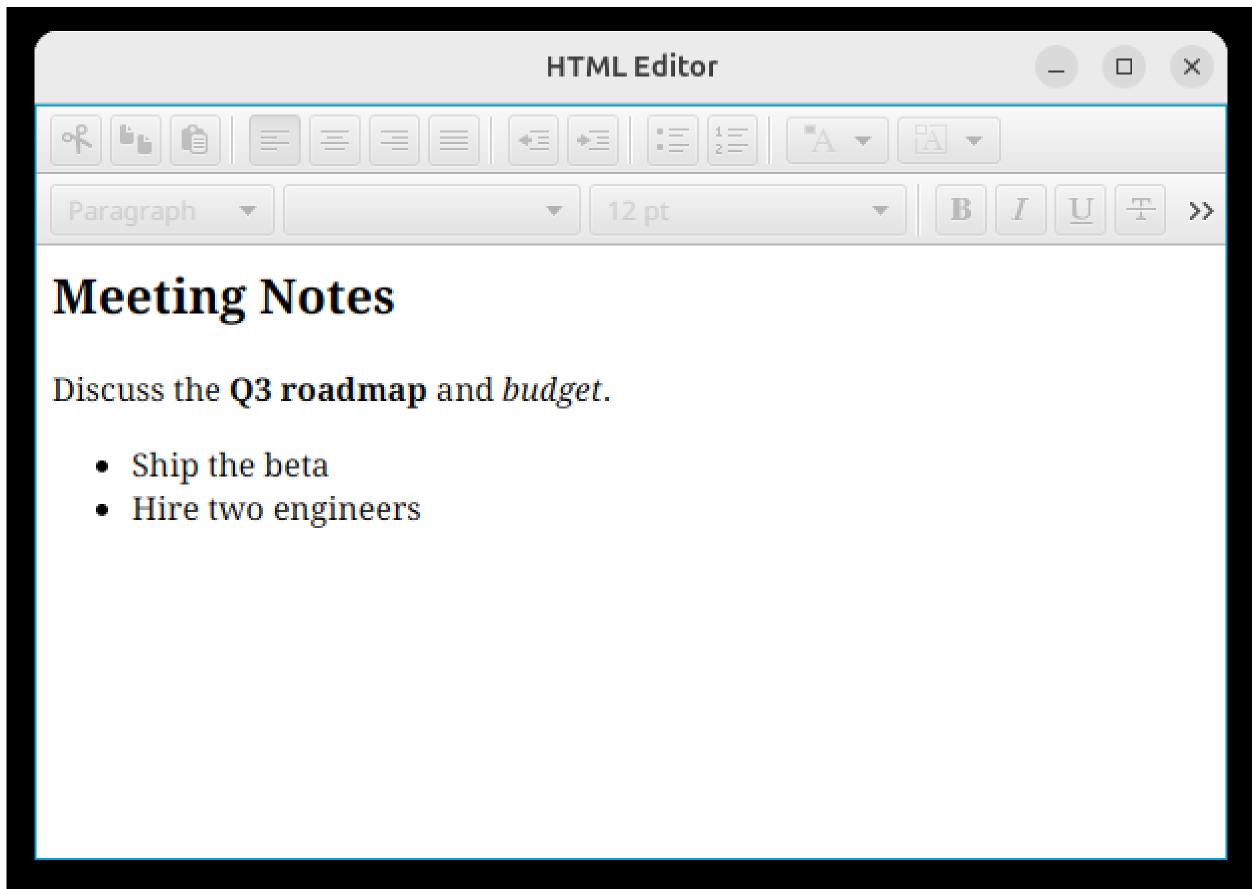


Figure 115: An HtmlEditor with a formatting toolbar above rendered rich text: a heading, a paragraph with bold and italic words, and a bulleted list

Recap

- A **`CodeArea`** is a code editor with **`setLineNumbers`** and syntax highlighting: hand **`setSyntaxHighlighter`** a method that runs on each edit and paints ranges with **`setStyleClass(from, to, class)`**, whose colors come from a stylesheet.
- It extends **`StyleClassedTextArea`** (style ranges with named **classes**); its cousin **`InlineCssTextArea`** styles ranges with **inline CSS** (`setStyle(from, to, css)`) and needs no stylesheet.
- An **`HtmlEditor`** is a full WYSIWYG editor; set content with **`setHtmlText`**, read it back with **`getHtmlText`**, and build it on the UI thread like a `WebView`.

Those are the specialized controls for editing text. In the final part before we ship an app, the

next chapter takes a wider view: the big **third-party control libraries** (ControlsFX and the HanSolo components) that add dozens more ready-made widgets.

Part XI

Extended Controls

Chapter 32 - Beyond the Core Controls

Everything so far has used **core JavaFX**, the controls that ship with the toolkit. That is a lot, but it is not everything. Two large, well-loved third-party libraries add dozens more ready-made widgets, and both are wrapped in the Aussom `fx` bindings so you can use them the same friendly way. This chapter is a **tour**, not a deep dive: enough to know what is out there and where to reach for it.

Every example in this chapter lives here: `32-extended-controls`.

ControlsFX at a Glance

ControlsFX is the best-known JavaFX add-on library. It fills gaps in the core set with controls people reach for constantly. In the bindings its controls are grouped into a few modules:

- **Basic controls** (`fx.controlsfx.BasicControls`), a **`ToggleSwitch`** (an on/off switch, nicer than a check box for settings) and a **`RangeSlider`** (a slider with *two* thumbs, for picking a low-to-high range).
- **Advanced controls** (`fx.controlsfx.AdvancedControls`), a **`Rating`** (the familiar star rating), a **`TaskProgressView`** (a list of running tasks each with its own progress bar), and a **`PropertySheet`** (an auto-built editor for a set of properties).
- **Layout controls** (`fx.controlsfx.LayoutControls`), a **`StatusBar`** for the bottom of a window, a **`MaskerPane`** (a “please wait” overlay that covers the UI while something loads), and a **`MasterDetailPane`** (a list beside a detail view).
- **Selection controls** (`fx.controlsfx.SelectionControls`), a **`CheckComboBox`** and a **`CheckListView`** (a combo box and a list whose items each have a check box), plus **`CheckBoxTreeItem`** for checkable trees.
- **FontAwesome** (`fx.controlsfx.FontAwesome`), the Font Awesome icon set as **`Glyph`** nodes, so you can drop a crisp scalable icon onto any button or label.

Here are four of them in one window, a toggle switch, a star rating, a range slider, and a check combo box, each built with a plain constructor like any core control:

HanSolo Components at a Glance

The second family comes from **Gerrit Grunwald** (known online as *HanSolo*), whose libraries are famous for looking gorgeous. Three are wrapped, and instead of plain constructors they use a **fluent builder**: you chain settings and finish with `build()`, which hands back a node you drop into your scene.

