

# ADVANCED SE233 PROGRAMMING

LECTURE HANDOUT 1/2020

BY LECT.PASSAKORN PHANNACHITTA, D.ENG.

# SE233 – ADVANCED PROGRAMMING

# Preface

**T**HIS lecture handout covers 30 lecture hours and 30 laboratory hours of the Advanced Programming course (SE233). This course is one of the major courses for the students who pursue a bachelor's degree in the Software engineering program at the College of Arts, Media and Technology, Chiang Mai University, Thailand.

Principally, this course is a step forward from the Object-oriented programming course, where many more advanced use cases are illustrated to enable the students to apply a wide range of programming techniques for problem-solving. This material is organized into six chapters. The first chapter introduces a programming paradigm named Event driven development along with JavaFX, the technology used throughout this course. The second chapter discusses the Exception handling technique, which is a practically useful technique allowing us to design a software application better. The third chapter illustrates the Multithreaded programming technique, in which its utilization will enable us to utilize the computational resources better and better understand how the standard Graphic user interface works. The fourth chapter discusses error localizing using debugging techniques. The fifth chapter provides backgrounds on software testing with the main focus on unit testing. Finally, the sixth chapter includes topics related to software building and dependency management.

The lecturer much wishes that this handout would be beneficial for the students who have their interested in the contents covered in this course.

Passakorn Phannachitta, D.Eng.  
September 1, 2020

# Contents

<b>Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>x</b>
<b>1 Event driven development</b>	<b>1</b>
1.1 GUI . . . . .	2
1.1.1 JavaFX . . . . .	3
1.2 Events . . . . .	8
1.3 Events driven programming . . . . .	9
1.3.1 Events and GUI . . . . .	10
More examples . . . . .	11
1.3.2 Mouse events . . . . .	15
Drag and drop . . . . .	17
Dragging and dropping items in a ListView . . . . .	24
1.3.3 Keyboard events . . . . .	27
1.4 Case Study . . . . .	29
1.4.1 Model components . . . . .	31
1.4.2 View components . . . . .	35
1.4.3 Controller components and programming logic . . . . .	39
1.4.4 Container's matter . . . . .	43
1.4.5 Drag-and-drop functionality . . . . .	46
1.4.6 Attributes propagation . . . . .	51
	iii

---

1.4.7	Exercise . . . . .	55
<b>2</b>	<b>Exception handling</b>	<b>56</b>
2.1	What to do with exceptions? . . . . .	57
2.1.1	Catching exceptions . . . . .	57
2.1.2	Try-catch block . . . . .	58
2.1.3	The finally block . . . . .	60
2.1.4	Throwing exceptions . . . . .	61
2.2	Categorization . . . . .	63
2.3	Exceptions handling . . . . .	63
2.4	Case study . . . . .	65
2.4.1	Looking for the data supplier . . . . .	67
2.4.2	Extracting information from the API . . . . .	68
2.4.3	Adding Exception handlers . . . . .	70
2.4.4	The view components . . . . .	72
2.4.5	The controller components . . . . .	75
2.4.6	Event handlers . . . . .	79
2.4.7	Requirement change handling . . . . .	81
2.4.8	Background process . . . . .	89
2.4.9	Exercise . . . . .	94
<b>3</b>	<b>Multithreaded programming</b>	<b>95</b>
3.1	Multithread and multiprocess . . . . .	95
3.2	Purpose . . . . .	97
3.2.1	Multithreaded and GUI programming . . . . .	97
3.2.2	Multithreaded and accelerated computing . . . . .	98
	Synchronization . . . . .	99
3.3	The JavaFX concurrent framework . . . . .	100
3.3.1	Components . . . . .	101
	The worker interface . . . . .	101
	The Task class . . . . .	102
	The Service and ScheduledService class . . . . .	102
3.3.2	Example . . . . .	102
3.4	A more real-world example project . . . . .	105

---

3.5	Case study . . . . .	113
3.5.1	Layouting the GUI using FXML . . . . .	114
3.5.2	The drop-file functionality . . . . .	116
3.5.3	PDF file extraction . . . . .	118
3.5.4	Map and ListView . . . . .	126
3.5.5	Multithreaded indexer . . . . .	129
3.5.6	Adding a loading widget . . . . .	133
3.5.7	Exercise . . . . .	135
4	<b>Debugging</b> . . . . .	<b>136</b>
4.1	Debugging methodologies . . . . .	136
4.1.1	Probing . . . . .	137
4.1.2	Logging . . . . .	138
	Logging frameworks . . . . .	139
	Logging levels . . . . .	139
	Loggers . . . . .	141
	Appender . . . . .	141
	Layout . . . . .	142
	Configuration . . . . .	144
4.1.3	Debugging . . . . .	145
4.2	Additional topic: Sprite animation . . . . .	148
4.3	Case study . . . . .	155
4.3.1	Game loop . . . . .	159
4.3.2	Keys . . . . .	162
4.3.3	Collision checking . . . . .	166
4.3.4	Jumping . . . . .	167
4.3.5	Interpolation . . . . .	174
4.3.6	Acceleration . . . . .	178
4.3.7	Logging . . . . .	181
4.3.8	Exercise . . . . .	184
5	<b>Unit test</b> . . . . .	<b>185</b>
5.1	Basic software testing . . . . .	186
5.1.1	Why is software testing important? . . . . .	186

---

5.1.2	What can be the sources of bugs? . . . . .	187
5.1.3	Common test methods in different software development phases . . . . .	188
5.1.4	Testing objectives . . . . .	191
5.2	Unit test . . . . .	192
5.2.1	Objective . . . . .	193
5.2.2	JUnit . . . . .	193
	Assert methods . . . . .	194
5.2.3	JUnit naming conventions . . . . .	196
5.2.4	JUnit structure conventions . . . . .	197
5.2.5	Unit testing tips . . . . .	197
5.2.6	JUnit test suites . . . . .	198
5.2.7	Reflection . . . . .	200
5.3	Test-driven development (TDD) . . . . .	203
5.3.1	Benefits . . . . .	203
5.4	Case study – unit testing . . . . .	204
5.4.1	Character collision . . . . .	205
5.4.2	Death scene rendering . . . . .	208
5.4.3	Scoreboard . . . . .	209
5.4.4	Unit testing . . . . .	212
5.4.5	Unit testing with reflection . . . . .	214
5.4.6	Exercise . . . . .	216
5.5	Case study – TDD . . . . .	217
5.5.1	The snake’s head, body, and tail . . . . .	218
5.5.2	Food colliding and snake growing . . . . .	221
5.5.3	Getting the snake killed . . . . .	224
5.5.4	The game loop . . . . .	226
5.5.5	The test suite . . . . .	230
5.5.6	Exercise . . . . .	230
6	<b>Build management system</b> . . . . .	<b>231</b>
6.1	Built tool and dependency management . . . . .	232
6.1.1	Build tool . . . . .	232
6.1.2	Dependency management . . . . .	232

---

6.2	Java build management system . . . . .	234
6.2.1	Apache Maven . . . . .	235
	Maven build lifecycle . . . . .	236
	Maven standard directory . . . . .	237
	Project object model (POM) . . . . .	238
6.3	Case study . . . . .	242
6.3.1	A Maven Project . . . . .	242
6.3.2	Testing with Maven . . . . .	246
6.3.3	Packaging with Maven . . . . .	247
6.3.4	Exercise . . . . .	250
	<b>Appendix</b>	<b>251</b>
	<b>Bibliography</b>	<b>256</b>



# List of Figures

1.1	A simple UI with a button . . . . .	2
1.2	JavaFX Embedded System Development (Arduino-er 2015) . . . . .	4
1.3	The UI of HelloWorld.java . . . . .	5
1.4	Adding a button to HelloWorld.java . . . . .	6
1.5	Parts of the JavaDoc of getChildren and ObservableList (Interface ObservableList, 2020) . . . . .	7
1.6	GUI system in Java (Wong, 2017) . . . . .	10
1.7	Explaining button clicking events (Wong, 2017) . . . . .	11
1.8	The current rendering of customizedEvent.java . . . . .	15
1.9	An example application with drag and drop functionality . . . . .	18
1.10	Drag and drop items between lists . . . . .	24
1.11	The GUI of the case study of Chapter 1 . . . . .	29
1.12	The initialized GUI of the project . . . . .	39
2.1	The GUI of the case study of Chapter 2 . . . . .	65
2.2	Free usage limit of free.currconv.com. . . . .	67
2.3	The updated GUI of the project . . . . .	80
3.1	An example application with two buttons . . . . .	97
3.2	A graphical explanation of the FxConcurrentExample . . . . .	105
3.3	An example download manager application . . . . .	105
3.4	An example GUI of a multithreaded downloader . . . . .	106
3.5	An example reverse index creation application . . . . .	113
3.6	A GUI structure tree created by the Scene Builder toolkit . . . . .	114

---

4.1	Defining a breakpoint . . . . .	145
4.2	Defining the program arguments for debugging . . . . .	146
4.3	An example Debug tool windows of the IntelliJ Idea IDE . . . . .	147
4.4	An example sprite sheet . . . . .	151
4.5	The initialized GUI of the case study of Chapter 4 . . . . .	158
5.1	The V model of software testing (Java T Point, 2020) . . . . .	189
5.2	The final version of this platformer game case study. . . . .	211
5.3	The rendered snake game window. . . . .	229
5.4	An example test results generated from a test suite. . . . .	230
6.1	Querying for a Maven dependency . . . . .	240
6.2	An example querying results . . . . .	241
6.3	The Maven menu in IntelliJ Idea . . . . .	243
6.4	Selecting the JDK version . . . . .	252
6.5	Project Structure menu item . . . . .	253
6.6	Adding JavaFX dependencies . . . . .	253
6.7	Configure the PATH_TO_FX global variable . . . . .	254
6.8	Configure the VM options . . . . .	254

# List of Tables

- 4.1 The relationship between the `curIndex`, `curColumnIndex`, and the `curRowIndex` variables given the size of the sprite sheet of Figure 4.4 . . . 154

# Event driven development

**S**INCE real-world software nowadays has been more adapted to Graphic user interfaces (GUI), modern programming languages have been fully equipped with libraries and toolsets to provide the GUI options. The essence of GUI development is to respond to the incoming user control signals as promptly as possible by implementing the functionality that keeps monitoring for the possible incoming signals, recording the information about them after they occur, and linking up to the user-defined ways to respond to the particular inputs. In other words, a GUI application has to be able to: (1) accept inputs from the user, (2) process the input, and (3) deliver the output as predefined by software requirements. Usually, a GUI application has a control element, such as a button that interacts with the user by waiting for the user to click on it. Once the control element is activated, the application will undergo some processes and show the user results. The output typically displays on the screen in terms of text, dialogue boxes, or other visual elements.

Overall, a programming paradigm to fully operate GUI by telling it which particular input type it needs to handle and the approach to handle the particular input. Formally, the development of this activity is called Event driven development.

## 1.1 GUI

Before we go through more details for the GUI development in Java, we may have to concede that Object-oriented programming (OOP) is fundamental to GUI. To demonstrate its importance, imagine that we have to compose the GUI illustrated in Figure 1.1, along with all the functionalities implementing its visual elements. In steps, we have to draw the entire message box, followed by two text labels, a text box, and, finally a button with the word *Ok* on it. Focusing only on the button, we initially have to draw a rounded rectangle and color it in blue, perhaps using color code. Then, we have to define a coordinate  $(x, y)$  inside the message box to place the button and decide whether the button should be attached to the message box or not. After that, we will have to write the word *Ok* on the button and color the text in white. Apart from the full list of kinds of stuffs related to graphics, more importantly, we will also have to program the button's behaviors. For example, we have to implement some methods to define how the application should respond to the user when she or he presses the button. Moreover, nowadays, a button element also shows some effects when it is hovered or clicked. In short, stuff related to buttons has become more and more complicated in recent days. Hence, it is not practically sensible to create every single button from scratch in real-world software development.



Figure 1.1: A simple UI with a button

Without OOP, creating a visual element is a pain because there appears to be a pretty number of visual elements in one single application. For example, inside the message box of Figure 1.1, a *Cancel* button is often placed alongside the *Ok* button. Therefore, if OOP's concept does not exist, we will also have to create everything from scratch for the *Cancel* button, i.e., from drawing another rounded rectangle up to code its behavior. Since OOP's essential goal is to make everything reusable, with OOP applied, we will be able to create a Button object for the *Ok* button, where many behaviors and actions can be inherited from some existing classes. For example, the rounded rectangle may be inherited from a Shape class in

which the coordinate placement is already implemented. To add the *Cancel* button, we may reuse the already implemented Button object, and more importantly, this Button object can also be reused in many other projects in the future. In sum, OOP is indispensable to GUI development. It is the lecturer's suggestion to review thoroughly on OOP before continuing further on this course.

