Beginner's Guide To JavaFX

Natalia Spence

October 22, 2025

Contents

Chapter 1

Introduction

First I'd like to make clear what this is and isn't - this pdf is not to function as a complete, or even near complete documentation of everything in JavaFX; that can be found here. This is solely to guide you through the stuff deemed most important to our group project, and is compiled as spike work, made almost entirely of knowledge gained through spike work. This is also not a Java guide, but a JavaFX guide. Some necessary Java will be explained, such as File and Image I/O and the basics of Lambdas, but for the most part it is assumed you understand Java as a language and are capable of figuring out or researching any code shown here that is not explained as part of this guide.

This pdf is made up of several chapters, each chapter containing several sections. Chapters should be for broad topics, such as setup, events, files and animations, whilst sections will be for more specific focusses, such as a specific class used, or an in depth example. The author of each section or chapter will be displayed with the header.

1.1 JavaFX Installation And Setup

Sometimes JavaFX will work out the box, and that's great! But it often will not do that, so this section will serve as an installation guide, and a setup guide assuming you are using IntelliJ.

First, you're going to need to download JavaFX. I used this website, you simply download the relevant SDK for your OS. Once you've done that, unzip it and put it anywhere in your file system that you can find easily in a moment.

Next you'll need to open up your project in IntelliJ. For the group project, this has already been created, simply open the **gp6** folder in IntelliJ. If you want to use it in another project, follow the guide in the next section to create a blank JavaFX project. If you try to run this, you will probably be given an error saying JavaFX is missing. Click "File > Project Structure > Libraries" and click the + in the top left, then navigate to your JavaFX folder and select the **lib** folder.

Now, go to "File > Settings > Appearance & Behaviour > Path Variables" and click the +. In the Name section, type "`PATH_TO_FX`", and in the value navigate to your JavaFX **lib** folder once again. Click OK to save this.

Finally, you need to add VM options to your run configuration. In the top right of the window, to the left of the run button, there should be a dropdown. It will either have the name of an existing run configuration, or say there is none. You need to click this, and click "Edit Configurations". If you do not have an existing run configuration, click the +, then "Application". For the "Main class" select your uk.aber.dcs.gp6.Main class, then click "Modify options" above that and select "Add VM Options" to make the VM options text box appear. Paste the following line into your VM options:

```
-path=${PATH_TO_FX} --add-modules=javafx.controls
```

With that, JavaFX is installed and set up, and you should be able to use this run configuration to run your program without issues.

1.2 Creating A Project

Setting up a project tends to vary depending on your IDE, but for the purposes of this guide it will be assumed that you are working in Jetbrains IntelliJ from an empty project[1].

The first thing you will need to do, just as with any Java project, is create a $uk.aber.dcs.gp6.Main$ class. The difference here, is that the $Main$ class must extend $javafx.application.Application$ and implement it's *start* method, and the *main* method should only call $launch()$:

```
lass Main extends Application {
rride
ic void start(Stage stage) throws IOException {



ic static void main(String[] args) {
launch();
```

With this code, your program should already be able to run, it just... doesn't do anything. You haven't told it to do anything except launch JavaFX, so it starts JavaFX's background processes and does absolutely nothing else. So the first thing you're going to want to do, would be create a basic UI.

A JavaFX UI is made with a *Stage*. Every JavaFX program has one, and it is already provided to you in the arguments of the *start* method. You simply need to populate it. To do that, you use a *Scene*. A *Scene* contains various UI elements, and is what is displayed by the *Stage* - you can think of a *Scene* as the window itself, and the *Stage* as the program that runs the window, but to create a *Scene*, you need to have something to go on said *Scene*. There are various options for this, but we are going to use a *Pane*.

A *Pane* functions as a group of UI elements, whether they are visible or not, and can be used as the *root* of a scene - the object the entire scene is based on[2].

With this knowledge, you can be able to create a working JavaFX program with a visible (but empty) UI, as per the next page.

[1]IntelliJ offers a JavaFX template for new projects, however that will create a JavaFX project using fxml, which we are going to be avoiding, therefore it causes less confusion to start from an empty project.

[2]This is why each UI in this project will inherit from *Pane*, rather than *Scene*, *Stage*, or anything else.

```
lass Main extends Application {
rride
ic void start(Stage stage) throws IOException {
// Create the Pane to be displayed
Pane root = new Pane();
// Create the Scene using the Pane as the root
// The int arguments are the dimensions of the window, width and height
Scene scene = new Scene(root, 1080, 920);
// Set the window name
stage.setTitle("Hello!");
// Set the current scene
stage.setScene(scene);
// Actually display the screen
stage.show();


ic static void main(String[] args) {
launch();
```

Figure 1.1: An initial JavaFX UI.

Chapter 2

Adding To A Pane

2.1 Shapes

Now you should have a UI, but an empty UI isn't very useful. To that end, we want to add things to the *Pane*, and by extension to the *Scene*. Anything that inherits from *javafx.scene.Node* can be added to a *Pane*. This may seem rather limited at first, but there are a *lot* of classes that inherit from *Node*, if not directly. One example (which will likely be the most used one in this project) is the *Rectangle* class, which creates, you guessed it, a rectangle. By default this will be a solid black rectangle, but that can easily be changed, as will be explained below.

To create a *Rectangle*, you can simply create a new instance of the class, passing in the x, y, *width* and *height* as arguments. Note that the x and y coordinates are of the top left point of the rectangle, not the centre. After that, you can simply call the *getChildren().add(node)* method on the *Pane* to add the *Rectangle* to it. If you want to add multiple *Nodes* at once, use *addAll()* instead of add, and list out the nodes, e.g. *addAll(rectangle1, rectangle2, rectangle3)*.

It's rather common to not want a solid black rectangle, so we can use the *setFill()* method - this can take an *ImagePattern* (which can be made by calling *newImagePattern(image)* with an *Image* object) or a *Color* object, and will set that image or colour as the background for that rectangle. Images will be stretched to fit. For more cohesive image manipulation, you would likely want something like an *ImageView* - for more information, refer to **??** on page ??.

Other notable *Shapes* that can be used in the same manner include *Circle* and *Line*.

```
e
oid start(Stage stage) throws IOException {


reate a rectangle
angle rectangle = new Rectangle(400, 400, 100, 150);
ake it green
angle.setFill(Color.GREEN);
dd it to the Pane
.getChildren().add(rectangle);
```

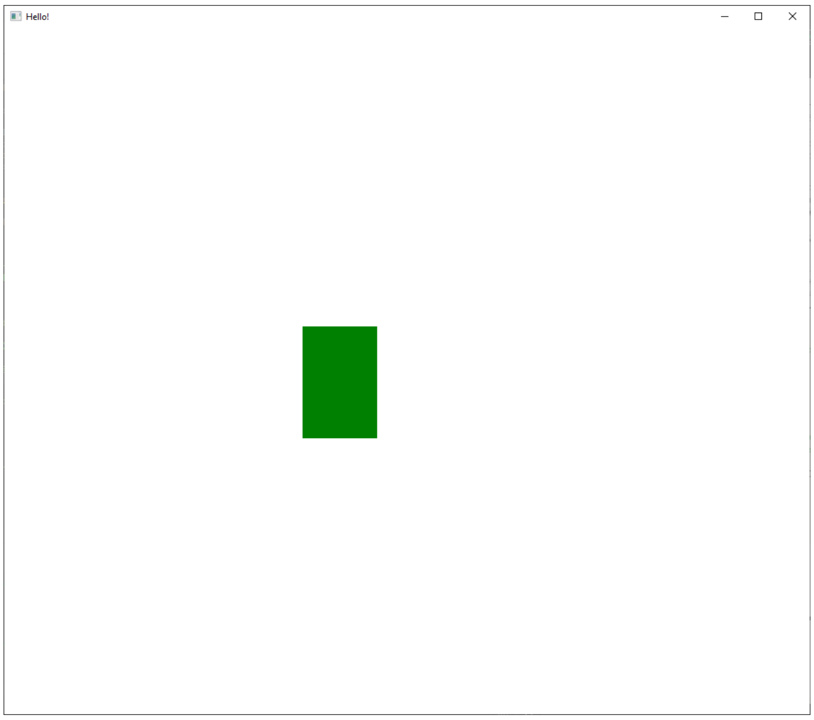


Figure 2.1: A green rectangle!