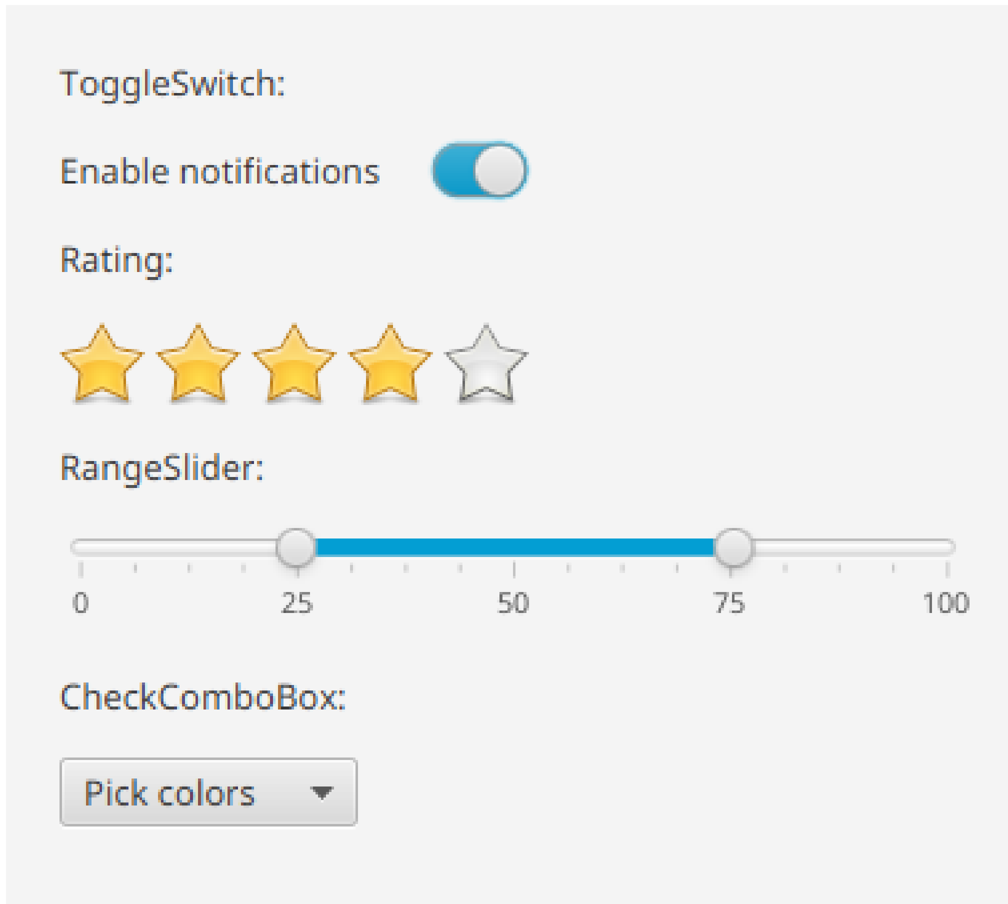


Figure 116: A window showing a ControlsFX ToggleSwitch turned on, a 4-of-5 star Rating, a RangeSlider with two thumbs, and a CheckComboBox

Medusa (`fx.hansolo.medusa.medusa`) is a library of **gauges and clocks**, the kind of dial you would put on a dashboard. You build one from `newGaugeBuilder()`:

```
gauge = medusa.newGaugeBuilder()
    .skinType("GAUGE")      // one of many built-in looks
    .minValue(0.0).maxValue(100.0).value(72.0)
    .title("CPU").unit("%")
    .build();
```

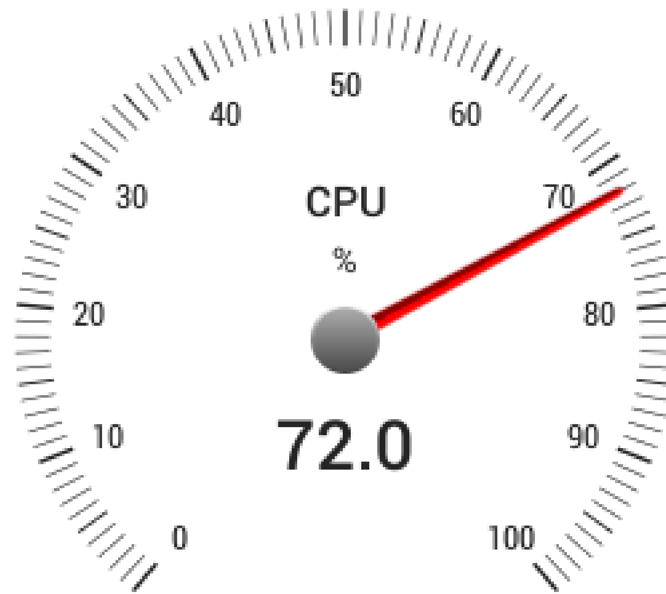


Figure 117: A Medusa gauge dial reading 72% with a red needle, titled CPU, on a 0 to 100 scale

TilesFX (`fx.hansolo.tilesfx.tilesfx`) builds **dashboard tiles**, self-contained widgets that each show one number, gauge, chart, or map, the building blocks of a monitoring dashboard. It uses the same builder style through `newTileBuilder()`:

HanSolo Charts (`fx.hansolo.charts.charts`) rounds out the family with fancier chart types than the core `Chart` classes from Chapter 30, built with `newXYSeries(...)` and friends. Reach for it when the built-in charts are not enough.

Where to Go Next: the API Reference Docs

This guide has walked you through JavaFX from a first empty window to 3D scenes, charts, and these extended libraries, but it has shown **one path** through each topic, not every method. Every control has more settings than we could cover, and there are more controls than fit in a book.

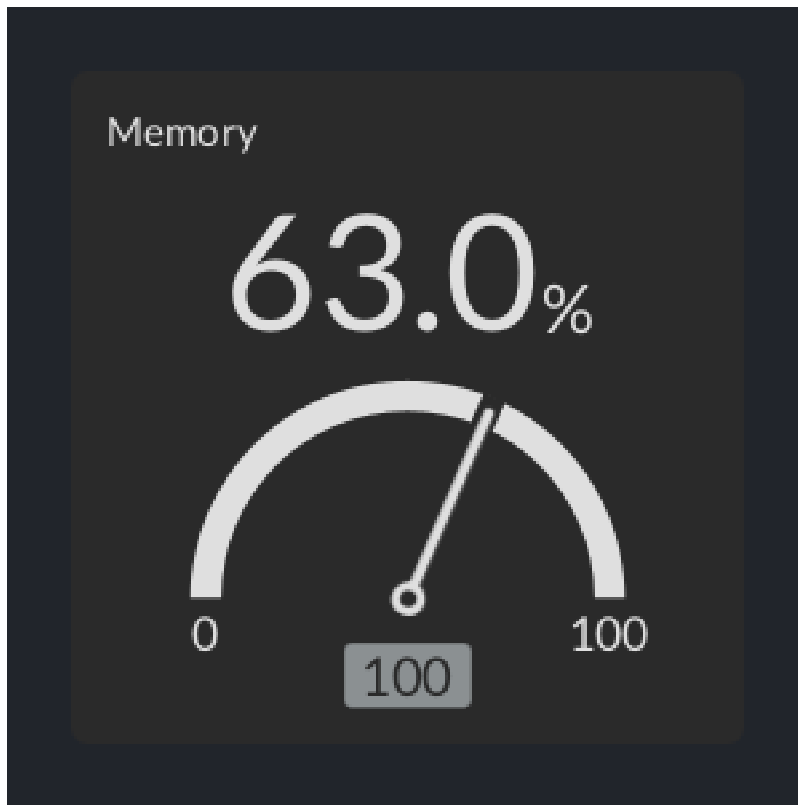


Figure 118: A dark TilesFX dashboard tile titled Memory showing 63% with a gauge arc