### 1.1.1 JavaFX

In the past, GUI in Java is commonly implemented by using two packages, namely `Java.awt` (2018) and `Javax.swing` (2018), a.k.a. Awt and Swing, respectively. However, more recently, trends have been steadily reshaped to a complete toolset named JavaFX (2020). It is the current approach of choice to user interfacing in Java. JavaFX is also known as the successor of Awt and Swing. Stated in the official website of JavaFX, it was emerged by a collaborative effort by many individuals and enterprises with the common goal of obtaining a modern, efficient, and fully featured toolkit for developing rich client applications. In this course, JavaFX will be used as a medium to study the GUI development and the Event driven development programming methodology. For the installation and configuration of Java and JavaFX, see the Appendix.

The name JavaFX is coined after Java, Adobe Flash, and Apache Flex, the somewhat common standard toolkits widely recognized and used in the past. In the lecturer's view, the reasons that make JavaFX a popular choice for client applications are as follows:

- JavaFX is 100% compatible with Java APIs where more than million libraries are available;
- GUI development in JavaFX is available for cross platform development;
- JavaFX is highly suitable for beginner where drag and drop component builders are available;

---

Java.awt (2018) Package java.awt. Retrieved May, 2020, from <https://docs.oracle.com/javase/10/docs/api/java/awt/package-summary.html>

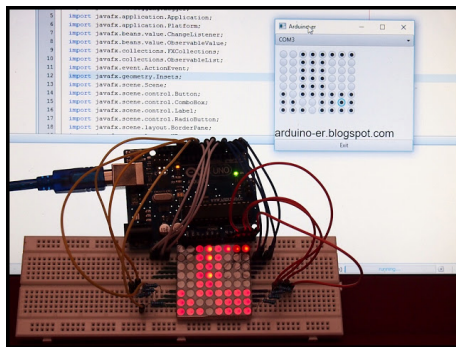
Javax.swing (2018) Package javax.swing. Retrieved May, 2020, from <https://docs.oracle.com/javase/10/docs/api/javax/swing/package-summary.html>

JavaFX (2020) JavaFX 14. Retrieved May, 2020, from <https://openjfx.io>

- Last but not least, JavaFX can be used implement an extensive range of applications including, desktop application, mobile application, web application, and embedded system. For example, Figure 1.2 shows an example of embedded system application development in JavaFX.

A GUI in JavaFX is constructed as a scene graph together with a collection of visual elements called nodes. All nodes are arranged hierarchically, analogous to how HTML elements are arranged in a DOM tree. There are numerous types of nodes, including buttons, labels, text boxes, shapes, and media (i.e., audio and video). As a collection, they can handle various types of user inputs, such as mouse move, mouse hover, drag over, and drag drop. In short, the available JavaFX visual elements are more than sufficient enough for developing client applications.

Script 1.1 illustrates a JavaFX example source code showing its basic anatomy. The launcher of a JavaFX application is a class that must be inherited from the `Application` class in the `javafx.application` package, i.e., the code fragment shown at Line 5 of the script. Also, we have to override the `start` method to indicate the application's entry point, as shown in Line 7 of the script. This `start` method will be called by the launcher when the application starts. This `start` method has one argument. It is an instance of the `JavaFX Stage` class, wherein this example, `primaryStage` is the name of the instance. Particularly, a JavaFX application defines the user interface container in terms of `Stage` and `Scene`, where a `Scene` element is displayed in the `Stage` element. The `Stage` can be considered as the top-level container of a JavaFX application, where `Scene` is the container for any other visual elements, such as text and drawing, which are what users may interact with.



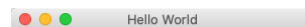
**Figure 1.2: JavaFX Embedded System Development (Arduino-er 2015)**

Arduino-er (2015) Java/JavaFX/jSSC control Arduino + 8x8 LED Matrix. Retrieved June, 2020, from <http://arduino-er.blogspot.com/2015/09/javajavafxjssc-control-arduino-8x8-led.html>

**Script 1.1: An example Hello World in JavaFX****HelloWorld.java**

```
+ 1 import javafx.application.Application;
+ 2 import javafx.scene.Scene;
+ 3 import javafx.stage.Stage;
+ 4 import javafx.scene.layout.StackPane;
+ 5 public class Launcher extends Application {
+ 6     @Override
+ 7     public void start(Stage primaryStage) throws Exception{
+ 8         StackPane root = new StackPane();
+ 9         primaryStage.setTitle("Hello World");
+10         primaryStage.setScene(new Scene(root, 300, 275));
+11         primaryStage.show();
+12     }
+13     public static void main(String[] args) {
+14         launch(args);
+15     }
+16 }
```

Script 1.1 creates Stage and Scene of the size 300px multiplied by 250px. The Scene content is represented as a hierarchical graph of nodes, wherein this example, a visual element of the type StackPane is the root node of the graph. Figure 1.3 shows the rendered GUI of this application. Note that, in front of the line number in Script 1.1, plus (+) and minus (-) symbols indicate the difference between versions of the same file name, i.e., helloWorld.java. Specifically, + indicates the lines of code added to the current version of the file, where as - indicates the lines of code deleted from the previous version of the file. In this script, since it is the first time the file helloWorld.java is presented in this document, all its lines of code are associated with a + sign. This notion will be used throughout the document.

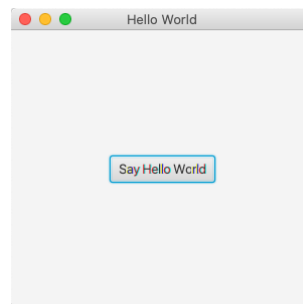
**Figure 1.3: The UI of HelloWorld.java**



**Script 1.2: Add a button to helloWorld.java****HelloWorld.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     @Override
4     public void start(Stage primaryStage) throws Exception{
+ 5         Button btn = new Button();
+ 6         btn.setText("Say Hello World");
+ 7         StackPane root = new StackPane();
+ 8         root.getChildren().add(btn);
+ 9         primaryStage.setTitle("Hello World");
10         primaryStage.setScene(new Scene(root, 300, 275));
11         primaryStage.show();
12     }
13     public static void main(String[] args) {
14         launch(args);
15     }
16 }
```

Script 1.2 adds a button to `helloWorld.java`. There are several steps to consider to implement the button, and these will be mostly the same for implementing any JavaFX visual elements. That is, firstly, we have to create the visual element itself as a JavaFX node, such as a button in this example. Then, we have to append the created node to an existing node in the Application Scene graph. This is analogous to how we append an HTML node to its parent node in the DOM tree. In the script, the new button is created in Line 5. Then, its labeled text is set at Line 6. Finally, at Line 8, the button is added to a `StackPane` element, making it appear in the Application Scene graph and presented when the application is launched. After the script is compiled and launched, there will be a button displayed in the middle of the application window, as shown in Figure 1.4.



**Figure 1.4: Adding a button to HelloWorld.java**

Up to the moment, we have already known that we can append a JavaFX node to any other specific nodes by using the `getChildren().add()` method. This can be implied that the `getChildren` method returns an object in a list type, that we can append something to it. To confirm this, we check the JavaDoc explaining this `getChildren` method (ObservableList 2020). As shown in the left-hand side of Figure 1.5, the return type of this `getChildren` method is `ObservableList<Node>` which seems to be close to our assumption. Scrutinizing the `ObservableList<Node>` further, as shown on the right-hand side of Figure 1.5, we can see that the class is an interface class inherited from multiple classes, which are `Collection<E>`, `List<E>`, `Iterable<E>`, and `Observable`. For the first three classes they are widely recognized classes related to collections. This shows that the JavaFX elements also allow us to utilize the common Java methods when needed. Particularly, we can use many predefined Java methods implemented for the operation over a `List`, such as the `add`, `contains`, `forEach`, and `toArray` methods. In contrast, the `Observable` class is not what we usually have seen elsewhere. The inherited methods of this class are `addListener` and `removeListener`. These are parts of events handling that will make an object of this type wait for and accept a signal from the user to process further what the user may need. Anyhow, it is the lecturer's thought that it must be better to go through its fundamentals, which are events, before we discuss the listeners in greater detail.

**getChildren**

```
protected ObservableList<Node> getChildren()
```

**Description copied from class: Parent**  
Gets the list of children of this Parent.

See the class documentation for `Node` for scene graph structure restrictions on setting a `Parent`'s children list. If these restrictions are violated by a change to the list of children, the change is ignored and the previous value of the children list is restored. An `IllegalArgumentException` is thrown in this case.

If this `Parent` node is attached to a `Scene` attached to a `Window` that is showing (`Window.isShowing()`), then its list of children must only be modified on the JavaFX Application Thread. An `IllegalStateException` is thrown if this restriction is violated.

Note to subclasses: if you override this method, you must return from your implementation the result of calling this super method. The actual list instance returned from any `getChildren()` implementation must be the list owned and managed by this `Parent`. The only typical purpose for overriding this method is to promote the method to be public.

**Overrides:**  
`getChildren` in class `Parent`

**Returns:**  
the list of children of this `Parent`.

**Module** `javafx.base`  
**Package** `javafx.collections`  
**Interface** `ObservableList<E>`

**Type Parameters:**  
`E` - the list element type

**All Superinterfaces:**  
`Collection<E>`, `Iterable<E>`, `List<E>`, `Observable`

**All Known Subinterfaces:**  
`ObservableListValue<E>`, `WritableListValue<E>`

**All Known Implementing Classes:**  
`FilteredList`, `ListBinding`, `ListExpression`, `ListProperty`, `ListPropertyBase`, `ModifiableObservableListBase`, `ObservableListBase`, `ReadOnlyListProperty`, `ReadOnlyListPropertyBase`, `ReadOnlyListWrapper`, `SimpleListProperty`, `SortedList`, `TransformationList`

**Figure 1.5: Parts of the JavaDoc of `getChildren` and `ObservableList` (Interface `ObservableList`, 2020)**

ObservableList (2020) Interface `ObservableList`. Retrieved May, 2020, from <https://openjfx.io/javadoc/14/javafx.base/javafx/collections/ObservableList.html>

## 1.2 Events

Events can be classified into two types: Foreground and Background events. The main difference between the two types of events is related to how do they interact with the user. That is, Foreground events are mostly, if not all, associated with human signals such as human interaction. Foreground events are such as button clicking, mouse moving, and dragging and dropping. Background events, on the other hand, do not interact directly with users. Mostly, they retrieved signals from other programs or computers. Background events include operating system interruptions, hardware or software failure, timer expiry, and operation completion.

In JavaFX, events have three fundamental components: event source, event listener, and event handler. An event source is a JavaFX visual element that users can interact with. An event listener is an object that actively listens to the events from a specific component. Finally, an event handler is a method that is created to process the input signal and generate the corresponding output. In terms of operation, a signal is created at an event source. An event listener will then respond to the signal by invoking an event handling method inside it, and further process the signal as to handle the user request. In terms of programming, we have to register an event listener to an event source and implement an event handling method for enabling a complete interaction possibility between GUI and users.

Script 1.3 added an event handler to `helloWorld.java`. After the button is clicked, the text *Hello World!* will be displayed at the terminal. At Line 7 of the script, an event instance is associated with the button instance created earlier. Note that we applied an overridden flag, i.e., `@Override`, here, and it is an example usage of the polymorphism concept in OOP. Another point worth mentioning here is that the event handler is simply created inside the `setOnAction` method without formally declaring an instance elsewhere. This is a utilization example of an anonymous class. This class implements the `EventHandler` interface and defines the handling method that displays the text *Hello World!* at the terminal.

Multiple events usually coincide in the real world, and the application usually responds to those events differently. For example, after a user clicks a form submitting button, the data may be sent to the server simultaneously as a popup message box appears to inform the user that the form submission is completed. In this sce-

Script 1.3: Add an event handler to helloWorld.java

HelloWorld.java

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     @Override
4     public void start(Stage primaryStage) throws Exception{
5         Button btn = new Button();
6         btn.setText("Say Hello World");
+ 7         btn.setOnAction(new EventHandler<ActionEvent>() {
+ 8             @Override
+ 9                 public void handle(ActionEvent event) {
+ 10                     System.out.println("Hello World!");
+ 11                 }
+ 12             });
13         StackPane root = new StackPane();
14         root.getChildren().add(btn);
15         primaryStage.setTitle("Hello World");
16         primaryStage.setScene(new Scene(root, 300, 275));
17         primaryStage.show();
18     }
19     public static void main(String[] args) {
20         launch(args);
21     }
22 }
```

nario, there is only one event source, i.e., the button, only one event listener, i.e., the one that actively checks if a button is clicked, and two event handlers that transmit the data and shows the popup message box, respectively.