2.2 Text

Handling text is unfortunately less simple than shapes. Whilst it can be added in exactly the same way, resizing it is far less convenient. Rather than specifying a width and height, then using *getWidth*() and *getHeight*(), you specify a font size, then check the size of the layout bounds of the text object afterwards. For example, if you want to centre some text on the screen:

```
the Text object
w Text("Centre Of Screen Text");
he font size to 40
ont(new Font(40));
e it
(this.getHeight()/2 - (txt.getLayoutBounds().getHeight()/2));
(this.getWidth()/2 - (txt.getLayoutBounds().getWidth()/2));
```

There are also a variety of other methods allowing for you to do various things with the text, such as underlining it or putting a line through it. The documentation for this can be found here.

Chapter 3

File I/O

Chapter 4

Image Manipulation

Chapter 5

Events

5.1 What Are Events?

JavaFX is very event based. This means that rather than looping endlessly to display things and waiting until something changes (like the C assignment), that part is done out of your view in JavaFX's internal code. So we have no endless loop running to make things happen, but we still need a way to do things - this is where events come in.

Rather than endlessly looping, we instead wait for an *event*. There are loads of events, and I would encourage you to look up the documentation for the class in question for a complete list, but some examples include *onMousePressed*, *onMouseDragged* or *onKeyPressed*. These allow us to bind methods to those events, so that when those events happen JavaFX automatically calls the methods we told it to. Events can be used with *Node* as you would expect, but they can also be used with the *Pane* itself to allow you to detect, for example, mouse clicks that are not on an object.

When an event occurs, it travels from the root *Node* of the *Scene* down to the target node, the one that called it. Along the way it goes through all event **filters**. It then starts travelling back, calling event **handlers** for each *Node* as it goes, starting with the *Node* that called it all the way down to the root of the *Scene*. At any point in both filters and handlers, you can stop this by calling *event.consume*(), which will use up the event, preventing it from calling anything else.

There are several ways to bind methods to events:

```
e
oid start(Stage stage) throws IOException {

reate an EventHandler
tHandler handler = new EventHandler<MouseEvent>() {
public void handle(MouseEvent event) {
    System.out.println("Handling event " + event.getEventType());
    event.consume();
}


angle.setOnMousePressed(handler);
```

This option creates an *EventHandler* with the relevant event type (in this case, *MouseEvent*), then creates a function inside it where you can do things. But do we really need a variable for this? The example below works exactly the same but without the additional variable:

```
e
oid start(Stage stage) throws IOException {

angle.setOnMousePressed((new EventHandler<MouseEvent>() {
@Override
```

```java
public void handle(MouseEvent mouseEvent) {
    // Do stuff here
}
```

Both of these methods have the same benefits and drawbacks - it's clear what is happening, but you do not have access to anything outside the event handler, so you cannot use methods from *rectangle* or anything at all except for things you can find inside the *EventHandler*.

The other option is a bit harder to fully understand but much simpler and much more useful:

```java
e
oid start(Stage stage) throws IOException {

angle.setOnMousePressed(mouseEvent -> {
// Do stuff here
```

This makes use of lambdas in order to create a method to handle this event on the fly. This allows all the functionality of the first option, but here you still have access to everything outside the method - you can freely use methods from *rectangle* and anything else you would have access to outside of this method.

5.1.1 Lambdas

This subsection will serve as a quick summary of the quick-start lambda guide linked above. It is not technically necessary to read this for the project, but will help you understand how event handling with lambdas works, and may clear up any confusion the previous section caused.

Lambdas can be confusing to use at first, but they are extremely useful in the right situation. They provide a way to represent a single anonymous method (a method with no name, commonly used when it is known the method will only be called once, often specifically in cases like this) using an expression.

Lambdas have a very simple syntax, since they are designed to be possible with a single readable line of code:

Argument List	Arrow Token	Body
(int x, int y)	->	x + y

The argument list works identically to a normal method, except in some cases where the argument types are already known (such as binding a lambda to an event as per section 5.1), in which case you can simply write the identifiers and omit the types.

The arrow token is simply made up of a dash (−) and a greater than sign (>), and it tells the program that you are defining a lambda using the arguments before it with the function body after it.

Finally you have the lambda's body, which works identically to any function body.

All of the following are valid lambda expressions:

```java
y) -> x + y; // Returns x + y
 Returns 42
> { System.out.println(s); } // Prints a string
```

Lambdas have a large variety of uses, but most of them should not be required for this project, so I will not be writing about them to avoid unnecessarily overcomplicating this section.

5.2 Events Examples

5.2.1 Rotating A Rectangle

Say we want to rotate a rectangle when we hover over it and scroll. We can use the *setOnScroll* method to run code when the *onScroll* event is called. One option is an anonymous method:

```
tOnScroll(new EventHandler<ScrollEvent>() {
oid handle(ScrollEvent event) {
otate shape here
```

But this is long and prevents us from accessing the rectangle itself, which makes it impossible to rotate it. So lets use a lambda instead:

```
tOnScroll(scrollEvent -> {
e shape here
```

With this, we can write code to be ran when the *onScroll* event is called, whilst still having access to all the variables we would outside this method.

So how do we actually rotate the rectangle? All events have a type, in this case *ScrollEvent*. We can use this to access information about how this event was called with the *scrollEvent* argument for this lambda. The documentation for *ScrollEvent* can be found here.

In this case, there is a method in *ScrollEvent* called *getDeltaY*(), which will return the vertical scroll amount that called the event. However, just directly incrementing the rectangle's rotation by this value would cause it to rotate in large intervals with minimal fine control, so we can divide it by 10, or any other value to slow this down.

This results in the following working code:

```
tOnScroll(scrollEvent -> {
e.setRotate(rectangle.getRotate() + scrollEvent.getDeltaY()/10);
```

5.2.2 Dragging A Rectangle

Rotating is a rather simple example, so lets look at a more complicated one - dragging a rectangle around with left click.

JavaFX has two useful events for this - *onMousePressed* and *onMouseDragged*. The former is called once when the mouse is pressed, and the latter continuously for as long as the node is dragged. Both use *MouseEvent*, the documentation for which can be found here.

To be able to move the rectangle, we need to know two things. We need to know where on the rectangle the user has clicked (otherwise we will not know how far to move the rectangle to keep the mouse at the same point on it), and we need to know where the mouse is moving to as it is being dragged.

To find where on the rectangle the user clicks, we will need 2 class variables and the *onMousePressed* event:

```
le anchorX;
le anchorY;
```

```

tOnMousePressed(mouseEvent -> {
eEvent.isPrimaryButtonDown()) {
orX = mouseEvent.getX();
orY = mouseEvent.getY();
```

In this example, we first check that the mouse button being clicked is the primary button, or left click. Since we only want to drag the rectangle on left click, we can ignore any other mouse button. We then get the mouse location and save it in a variable outside the method that will persist and be readable from other events.