When you need the full picture, the **API reference documentation** at aussom-lang.com lists every class in the `fx` bindings, core, ControlsFX, and HanSolo alike, with every method and property. The habit that will serve you best from here on is: **build from the patterns in this book, and dip into the reference for the exact method name or argument when you need it.**

Recap

- **ControlsFX** adds many everyday controls, grouped into **basic** (`ToggleSwitch`, `RangeSlider`), **advanced** (`Rating`, `TaskProgressView`, `PropertySheet`), **layout** (`StatusBar`, `MaskerPane`, `MasterDetailPane`), **selection** (`CheckComboBox`, `CheckListView`), and **FontAwesome** icons (`Glyph`).
- The **HanSolo** libraries add beautiful widgets built with a **fluent builder**: **Medusa** gauges and clocks (`newGaugeBuilder`), **TilesFX** dashboard tiles (`newTileBuilder`), and **HanSolo Charts**.
- For everything this book did not cover, the **API reference** at aussom-lang.com is the complete map.

That closes our tour of the controls. In the final part, we step away from building screens and look at **shipping**: how to package your finished application into an installer people can download and run.

Part XII

Shipping Your Application

Chapter 33 - Packaging and Distribution

You have been running your apps with `aussom myapp.aus`, which is perfect while you build them. But that command needs Aussom (and a Java runtime) installed first, so you cannot just hand the `.aus` file to a friend and expect it to run. To **ship** your app, you turn it into a **native installer**, a `.deb`, `.msi`, or `.pkg` that someone double-clicks to install, with everything it needs bundled inside. This is **Part XII**, and it is the last step before your app is a real, downloadable program.

The example for this chapter lives here: `33-packaging`.

Aussom does the packaging with a separate command-line tool called **`apac`** (the Aussom package tool). It reads a small config file, gathers your app together with a Java runtime and JavaFX, and calls the JDK's `jpackage` to produce the installer:

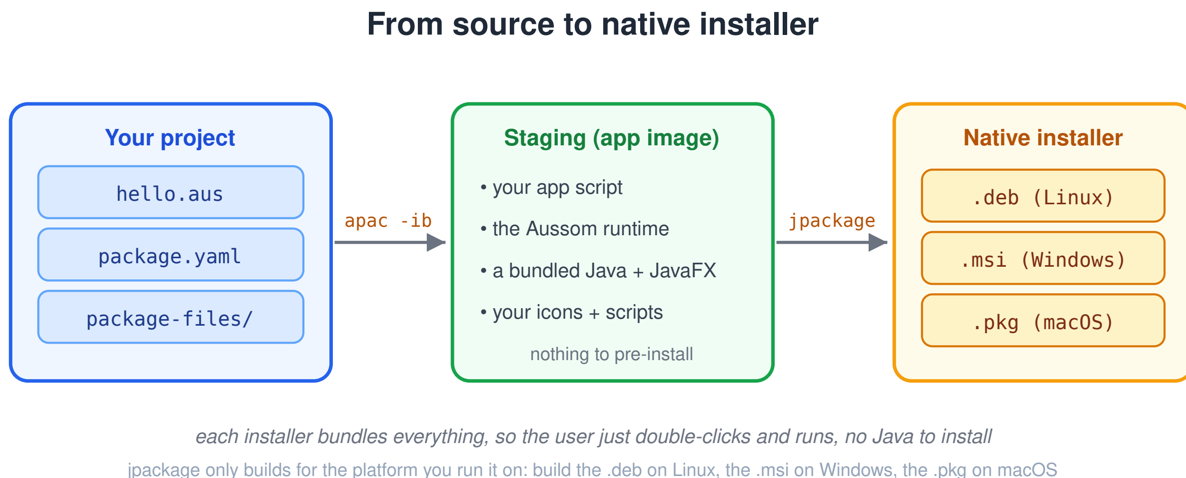


Figure 119: A pipeline diagram: your project (hello.aus, package.yaml, package-files) goes through apac -ib to a staging app image, then jpackage produces a .deb, .msi, or .pkg

The app we will package is deliberately tiny, a window with a greeting, because the point of this chapter is the *packaging*, not the app:

The `package.yaml` File

Everything about your installer is described in one file, **`package.yaml`**. You do not write it from scratch; **`apac -ii`** (installer-init) scaffolds it for you, naming your app:

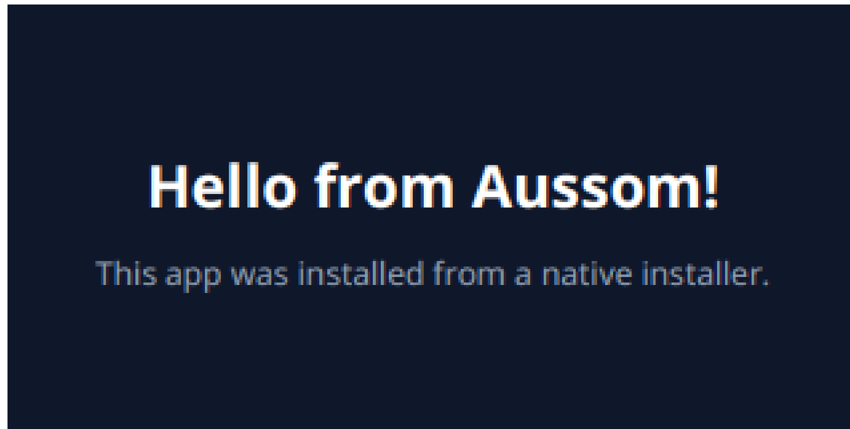


Figure 120: A small dark window reading “Hello from Aussom!” with the subtitle “This app was installed from a native installer.”

```
apac -ii hello
```

That writes a `package.yaml` and a `package-files/` folder (for icons and other assets, covered below). The `package.yaml` has two parts. The top is basic project info; the **`installer`** block is where the packaging settings live:

```
name: hello
version: 1.0.0
description: A small JavaFX greeting app, packaged as a native installer.
license: Apache-2.0
author: Austin Lehman

installer:
  enabled: true
  appName: hello                # the installed program's name
  version: 1.0.0
  vendor: Aussom Examples       # shown in the installer and OS
                                # "installed apps" list
  copyright: Copyright 2026 Austin Lehman
  description: A small JavaFX greeting app built with Aussom.
  mainScript: hello.aus         # the .aus file to run when the app
                                # starts
  appArgs: [ ]                  # command-line arguments to pass your
                                # app
  javaOptions:                  # options for the bundled Java runtime
  - --enable-native-access=ALL-UNNAMED
  - --add-opens=java.base/jdk.internal.module=ALL-UNNAMED
  linux:
    types: [ deb ]              # which installer files to build on
                                # Linux
    shortcut: true              # add a desktop/menu shortcut
```

```
    bundle:
      fx: true                         # bundle JavaFX (see the last section)
  windows:
    types: [ msi ]
    ...
  macos:
    types: [ pkg ]
    ...
```

The key fields are **`mainScript`** (which `.aus` starts your app) and, under each platform, **`types`** (which installer files to build). Each platform, `linux`, `windows`, and `macos`, has its own block so you can pick a different installer format and options for each.