## 1.3 Events driven programming

Event driven programming can be considered as a methodology to develop a program with numerous subprograms inside it. A subprogram will be launched after the arrival of a specific user signal meant for it. For example, in an action game, moving a game character forward can be considered a subprogram of the entire game, which will only be executed after the user emits the particular signal, such as a keyboard pressing of the key *w*. In particular, this subprogram will not do any-

thing unless the user presses the key *w*, whereas pressing other action keys, such as the *spacebar*, will trigger another subprogram that generates a different outcome such as damage dealing or speed boosting in the game.

Nowadays, the Event driven programming concept has become a dominant programming paradigm. It has been widely adopted in many cutting-edge technologies, such as in the Internet of Things (IoT) and numerous automated technologies harnessing the power of artificial intelligence (AI). Many tasks under these systems can be considered as subprograms in the systems. A subprogram will be triggered only if a specific signal arrives. For example, an automobile's adaptive cruise control functionality will be triggered based on the signals sensed from the surrounding environment. For another example, in a smart watering system in smart farms; there are numerous sensors where each actively monitors the condition of crops and the surrounding environment. In both examples, the systems will decide what it should do next based on the data retrieved from sensors and the algorithms.

### 1.3.1 Events and GUI

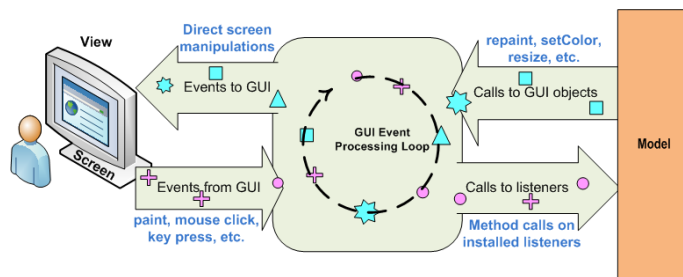


Figure 1.6: GUI system in Java (Wong, 2017)

Figure 1.6 illustrates the general concept of GUI in Java (Wong 2017). The core engine of the Java GUI system is the GUI event process loop, presented in the middle

Wong S (2017) Java GUI Programming Primer. Retrieved May, 2020, from <https://www.clear.rice.edu/comp310/JavaResources/GUI/>

of the figure. The process loop is an active background process performing like an infinite loop for executing subprograms related to event driven programming. For an action that affects the GUI, e.g., a mouse clicking on a button clicking, the action will be wrapped as an event and enqueued in the GUI event process loop. An action listener that is already registered to the visual element will retrieve the event from the loop and process it in the associated subprogram. Once the computation is done, the results will be sent back to the loop and wait until it can move toward the event's target output.

Referred to Script 1.3, the event is triggered by the user mouse click. The action of mouse clicking will be encapsulated and placed in the GUI event processing loop. When the loop becomes more available, it will pick up this mouse-clicking event and send the event to the backend to process it. The system knows whom the actual method should involve with this button clicking by preregistered the method to this mouse clicking event using a particular type of object called listeners. In this example, the listener is an instance that implements the `javafx.event`. Note that it is also possible to associate multiple listeners to one single action. Figure 1.7 graphically explains this button clicking event handling.

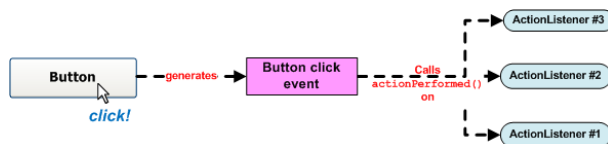


Figure 1.7: Explaining button clicking events (Wong, 2017)

### More examples

Script 1.4 adds two more events to the file `helloWorld.java`. The listeners of the two events monitor the application scene, and the events will be triggered when the size of the application window is changed. At Line 16 and Line 17 of the script, the `setScene` method is extracted because we have to modify the scene instance and associate two events with it. In other words, this scene is the event source of the two events added in this example. The first event is the code fragments between Line 18 to Line 23, which monitors the width property of the application

scene. The overridden method inside it is its event handler, which will be triggered when the scene window's width value is changed. After triggered, the method will send the result back to the GUI to show the updated screen width at the terminal. The second event of this example, shown between Line 24 and Line 29 of the script, is almost identical to that of between Line 18 and Line 23, whereas it monitors the application scene's height property. What should be further observed here is the code pattern of JavaFX's event handlers. As can be seen from all the code fragments added to the current update of the script, they all are in the form of `[source element].[component to bound with events].addListener([interface to the event listener])[an overridden method that response to the event]`.

Script 1.5 shows a concrete object design example, where we bind an event source with event handlers to bundle the product as a new object. This will allow us to reuse the entire object in the future. The task carried out in this script is to create a circle with a label that says whether the mouse pointer is being inside this circle or not. If it is, then the application will display the mouse coordinate on the label; otherwise, it will just merely say that the mouse pointer is not in the circle. Without Event driven programming, all the time, we may have to check the position of the mouse pointer, whether it is being overlapped with the circle or not. However, with Event driven programming, we can simplify the programming logic by checking only when changes in any visual element in the application have been detected, and examining whether the detected changes are related to the task we are focusing on or not. The programming logic design related to these changes may require us to reshape our logical thinking somewhat to consider the program's state. Explained briefly, the state is data or information that will be changed or manipulated throughout a program's runtime. For this example, we are interested in the mouse state at the time the pointer is inside the circle, and that of when it just moves out of the circle. Therefore, we will implement event handlers to track these states. On the other hand, when the mouse pointer is anywhere outside the circle, we do not need to implement anything for its handling.

Scrutinizing Script 1.5, the circle is created by the `createMonitoredCircle` method at the lines of code between Line 13 and Line 33 of the script. The events associated with the circle are: the `setOnMouseMoved` and `setOnMouseExit` methods. In particular, the `setOnMouseMoved` method will only be executed when the mouse pointer is inside the circle, and the `setOnMouseExit` method will only be executed at the exact

Script 1.4: Add two more event handlers to helloWorld.java

HelloWorld.java

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     @Override
4     public void start(Stage primaryStage) throws Exception{
5         Button btn = new Button();
6         btn.setText("Say Hello World");
7         btn.setOnAction(new EventHandler<ActionEvent>() {
8             @Override
9             public void handle(ActionEvent event) {
10                 System.out.println("Hello World!");
11             }
12         });
13         StackPane root = new StackPane();
14         root.getChildren().add(btn);
15         primaryStage.setTitle("Hello World");
16         primaryStage.setScene(new Scene(root, 300, 275));
17         Scene scene = new Scene(root, 300, 275);
18         scene.widthProperty().addListener(new ChangeListener<Number>() {
19             @Override
20             public void changed(ObservableValue<? extends Number> observableValue,
21                 Number oldSceneWidth, Number newSceneWidth) {
22                 System.out.println("Width: " + newSceneWidth);
23             }
24         });
25         scene.heightProperty().addListener(new ChangeListener<Number>() {
26             @Override
27             public void changed(ObservableValue<? extends Number> observableValue,
28                 Number oldSceneHeight, Number newSceneHeight) {
29                 System.out.println("Height: " + newSceneHeight);
30             }
31         });
32         primaryStage.setScene(scene);
33         primaryStage.show();
34     }
35     public static void main(String[] args) {
36         launch(args);
37     }
38 }
```

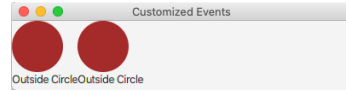


time the mouse pointer is moving out of the circle.

**Script 1.5: Customized event handlers****CustomizedEvent.java**

```
+ 1 //Imports are omitted
+ 2 public class CustomizedEvent extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) {
+ 5         FlowPane pane = new FlowPane();
+ 6         StackPane circle1 = createMonitoredCircle();
+ 7         pane.getChildren().addAll(circle1);
+ 8         Scene scene = new Scene(pane);
+ 9         stage.setTitle("Customized Events");
+10         stage.setScene(scene);
+11         stage.show();
+12     }
+13     private StackPane createMonitoredCircle() {
+14         StackPane pane = new StackPane();
+15         VBox layout = new VBox();
+16         Label reporter = new Label("Outside Circle");
+17         Circle circle = new Circle(30);
+18         circle.setOnMouseMoved(new EventHandler<MouseEvent>() {
+19             @Override
+20             public void handle(MouseEvent event) {
+21                 reporter.setText("(x: " + event.getX() + ", y: " + event.getY() + ")");
+22             }
+23         });
+24         circle.setOnMouseExited(new EventHandler<MouseEvent>() {
+25             @Override
+26             public void handle(MouseEvent event) {
+27                 reporter.setText("Outside Circle");
+28             }
+29         });
+30         layout.getChildren().setAll(circle, reporter);
+31         pane.getChildren().add(layout);
+32         return pane;
+33     }
+34     public static void main(String[] args) { launch(args); }
+35 }
```

Based on this design, we may think that whenever we want to reuse the circle created by the `createMonitoredCircle` method in the future, the label should be associated with the circle all the time. Thus, bundling them together as a new object should be a practical idea. All the associated visual elements are aligned in a JavaFX VBox (Short for Vertical box) instance and wrapped in a JavaFX StackPane instance. The VBox and the StackPane can be considered as containers. The return type of this `createMonitoredCircle` method is a ready-to-use JavaFX node of the type StackPane. To reuse the object, we can add this node to any parent node at the higher level of the application graph, such as, in this example, it is added to a JavaFX FlowPane at Line 7 of the script. After compiled, the application will be rendered as shown in Figure 1.8.



**Figure 1.8:** The current rendering of `customizedEvent.java`

### 1.3.2 Mouse events

Script 1.6 presents the reuse of the bundled object between Line 7 and Line 9 of the script. Also, the script introduces several events that utilize the mouse move and mouse exit signals. In JavaFX there are many more mouse events available. If we look through the JavaDoc (JavaFX 14 2020), the available mouse events of the current version of JavaFX include:

- `setOnMouseEntered`
- `setOnMouseExited`
- `setOnMouseMoved`
- `setOnMousePressed`
- `setOnMouseReleased`
- `setOnMouseDragged`
- `setOnMouseDragEntered`
- `setOnMouseDragExited`
- `setOnMouseDragOver`
- `setOnMouseDragReleased`

## Script 1.6: Reusing the object group

CustomizedEvent.java

```
1 //Imports are omitted
2 public class CustomizedEvent extends Application {
3     @Override
4     public void start(Stage stage) {
5         FlowPane pane = new FlowPane();
6         StackPane circle1 = createMonitoredCircle();
+ 7         StackPane circle2 = createMonitoredCircle();
- 8         pane.getChildren().addAll(circle1);
+ 9         pane.getChildren().addAll(circle1, circle2);
10         Scene scene = new Scene(pane);
11         stage.setTitle("Customized Events");
12         stage.setScene(scene);
13         stage.show();
14     }
15     private StackPane createMonitoredCircle() {
16         StackPane pane = new StackPane();
17         VBox layout = new VBox();
18         Label reporter = new Label("Outside Circle");
19         Circle circle = new Circle(30);
20         circle.setOnMouseMoved(new EventHandler<MouseEvent>() {
21             @Override
22             public void handle(MouseEvent event) {
23                 reporter.setText("x: " + event.getX() + ", y: " + event.getY() + ");");
24             }
25         });
26         circle.setOnMouseExited(new EventHandler<MouseEvent>() {
27             @Override
28             public void handle(MouseEvent event) {
29                 reporter.setText("Outside Circle");
30             }
31         });
32         layout.getChildren().setAll(circle, reporter);
33         pane.getChildren().add(layout);
34         return pane;
35     }
36     public static void main(String[] args) { launch(args); }
37 }
```

### Drag and drop

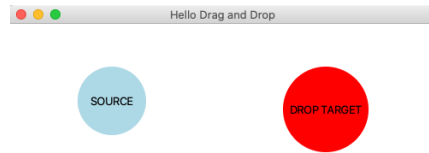
One of the very best approaches to learn about various mouse events at once is to implement a drag and drop functionality. There are several things to consider to implement it, which are as follows:

- How to make us able to start dragging?
- What can we drag?
- How do we know that the mouse is on the object which we want to drag?
- What happen during the dragging?
- Will the object move along with the mouse pointer?
- When and how to stop dragging?
- Should the movement distance also be considered?
- Should the coordinate also be considered?

The following in this section walkthroughs a drag-and-drop task illustrated in Figure 1.9. Specifically, the task has the following requirements:

1. There are two circles, the source is in blue, and the target is in red;
2. Once we click on the source, it will be brought to the front and moved with the mouse pointer;
3. If we drag the source and release the mouse button over the target, the source object will be snapped in front of the target object.
4. If we drag the source and release the mouse button outside the target, the source object moves back to the origin coordinate.

Even if these requirements may read simple, there are numerous things to consider to complete them. As can be observed in the requirement, the two circles have multiple common characteristics, such as both of them have a label on it. Thus, it is recommended to separate the object creation into a new class and let the application launcher or any other class use it. This will also allow us to reuse it elsewhere in the future. Script 1.7 shows this FigureGroup class.



**Figure 1.9: An example application with drag and drop functionality**

**Script 1.7: The Figure group object for a drag and drop task**

FigureGroup.java