Now we need to actually move the rectangle. We can do this with the *setLayoutX* and *setLayoutY* methods, which, contrary to their names, do not set the x and y location of the rectangle. These methods actually set the amount the rectangle should be moved by in the x or y direction to reach its new location. So *setLayoutX*(5) would move the rectangle 5 to the right. This can be implemented rather easily:

```
useDragged(mouseEvent -> {
eEvent.isPrimaryButtonDown()) {
.setLayoutX(mouseEvent.getSceneX() - anchorX);
.setLayoutY(mouseEvent.getSceneY() - anchorY);
```

Again, we only want to drag the rectangle if it is left click being pressed, so we check for that. We then move the rectangle by the difference between where the mouse is, and where it was originally. This will continuously update the rectangle's position to follow the mouse, and it will remain in place once left click is released.

If you needed to check for overlap between this and other $Node$s, you would do so after the *setLayout* calls.

Chapter 6

Animations

Animations in JavaFX can be used to create, well, animations, along with some transitions. An animation will progress in the direction and speed specified by its *Rate*, and will progress until its duration is complete, unless paused. You can also set the *cycleCount* to *Animation.INDEFINITE*, in which case it will repeat the animation forever unless you stop it.

There are some basic controls shared by all types of animations in JavaFX. The important ones are as follows:

```
lay an animation
ill resume if previously paused
ation.play();
lternative method
ill restart if previously paused
ation.playFromStart();

ause an animation
ation.pause();
top an animation
ation.stop();

ill make the animation reverse direction
n alternating cycles
ation.setAutoReverse(true);

efines the number of cycles in this animation
et to Animation.INDEFINITE to repeat indefinitely,
therwise must be a positive integer
ation.setCycleCount(2);

elay the start of the animation
ation.setDelay(Duration.seconds(2));

ontrols the timing for accelleration and
eceleration of the animation
an be set to DISCRETE, EASE_BOTH, EASE_IN, EASE_OUT, LINEAR
efaults to EASE_BOTH
ation.setInterpolator(Interpolator.LINEAR);
```

All animations also have an *onFinished* event (see chapter 5 on page 9) that can be set to run code when the animation completes.

There are 2 types of animations in JavaFX - *Timelines*, and *Transitions*.

Timelines are used to define a free form animation of any JavaFX property, one *KeyFrame* at a time. *Timelines* are mainly useful for more specific animations that the *Transition* subclasses do not cover, for when you cannot plan out the whole animation beforehand, or for non-blocking UI updates that you want to run on a timer (for which you should also look into an *AnimationTimer* before deciding on a *Timeline*).
Timelines still run on the main JavaFX application thread however, so avoid using them for any long tasks; consider whether the task is absolutely necessary - if so, multithreading is possible, but comes with many risks you have to account for. Multithreading is a long and complex topic that will not be covered in this guide. Avoid it as much as possible.

Transitions are much easier to use, and should be what is being used unless a *Timeline* is specifically required. *Transition*, unlike *Timeline*, is an abstract class - this allows you to create your own transition if you need to, however in most cases the existing subclasses will prove sufficient.

The rest of this chapter will go over each *Transition* subclass one at a time. All code examples here are based on or taken from their individual documentation pages.

6.1 TranslateTransition

The *TranslateTransition* simply creates a move animation that takes as long as its duration to complete by updating the *translateX*, *translateY* and *translateZ* variables of the node at regular intervals.

You can specify a specific position for it to start from, or leave it to use the *Nodes* current position instead. Similarly, you can specify x, y and z values to stop at, otherwise it simply moves it by the specified amount.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

e a TranslateTrantition
eTransition tt = new TranslateTransition(Duration.seconds(2), rect);
X(200f);
toReverse(true);

the animation
);
```

6.2 PathTransition

The *PathTransition* works similarly to the *TranslateTransition*, except instead of simply moving from one point to another, it will make the *Node* follow a specified path instead.

You can also set the orientation of the *PathTransition* to $OrientationType.ORTHOGONAL_{T}O_{T}ANGENT$, in which case the rotation of the *Node* will be updated as it moves.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

e a Path
h = new Path();
Elements().add (new MoveTo (0f, 50f));
Elements().add (new CubicCurveTo (40f, 10f, 390f, 240f, 1904, 50f));

e a PathTransition
sition pt = new PathTransition();
ration(Duration.seconds(10));
de(rect);
th(path);
ientation(OrientationType.ORTHOGONAL_TO_TANGENT);
toReverse(true);

the animation
);
```

6.3 ScaleTransition

ScaleTransition will scale the *Node* by a set amount at a regular interval over its duration. You can set limits on the x, y and z scale values similarly to *TranslateTransition*.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

e a ScaleTransition
nsition st = new ScaleTransition(Duration.seconds(2), rect);
X(1.5f);
Y(1.5f);
toReverse(true);

the animation
);
```

6.4 RotateTransition

The *RotateTransition* will rotate its *Node* at a regular interval by an angle specified in degrees. Limits and start position can be set, similarly to other animations.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
```

```
ArcWidth(50);
Fill(Color.VIOLET);

e a RotateTransition
ansition rt = new RotateTransition(Duration.seconds(3), rect);
Angle(180);
toReverse(true);

the animation
);
```

6.5 FillTransition

Onto more niche *Transitions*, the *FillTransition* creates an animation that changes the filling of a *Shape* over a duration. In this situation, the fill must be a *Color*. Limits can be set similarly to other animations.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);

e a FillTransition
sition ft = new FillTransition(Duration.seconds(3), rect, Color.RED,
   Color.BLUE);
toReverse(true);

the animation
);
```

6.6 FadeTransition

A *FadeTransition* creates a fade effect spanning its duration by updating the *Nodes opacity* at regular intervals. Limits can be set similarly to other animations.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

e a FillTransition
sition ft = new FadeTransition(Duration.seconds(3), rect);
omValue(1.0);
Value(0.3);
toReverse(true);

the animation
);
```

6.7 SequentialTransition

Moving onto the more unique *Transitions*, the *SequentialTransition* does nothing by itself. Instead, it allows you to chain together animations, allowing you to play a list of animations one after the other without creating new animations in the previous one's *onFinished* event.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

ration SEC_2 = Duration.seconds(2);
ration SEC_3 = Duration.seconds(3);

e several animations
sition ft = new FadeTransition(SEC_3);
omValue(1.0f);
Value(0.3f);
eTransition tt = new TranslateTransition(SEC_2);
omX(-100f);
X(100f);
ansition rt = new RotateTransition(SEC_3);
Angle(180f);
nsition st = new ScaleTransition(SEC_2);
X(1.5f);
Y(1.5f);

all animations in order
alTransition seqT = new SequentialTransition (rect, ft, tt, rt, st);
y();
```

6.8 ParallelTransition

Another unique *Transition*, *ParallelTransition* works similarly to *SequentialTransition*, except instead of playing the animations one after the other, they are played in parallel, allowing you to execute multiple animations at the same time.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
ArcHeight(50);
ArcWidth(50);
Fill(Color.VIOLET);

ration SEC_2 = Duration.seconds(2);
ration SEC_3 = Duration.seconds(3);

e several animations
sition ft = new FadeTransition(SEC_3);
omValue(1.0f);
Value(0.3f);
eTransition tt = new TranslateTransition(SEC_2);
omX(-100f);
X(100f);
ansition rt = new RotateTransition(SEC_3);
Angle(180f);
nsition st = new ScaleTransition(SEC_2);
X(1.5f);
Y(1.5f);

all animations simultaneously
Transition pt = new ParallelTransition(rect, ft, tt, rt, st);
);
```

6.9 PauseTransition

The simplest by far of the *Transition* subclasses, *PauseTransition* just waits for its duration to finish and then executes its *onFinished* event. This can be used to do a delayed action, or much more commonly with *SequentialAnimation* to add a delay in between multiple *Transitions* in a list, or with *ParallelTransition* to execute code at a certain point in the middle of other animations.

```
e a Rectangle
e rect = new Rectangle (100, 40, 100, 100);
Fill(Color.VIOLET);

e RotateTransition
ansition rt = new RotateTransition(Duration.seconds(3), rect);
Angle(180);

e PauseTransition
nsition pt = new PauseTransition(Duration.seconds(1));

e SequentialTransition for a delayed rotation
alTransition seqTransition = new SequentialTransition (pt, rt);
ition.play();
```

Chapter 7

An In-Depth Example - Draughts

This chapter is going to serve as one large example that will be built on in each subsection in order to create a whole system. For lack of better idea, I'm going to walk through the creation of a JavaFX version of draughts. This is going to follow my thought process as I create this from scratch, and should help show how to put the elements in the chapters above together into one complex system. It's worth noting that some of the logic here is inspired by this chess implementation using Swing.