Building Native Installers

With the config in place, one command builds the installer, **`apac -ib`** (installer-build):

```
apac -ib
```

It walks through staging your app, bundling the runtime, and calling `jpackage`:

```
→ Loading package.yaml
→ Building for platforms: [LINUX]
→ Staging for LINUX
→ Invoking jpackage for LINUX/deb
✓ Build complete; output in .../target/installer
```

The finished installer lands in **`target/installer/<platform>/`**, for our app, `hello_1.0.0_amd64.deb`. It is large (over 100 MB) because it carries a complete Java runtime and JavaFX inside, which is exactly what lets the user run it with **nothing pre-installed**. Installing it lays the app down under `/opt/hello/` with a launcher on the path, so the user can run `hello` or click its shortcut.

Two commands help while you work:

- **`apac -ib --dry-run`** prints the exact `jpackage` command it *would* run, without building anything, handy for checking your config.
- **`apac -ib -t deb`** builds just one installer type (here `deb`); the types are `deb`, `rpm`, `msi`, `exe`, `pkg`, `dmg`, and `app-image`.

One important limit: `jpackage` can only build an installer for the **platform it runs on**. You build the `.deb` on Linux, the `.msi` on Windows, and the `.pkg` on macOS, so shipping for all three usually means a build step on each (often a per-platform CI job), all reading the same `package.yaml`.

Bundling JavaFX and Extra Resources

A plain Aussom app needs only the Java runtime, but a **JavaFX** app also needs JavaFX, and that is not bundled unless you ask. In each platform's block, set **`bundle.fx`** to `true`:

```
linux:
  bundle:
    fx: true    # include JavaFX in the installer
```

Leave it `false` and a JavaFX app will install but crash on launch with no toolkit to draw with, so for every app in this book, set it to `true`. There is a matching `bundle.gtk4` for GTK apps.

The **`package-files/`** folder holds everything else the installer needs, and `apac` scaffolds it with a README explaining each part:

- **`icons/`** - drop `app.png` (Linux, 1024x1024), `app.ico` (Windows), and `app.icns` (macOS), then uncomment the `icon:` line in the matching platform block.
- **`deb/`** - Debian maintainer scripts (`postinst`, `prerm`) that run on install and removal; the scaffold's `postinst` links the launcher onto the user's path.
- **`lib/`** - native libraries (`.so` / `.dll` / `.dylib`) your app loads at runtime.
- **`license.txt`** - the license text the installer shows.

And back in `package.yaml`, each platform's `bundle` block can carry extra files into the installer:

- **`extraResources`** - a list of files or folders to bundle alongside your app (data files, configuration, images your app reads at runtime).
- **`extraJars`** - extra Java `.jar` libraries to include (each needs a matching license file in `package-files/extra-licenses/`).

Recap

- Ship an app by building a **native installer** with the **apac** tool; the user double-clicks it and runs your app with nothing else to install.
- **`apac -ii <name>`** scaffolds a **`package.yaml`** (the `installer` block: `appName`, `mainScript`, per-platform `types`, and options) and a `package-files/` folder.
- **`apac -ib`** builds the installer into `target/installer/`; **`--dry-run`** previews the command and **`-t <type>`** builds one format. `jpackage` builds only for the **current platform**, so build each OS's installer on that OS.
- Set **`bundle.fx: true`** so JavaFX apps have their toolkit; use `package-files/` for icons, scripts, and native libs, and **`extraResources`** / **`extraJars`** for anything else your app needs.

You can now turn any app in this book into an installer. In the final chapter we put the whole book together: we build one **complete application** from scratch and carry it all the way from source to a finished installer.

Chapter 34 - A Complete Application

Every chapter so far has taught one piece. This one puts the pieces together. We will build a small but **complete** application from scratch, a task list, organize it the way a real project is organized, and carry it all the way to a finished installer. Nothing here is new; it is the whole book working at once.

The example for this chapter lives here: `34-complete-app`.

Putting the Pieces Together

The app lets you type a task, add it to a list, and remove the one you have selected, with a live count at the bottom. It is deliberately modest, but it draws on a good slice of the book:

Look at what is in that one screen: a **`VBox`** and **`HBox`** for layout (Chapters 4-6), a **`TextField`** and **`Button`** and **`Label`** (Chapters 7-9), **events** wiring the buttons to methods (Chapter 8), a **`ListView`** holding the tasks (Chapter 11), and **CSS** making it look finished (Chapter 16). The interesting part is how it is put together. The UI code stays short because the data lives somewhere else:

```
public onAdd(event) {
    this.store.add(this.field.getText());   // ask the store to add it
    this.field.clear();
    this.refresh();                           // redraw from the store
}

public refresh() {
    this.list.setItems(this.store.all());
    this.countLabel.setText(this.store.count() + " tasks");
}
```

The button handlers never touch a list directly; they call a **`store`** that owns the data. That store is a separate class in a separate file.

Project Layout and Structure

Up to now every example has been a single `.aus` file. A real app grows past that, so Aussom lets you split it into several files, **one class per file**, and pull one into another with **`include`**, exactly how you have included `fx` modules all along:

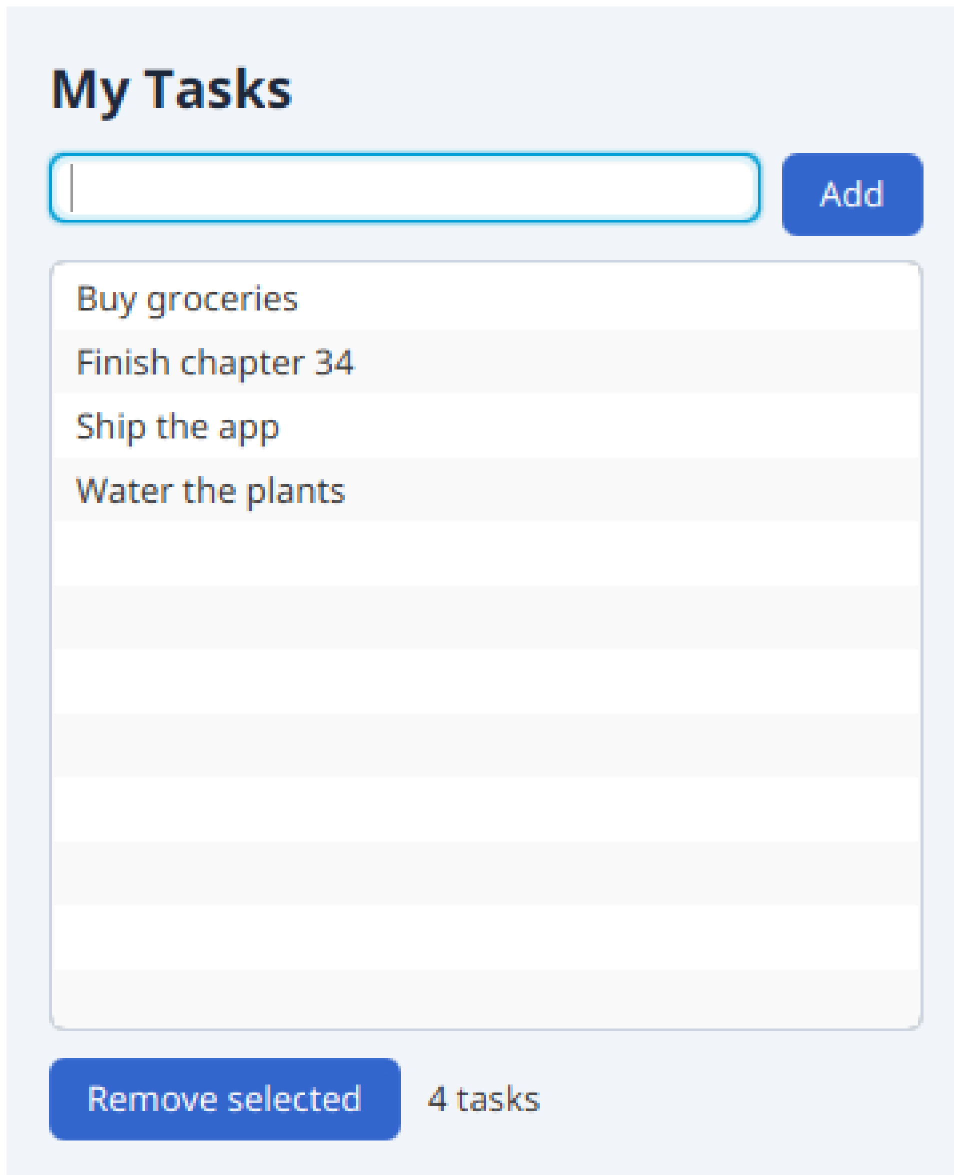


Figure 121: A styled task list window: a “My Tasks” heading, a text field with an Add button, a list of four tasks, and a “Remove selected” button with a “4 tasks” count

```
include TaskStore;    // loads TaskStore.aus from the same folder

// ... later ...
this.store = new TaskStore();
```

`TaskStore.aus` is a plain class with no UI at all, just the list and the operations on it (`add`, `removeAt`, `all`, `count`). Keeping the **data and logic** apart from the **screen** is the single most useful habit for a growing project: each file stays small, and you can change how tasks are stored without touching the UI.

So the finished project is a handful of focused files, the app itself plus the two files that describe its installer:

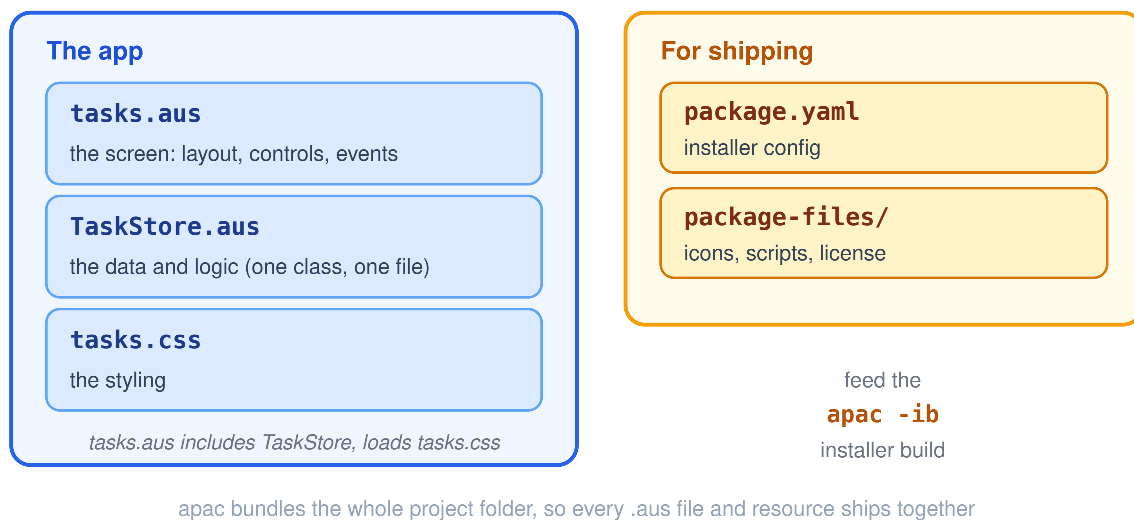


Figure 122: A diagram of the project layout: the app group (tasks.aus, TaskStore.aus, tasks.css) and the shipping group (package.yaml, package-files) feeding apac -ib

From Source to Installer

With the app written, shipping it is the flow from Chapter 33, start to finish. First you run it the whole time you are building it:

```
aussom tasks.aus
```

When it is ready, you scaffold the installer config and turn on JavaFX bundling:

```
apac -ii tasks          # writes package.yaml + package-files/
# then set  bundle.fx: true  under each platform in package.yaml
```

And you build the installer:

```
apac -ib -t deb             # ->
                            # target/installer/linux/tasks_1.0.0_amd64.deb
```

That produces a real, installable `tasks_1.0.0_amd64.deb`. Because **apac bundles the whole project folder**, `TaskStore.aus` and `tasks.css` are carried along automatically, the multi-file app and its stylesheet all ship together, with no extra configuration. Install it, and the task list is a program in the menu like any other, with no Aussom or Java for the user to set up first.

That is the entire journey this book set out to make: **from a blank window in Chapter 1 to a finished application anyone can download and run.**

Recap

- A **complete app** is just the book's pieces working together: layout, controls, events, a data control, and styling, wired up around your own logic.
- Split a growing app into **one class per file** and pull them together with **`include`**; keep the **data and logic** (a class like `TaskStore`) separate from the **screen**.
- Ship it with the Chapter 33 flow, **`apac -ii`** to configure, **`apac -ib`** to build, and `apac` bundles every file in the project folder, so multi-file apps and their resources package with no extra work.

That closes the core of the book. You can now design an interface, wire it to behavior, style it, draw and animate, chart data, edit rich text, and package the result into an installer, everything you need to build and ship a real desktop application in Aussom. The next part steps outside JavaFX to look briefly at **FXGL**, the game engine built on top of it, and the appendices are there as a quick reference whenever you need them.

Part XIII

Game Development with FXGL

Chapter 35 - What FXGL Is

This book has been about **applications**, windows, controls, and documents. But JavaFX is also a fine foundation for **games**, and if that is where you want to go next, there is a whole engine waiting for you. This short chapter is a **high-level tour** of it: enough to know what it is and where to start. Games are a big topic, so FXGL really deserves a book of its own; this is just the map.

What FXGL Adds on Top of JavaFX

FXGL is a mature, open-source **game engine** built on top of JavaFX, created by Almas Baim (known online as *AlmasB*). Plain JavaFX gives you the raw ingredients for a game, a drawing surface (Chapter 18), shapes and images, animation (Chapter 24), and keyboard and mouse input (Chapter 8), but you would have to build all the plumbing around them yourself. FXGL *is* that plumbing, and a lot more. Aussom ships it as the **`fxgl`** module, and a game starts with one call:

```
include fxgl;

app = fxgl.gameApp("My Game", 800, 600);   // then register your game
                                            // callbacks
```

Out of the box FXGL hands you the things every game needs:

- a **game loop** that updates and redraws the screen many times a second,
- an **entity system** for the objects in your game, with **physics** and **collision detection** between them,
- rich **input** handling, keyboard, mouse, gamepads, and on-screen virtual controllers and joysticks,
- **asset loading** for images, sounds, music, and levels,
- and higher-level extras like **particles**, **menus** (main menu, pause screen), **save/load**, **achievements**, and simple **AI**.