```
+ 1 //Imports are omitted
+ 2 public class FigureGroup {
+ 3     private StackPane figure;
+ 4     public FigureGroup(int x, int y, int radius, Color color, String str) {
+ 5         Circle circle = new Circle(radius, color);
+ 6         Text text = new Text(str);
+ 7         this.figure = new StackPane( circle, text );
+ 8         this.figure.setLayoutX(x);
+ 9         this.figure.setLayoutY(y);
+10     }
+11     public StackPane getFigure() { return figure; }
+12 }
```

Script 1.8 shows the application launcher that uses the FigureGroup object by creating two instances of the class, i.e., two circles with a label on it, at Line 9 and Line 10. If we launch the script now, it will render the GUI shown in Figure 1.9.

**Script 1.8: The launcher of a drag and drop task****DragAndDrop.java**

```
+ 1 //Imports are omitted
+ 2 public class DragAndDrop extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) {
+ 5         stage.setTitle("Hello Drag and Drop");
+ 6         Group root = new Group();
+ 7         Scene scene = new Scene(root, 500, 200);
+ 8         scene.setFill(Color.WHITE);
+ 9         FigureGroup source = new FigureGroup(80,50,40,Color.LIGHTBLUE,"SOURCE");
+10         FigureGroup target = new FigureGroup(320,50,50,Color.RED,"DROP TARGET");
+11         root.getChildren().addAll(source.getFigure(),target.getFigure());
+12         stage.setScene(scene);
+13         stage.show();
+14     }
+15     public static void main(String[] args) {
+16         Application.launch(args);
+17     }
+18 }
```

The next step is the drag-and-drop-actions. Let us make the circle moved along with the mouse pointer. This requires the coordinate of the circle for further computation. Furthermore, the requirement #4 says that if we release the mouse button outside the target, the source object shall move back to the origin location. Therefore, we have to store the mouse coordinate and the origin, which is the coordinate before the circle is moved. In brief, the event handlers we have to associate with the source circle are `setOnMousePressed`, `setOnDragDetected`, `setOnMouseDragged`, and `setOnMouseReleased`.

The updated `FigureGroup` and `DragAndDrop` classes are presented in Script 1.9 and Script 1.10, respectively. The current updates in the `FigureGroup` class are related to the implementation of the move back to origin functionality, where the variable declaration is at Line 4. The programming logic to set or reset the figure's location are shown between Line 12 and 19. In addition, four event handlers are added to the `DragAndDrop` class in this updated Script 1.10. The first event handler is a `setOnMousePressed` method between Line 13 and Line 19; the han-

Script 1.9: Add origin coordinate to Figure group

FigureGroup.java

```
1 //Imports are omitted
2 public class FigureGroup {
3     private StackPane figure;
+ 4     private double origin_x, origin_y;
5     public FigureGroup(int x, int y, int radius, Color color, String str) {
6         Circle circle = new Circle(radius, color);
7         Text text = new Text(str);
8         this.figure = new StackPane( circle, text );
9         this.figure.setLayoutX(x);
10        this.figure.setLayoutY(y);
11    }
+ 12    public void setBackOrigin() {
+ 13        this.figure.setLayoutX(origin_x);
+ 14        this.figure.setLayoutY(origin_y);
+ 15    }
+ 16    public void setPosition(double x, double y) {
+ 17        this.figure.setLayoutX(this.figure.getLayoutX()+x);
+ 18        this.figure.setLayoutY(this.figure.getLayoutY()+y);
+ 19    }
20    public StackPane getFigure() { return figure; }
21 }
```

dling method keeps detecting whether the mouse button is clicked over the event source. Then, it will store the coordinate in the `mouse_x` and `mouse_y` variables. The second method added to this script is `setOnDragDetected`; once the drag is started, this method brings the object to the front and let the mouse pointer control affect only this source object until the drag is over. The third method added to the script is `setOnMouseDragged`. The method is for updating the position of the source object regarding the coordinate of the mouse pointer. The last method is `setOnMouseReleased`. It will put the source circle back to the origin coordinate when the mouse button is released. The source circle will simply go back to its origin after we release the mouse button.

Script 1.10: Add mouse events to the source circle

DragAndDrop.java

```
1 //Imports are omitted
2 public class DragAndDrop extends Application {
+ 3     double mouse_x, mouse_y;
+
+     ...
+ 13     source.getFigure().setOnMousePressed(new EventHandler<MouseEvent>() {
+ 14         @Override
+ 15         public void handle(MouseEvent event) {
+ 16             mouse_x = event.getScreenX();
+ 17             mouse_y = event.getScreenY();
+ 18         }
+ 19     });
+ 20     source.getFigure().setOnDragDetected(new EventHandler<MouseEvent>() {
+ 21         @Override
+ 22         public void handle(MouseEvent event) {
+ 23             source.getFigure().setMouseTransparent(true);
+ 24             source.getFigure().ToFront();
+ 25             source.getFigure().startFullDrag();
+ 26         }
+ 27     });
+ 28     source.getFigure().setOnMouseDragged(new EventHandler<MouseEvent>() {
+ 29         @Override
+ 30         public void handle(MouseEvent event) {
+ 31             double deltaX = event.getScreenX() - mouse_x;
+ 32             double deltaY = event.getScreenY() - mouse_y;
+ 33             source.setPosition(deltaX, deltaY);
+ 34             mouse_x = event.getScreenX();
+ 35             mouse_y = event.getScreenY();
+ 36         }
+ 37     });
+ 38     source.getFigure().setOnMouseReleased(new EventHandler<MouseEvent>() {
+ 39         @Override
+ 40         public void handle(MouseEvent event) {
+ 41             source.setBackOrigin();
+ 42             source.getFigure().setMouseTransparent(false);
+ 43         }
+ 44     });
+ 45     root.getChildren().addAll(source.getFigure(), target.getFigure());
+ 46     stage.setScene(scene);
+ 47     stage.show();
+ 48 }
+ 49 public static void main(String[] args) {
+ 50     Application.launch(args);
+ 51 }
+ 52 }
```



Script 1.11: Add mouse events to the target circle

DragAndDrop.java

```
1 //Imports are omitted
2 public class DragAndDrop extends Application {
3     double mouse_x, mouse_y;
+ 4     boolean isEntered;
5     @Override
6     public void start(Stage stage) {
7         stage.setTitle("Hello Drag and Drop");
8         Group root = new Group();
9         Scene scene = new Scene(root, 500, 200);
10        scene.setFill(Color.WHITE);
11        FigureGroup source = new FigureGroup(80,50, 40, Color.LIGHTBLUE, "SOURCE");
12        FigureGroup target = new FigureGroup(320,50, 50, Color.RED, "DROP TARGET");
+ 13        isEntered = false;
14        ...
41        source.getFigure().setOnMouseReleased(new EventHandler<MouseEvent>() {
42            @Override
43                public void handle(MouseEvent event) {
- 44                source.setBackOrigin();
- 45                source.getFigure().setMouseTransparent(false);
+ 46                if(!isEntered) {
+ 47                    source.setBackOrigin();
+ 48                    source.getFigure().setMouseTransparent(false);
+ 49                } else {
+ 50                    target.getFigure().getChildren().add(source.getFigure());
+ 51                }
52            }
53        });
+ 54        target.getFigure().setOnMouseDragEntered(new EventHandler<MouseEvent>() {
+ 55            @Override
+ 56                public void handle(MouseEvent event) {
+ 57                    isEntered = true;
+ 58                }
+ 59        });
+ 60        target.getFigure().setOnMouseDragExited(new EventHandler<MouseEvent>() {
+ 61            @Override
+ 62                public void handle(MouseEvent event) {
+ 63                    isEntered = false;
+ 64                }
+ 65        });
        ...
```

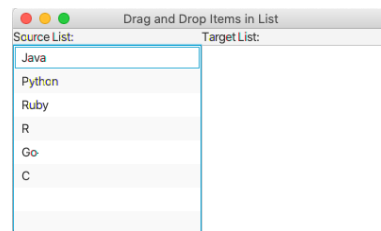
The final step is to complete requirement #3. Since the response to the mouse release action are different regardless of where the pointer is released, the most suitable tool to assist this implementation is to use a flag variable. A flag is a logical concept that is commonly implemented by using a boolean variable. The principal concept is similar to using an electronic switch to control an electric device's on/off state, such as a light bulb. In programming, a flag variable is commonly used to track the event states by means of whether the event is happening or not, or whether the event was happened already or not. In the introductory programming course, we may have already practiced using this flag variable in fundamental problem solving, such as the problem of prime number checking. That is, given an integer  $n$  we probed whether there is a number in the range between two, and the rounded-up of its square root is divisible by it. Before the beginning of the loop, we implemented a flag variable and initiated its value as `false`, and an integer  $i$  with an initial value of two. After we start the loop  $i$  will be increment one by one until it reaches the rounded up of the square root of  $n$ . This flag variable's value will be changed from `false` to `true` if  $n$  is divisible by a particular  $i$ . After the end of the loop, we check whether the flag variable's value has been changed from `false` to `true` or not. If it was, it means that the  $n$  is not a prime number. The notion of a flag variable is commonly used in GUI programmings, such as for implementing a toggle button, a toggle menu, or a conditional rendering component. Similarly, in this example of drag and drop implementation, a flag variable is necessary for completing the requirement #3 because, despite that, the mouse release means the end of the drag-and-drop activity; however, the location it is released will lead to different outcomes. In particular, if the mouse is released outside the target object, the source object should move back to its origin. On the other hand, if the mouse pointer is released over the target object, the source object should be snapped to the target object.

Script 1.11 further modifies Script 1.10 by adding a flag variable and two event handlers to it. Initially, the flag variable named `isEntered` is added to the script at Line 4, and its value is initialized as `false` at Line 13. The two event handlers added to this example are between Line 54 and 65, where the programming logics they carry out are to toggle the flag variable. The `setOnMouseDragEntered` and the `setOnMouseDragExited` methods are binding to the event targets, and they have to toggle the `isEntered` variable when their control signal arrives. The value of the

flag variable will be used in the modified `setOnMouseReleased` method of the event source. Between Line 46 and Line 51, when the mouse is released the method will check the flag variable to know whether the mouse button is released inside or outside the target circle.

### Dragging and dropping items in a ListView

Figure 1.10 shows another typical application example of the drag-and-drop functionality, which is to drag and drop items between lists. Different from that of discussed in the previous example, the drag and drop functionality in this example additionally requires data transferring along with the dragging and dropping. For example, if we drag the word Java from the list on the left-hand side to the list on the right-hand side, it means that not only we move the word Java between the lists, but also all the data and information which are associated with the word Java as well. Thus, in this case, aside from a simple dragging and dropping shapes we did in the previous example, this task has more things to consider, which are as follows:



**Figure 1.10: Drag and drop items between lists**

- How to transfer the data associated with each word between lists.
- When and how to let the target list accept the data being dragged from the source list.
- After dropped, what are needed to be done on the source list and the target list.

At least four mouse events are required to implement all these functionalities. They are `dragDetected`, `dragOver`, `dragDropped`, and `dragDone`. Since either the list on the left-hand side or that on the right-hand side can alternately be the source list and target list, it seems to be better to create a list object once and use them twice on both sides of the application window. Script 1.12 shows the class of this list object. A `ListView` control element is created at Line 5 of the script. The four mouse events are associated with the `ListView` between Line 7 and Line 40. The `setOnDragDetected` method initially creates a `DragBoard`, a universal object container in JavaFX that can be used as a carrier for transferring the data between the drag source and the drop target. In this example, only the string, e.g., Java, is transferred through this drag board. However, in practice, we usually let this drag board store and transfer the entire object. Next, in the `setOnDragOver` method, when a `ListView` detects an object being dragged over it, it will check whether or not it can accept the item for transferring. In this example, the `ListView` will check if the drag board is storing an item with the type string or not. Next, in the `setOnDragDropped` method, we use a flag variable to monitor the state of drag and drop completion. The variable is declared at Line 27, and its state will be changed after a string type variable is successfully transferred to the target list. Finally, the `dragDone` method will remove the item from the source list view if it is successfully transferred to the target `ListView`. The application launcher of this object is shown in Script 1.13. The launcher simply creates a list of strings at Line 7 and Line 8. It then creates two `ListView` elements at Line 9 and Line 10, and adds the string lists as the initial items of the `ListView` on the left-hand side. Once this script is launched, it will show the GUI illustrated in Figure 1.10.

Script 1.12: Drag and drop over ListView elements

DragListModel.java