Is this mostly because I was bored? Yes.

Does it make a really good example? Also yes.

Am I doing draughts because I don't want to deal with image copyright? Still yes.

If you want to skip the game logic and go straight to the parts that involve more than a basic JavaFX window, given that that's what this document is about, head down to any of the following:

- **Displaying On The GUI - section 7.4 on page 26**
- **Movement With JavaFX - section 7.5.2 on page 31**

The creation of the initial project is going to be completely identical to the process outlined in section 1.2 on page 3. Once that has been done, the first thing we will need is the board.

7.1 The Board (And Table)

Draughts is played on a chess board, but not every aspect of the game takes place on the board itself; there are game settings we might want to add, we want the board to be facing a different direction for each player (so for local multiplayer, it will have to flip after each turn), and if we want to get really fancy, we might want to show the taken pieces and a move log to the side. All of this means we can't just use a normal *Pane* to display the board on.

To start off, we should create some packages and empty classes for the project. We need a *gui* package for the gui itself, a *board* package for the *Board*, its *Tile*s and its *Piece*s, and a *turn* package for the moves and teams.

For this example, we're going to have the game itself, and the UI. The UI will provide user input and display, whilst the rest of the code handles all game logic and simply tells the UI what the board looks like at any given moment. We are *not* going to have things move on the UI then update an internal representation - this will be the other way around.

First, we're going to modify the *Table* constructor to take the *Stage* and create the *Scene*, so that the only code required in the *start* method of the *Main* class is creating a table. The *Table* needs a *Board*, so we'll make a variable for that as well.

7.1: Main.Java

```
e
oid start(Stage stage) throws IOException {
e root = new Table(stage);
```

7.2: Table.Java

```
Board board;

able(Stage stage) {
e scene = new Scene(this, 600, 600);
e.setTitle("Draughts!");
e.setScene(scene);
e.show();
```

Next up we need a basic *Board*. The board will need a list of all the *Tile*s on the board, but it will also need to know where all the white and black pieces are for easy access, along with knowing the current player. There will only ever be one board, so the smartest option would be making this a singleton - a class with a private constructor that creates a single instance when the program starts and just returns that instance whenever asked.

7.3: Board.java

```
lass Board {
ate final List<Tile> board = new ArrayList<>();
ate final List<Piece> whitePieces;
ate final List<Piece> blackPieces;
ate final Team currentTeam;
ate static final Board Instance = new Board();

ate Board() {



ic static Board getInstance() {
return Instance;


ic Team getCurrentTeam() {
return this.currentTeam;
```

The board will need to be created in it's standard layout, so we can do that like this:

7.4: Board.java

```java
Board() {
(int i = 0; i < 64; i++) {
this.board.add(new Tile(i, getStartingPiece(i)));

.whitePieces = getActivePieces(this.board, Team.WHITE);
.blackPieces = getActivePieces(this.board, Team.BLACK);
.currentTeam = Team.WHITE;
```

Now we have a lot of unimplemented references, so we need to make them. Starting with the methods in the *Board* class.
We need a method that just makes pieces for the starting layout, so we can do this by checking if each location is in the row, then checking if it is on a square that should have a piece. To get the active pieces for a *Team*, we just need to iterate over the board, get the *Piece* from each *Tile*, and add it to a list if it belongs to the required *Team*.

7.5: Board.java

```java
static Piece getStartingPiece(final int location) {
lack Side
location < 8 && location % 2 == 1)
rn new Piece(Team.BLACK, location);
location >= 8 && location < 16 && location % 2 == 0)
rn new Piece(Team.BLACK, location);
location >= 16 && location < 24 && location % 2 == 1)
rn new Piece(Team.BLACK, location);
hite Side
location >= 40 && location < 48 location % 2 == 0)
rn new Piece(Team.WHITE, location);
location >= 48 && location < 56 && location % 2 == 1)
rn new Piece(Team.WHITE, location);
location >= 56 && location < 64 && location % 2 == 0)
rn new Piece(Team.WHITE, location);
verywhere Else
rn null;


static List<Piece> getActivePieces(final List<Tile> board, final Team
    team) {
l List<Piece> active = new ArrayList<>();
(final Tile tile : board) {
if (tile.isOccupied()) {
    final Piece piece = tile.getPiece();
    if (piece.getTeam() == team) active.add(piece);
}

rn active;
```

Finding the legal moves for pieces is going to mostly be done by the pieces themselves, so the code here just involves iterating over each one:

7.6: Board.java

```
List<Move> findLegalMoves(final Piece piece) {
rn piece.findLegalMoves(this.board);
```

The final thing we need for the board currently is a way to display it. Directly displaying it on the GUI immediately would be nice, but then if something goes wrong with the UI code we have no way to know if that's a problem with the GUI or with the board itself. So, in order to give a nice representation of the board without the GUI, we can override *toString*() for a custom view of the board in the console. We want each *Tile* to give a string representation of itself, and the board should simply use a *StringBuilder* to combine them.

7.7: Board.java

```
e
tring toString() {
l StringBuilder bob = new StringBuilder();
(int i = 0; i < 64; i++) {
bob.append(String.format("%3s", this.board.get(i).toString()));
if ((i + 1) % 8 == 0) {
    bob.append("\n");
}

rn bob.toString();
```

With that, we can move on from the *Board* and start making some *Tiles* and *Pieces*.

7.2 Tiles And Pieces

A *Tile* is pretty simple - it needs to know it's location and the piece on it, and it needs to be able to tell the program that. That's it.

7.8: Tile.java

```
lass Tile {
ate final int location;
ate Piece piece;

ic Tile(final int location, final Piece piece) {
this.location = location;
this.piece = piece;


ic boolean isOccupied() {
return this.piece != null;


ic Piece getPiece() {
return this.piece;


ic int getLocation() {
return this.location;


```

```
rride
ic String toString() {
return this.piece == null ? "-" : this.piece.toString();
```

Yep, that's the whole class. It's that simple. So now moving on to *Piece*.

Draughts has two types of pieces - single and double[1]. If there were more types, such as in chess, it would be worth making an abstract *Piece* class and inheriting from it, but since there is only the two, we can simply store a boolean *isDoublePiece*.

As for the rest, a *Piece* will need to know it's location and it's *Team*, be able to move and be captured, be able to create a list of legal moves for itself, and it'll need to be able to be promoted. Ignoring the moves for now, this can be done as follows:

7.9: Piece.java

```
lass Piece {
ate boolean isDoublePiece = false;
ate final Team team;
ate int location;

ic Piece(final Team team, final int location) {
this.team = team;
this.location = location;


ic void promote() {
this.isDoublePiece = true;


ic boolean isDoublePiece() {
return this.isDoublePiece;


ic void capture(final List<Tile> board) {
this.location = -1;


ic int getLocation() {
return this.location;


ic void setLocation(final int location) {
this.location = location;


ic Team getTeam() {
return this.team;


ic void move(Move move) {
// TODO


ic List<Move> findLegalMoves(final List<Tile> board) {
// TODO
final List<Move> legalMoves = new ArrayList<>();
return legalMoves;
```

[1]There is probably an actual name other than that, but that's how I learnt it so that's what I'm going with.