The Game Loop, Entities, and Components, in Brief

Two ideas make a game engine different from the apps in this book.

The first is the **game loop**. An app mostly *waits*, it sits still until the user clicks or types, then reacts. A game never sits still: it runs a loop that ticks dozens of times a second, and on every tick it reads the input, steps the physics, moves everything a little, and redraws. FXGL runs that loop for you.

The second is how you build the objects in the game. Instead of writing a big class for "player" and another for "enemy," FXGL uses an **entity-component** model. An **entity** is just a game object, and it is really a *bag of* **components**, small, reusable pieces that each add one capability: a position, a view (any JavaFX node), physics, health, your own behavior. You compose an entity by adding components rather than by subclassing:

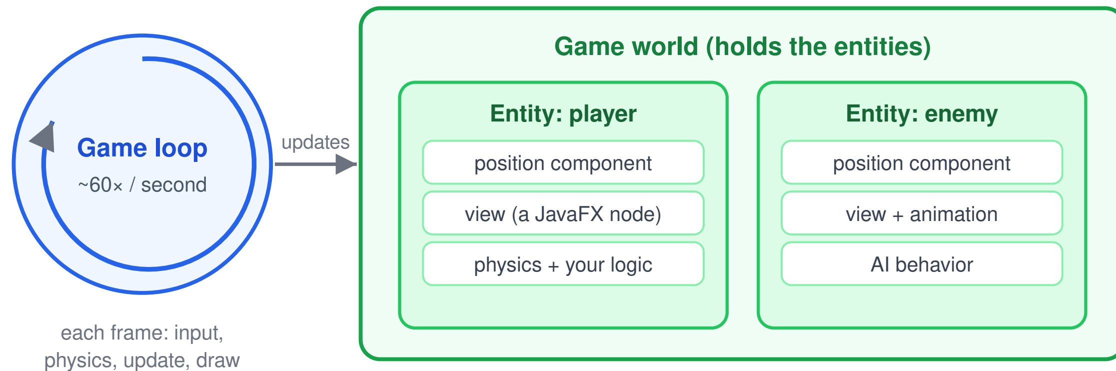


Figure 123: A diagram: a game loop running ~60 times a second updates a game world holding entities, each entity being a bag of components (position, view, physics, behavior)

FXGL runs the loop, keeps the world of entities moving, and routes input and collisions to your code. **Your** job is to define the entities, write the components and rules that make them behave, and react when they collide, win, or lose.

Where to Go Next: the API Reference Docs and Example Games

This is where a full FXGL book would begin. For now, the trail forward:

- The **`fxgl` module reference** at aussom-lang.com lists everything Aussom exposes, `gameApp`, entities and components, input, physics, and the rest.
- **FXGL's own project** (github.com/AlmasB/FXGL) has extensive documentation, a wiki, and a large collection of **sample games** whose ideas carry straight over to Aussom.

The concepts you learned in this book, nodes, layout, events, the canvas, animation, are the same ones FXGL builds on, so you are not starting over. You are just adding a game loop and an entity system on top of everything you already know.

Recap

- **FXGL** is an open-source **game engine** on top of JavaFX (by AlmasB), available in Aussom as the **`fxgl`** module and started with **`fxgl.gameApp(...)`**.

- It provides what games need beyond plain JavaFX: a **game loop**, an **entity-component** system, **physics** and **collisions**, rich **input**, asset loading, and extras like particles, menus, and save/load.
- Build game objects as **entities** made of **components** rather than by subclassing; FXGL runs the loop and the world, you write the pieces and the rules.
- For the full story, see the `fxgl` reference at aussom-lang.com and the FXGL project's docs and sample games.

That is the last of the guide's chapters. What remains are the **appendices**, a quick map of the `fx` module, a table for readers coming from Java, and a troubleshooting list, there to help you whenever you need a fast answer.

Appendix A - The fx Module Reference at a Glance

This is a bird's-eye map of the fx module, the whole surface of JavaFX in Aussom on one page. It is not the full reference (that lives at aussom-lang.com); it is a place to jog your memory: *which area is that in, and which chapter covered it?* Everything is loaded the same way, include fx.<Name>;, and created with new <Name>(...).

The core fx module

A handful of helpers live on the fx module itself (include fx;), not on a wrapper class. These are the pieces that start and steer an app:

Helper	What it does	Chapter
fx.fxApp(title, w, h)	create the application window	1
fx.runLater(::method)	run code on the UI thread	2
fx.shutdown()	end the JavaFX runtime	1
fx.fxDialog(...)	create a custom dialog window	13
fx.actionEventHandler(::method)	build a reusable action handler	8
fx.getClosure(interface, ::method)	bridge to any Java listener interface	8
fx.toHexString(color)	turn a Color into a #RRGGBB string	10

The **fx* enums** also live here (fxpos, fxside, fxorient, fxweight, fxdrawmode, and about thirty more); you pass them as values like fxpos.CENTER.

The wrapper classes, by area

Each area is a group of fx.<Name> classes. Skim for the area you need, then open the chapter for the details.

Layout (Part II) - the panes that arrange nodes: StackPane, HBox, VBox, FlowPane, TilePane, GridPane, BorderPane, AnchorPane, and the grouping and scrolling containers ScrollPane, SplitPane, TabPane, Accordion, Pagination. *(Chapters 4-6.)*

Controls (Part III) - what the user reads and operates: `Label`, `Button` and its family (`ToggleButton`, `RadioButton`, `CheckBox`, `Hyperlink`, `MenuButton`); text input (`TextField`, `PasswordField`, `TextArea`, `Spinner`); choosers (`ChoiceBox`, `ComboBox`, `Slider`, `ColorPicker`, `DatePicker`); progress (`ProgressBar`, `ProgressIndicator`); and the data views `ListView`, `TableView`, `TreeView`. *(Chapters 7-11.)*

Menus, dialogs, and chrome (Part III) - `MenuBar`, `Menu`, `MenuItem`, `ContextMenu`, `ToolBar`, `Tooltip`; `Alert`, `TextInputDialog`, `ChoiceDialog`, `FileChooser`, `DirectoryChooser`. *(Chapters 12-13.)*

Text and color (Part IV) - `Text` and `TextFlow` for drawable, styled text; `Font`; and `Color`, `LinearGradient`, `RadialGradient`, and `Stop` for paint. *(Chapters 14-15.)*

Styling - not a class but a technique: `setStyle`, style classes, and stylesheets (`addStyleSheet`), with live reload via CSSFX. *(Chapter 16.)*

Graphics (Part V) - vector shapes (`Rectangle`, `Circle`, `Ellipse`, `Line`, `Polygon`, `Polyline`, `Arc`, `SVGPath`); the immediate-mode `Canvas`; visual `Effects` (`GaussianBlur`, `DropShadow`, `Glow`, `ColorAdjust`, `Reflection`, and more); `Image` and `ImageView`. *(Chapters 17-19, 22.)*