```
+ 1 //Imports are omitted
+ 2 public class DragListModel {
+ 3     ListView<String> lv;
+ 4     public DragListModel(){
+ 5         lv = new ListView<String>();
+ 6         lv.setPrefSize(200, 200);
+ 7         lv.setOnDragDetected(new EventHandler<MouseEvent>() {
+ 8             public void handle(MouseEvent event) {
+ 9                 Dragboard dragboard = lv.startDragAndDrop(TransferMode.MOVE);
+10                 String selectedItems = lv.getSelectionModel().getSelectedItem();
+11                 ClipboardContent content = new ClipboardContent();
+12                 content.putString(selectedItems);
+13                 dragboard.setContent(content);
+14             }
+15         });
+16         lv.setOnDragOver(new EventHandler <DragEvent>() {
+17             public void handle(DragEvent event) {
+18                 Dragboard dragboard = event.getDragboard();
+19                 if (event.getGestureSource() != lv && dragboard.hasString()) {
+20                     event.acceptTransferModes(TransferMode.MOVE);
+21                 }
+22             }
+23         });
+24         lv.setOnDragDropped(new EventHandler <DragEvent>() {
+25             public void handle(DragEvent event) {
+26                 boolean dragCompleted = false;
+27                 Dragboard dragboard = event.getDragboard();
+28                 if(dragboard.hasString()) {
+29                     String list = dragboard.getString();
+30                     lv.getItems().addAll(list);
+31                     dragCompleted = true;
+32                 }
+33                 event.setDropCompleted(dragCompleted);
+34             }
+35         });
+36         lv.setOnDragDone(new EventHandler <DragEvent>() {
+37             public void handle(DragEvent event) {
+38                 lv.getItems().remove(lv.getSelectionModel().getSelectedItem());
+39             }
+40         });
+41     }
+42 }
```

**Script 1.13: The launcher of the problem of Figure 1.10** DragListExample.java

```
+ 1 //Imports are omitted
+ 2 public class DragListExample extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) {
+ 5         Label sourceListLbl = new Label("Source List: ");
+ 6         Label targetListLbl = new Label("Target List: ");
+ 7         ObservableList<String> list = FXCollections<String>.observableArrayList();
+ 8         list.addAll("Java", "Python", "Ruby", "R", "Go", "C");
+ 9         DragListModel sourceView = new DragListModel();
+10         DragListModel targetView = new DragListModel();
+11         sourceView.listView.getItems().addAll(list);
+12         GridPane pane = new GridPane();
+13         pane.addRow(1, sourceListLbl, targetListLbl);
+14         pane.addRow(2, sourceView.listView, targetView.listView);
+15         VBox root = new VBox();
+16         root.getChildren().add(pane);
+17         Scene scene = new Scene(root);
+18         stage.setScene(scene);
+19         stage.setTitle("Drag and Drop Items in List");
+20         stage.show();
+21     }
+22     public static void main(String[] args) {
+23         Application.launch(args);
+24     }
+25 }
```

### 1.3.3 Keyboard events

Handling a mouse event and a keyboard are not entirely identical. For a mouse event, the user obviously knows the visual element that she or he may intend to control by using the mouse pointer over the specific visual element. Thus, programming is less complicated because we can simply imply that the user wants to control the particular visual element because they have to move the mouse pointer to the exact coordinate on the application by themselves. However, it is not as simple as that of a mouse event for a keyboard event. That is because the user may not be

Script 1.14: A keyboard event example

KeyboardEvent.java

```
+ 1 //Imports are omitted
+ 2 public class KeyboardEventExample extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) {
+ 5         stage.setTitle("Keyboard Event Example");
+ 6         Group root = new Group();
+ 7         Scene scene = new Scene(root, 500, 200);
+ 8         scene.setFill(Color.WHITE);
+ 9         FigureGroup obj = new FigureGroup(200,50,50,Color.CORAL,"FIGURE");
+10         scene.setOnKeyPressed(new EventHandler<KeyEvent>() {
+11             @Override
+12             public void handle(KeyEvent event) {
+13                 if (event.getCode() == KeyCode.UP) {
+14                     obj.setPosition(0,-5);
+15                 } else if (event.getCode() == KeyCode.DOWN) {
+16                     obj.setPosition(0,5);
+17                 } else if (event.getCode() == KeyCode.LEFT) {
+18                     obj.setPosition(-5,0);
+19                 } else if (event.getCode() == KeyCode.RIGHT) {
+20                     obj.setPosition(5,0);
+21                 }
+22             }
+23         });
+24         root.getChildren().add(obj.getFigure());
+25         stage.setScene(scene);
+26         stage.show();
+27     }
+28     public static void main(String[] args) {
+29         Application.launch(args);
+30     }
+31 }
```

able to simply know the outcome of a particular key pressing, without any instruction. Thus, in terms of programming, all the keyboard events must be predefined, and the event source cannot be anything but the entire application window itself. For example, we use a keyboard to move a game character in action games. The associated keys are only meant for such movement for the entire application. Thus, the application screen is typically the event source for keyboard events. Script 1.14 shows an example application, where we can use arrow keys to move the figure in the middle of the application window. Note that the figure used in this example is reused from the previous example illustrated in Script 1.9. The code fragments between Line 12 and 21 show the movement functionality. It is done by redrawing the figure in a relative location to that of before the movement after the event source receives a key code signal from the keyboard.

## 1.4 Case Study

This is the first case study of the course. We will implement a GUI for a game inventory, as presented in Figure 1.11. The GUI is commonly seen in role-playing games (a.k.a., RPG games). All the classes to be implemented in this case study application are related to two entities: Character and Item. For Character, we will initially create two types of characters: Physical and Magical characters. Alongside these two types of characters, we will also create two distinct item types: Weapon and Armor. At the end of this exercise, the final GUI will be the same as that being shown in Figure 1.11 where the character avatar is shown on the right-hand side of the inventory window along with its character attributes. Below the char-



Figure 1.11: The GUI of the case study of Chapter 1



acter attributes is a button labeled as *Generate Character*. The functionality of this button is to recreate a character with random attributes. This is the same as the character re-rolling features in many recent online games. On the left-hand side of the application window, there are two item slots, where we will be able to drag an item from the listed at the bottom of the application window and drop in these slots to enhance the character's attribute. For example, a sword will increase the attack power for a physical character. Note that these items can also be unequipped, bringing the character's attribute back to its original values.

The directory and file structures of the project are as given below:

```
src
├── assets
├── controller
│   ├── AllCustomHandler.java
│   ├── GenCharacter.java
│   ├── GenItemList.java
│   └── Launcher.java
├── model
│   ├── Character
│   │   ├── BasedCharacter.java
│   │   ├── BattleMageCharacter.java
│   │   ├── MagicalCharacter.java
│   │   └── PhysicalCharacter.java
│   ├── Item
│   │   ├── Armor.java
│   │   ├── BasedEquipment.java
│   │   └── Weapon.java
│   └── DamageType.java
└── view
    ├── CharacterPan.java
    ├── EquipPane.java
    └── InventoryPane.java
```

The `assets` folder is the place to store all the program's asset files, which is made available at <http://myweb.cmu.ac.th/passakorn.p/953233/materials/chapter01.zip>. The `Launcher` class is the application launcher where the components of GUI and other global variables are implemented. The view component is all that for presentation, such as to display the character attribute values being stored in a `Character` instance. The model component is for data modeling, data structure, and computations related to the data themselves. They are implemented as objects. For example, the `BasicEquipment` is the parent class of the `Armor` and `Weapon` classes, where the familiar entities of the two classes are implemented. The controller component consists of the fundamental logic of the application.

#### 1.4.1 Model components

Let us begin with the model components. Script 1.15, Script 1.16, Script 1.17, and Script 1.18 illustrate all the classes associated with the game `Character`. In brief, the two types of characters, i.e., `Physical` and `Magical` characters, are extended from the `BasedCharacter` class. One of the most significant benefits of applying this technique is that we can simply render either `Physical` or `Magical` characters as the type of `BasedCharacter` on the GUI. This example lets a `Physical`-type character have a higher value of full health point (`FullHp`) than a `Magical`-type character. This higher `FullHp` is traded-off with its lower based power in this game. Observing Script 1.16 and Script 1.17, we will see that they do not declare any variables. This is because all the variables are in common between the two classes. Thus, it is a better idea to declare these variables in their super class, i.e., the `BasedCharacter` class. The common variables are such as `name`, `fullHP`, and `basedPow`.

**Script 1.15: /src/model/basedCharacter.java**

```
+ 1 //Imports are omitted
+ 2 public class BasedCharacter {
+ 3     protected String name, imgpath;
+ 4     protected DamageType type;
+ 5     protected Integer fullHp, basedDef, basedDef, basedRes;
+ 6     protected Integer hp, power, defense, resistance;
+ 7     protected Weapon weapon;
+ 8     protected Armor armor;
+ 9     public String getName () { return name; }
+10     public String getImagepath() { return imgpath; }
+11     public Integer getHp() { return hp; }
+12     public Integer getFullHp() { return fullHp; }
+13     public Integer getPower() { return power; }
+14     public Integer getDefense() { return defense; }
+15     public Integer getResistance() { return resistance; }
+16     @Override
+17     public String toString() { return name; }
+18 }
```

**Script 1.16: /src/model/Character/PhysicalCharacter.java**

```
+ 1 //Imports are omitted
+ 2 public class PhysicalCharacter extends BasedCharacter {
+ 3     public PhysicalCharacter(String name,String imgpath,int basedDef,int basedRes) {
+ 4         this.name = name;
+ 5         this.type = DamageType.physical;
+ 6         this.imgpath = imgpath;
+ 7         this.fullHp = 50;
+ 8         this.basedPow = 30;
+ 9         this.basedDef = basedDef;
+10         this.basedRes = basedRes;
+11         this.hp = this.fullHp;
+12         this.power = this.basedPow;
+13         this.defense = basedDef;
+14         this.resistance = basedRes;
+15     }
+16 }
```

**Script 1.17: /src/model/Character/MagicalCharacter.java**

```
+ 1 //Imports are omitted
+ 2 public class MagicalCharacter extends BasedCharacter {
+ 3     public MagicalCharacter(String name,String imgpath,int basedDef,int basedRes) {
+ 4         this.name = name;
+ 5         this.type = DamageType.magical;
+ 6         this.imgpath = imgpath;
+ 7         this.fullHp = 30;
+ 8         this.basedPow = 50;
+ 9         this.basedDef = basedDef;
+10         this.basedRes = basedRes;
+11         this.hp = this.fullHp;
+12         this.power = this.basedPow;
+13         this.defense = basedDef;
+14         this.resistance = basedRes;
+15     }
+16 }
```

**Script 1.18: /src/model/Character/DamageType.java**