Piece also needs to override the *toString*() method so we can print the board. This would simply return either *S* or *D* depending on if it is a single or double, and change it to lowercase for one team so we can differentiate them.

7.10: Piece.java

```
e
tring toString() {
this.isDoublePiece)
rn this.team == Team.WHITE ? "D" : "d";
rn this.team == Team.WHITE ? "S" : "s";
```

The last part of *Piece* is finding where it can move. The movement logic itself is going to be covered in another section, but we still want the program to compile, so we need to know what a *Move* is.
A *Move* will require a *Piece*, a start location, an end location, and a list of *Piece*s being captured.

7.11: Move.java

```
lass Move {
ate final Piece movingPiece;
ate final int destination;
ate final Piece pieceCaptured;

ic Move(final Piece movingPiece, final int destination, final Piece
   pieceCaptured) {
this.movingPiece = movingPiece;
this.destination = destination;
this.pieceCaptured = pieceCaptured;
```

We can write the methods for the *Move* class later, for now we just need the constructor so the program is one step closer to compilation.

That's all for *Tile*s and *Piece*s, so we should only need to implement *Team* for the program to compile.

7.3 Teams And Running The Program

For *Team*, we are going to use an enum. An enum is essentially just a labelled integer - in this case, we will have *Team.WHITE* and *Team.BLACK*, but internally that's just a more readable way of saying 0 and 1. This may not seem very useful, but it makes the code much easier to read and in other programs where you could have 20 or more option, it becomes a life saver.

7.12: Team.java

```
num Team {
E,
K
```

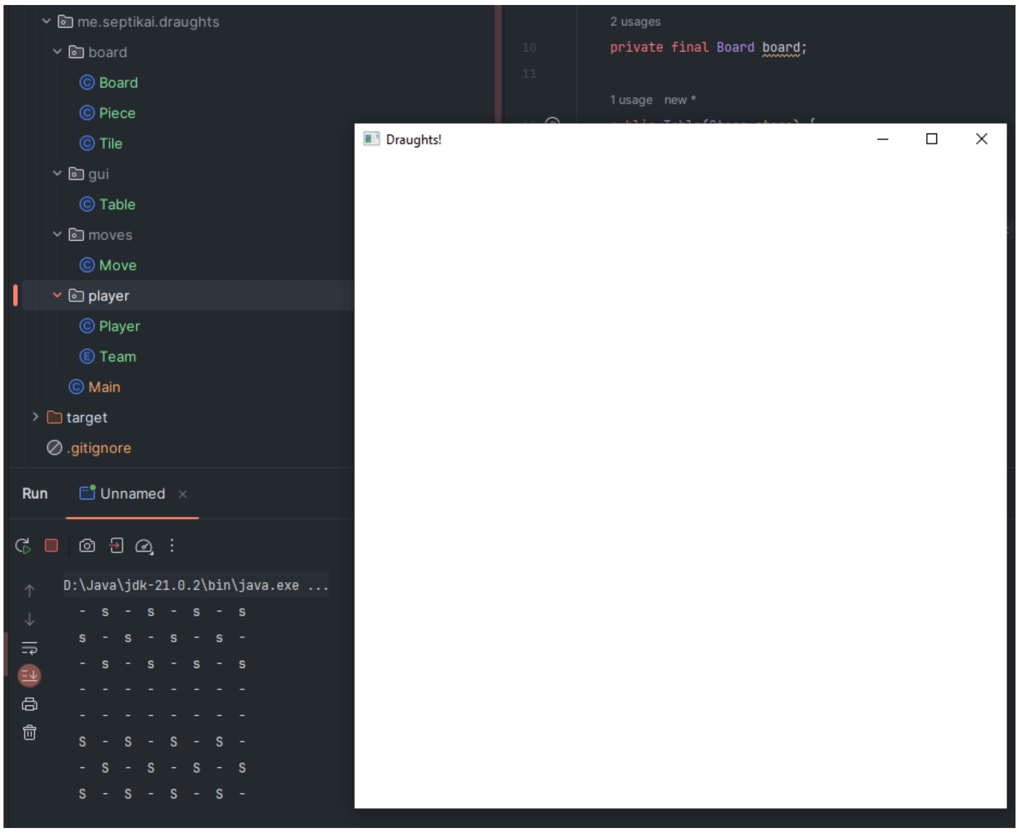
Now, we should have everything needed to run the program. Before we do that though, we need to actually get the board onto the table. So in *Table*, before creating the *Scene*, we should get the instance of *Board* and print it.

7.13: Table.java

```
lass Table extends Pane {

ate final Board board;

ic Table(Stage stage) {
```

```java
board = Board.getInstance();
System.out.println(board);

Scene scene = new Scene(this, 600, 600);
stage.setTitle("Draughts!");
stage.setScene(scene);
stage.show();
```

And just like that, we now have a draughts board being created, filled with the classes we just wrote! They don't show up on the GUI or do anything yet though, so next up is displaying them properly.



```java
2 usages
private final Board board;

1 usage   new *
```

```
D:\Java\jdk-21.0.2\bin\java.exe ...
 - s - s - s - s
 s - s - s - s -
 - s - s - s - s
 - - - - - - - -
 - - - - - - - -
 S - S - S - S -
 - S - S - S - S
 S - S - S - S -
```

Figure 7.1: The program running!

7.4 Displaying On The GUI

We want to get the board, and everything on it, to be displayed on the JavaFX GUI. So lets think about how to do that.

The *Board* is made entirely of *Tile*s - there is no need for anything to be displayed that is part of the *Board* and not part of a *Tile*. So we can make the *Board* inherit from *javafx.scene.Group*.

A *Tile* is going to be a square of a single colour, so inheriting from *javafx.scene.shape.Rectangle* would be a perfect choice here.

Finally, a *Piece* is going to be a coloured circle. It will also need some indicator for if it is doubled up, but we can ignore that for now. So we can inherit from *javafx.scene.shape.Circle*.

7.14: Board.java

```java
lass Board extends Group {
```

7.15: Tile.java

```java
ic class Tile extends Rectangle {
```

7.16: Piece.java

```java
public class Piece extends Circle {
```

This will necessitate modifying the constructors for each of these classes to ensure they display everything properly.

For *Board*, this is as simple as calling *super*() then *getChildren*().*addAll*() to add all the *Tile*s.

7.17: Board.java

```java
    private Board() {
        super();
        for (int i = 0; i < 64; i++) {
            this.board.add(new Tile(i, getStartingPiece(i)));
        }
        this.getChildren().addAll(this.board);
        ...
    }
```

Next is *Tile*. Each tile has to position itself, but each *Tile* also stores it's location, so this is not too hard. We need to call *super*() with the location of each tile. We can determine the x coordinate by doing $location \ \% \ 8$ as there will be 8 tiles per row, then we multiply that by $totalWidth/8$. Since we are using a 600×600 square for the table, $totalWidth$ would be 600. Figuring out the y coordinate works identically, except instead of doing $location \ \% \ 8$, we do $location/8$ and round it down with $Math.floor()$. The width and height are then just $600/8$ as you might expect.