Media and web (Part VI) - `Media`, `MediaPlayer`, `MediaView` for audio and video; `WebView` for embedding web pages. *(Chapters 20-21.)*

Motion (Part VI) - `Transforms` (`Translate`, `Rotate`, `Scale`, `Shear`, `Affine`) and animation (`Timeline`, and the `...Transition` classes). *(Chapters 23-24.)*

Properties and binding (Part VII) - the observable values `StringProperty`, `BooleanProperty`, the numeric properties, and `ObjectProperty`, plus `bind` / `bindBidirectional`. *(Chapters 25-26.)*

3D (Part VIII) - `SubScene` and cameras (`PerspectiveCamera`, `ParallelCamera`); solids (`Box`, `Sphere`, `Cylinder`) and custom `MeshView / TriangleMesh`; `PhongMaterial`; lights (`AmbientLight`, `PointLight`); `Model3D` for loading model files. *(Chapters 27-29.)*

Charts (Part IX) - `BarChart`, `StackedBarChart`, `LineChart`, `AreaChart`, `StackedAreaChart`, `ScatterChart`, `BubbleChart`, and `PieChart`, with the axes `NumberAxis` and `CategoryAxis` and the data classes `ChartSeries` and `PieSlice`. *(Chapter 30.)*

Rich text (Part X) - `CodeArea`, `StyleClassedTextArea`, `InlineCssTextArea`, and `HtmlEditor`. *(Chapter 31.)*

Extended controls (Part XI) - `fx.controlsfx.*` (ToggleSwitch, Rating, RangeSlider, CheckComboBox, and more) and `fx.hansolo.*` (Medusa gauges, TilesFX tiles, HanSolo charts). *(Chapter 32.)*

Fonts and icons - `fx.fontawesome.fontawesome` provides Font Awesome glyph icons for buttons and labels. *(Chapter 12.)*

Where the full detail is

- **Appendix B** maps each JavaFX class to its `fx` wrapper, for readers coming from Java.
- The **API reference** at aussom-lang.com lists every class, method, and property in the `fx` module, including everything this guide did not have room to cover.

Appendix B - Mapping JavaFX Classes to Aussom Wrappers

If you already know JavaFX, or you are reading JavaFX documentation and want the Aussom equivalent, this table is your bridge. The rule is almost always the same:

> A JavaFX class **javafx.<area>.<Name>** is wrapped as **fx.<Name>**, which you load with include fx.<Name>; and create with new <Name>(...).

So javafx.scene.control.Button becomes fx.Button: include fx.Button; then new Button("OK"). The wrapper keeps the JavaFX name, so once you know the pattern you rarely need to look a class up. The tables below list the common wrappers by area, with the chapter that covers each.

Layout (javafx.scene.layout, javafx.scene)

JavaFX class	Aussom wrapper	Chapter
StackPane	fx.StackPane	5
HBox	fx.HBox	5
VBox	fx.VBox	5
FlowPane	fx.FlowPane	5
TilePane	fx.TilePane	5
GridPane	fx.GridPane	5
BorderPane	fx.BorderPane	5
AnchorPane	fx.AnchorPane	5
Group	fx.Group	17, 27
ScrollPane	fx.ScrollPane	6
SplitPane	fx.SplitPane	6
TabPane / Tab	fx.TabPane / fx.Tab	6
Accordion / TitledPane	fx.Accordion / fx.TitledPane	6
Pagination	fx.Pagination	6

Controls (javafx.scene.control)

JavaFX class	Aussom wrapper	Chapter
Label	fx.Label	7
Button	fx.Button	7
ToggleButton / ToggleGroup	fx.ToggleButton / fx.ToggleGroup	7
RadioButton	fx.RadioButton	7
CheckBox	fx.CheckBox	7
Hyperlink	fx.Hyperlink	7
TextField / PasswordField	fx.TextField / fx.PasswordField	9
TextArea	fx.TextArea	9
Spinner	fx.Spinner	9
ChoiceBox / ComboBox	fx.ChoiceBox / fx.ComboBox	10
Slider / ScrollBar	fx.Slider / fx.ScrollBar	10
ProgressBar / ProgressIndicator	fx.ProgressBar / fx.ProgressIndicator	10
ColorPicker / DatePicker	fx.ColorPicker / fx.DatePicker	10
ListView	fx.ListView	11
TableView / TableColumn	fx.TableView / fx.TableColumn	11
TreeView / TreeItem	fx.TreeView / fx.TreeItem	11
MenuBar / Menu / MenuItem	fx.MenuBar / fx.Menu / fx.MenuItem	12
ContextMenu	fx.ContextMenu	12
ToolBar / ButtonBar / Separator	fx.ToolBar / fx.ButtonBar / fx.Separator	12
Tooltip	fx.Tooltip	12
Alert / DialogPane	fx.Alert / fx.DialogPane	13

Shapes, Text, and Paint (`javafx.scene.shape` / `.text` / `.paint`)

JavaFX class	Aussom wrapper	Chapter
Rectangle / Circle / Ellipse	fx.Rectangle / fx.Circle / fx.Ellipse	17
Line / Polygon / Polyline	fx.Line / fx.Polygon / fx.Polyline	17
Arc / SVGPath	fx.Arc / fx.SVGPath	17
Text / TextFlow	fx.Text / fx.TextFlow	14
Font	fx.Font	14
Color	fx.Color	15
LinearGradient / RadialGradient / Stop	fx.LinearGradient / fx.RadialGradient / fx.Stop	15

JavaFX class	Aussom wrapper	Chapter
Canvas (javafx.scene.canvas)	fx.Canvas	18

Effects (javafx.scene.effect)

JavaFX class	Aussom wrapper	Chapter
GaussianBlur / BoxBlur	fx.GaussianBlur / fx.BoxBlur	22
DropShadow / InnerShadow	fx.DropShadow / fx.InnerShadow	22
Glow / Bloom / SepiaTone	fx.Glow / fx.Bloom / fx.SepiaTone	22
ColorAdjust / Reflection	fx.ColorAdjust / fx.Reflection	22

Images, Media, and Web

JavaFX class	Aussom wrapper	Chapter
Image / ImageView (.image)	fx.Image / fx.ImageView	19
Media / MediaPlayer / MediaView (.media)	fx.Media / fx.MediaPlayer / fx.MediaView	20
WebView (.web)	fx.WebView	21
HTMLEditor (.web)	fx.HtmlEditor	31

Transforms and Animation

JavaFX class	Aussom wrapper	Chapter
Translate / Rotate / Scale (.transform)	fx.Translate / fx.Rotate / fx.Scale	23
Shear / Affine (.transform)	fx.Shear / fx.Affine	23
Timeline (.animation)	fx.Timeline	24
FadeTransition / TranslateTransition / … (.animation)	fx.FadeTransition / fx.TranslateTransition / …	24
ParallelTransition / SequentialTransition / PauseTransition	fx.ParallelTransition / fx.SequentialTransition / fx.PauseTransition	24

3D (`javafx.scene`, `javafx.scene.shape`, `javafx.scene.paint`)