```
+ 1 //Imports are omitted
+ 2 public enum DamageType {
+ 3     physical,
+ 4     magical
+ 5 }
```

Next, we will implement all the classes associated with `Item`. Script 1.19, Script 1.20, and Script 1.21 illustrate the three associated classes. Like the `Character` class, for both item types, i.e., `Weapon` and `Armor`, their classes are extended from a super class. Later, when we implement the drag and drop equipment functionality. Both the inventory list and the equipment slot can simply be the type of `BasedEquipment` class, and we can store either `Weapon` or `Armor` instance inside it. As can be observed in Script 1.20 and Script 1.21, not every variable is inherited from the `BasedEquipment` class. This is because attributes like defense and resistance are not in common for both `Weapon` and `Armor` classes. For the other components, all the functionalities are similar to that of the `Character` class.

**Script 1.19: /src/model/Item/BasedEquipment.java**

```
+ 1 //Imports are omitted
+ 2 public class BasedEquipment {
+ 3     protected String name;
+ 4     protected String imgpath;
+ 5     public String getName() { return name; }
+ 6     public String getImagepath() { return imgpath; }
+ 7     public void setImagepath(String imgpath) { this.imgpath = imgpath; }
+ 8 }
```

**Script 1.20: /src/model/Item/Weapon.java**

```
+ 1 //Imports are omitted
+ 2 public class Weapon extends BasedEquipment{
+ 3     private int power;
+ 4     private DamageType damageType;
+ 5     public Weapon(String name, int power, DamageType damageType, String imgpath) {
+ 6         this.name = name;
+ 7         this.imgpath = imgpath;
+ 8         this.power = power;
+ 9         this.damageType = damageType;
+10     }
+11     public int getPower() { return power; }
+12     public void setPower(int power) { this.power = power; }
+13     public void setDamageType(DamageType weaponType) {
+14         this.damageType = weaponType;
+15     }
+16     public DamageType getDamageType() { return damageType; }
+17     @Override
+18     public String toString() { return name; }
+19 }
```

**Script 1.21: /src/model/Item/Armor.java**

```
+ 1 //Imports are omitted
+ 2 public class Armor extends BasedEquipment {
+ 3     private int defense, resistance;
+ 4     public Armor(String name, int defense, int resistance, String imgpath) {
+ 5         this.name = name;
+ 6         this.imgpath = imgpath;
+ 7         this.defense = defense;
+ 8         this.resistance = resistance;
+ 9     }
+10     public int getDefense() { return defense; }
+11     public void setDefense(int defense) { this.defense = defense; }
+12     public int getResistance() { return resistance; }
+13     public void setResistant(int resistance) { this.resistance = resistance; }
+14     @Override
+15     public String toString() { return name; }
+16 }
```

### 1.4.2 View components

After all the classes under the model components are implemented, we usually continue to implement view components, mainly to observe the model behaviors. As shown in Figure 1.11, the application has three sub-windows (we also call them as sub panes in JavaFX). Thus, we will implement each of which as a different class. Based on functionality, let the name of the sub pane on the left hand side as EquipPane, that on the right hand side as CharacterPane, and that on the bottom as InventoryPane. Script 1.22, Script 1.23, and Script 1.24 illustrate the implementation of the three view component classes.

In Script 1.22, an Imageview array is created at Line 10. Its values are initialized in the for-loop between Line 11 and Line 15 of the script. The array has the same size as that of the total number of equipments. The functionality of this ImageView is only to display item figures on the screen, where the real object is stored elsewhere in the equipmentArray variable. The return type of the getDetailPane method in the script is Pane, which can be attached directly to any JavaFX node. The drawPane method at the bottom of the script simply attaches the Pane returned from the called method to itself, where the class itself extends the JavaFX ScrollPane visual ele-

**Script 1.22: /src/view/InventoryPane.java**

```

+ 1 //Imports are omitted
+ 2 public class InventoryPane extends ScrollPane {
+ 3     private BasedEquipment[] equipmentArray;
+ 4     public InventoryPane() { }
+ 5     private Pane getDetailsPane() {
+ 6         Pane inventoryInfoPane = new HBox(10);
+ 7         inventoryInfoPane.setBorder(null);
+ 8         inventoryInfoPane.setPadding(new Insets(25, 25, 25, 25));
+ 9         if (equipmentArray!=null) {
+10             ImageView[] imageViewList = new ImageView[equipmentArray.length];
+11             for(int i=0 ; i< equipmentArray.length ; i++) {
+12                 imageViewList[i] = new ImageView();
+13                 imageViewList[i].setImage(new Image(getClass().getClassLoader().
+14                     getResource(equipmentArray[i].getImagepath()).toString()));
+15             }
+16             inventoryInfoPane.getChildren().addAll(imageViewList);
+17         }
+18         return inventoryInfoPane;
+19     }
+20     public void drawPane(BasedEquipment[] equipmentArray) {
+21         this.equipmentArray = equipmentArray;
+22         Pane inventoryInfo = getDetailsPane();
+23         this.setStyle("-fx-background-color:Red;");
+24         this.setContent(inventoryInfo);
+25     }
+26 }

```

ment. Since a `ScrollPane` is also a type of JavaFX node, we can simply attach the entire class to the main application window as to display all its visual elements. Next, in Script 1.23, the essential functionality of the script is to generate a different kind of response based on whether an item is equipped or not. For example, between Line 14 and 20 of the script, the code snippet `equippedWeapon != null` checks whether a weapon is equipped or not. This is a common technique used in GUI known as conditional rendering. If a weapon is equipped, then the weapon image whose file path is stored in the `equippedWeapon` variable will be rendered. Otherwise, `blank.png` will be rendered. This technique is also applied for character image rendering, presented at Line 21 of Script 1.24.

**Script 1.23: /src/view/EquipPane.java**

```
+ 1 //Imports are omitted
+ 2 public class EquipPane extends ScrollPane {
+ 3     private Weapon equippedWeapon;
+ 4     private Armor equippedArmor;
+ 5     public EquipPane() { }
+ 6     private Pane getDetailsPane() {
+ 7         Pane equipmentInfoPane = new VBox(10);
+ 8         equipmentInfoPane.setBorder(null);
+ 9         ((VBox) equipmentInfoPane).setAlignment(Pos.CENTER);
+10         equipmentInfoPane.setPadding(new Insets(25, 25, 25, 25));
+11         Label weaponLbl,armorLbl;
+12         ImageView weaponImg = new ImageView();
+13         ImageView armorImg = new ImageView();
+14         if (equippedWeapon != null) {
+15             weaponLbl = new Label("Weapon:\n"+equippedWeapon.getName());
+16             weaponImg.setImage(new Image(getClass().getClassLoader().getResource(
+17                 equippedWeapon.getImagepath()).toString()));
+18         } else {
+19             weaponLbl = new Label("Weapon:");
+20             weaponImg.setImage(new Image(getClass().getClassLoader().getResource("
+21                 assets/blank.png").toString()));
+22         }
+23         if (equippedArmor != null) {
+24             armorLbl = new Label("Armor: \n"+equippedArmor.getName());
+25             armorImg.setImage(new Image(getClass().getClassLoader().getResource(
+26                 equippedArmor.getImagepath()).toString()));
+27         } else {
+28             armorLbl = new Label("Armor:");
+29             armorImg.setImage(new Image(getClass().getClassLoader().getResource("
+30                 assets/blank.png").toString()));
+31         }
+32         equipmentInfoPane.getChildren().addAll(weaponLbl, weaponImg, armorLbl,
+33             armorImg);
+34         return equipmentInfoPane;
+35     }
+36     public void drawPane(Weapon equippedWeapon, Armor equippedArmor) {
+37         this.equippedWeapon = equippedWeapon;
+38         this.equippedArmor = equippedArmor;
+39         Pane equipmentInfo = getDetailsPane();
+40         this.setStyle("-fx-background-color:Red;");
+41         this.setContent(equipmentInfo);
+42     }
+43 }
```



**Script 1.24: /src/view/CharacterPane.java**

```

+ 1 //Imports are omitted
+ 2 public class CharacterPane extends ScrollPane {
+ 3     private BasedCharacter character;
+ 4     public CharacterPane() { }
+ 5     private Pane getDetailsPane() {
+ 6         Pane characterInfoPane = new VBox(10);
+ 7         characterInfoPane.setBorder(null);
+ 8         characterInfoPane.setPadding(new Insets(25, 25, 25, 25));
+ 9         Label name,type,hp,atk,def,res;
+10         ImageView mainImage = new ImageView();
+11         if (this.character != null) {
+12             name = new Label("Name: "+character.getName());
+13             mainImage.setImage(new Image(getClass().getClassLoader().getResource(
+14                 character.getImagepath().toString()));
+15             hp = new Label("HP: "+character.getHp().toString()+"/"+character.
+16                 getFullHp().toString());
+17             type = new Label("Type: "+character.getType().toString());
+18             atk = new Label("ATK: "+character.getPower());
+19             def = new Label("DEF: "+character.getDefense());
+20             res = new Label("RES: "+character.getResistance());
+21         } else {
+22             name = new Label("Name: ");
+23             mainImage.setImage(new Image(getClass().getClassLoader().getResource("
+24                 assets/unknown.png").toString()));
+25             hp = new Label("HP: ");
+26             type = new Label("Type: ");
+27             atk = new Label("ATK: ");
+28             def = new Label("DEF: ");
+29             res = new Label("RES: ");
+30         }
+31         Button genCharacter = new Button();
+32         genCharacter.setText("Generate Character");
+33         characterInfoPane.getChildren().addAll(name,mainImage,type,hp,atk,def,res,
+34             genCharacter);
+35         return characterInfoPane;
+36     }
+37     public void drawPane(BasedCharacter character) {
+38         this.character = character;
+39         Pane characterInfo = getDetailsPane();
+40         this.setStyle("-fx-background-color:Red;");
+41         this.setContent(characterInfo);
+42     }
+43 }

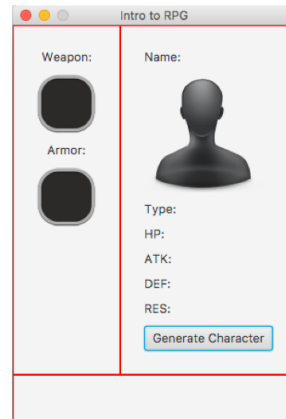
```

### 1.4.3 Controller components and programming logic

After implementing the view components, it is the time to compose all of the implemented classes in the Launcher class. Script 1.25 depicts an example Launcher class of this application. The essential functionality of this Launcher class is between Line 20 and Line 30 of the script, where all the three sub panes are composed into one single JavaFX visual element of the type `BorderPane`. Note that the `refreshPane` method implemented between Line 31 and Line 35 also applies another common GUI technique, which uses only one method to redraw (or recalculate)

all the application sub-components. After completing this step, it is the first time we can launch the application, and the application will show the GUI displayed in Figure 1.12.

Our next task is to implement programming logic. Initializing the Character and the Item list when starting the application is the first logic set we will implement. This will transform the GUI of Figure 1.12 into that of Figure 1.11. Script 1.26 consists of only one method named `setUpCharacter`. This method initially declares a character variable with the type `BasedCharacter` at Line 4. After that, the character variable is instantiated as the type of a `BasedCharacter`'s subclass, i.e., the `MegicalCharacter` and `PhysicalCharacter` classes shown at Line 10 and Line 12 of the script, respectively. This presents how convenience the OOP principle can reduce the complexity of a complicated programming design. Next, Script 1.27 simply initiate an array of `BasedEquipment` whose member are item objects. Finally, the two scripts are integrated into the application by modifying the Launcher class at Line 19 and Line 20 of Script 1.28.



**Figure 1.12: The initialized GUI of the project**

**Script 1.25: /src/controller/Launcher.java**

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     private static Scene mainScene;
+ 4     private static BasedCharacter mainCharacter = null;
+ 5     private static BasedEquipment[] allEquipments = null;
+ 6     private static Weapon equippedWeapon = null;
+ 7     private static Armor equippedArmor = null;
+ 8     private static CharacterPane characterPane = null;
+ 9     private static EquipPane equipPane = null;
+10     private static InventoryPane inventoryPane = null;
+11     @Override
+12     public void start(Stage primaryStage) throws Exception{
+13         primaryStage.setTitle("Intro to RPG");
+14         primaryStage.setResizable(false);
+15         primaryStage.show();
+16         Pane mainPane = getMainPane();
+17         mainScene = new Scene(mainPane);
+18         primaryStage.setScene(mainScene);
+19     }
+20     public Pane getMainPane() {
+21         BorderPane mainPane = new BorderPane();
+22         characterPane = new CharacterPane();
+23         equipPane = new EquipPane();
+24         inventoryPane = new InventoryPane();
+25         refreshPane();
+26         mainPane.setCenter(characterPane);
+27         mainPane.setLeft(equipPane);
+28         mainPane.setBottom(inventoryPane);
+29         return mainPane;
+30     }
+31     public static void refreshPane() {
+32         characterPane.drawPane(mainCharacter);
+33         equipPane.drawPane(equippedWeapon,equippedArmor);
+34         inventoryPane.drawPane(allEquipments);
+35     }
+36     public static BasedCharacter getMainCharacter() { return mainCharacter; }
+37     public static void setMainCharacter(BasedCharacter mainCharacter) {
+38         Launcher.mainCharacter = mainCharacter;
+39     }
+40     public static void main(String[] args) {
+41         launch(args);
+42     }
+43 }
```