7.18: Tile.java

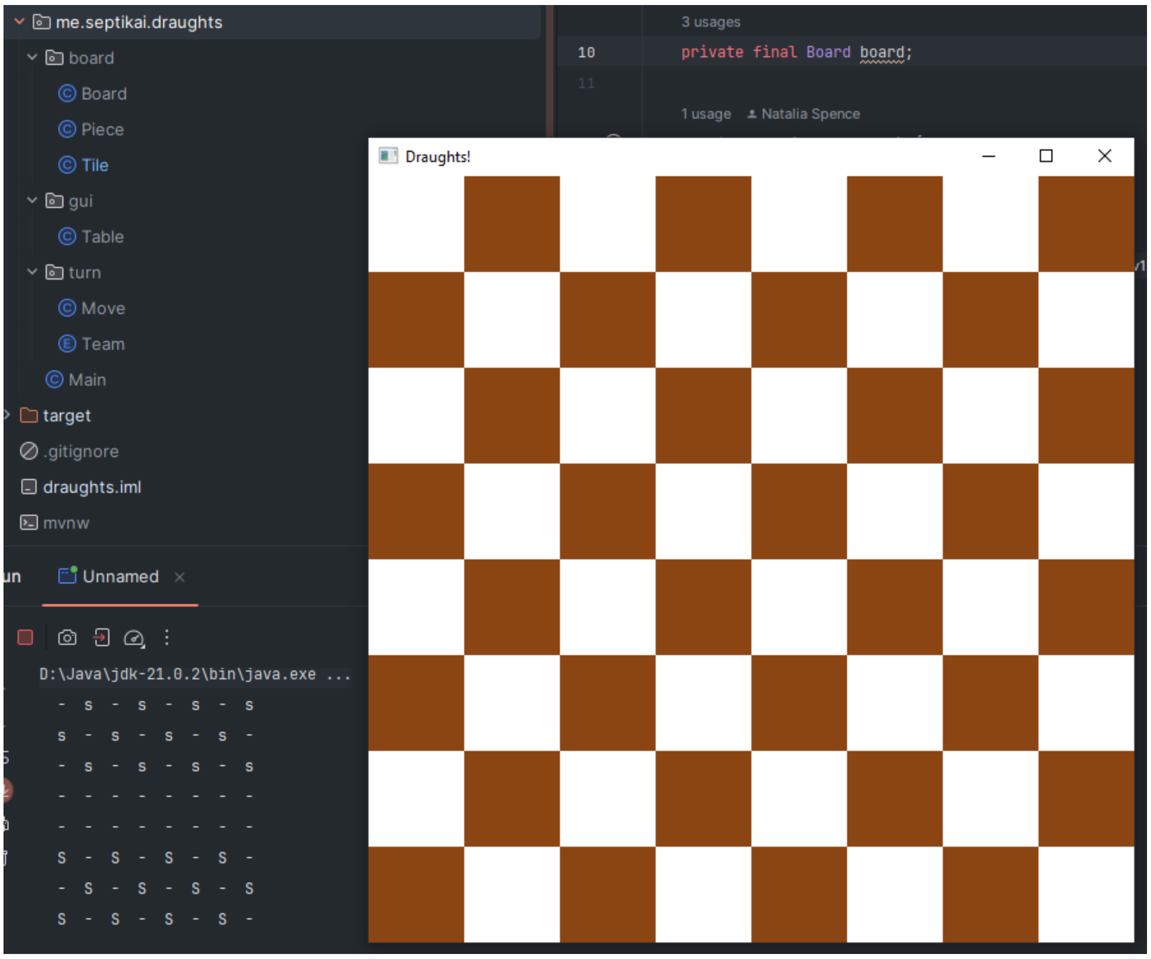
```java
    public Tile(final int location, final Piece piece) {
        super((location % 8) * 600.0/8, Math.floor(location / 8.0) *
            600.0/8, 600.0/8, 600.0/8);
        ...
    }
```

Figuring out which tiles should be which colour is a little different. We can find all even tiles with $location \ \% \ 2$, but only half of the tiles are on even numbers. We can find whether the tile is on an even row with $(location/8) \ \% \ 2$ though, and when you combine those you can find every even tile on an even row, and every odd tile on an odd row, as shown below.

7.19: Tile.java

```java
    public Tile(final int location, final Piece piece) {
        super(...);
        if ((location % 2) == 0 && (location / 8) % 2 == 0 || (location
            % 2) == 1 && (location / 8) % 2 == 1) {
            this.setFill(Color.WHITE);
        } else this.setFill(Color.SADDLEBROWN);
        ...
    }
```

If you run the program at this point, you should find that the *Tiles* are all displaying properly on the GUI, but that the *Pieces* are nowhere to be seen since we haven't reached them yet.



```java
private final Board board;
```

```
D:\Java\jdk-21.0.2\bin\java.exe ...
 - s - s - s - s
 s - s - s - s -
 - s - s - s - s
 - - - - - - - -
 - - - - - - - -
 S - S - S - S -
 - S - S - S - S
 S - S - S - S -
```

Figure 7.2: Displaying Tiles

For the *Pieces*, we start with the *super*() call just like with the previous two classes, but this time we are going to call it without any arguments. It does allow you to specify the location, radius, and even fill colour, but whilst we do know where the *Piece* should go thanks to its *location* attribute, this would require recalculating the coordinates again. Instead, we can set just the fill colour in the piece, and call a *setPiece*() method in the *Tile* constructor (this will also be used for moving *Pieces*), where we already know the *Tile* coordinates.

7.20: Piece.java

```java
    public Piece(final Team team, final int location) {
        super();
        ...
    }
```

7.21: Tile.java

```
    public Tile(final int location, final Piece piece) {
        ...
    } else this.setFill(Color.SADDLEBROWN);

    if (piece != null) {
        piece.setCenterX(this.getX() / 2);
        piece.setCenterY(this.getY() / 2);
        piece.setRadius(this.getWidth() * 2 / 3);
    }

    this.location = location;
    ...
}
```