JavaFX class	Aussom wrapper	Chapter
`SubScene`	`fx.SubScene`	27
`PerspectiveCamera` / `ParallelCamera`	`fx.PerspectiveCamera` / `fx.ParallelCamera`	27
`Box` / `Sphere` / `Cylinder`	`fx.Box` / `fx.Sphere` / `fx.Cylinder`	28
`MeshView` / `TriangleMesh`	`fx.MeshView` / `fx.TriangleMesh`	28
`Point3D` (`.geometry`)	`fx.Point3D`	28
`PhongMaterial`	`fx.PhongMaterial`	29
`AmbientLight` / `PointLight`	`fx.AmbientLight` / `fx.PointLight`	29

Charts (`javafx.scene.chart`)

JavaFX class	Aussom wrapper	Chapter
`NumberAxis` / `CategoryAxis`	`fx.NumberAxis` / `fx.CategoryAxis`	30
`BarChart` / `StackedBarChart`	`fx.BarChart` / `fx.StackedBarChart`	30
`LineChart` / `AreaChart` / `StackedAreaChart`	`fx.LineChart` / `fx.AreaChart` / `fx.StackedAreaChart`	30
`ScatterChart` / `BubbleChart`	`fx.ScatterChart` / `fx.BubbleChart`	30
`PieChart`	`fx.PieChart`	30
`XYChart.Series`	`fx.ChartSeries` (in `fx.ChartData`)	30
`PieChart.Data`	`fx.PieSlice` (in `fx.ChartData`)	30

Properties (`javafx.beans.property`)

The property wrappers map to JavaFX's **`Simple...`** implementations:

JavaFX class	Aussom wrapper	Chapter
`SimpleStringProperty`	`fx.StringProperty`	25
`SimpleBooleanProperty`	`fx.BooleanProperty`	25
`SimpleIntegerProperty` / `SimpleLongProperty`	`fx.IntegerProperty` / `fx.LongProperty`	25

JavaFX class	Aussom wrapper	Chapter
`SimpleFloatProperty` / `SimpleDoubleProperty`	`fx.FloatProperty` / `fx.DoubleProperty`	25
`SimpleObjectProperty`	`fx.ObjectProperty`	25

A few things that are not one-to-one

- The **application window** is not a raw JavaFX `Stage`; you create it with the `fx` module's **`fx.fxApp(title, width, height)`** helper (Chapter 1), which sets up the `Stage` and `Scene` for you.
- **Enums** (JavaFX `Pos`, `Side`, `DrawMode`, …) become the `fx*` enums (`fxpos`, `fxside`, `fxdrawmode`, …), passed as values like `fxpos.CENTER`.
- **Extended controls** (ControlsFX, HanSolo) live under `fx.controlsfx.*` and `fx.hansolo.*` (Chapter 32), not under plain `fx.*`.

Appendix C - Troubleshooting and Lessons Learned

Every problem in this list is one the book ran into for real. When something misfires, scan the **symptoms** for what you are seeing; each entry gives the cause and the fix.

Threading and the UI thread

Symptom: an error like "Not on FX application thread; currentThread = main." JavaFX draws on a single **UI thread**, and only that thread may touch the interface. Code in `main`, or in a background task, is *not* on it. Wrap any UI work from those places in **`fx.runLater(::method)`** (Chapter 2). You do **not** need this inside event handlers, they already run on the UI thread (Chapter 8).

Symptom: a `WebView`, `HtmlEditor`, or media player throws a threading error when you create it. These heavier controls must be **built** on the UI thread, not just updated there. Create them inside `fx.runLater` (Chapters 20, 21, 31).

Blank windows and missing content

Symptom: a `WebView` or `HtmlEditor` area is blank, even though the app runs. The web engine only paints when its window is actually **visible on screen**. This most often bites when taking off-screen snapshots; on a normal visible window the content appears (Chapters 21, 31).

Symptom: an installed app shows no window, or exits immediately. A JavaFX app needs JavaFX bundled into its installer. In `package.yaml`, set **`bundle.fx: true`** under each platform; leave it `false` and the app installs but has no toolkit to draw with (Chapter 33).

Symptom: a 3D scene is empty. A `SubScene` shows nothing until you attach a **camera** with `setCamera`, and the camera has to be positioned to see the shapes (pull a `PerspectiveCamera` back with a negative `setTranslateZ`). Remember that in 3D y points **down** and larger `z` is farther away (Chapter 27).

Input and focus

Symptom: a key handler on a layout pane (like a `StackPane`) never runs. Layout containers cannot hold keyboard focus by default, so key events never reach them. Call **`setFocusTraversable(true)`** to make the pane eligible, then **`requestFocus()`** (in an `onShow` handler,

once the window is up) to give it focus. Interactive controls like buttons and text fields are focusable already (Chapter 8).

Building and packaging

Symptom: you can only build one installer type. `jpackage` builds an installer only for the **platform it runs on**. Build the `.deb` on Linux, the `.msi` on Windows, and the `.pkg` on macOS, usually a per-platform CI step, all reading the same `package.yaml` (Chapter 33).

Symptom: `apac -ib` fails on missing tools. The build needs the JDK's `jpackage` (and, per format, tools like `dpkg-deb` or WiX). Use **`apac -ib --dry-run`** to check your configuration and see the exact command it would run before building for real (Chapter 33).

Note: the installer is large (100 MB+). That is expected, it bundles a whole Java runtime and JavaFX so the user needs nothing pre-installed. The generated `target/` folder is disposable; keep it out of version control.

Charts

Symptom: creating a chart or axis throws a "constructor instantiation" error. Chart parts (`NumberAxis`, `CategoryAxis`, the chart itself) must be built **after the JavaFX toolkit has started**, that is, after `fx.fxApp(...)`. In a normal app this just happens, since `fx.fxApp` is the first line of `main` (Chapter 30).

Language and API gotchas

The ternary operator (`? :`) is not supported. Use a plain `if` / `else` instead.

`setPadding` needs an `Insets` object, not a number. The simplest way to add padding is CSS: `node.setStyle("-fx-padding: 10;")` (or four values for per-edge) (Chapter 4).

Prefer the `fx*` enums over raw strings. Writing `fxpos.CENTER` instead of `"CENTER"` means a typo is caught right away rather than failing deep inside JavaFX. The enums cover positions, sides, orientations, draw modes, and more.

`fxApp` wants whole-number sizes. `fx.fxApp(title, width, height)` takes **int** width and height, while a `SubScene` and camera take **doubles**, convert with `value + 0.0` if you are passing sizes around (Chapter 27).

Selection indexes are `-1` when nothing is selected. `getSelectedIndex` on a `ListView` or `TableView` returns `-1` if there is no selection, guard for it before using the value (Chapter 11).

The string type is deliberately small. It has `indexOf` (first match only), `lastIndexOf`, `charAt`, `length`, `contains`, `replace`, and `split`, but no `substring` and no from-a-position `indexOf`. You can still do real text work by walking characters with `charAt` (Chapter 31).

When you are still stuck

- Re-read the relevant chapter's example; every one in this book **runs**, so it is a known-good starting point.
- Check the class in **Appendix B** to find the JavaFX class it wraps, then read that class's JavaFX documentation for behavior this guide did not cover.
- The full **API reference** at aussom-lang.com lists every method and property available.