**Script 1.26: /src/controller/GenCharacter.java**

```
+ 1 //Imports are omitted
+ 2 public class GenCharacter {
+ 3     public static BasedCharacter setUpCharacter() {
+ 4         BasedCharacter character;
+ 5         Random rand = new Random();
+ 6         int type = rand.nextInt(2)+1;
+ 7         int basedDef = rand.nextInt(50)+1;
+ 8         int basedRes = rand.nextInt(50)+1;
+ 9         if (type == 1) {
+10             character = new MagicalCharacter("MagicChar1", "assets/wizard.png",
+11                 basedDef, basedRes);
+12         } else {
+13             character = new PhysicalCharacter("PhysicalChar1", "assets/knight.png",
+14                 basedRes, basedRes);
+15         }
+16     }
+17 }
```

**Script 1.27: /src/controller/GenItemList.java**

```
+ 1 //Imports are omitted
+ 2 public class GenItemList {
+ 3     public static BasedEquipment[] setUpItemList() {
+ 4         BasedEquipment[] itemLists = new BasedEquipment[5];
+ 5         itemLists[0] = new Weapon("Sword", 10, DamageType.physical, "assets/sword1.png");
+ 6         itemLists[1] = new Weapon("Gun", 20, DamageType.physical, "assets/gun1.png");
+ 7         itemLists[2] = new Weapon("Staff", 30, DamageType.magical, "assets/staff1.png");
+ 8         itemLists[3] = new Armor("shirt", 0, 50, "assets/shirt1.png");
+ 9         itemLists[4] = new Armor("armor", 50, 0, "assets/armor1.png");
+10         return itemLists;
+11     }
+12 }
```

**Script 1.28: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     ...
14     @Override
15     public void start(Stage primaryStage) throws Exception{
16         primaryStage.setTitle("Intro to RPG");
17         primaryStage.setResizable(false);
18         primaryStage.show();
+ 19         mainCharacter = GenCharacter.setUpCharacter();
+ 20         allEquipments = GenItemList.setUpItemList();
21         Pane mainPane = getMainPane();
22     }
23     ...
```

**Script 1.29: /src/view/CharacterPane.java**

```
1 //Imports are omitted
2
3 public class CharacterPane extends ScrollPane {
4     ...
36     Button genCharacter = new Button();
37     genCharacter.setText("Generate Character");
+ 38     generateCharacter.setOnAction(new AllCustomHandler.GenHeroHandler());
39     characterInfoPane.getChildren().addAll(name,mainImage,type,hp,atk,def,res,
40         genCharacter);
41     return characterInfoPane;
42 }
43 ...
```

Continual to the second programming logic set, Script 1.29 adds an event handler that responds to the user signal generated from clicking the *Generate Character* button. Since this application will later implement many more event handlers, it is the lecture decision to collect the event handling methods in a separate class, i.e., the *AllCustomHandler* class illustrated in Script 1.30. The only task the handling method for the *Generate Character* button is to call the *setUpCharacter* method to recreate a new character with randomized attributes and let the *Launcher* class re-render the GUI.

**Script 1.30: /src/controller/AllCustomHandler.java**

```
+ 1 //Imports are omitted
+ 2 public class AllCustomHandler {
+ 3     public static class GenHeroHandler implements EventHandler<ActionEvent> {
+ 4         @Override
+ 5         public void handle(ActionEvent event) {
+ 6             Launcher.setMainCharacter(GenCharacter.setUpCharacter());
+ 7             Launcher.refreshPane();
+ 8         }
+ 9     }
+10 }
```

**1.4.4 Container's matter**

In general, we use an array just to store a set of variables where the size of members is predefined, and the number of members is not subject to any change throughout the program execution. However, in this application, the number of items may not be static. This is because game items can be added or removed anytime throughout the gameplay. Hence, an `ArrayList` appears to be a more appropriate data structure of choice for implementing the inventory slots. Script 1.31 depicts the changes in the item container's type from an array to into an `ArrayList`.

We apply the changes made from the current version of the `GenItemList` class, i.e., Script 1.31, to the application by modifying the `InventoryPane` and the `Launcher` classes as illustrated in Script 1.32 and Script 1.33, respectively. All the changes are to replace all the methods associating with an array with that of associating with an `ArrayList`. For example, The `length` method used in querying an array's size has equivalent functionality as the `size` method used for an `ArrayList` container. Also, different from to simply subscript an array index to retrieve the value, we need to use the `get` method for an `ArrayList`. These changes are shown at Line 13, Line 14, Line 16, and Line 17 of Script 1.32. For the `Launcher` class, the only change is the type of the `allEquipments` variable depicted at Line 7 and Line 8 of the Script 1.33.

**Script 1.31: /src/controller/GenItemList.java**

```
1  //Imports are omitted
2  public class GenItemList {
- 3      public static BasedEquipment[] setUpItemList() {
- 4          BasedEquipment[] itemLists = new BasedEquipment[5];
- 5          itemLists[0] = new Weapon("Sword",10,DamageType.physical,"assets/sword1.png");
- 6          itemLists[1] = new Weapon("Gun",20,DamageType.physical,"assets/gun1.png");
- 7          itemLists[2] = new Weapon("Staff",30,DamageType.magical,"assets/staff1.png");
- 8          itemLists[3] = new Armor("shirt",0,50,"assets/shirt1.png");
- 9          itemLists[4] = new Armor("armor",50,0,"assets/armor1.png");
+10      public static ArrayList<BasedEquipment> setUpItemList() {
+11          ArrayList<BasedEquipment> itemLists = new ArrayList<BasedEquipment>(5);
+12          itemLists.add(new Weapon("Sword",10,DamageType.physical,"assets/sword1.png"));
+13          itemLists.add(new Weapon("Gun",20,DamageType.physical,"assets/gun1.png"));
+14          itemLists.add(new Weapon("Staff",30,DamageType.magical,"assets/staff1.png"));
+15          itemLists.add(new Armor("shirt",0,50,"assets/shirt1.png"));
+16          itemLists.add(new Armor("armor",50,0,"assets/armor1.png"));
17      return itemLists;
18  }
19 }
```

**Script 1.32: /src/view/InventoryPane.java**

```

1  //Imports are omitted
2  public class InventoryPane extends ScrollPane {
- 3      private BasedEquipment[] equipmentArray;
+ 4      private ArrayList<BasedEquipment> equipmentArray;
5      public InventoryPane() { }
6      private Pane getDetailsPane() {
7          Pane inventoryInfoPane = new HBox(10);
8          inventoryInfoPane.setBorder(null);
9          inventoryInfoPane.setPadding(new Insets(25, 25, 25, 25));
10         if (equipmentArray!=null) {
- 11             ImageView[] imageViewList = new ImageView[equipmentArray.length];
+ 12             ImageView[] imageViewList = new ImageView[equipmentArray.size()];
- 13             for(int i=0 ; i< equipmentArray.length ; i++) {
+ 14             for(int i=0 ; i< equipmentArray.size() ; i++) {
15                 imageViewList[i] = new ImageView();
- 16                 imageViewList[i].setImage(new Image(getClass().getClassLoader().
                     getResource(equipmentArray[i].getImagepath()).toString()));
+ 17                 imageViewList[i].setImage(new Image(getClass().getClassLoader().
                     getResource(equipmentArray.get(i).getImagepath()).toString()));
18             }
19             inventoryInfoPane.getChildren().addAll(imageViewList);
20         }
21         return inventoryInfoPane;
22     }
- 23     public void drawPane(BasedEquipment[] equipmentArray) {
+ 24     public void drawPane(ArrayList<BasedEquipment> equipmentArray) {
        ...

```

**Script 1.33: /src/controller/Launcher.java**

```

1  //Imports are omitted
2  public class Launcher extends Application {
3      private static Scene mainScene;
4      private static BasedCharacter mainCharacter = null;
- 5      private static BasedEquipment[] allEquipments = null;
+ 6      private static ArrayList<BasedEquipment> allEquipments = null;
7      private static Weapon equippedWeapon = null;
8      private static Armor equippedArmor = null;
        ...

```



### 1.4.5 Drag-and-drop functionality

After completely implementing all the infrastructural components, it is time for the drag-and-drop functionality. In this example case study, the drag and drop functionality is mainly for data transferring. Notably, we drag and drop the item image to associate it with the game character to modify the character status attribute. The event source here is the item image in the InventoryPane, in which we will drag an item image and drop it in our event target, which is the two boxes in the EquipPane. Several methods will be added to Script 1.34. The first one is the `setOnDragDetected` method, which performs in the following steps: First, it defines a variable named `db` of the type `Dragboard` to store immediate attribute values during the data transfers. The value is emitted from the `imgView.startDragAndDrop(TransferMode.ANY)` method, which can be viewed as a clipboard used for transferring an object between a drag source and the drop target. The updated Script 1.35 binds the event sources with the `setOnDragDetected` method. Note that all the item images are event sources so that we have to bind all of them using a `for`-loop as shown between Line 16 and Line 21 of the script. The last thing to do for setting up the event sources is to declare the data format and the item we are going to use a `DragBoard` with. This declaration is shown at Line 6 of Script 1.36.

#### Script 1.34: `/src/controller/AllCustomHandler.java`

```
1 //Imports are omitted
2 public class AllCustomHandler {
3
4     ...
+ 11     public static void onDragDetected(MouseEvent event, BasedEquipment equipment,
+ 12         ImageView imgView) {
+ 13         Dragboard db = imgView.startDragAndDrop(TransferMode.ANY);
+ 14         db.setDragView(imgView.getImage());
+ 15         ClipboardContent content = new ClipboardContent();
+ 16         content.put(equipment.DATA_FORMAT, equipment);
+ 17         db.setContent(content);
+ 18         event.consume();
+ 19     }
+ 20 }
```

**Script 1.35: /src/view/InventoryPane.java**

```

1 //Imports are omitted
2 public class InventoryPane extends ScrollPane {
3     private ArrayList<BasedEquipment> equipmentArray;
4     ...
13     for(int i=0 ; i< equipmentArray.size() ; i++) {
14         imageViewList[i] = new ImageView();
15         imageViewList[i].setImage(new Image(getClass().getClassLoader().
+ 16             getResource(equipmentArray.get(i).getImagepath()).toString());
+ 17         int finalI = i;
+ 18         imageViewList[i].setOnDragDetected(new EventHandler<MouseEvent>() {
+ 19             public void handle(MouseEvent event) {
+ 20                 onDragDetected(event, equipmentArray.get(finalI), imageViewList[
+ 21                     finalI]);
22             }
23         });
24     }
25     inventoryInfoPane.getChildren().addAll(imageViewList);
26     ...

```

**Script 1.36: /src/model/Item/BasedEquipment.java**