7.22: Tile.java

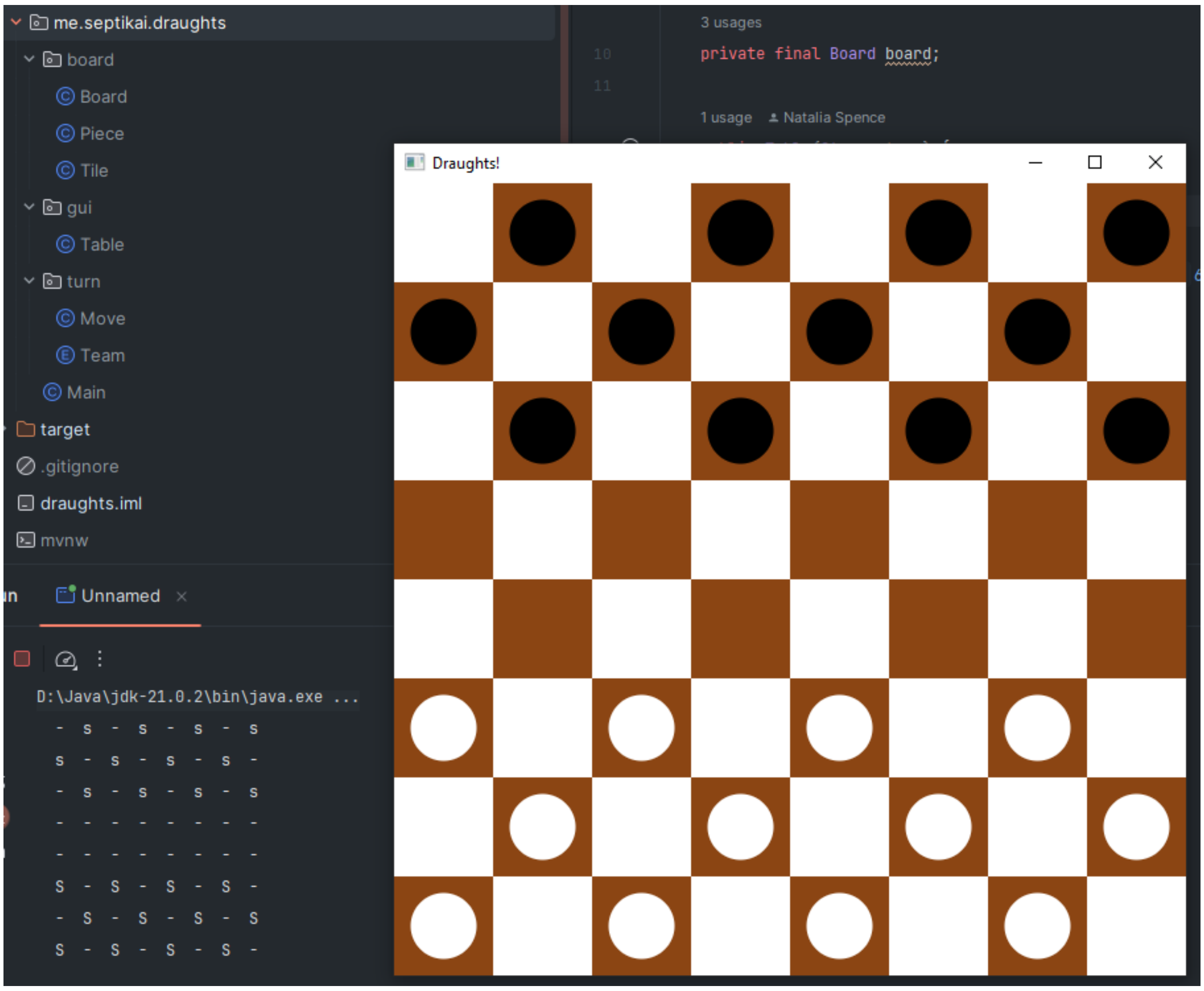
```
ic void setPiece(final Piece piece) {
if (piece != null) piece.setLocation(this.location);
this.piece = piece;
if (piece != null) {
    piece.setCenterX(this.getX() + (this.getWidth() / 2));
    piece.setCenterY(this.getY() + (this.getWidth() / 2));
    piece.setRadius(this.getWidth() * 1 / 3);
}
```

However, we can't add the *Pieces* to the board from here. We could do it from the *Tile* class, but the *Tile* has not been added to the *Board* and has no knowledge of the *Board* in it's constructor. Instead, we can add them in *Board* after we have all active pieces, to prevent extra loops.

7.23: Board.java

```
private Board() {
    ...
    this.whitePieces = getActivePieces(this.board, Team.WHITE);
    this.blackPieces = getActivePieces(this.board, Team.BLACK);

    this.getChildren().addAll(this.whitePieces);
    this.getChildren().addAll(this.blackPieces);

    ...
}
```

And with that, everything should now be successfully displayed on the board, and the next step is to make *Pieces* move.



```java
private final Board board;
```

```
D:\Java\jdk-21.0.2\bin\java.exe ...
 -  s  -  s  -  s  -  s
 s  -  s  -  s  -  s  -
 -  s  -  s  -  s  -  s
 -  -  -  -  -  -  -  -
 -  -  -  -  -  -  -  -
 S  -  S  -  S  -  S  -
 -  S  -  S  -  S  -  S
 S  -  S  -  S  -  S  -
```

Figure 7.3: The Initial Board

7.5 Moving Pieces

Moving *Pieces* is going to be split into two subsections - how movement works internally, and the user input for it.
The former will focus on actually moving the *Pieces* around the board, whilst the latter will cover how JavaFX takes the user input, displays possible move locations, and changes where the *Piece* is displayed.
Animating the move will be in another section.

7.5.1 Movement Behind The Scenes

First, we need to know where a *Piece* can move to. In draughts, pieces can move diagonally forwards, and backwards if doubled up. So in an 8×8 grid, diagonally forwards would be moving -7 or -9 places on the board, and backwards would be 7 or 9.

7.24: Piece.java

```java
ate int[] getPossibleMoveOffsets() {
if (this.isDoublePiece) return new int[]{-9, -7, 7, 9};
if (this.team == Team.BLACK) return new int[]{7, 9};
return new int[]{-7, -9};
```

Now, we need to iterate over these positions, check that each one is on the board, and see if it is occupied or not. If not, we can add a *Move* to there. We also need to ensure that if the *Piece* is on the left column, it's move does not wrap around the board, and the same for the right.

7.25: Piece.java

```java
ic List<Move> findLegalMoves(final List<Tile> board) {
final List<Move> legalMoves = new ArrayList<>();

for (final int possibleLocation : getPossibleMoveOffsets()) {
    int possibleEnd = this.location + possibleLocation;
    if (!isValidTile(possibleEnd)) continue;
    if (board.get(possibleEnd).isOccupied()) {
        // In Progress
    } else {
        if (this.location % 8 == 0 && (possibleLocation == -9 ||
           possibleLocation == 7)) continue;
        if ((this.location + 1) % 8 == 0 && (possibleLocation == -7 ||
           possibleLocation == 9)) continue;
        legalMoves.add(new Move(this, possibleEnd, null));
    }
}
return legalMoves;


ate boolean isValidTile(final int location) {
return location >= 0 && location <= 63;
```

If it is occupied, we need to see if the next *Tile* in that direction also exists and is occupied - if it exists and is empty, we can add a capturing *Move* to there.

7.26: Piece.java

```java
board.get(possibleEnd).isOccupied()) {
if (board.get(possibleEnd).getPiece().getTeam() == this.team) continue;
possibleEnd += possibleLocation;
if (!isValidTile(possibleEnd) || board.get(possibleEnd)
.isOccupied())
continue;
legalMoves.add(new Move(this, possibleEnd,
board.get(possibleEnd - possibleLocation).getPiece()));
se {
```

We can implement chaining capturing moves by simply not passing the turn and limiting legal moves to a single piece, so that's all for determining moves. To actually carry out a move, we need to update the moving *Piece*'s location and remove the *Piece* from the origin *Tile*.

7.27: Piece.java

```java
public void capture(final List<Tile> board) {
    board.get(this.location).setPiece(null);
    this.setLocation(-1);
}
...
public void move(final List<Tile> board, final Move move) {
    board.get(this.location).setPiece(null);
    board.get(move.getDestination()).setPiece(this);
    move.getCapturedPiece().capture(board);
}
```

7.28: Move.java

```java
public Piece getMovingPiece() {
    return this.movingPiece;
}

public int getDestination() {
    return this.destination;
}

public Piece getCapturedPiece() {
    return this.pieceCaptured;
}
```

With that, movement works internally! We just have no way for the user to actually tell it to move, and the display doesn't update, so now we can move to the JavaFX part of movement.

7.5.2 Movement With JavaFX

Piece movement via JavaFX has 2 parts - selecting the piece and displaying it's possible moves, and selecting and carrying out a move. Also deselecting a *Piece* if we want to stop displaying it's moves.

We can start in the *Piece* constructor. We need to add an *onMousePressed* event to select a *Piece*, which will need to check the *Piece* is on the board and then display valid moves for that piece if it belongs to the current team and is on the board.

In *Piece* we are going to simply add the event handler, and it should consume the event when done to be sure the mouse click does not register anywhere else as well as on the *Piece*.

7.29: Piece.java

```java
public Piece(final Team team, final int location) {
    ...

    this.setOnMouseClicked(mouseEvent -> {
        Node parent = this.getParent();
        if (parent instanceof Board) {
            if (this.team == ((Board) parent).getCurrentTeam()) {
                ((Board) parent).displayMoves(this);
                mouseEvent.consume();
            }
        }
    });
}
```

To actually display the moves, we need to track which *Tiles* are highlighted so we can unhighlight previous ones when we click off their piece, and which *Piece* is selected along with it's moves so we can move it later. Then we need to check each move individually. If the move is an attack then the destination should be highlighted red, otherwise yellow.

7.30: Board.java

```java
...
private final List<Integer> highlightedTiles = new ArrayList<>();
private Piece selected;
private List<Move> selectedPieceMoves;
...
private void unhighlightAll() {
    for (final int location : this.highlightedLocations) {
        this.board.get(location).setFill(Color.SADDLEBROWN);
    }
    this.highlightedLocations.clear();
}
...
public void displayMoves(final Piece piece) {
    if (!this.highlightedLocations.isEmpty())  unhighlightAll();
    this.selected = piece;
    this.selectedPieceMoves = findLegalMoves(piece);
    for (final Move move : this.selectedPieceMoves) {
        if (move.getCapturedPiece() != null) this.board.get(move.
            getDestination()).setFill(Color.RED);
        else this.board.get(move.getDestination()).setFill(Color.YELLOW
            );
        this.highlightedLocations.add(move.getDestination());
    }
}
```

Now we need to check for clicks on the tiles, both so we can deselect *Pieces* and so we can move them. We make sure to consume the event again as we do not want this click to register elsewhere.

7.31: Tile.java

```java
public Tile(final int location, final Piece piece) {
    ...

    this.setOnMouseClicked(mouseEvent -> {
        Node parent = this.getParent();
        if (!(parent instanceof Board)) return;
        if (((Board) parent).getHighlightedLocations().contains(this.
            location)) {
             ((Board) parent).moveTo(this.location);
        } else ((Board) parent).deselectTile();
        mouseEvent.consume();
    });
}
```

Back in *Board* we then need to implement those methods, all three of which are pretty simple, as they just return a list or call other methods we already wrote. We can also print the *Board* after a move to be sure everything is working.

7.32: Board.java

```java
public List<Integer> getHighlightedTiles() {
    return this.highlightedTiles;
}
...
public void deselectTile() {
    if (selected == null) return;
    this.selected = null;
    this.selectedPieceMoves.clear();
    unhighlightAll();
}
...
public void moveTo(final int destination) {
    for (final Move move : this.selectedPieceMoves) {
        if (move.getDestination() != destination) continue;
        this.selected.move(this.board, move);
        deselectTile();
        System.out.println(this);
        passTurn()
        break;
    }
}
```

Once we have movement working, we need to pass the turn. To do that, we're just going to change the *currentTeam* attribute of *Board*:

7.33: Board.java

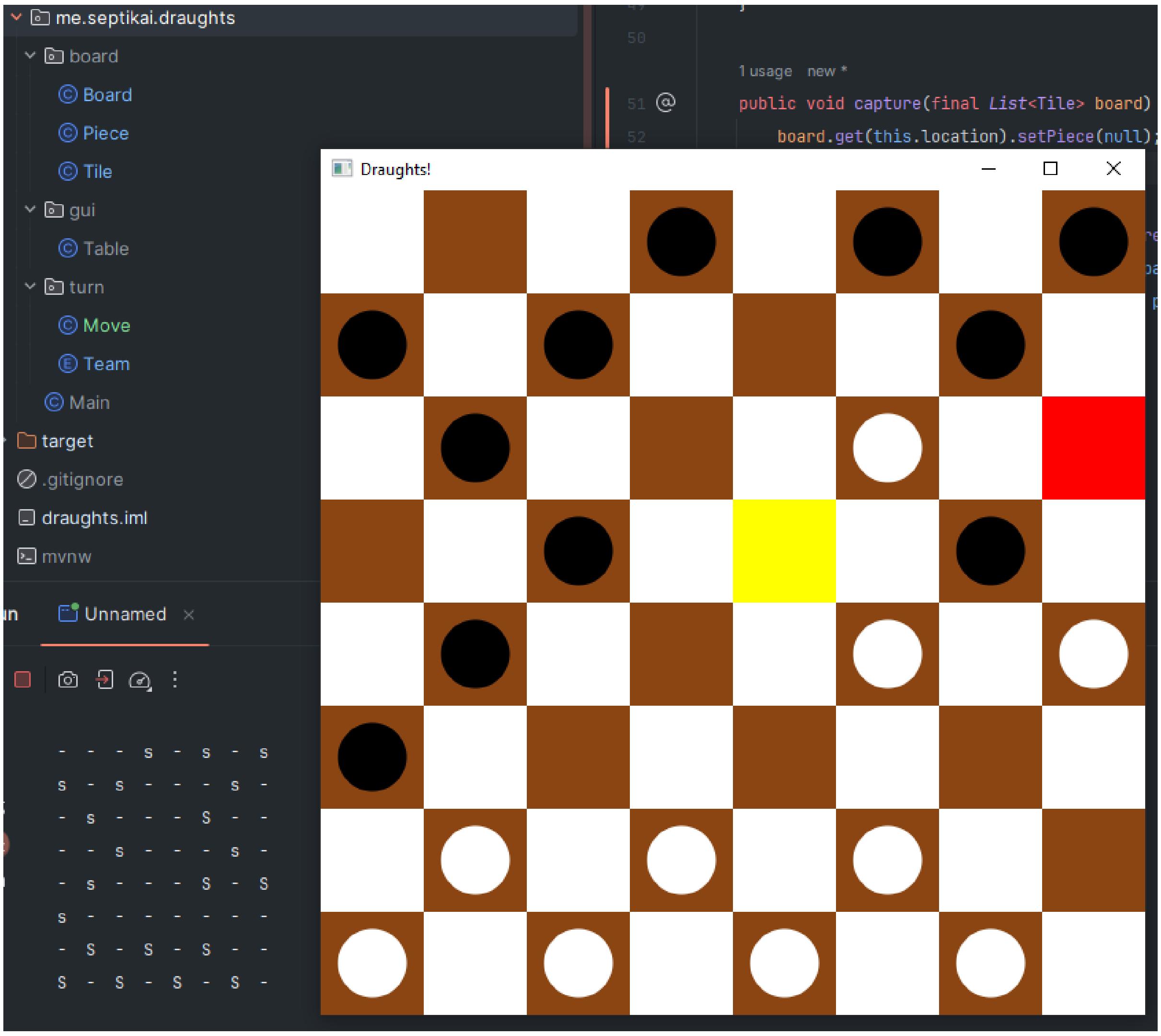
```java
private void passTurn() {
    if (this.currentTeam == Team.WHITE) this.currentTeam = Team.BLACK;
    else this.currentTeam = Team.WHITE;
}
```

Finally, we need to remove captured pieces from the UI. We've already removed them from the board in the *capture* method of *Piece*, but they will still show up on the board currently. We can fix this by searching the *Board*s children for the *Piece* and removing it.

7.34: Board.java

```java
public void capture(final List<Tile> board) {
    ...
    Parent parent = this.getParent();
    for (final Node node : parent.getChildrenUnmodifiable()) {
        if (!(node instanceof Piece) || !(parent instanceof Board))
            continue;
        if ((Piece) node == this) ((Board) parent).getChildren().remove
            (this);
    }
}
```

And that's movement complete!



```
1 usage  new *
public void capture(final List<Tile> board)
    board.get(this.location).setPiece(null);
```

```
- - - S - S - S
S - S - - - S -
- S - - - S - -
- - S - - - S -
- S - - - S - S
S - - - - - - -
- S - S - S - -
S - S - S - S -
```

Figure 7.4: Pieces moving and attacking