```

1 //Imports are omitted
- 2 public class BasedEquipment {
+ 3 public class BasedEquipment implements Serializable {
+ 4     public static final DataFormat DATA_FORMAT = new DataFormat( "src.model.Item.
        BasedEquipment");
5     protected String name;
6     protected String imgpath;
7     public String getName() { return name; }
8     public String getImagepath() { return imgpath; }
9     public void setImagepath(String imgpath) { this.imgpath = imgpath; }
10 }

```

After implementing the event sources, let us continue on the event targets. Script 1.37 adds two event handlers to the `allCustomHandler` class. They are `onDragOver` and `onDragDropped` methods. Line 23 of Script 1.37 shows a programming technique that checks the class name, whether it can be accepted and processed by the particular event handler or not. We sent type equals to `Weapon` or `Armor` to the method. Hence, only the `Weapon`-type equipment can be equipped in the weapon slot, and also for the `Armor`-type equipment. Line 37 of Script 1.37 presents another programming technique utilizing the container size as an identifier to check whether there is an item being equipped or not. For example, we will stack an `Item` image to a `blank.png` background to define the state of equipping an item. So, if the item container's size is not equal to one, it means there is an equipped item in that specific slot. In such a case, we will unequip the current equipment and equip the new one.

Next, we have to bind the two handling methods with the view component. Between Line 40 and Line 55 of Script 1.38, the two event handling methods depicted in 1.37 are bound with the `weaponIngGroup` and `armorIngGroup` elements of the `EquipPane` class. In addition, the `blank.png` image will be shown when there is no item being equipped. Otherwise, it will show the item image. However, this does not seem to be what we commonly see in the real-world application, where an item is usually stacked with an item slot. Thus, we have to somewhat modify the `EquipPane` class between Line 14 and Line 17, between Line 20 and Line 23, Line 27, Line 35, and Line 57 of Script 1.38 to let an equipping item image stack above the `blank.jpg` image.

Now the drag and drop functionalities should be almost ready. The compiler may tell us about the missing of Getter and Setter methods for several classes. Please add all of them all and rerun the application.

**Script 1.37: /src/controller/AllCustomHandler.java**

```
1 //Imports are omitted
  ...
+ 20 public static void onDragOver(DragEvent event, String type) {
+ 21     Dragboard dragboard = event.getDragboard();
+ 22     BasedEquipment retrievedEquipment = (BasedEquipment)dragboard.getContent(
        BasedEquipment.DATA_FORMAT);
+ 23     if (dragboard.hasContent(BasedEquipment.DATA_FORMAT) && retrievedEquipment.
        getClass().getSimpleName().equals(type))
+ 24         event.acceptTransferModes(TransferMode.MOVE);
+ 25 }
+ 26 public static void onDragDropped(DragEvent event, Label lbl, StackPane imgGroup) {
+ 27     boolean dragCompleted = false;
+ 28     Dragboard dragboard = event.getDragboard();
+ 29     if(dragboard.hasContent(BasedEquipment.DATA_FORMAT)) {
+ 30         BasedEquipment retrievedEquipment = (BasedEquipment)dragboard.getContent(
            BasedEquipment.DATA_FORMAT);
+ 31         if(retrievedEquipment.getClass().getSimpleName().equals("Weapon")) {
+ 32             Launcher.setEquippedWeapon((Weapon) retrievedEquipment);
+ 33         } else {
+ 34             Launcher.setEquippedArmor((Armor) retrievedEquipment);
+ 35         }
+ 36         ImageView imgView = new ImageView();
+ 37         if (imgGroup.getChildren().size()!=1) {
+ 38             imgGroup.getChildren().remove(1);
+ 39             Launcher.refreshPane();
+ 40         }
+ 41         lbl.setText(retrievedEquipment.getClass().getSimpleName() + ":\n" +
            retrievedEquipment.getName());
+ 42         imgView.setImage(new Image(AllCustomHandler.class.getClassLoader().
            getResource(retrievedEquipment.getImagepath()).toString()));
+ 43         imgGroup.getChildren().add(imgView);
+ 44         dragCompleted = true;
+ 45     }
+ 46     event.setDropCompleted(dragCompleted);
+ 47 }
+ 48 }
```

**Script 1.38: /src/view/EquipPane.java**

```

1  //Imports are omitted {
    ...
13      Label weaponLbl,armorLbl;
+ 14      StackPane weaponImgGroup = new StackPane();
+ 15      StackPane armorImgGroup = new StackPane();
+ 16      ImageView bg1 = new ImageView();
+ 17      ImageView bg2 = new ImageView();
18      ImageView weaponImg = new ImageView();
19      ImageView armorImg = new ImageView();
+ 20      bg1.setImage(new Image(getClass().getClassLoader().getResource("assets/blank.png")
        .toString()));
+ 21      bg2.setImage(new Image(getClass().getClassLoader().getResource("assets/blank.png")
        .toString()));
+ 22      weaponImgGroup.getChildren().add(bg1);
+ 23      armorImgGroup.getChildren().add(bg2);
24      if (equippedWeapon != null) {
25          weaponLbl = new Label("Weapon:\n"+equippedWeapon.getName());
26          weaponImg.setImage(new Image(getClass().getClassLoader().getResource(
            equippedWeapon.getImagepath()).toString()));
+ 27          weaponImgGroup.getChildren().add(weaponImg);
        ...
32      if (equippedArmor != null) {
33          armorLbl = new Label("Armor: \n"+equippedArmor.getName());
34          armorImg.setImage(new Image(getClass().getClassLoader().getResource(
            equippedArmor.getImagepath()).toString()));
+ 35          armorImgGroup.getChildren().add(armorImg);
        ...
+ 40      weaponImgGroup.setOnDragOver(new EventHandler<DragEvent>() {
+ 41          @Override
+ 42          public void handle(DragEvent e) { onDragOver(e,"Weapon"); }
+ 43      });
+ 44      armorImgGroup.setOnDragOver(new EventHandler<DragEvent>() {
+ 45          @Override
+ 46          public void handle(DragEvent e) { onDragOver(e, "Armor"); }
+ 47      });
+ 48      weaponImgGroup.setOnDragDropped(new EventHandler<DragEvent>() {
+ 49          @Override
+ 50          public void handle(DragEvent e) { onDragDropped(e, weaponLbl, weaponImgGroup); }
+ 51      });
+ 52      armorImgGroup.setOnDragDropped(new EventHandler<DragEvent>() {
+ 53          @Override
+ 54          public void handle(DragEvent e) { onDragDropped(e, armorLbl, armorImgGroup); }
+ 55      });
- 56      equipmentInfoPane.getChildren().addAll(weaponLbl,weaponImg,armorLbl,armorImg);
+ 57      equipmentInfoPane.getChildren().addAll(weaponLbl,weaponImgGroup,armorLbl,
        armorImgGroup);
    ...

```

### 1.4.6 Attributes propagation

The main purpose of this drag-and-drop implementation is to modify the character status attributes based on the equipment. It can be achieved by transferring the equipment status being stored in the particular object we drag its image to adjust the character status. Notably, we have to implement the status modification logic to propagate the value change to the model and the view components. Soon after the mouse button is released and the drop object is accepted by the target event node, the `BasedCharacter` object will be retrieved from the `Launcher` class and have its status values modified. After the modification is completed we update the object state by using its setter method. These modifications are shown at Line 29, Line 32, Line 35, Line 37, and Line 38 of Script 1.39. The attribute value adjustment programming logics are shown between Line 19 and 27 of Script 1.40.

#### Script 1.39: /src/controller/AllCustomHandler.java

```
1 //Imports are omitted
...
27 if(dragboard.hasContent(BasedEquipment.DATA_FORMAT)) {
28     BasedEquipment retrievedEquipment = (BasedEquipment)dragboard.getContent(
        BasedEquipment.DATA_FORMAT);
+ 29     BasedCharacter character = Launcher.getMainCharacter();
30     if(retrievedEquipment.getClass().getSimpleName().equals("Weapon")) {
31         Launcher.setEquippedWeapon((Weapon) retrievedEquipment);
+ 32         character.equipWeapon((Weapon) retrievedEquipment);
33     } else {
34         Launcher.setEquippedArmor((Armor) retrievedEquipment);
+ 35         character.equipArmor((Armor) retrievedEquipment);
36     }
+ 37     Launcher.setMainCharacter(character);
+ 38     Launcher.refreshPane();
39     ImageView imgView = new ImageView();
...

```

**Script 1.40: /src/modelbasedCharacter.java**

```
1 //Imports are omitted
2 public class BasedCharacter {
3     ...
+ 19     public void equipWeapon( Weapon weapon) {
+ 20         this.weapon = weapon;
+ 21         this.power = this.basedPow + weapon.getPower();
+ 22     }
+ 23     public void equipArmor( Armor armor) {
+ 24         this.armor = armor;
+ 25         this.defense = this.basedDef + armor.getDefense();
+ 26         this.resistant = this.basedRes + armor.getResistant();
+ 27     }
28     @Override
29     public String toString() { return name; }
30 }
```

The final functionality to be implemented this case study is to remove the item from the inventory list after it is equipped, as well as to bring it back to the inventory list after the character is equipped by any other items. The methodology is implemented in Script 1.41 and is similar to that was implemented for the character status modification. The implementation is shown at the modified lines of code above Line 66 of Script 1.41. For the code depicted between Line 86 and Line 93 of Script 1.41, its functionality is to remove the equipment entity from the allEquipments ArrayList once the character has equipped it. This implementation utilizes another technique which can be viewed as a variation of a flag variable. That is, the pos variable is initialized as -1. In the for-loop at Line 86, if the search hit by means that the name of the equipped item and the name of the item in the list are the same, we will set this pos variable as the index in the ArrayList where the search is hit. After the loop, the pos variable will be checked whether its value is still -1 or not. If the value is not equal to -1, we can simply remove the item from the ArrayList because it shows that the item is being equipped, it should no longer appear in the inventory list anymore. Script 1.42 binds the onEquipDone method with the event sources between Line 22 and Line 27. All these events are bound with their event sources between Line 22 and 27 of Script 1.42.

**Script 1.41: intro\_to\_rpg/controller/AllCustomHandler.java**

```

1 //Imports are omitted
  ...
47 public static void onDragDropped(DragEvent event, Label lbl, StackPane imgGroup) {
48     boolean dragCompleted = false;
49     Dragboard dragboard = event.getDragboard();
+ 50     ArrayList<BasedEquipment> allEquipments = Launcher.getAllEquipments();
51     if(dragboard.hasContent(BasedEquipment.DATA_FORMAT)) {
52         BasedEquipment retrievedEquipment = (BasedEquipment)dragboard.getContent(
            BasedEquipment.DATA_FORMAT);
53         BasedCharacter character = Launcher.getMainCharacter();
54         if(retrievedEquipment.getClass().getSimpleName().equals("Weapon")) {
+ 55             if (Launcher.getEquippedWeapon() != null) {
+ 56                 allEquipments.add(Launcher.getEquippedWeapon());
+ 57                 Launcher.setEquippedWeapon((Weapon) retrievedEquipment);
58                 character.equipWeapon((Weapon) retrievedEquipment);
59             } else {
+ 60                 if (Launcher.getEquippedArmor() != null)
+ 61                     allEquipments.add(Launcher.getEquippedArmor());
+ 62                 Launcher.setEquippedArmor((Armor) retrievedEquipment);
63                 character.equipArmor((Armor) retrievedEquipment);
64             }
65             Launcher.setMainCharacter(character);
+ 66             Launcher.setAllEquipments(allEquipments);
67             Launcher.refreshPane();
        ...
+ 81 public static void onEquipDone(DragEvent event) {
+ 82     Dragboard dragboard = event.getDragboard();
+ 83     ArrayList<BasedEquipment> allEquipments = Launcher.getAllEquipments();
+ 84     BasedEquipment retrievedEquipment = (BasedEquipment)dragboard.getContent(
        BasedEquipment.DATA_FORMAT);
+ 85     int pos = -1;
+ 86     for(int i=0; i<allEquipments.size() ; i++) {
+ 87         if (allEquipments.get(i).getName().equals(retrievedEquipment.getName())) {
+ 88             pos = i;
+ 89         }
+ 90     }
+ 91     if (pos != -1) {
+ 92         allEquipments.remove(pos);
+ 93     }
94     Launcher.setAllEquipments(allEquipments);
95     Launcher.refreshPane();
96 }
  ...

```



**Script 1.42: intro\_to\_rpg/view/InventoryPane.java**

```

1  //Imports are omitted
2  public class InventoryPane extends ScrollPane {
    ...
13     for(int i=0 ; i< equipmentArray.size() ; i++) {
14         imageViewList[i] = new ImageView();
15         imageViewList[i].setImage(new Image(getClass().getClassLoader().
16             getResource(equipmentArray.get(i).getImagepath()).toString());
17         int finalI = i;
18         imageViewList[i].setOnDragDetected(new EventHandler<MouseEvent>() {
19             public void handle(MouseEvent event) {
20                 onDragDetected(event, equipmentArray.get(finalI), imageViewList[
21                     finalI]);
22             }
23         });
24         imageViewList[i].setOnDragDone(new EventHandler<DragEvent>() {
25             @Override
26             public void handle(DragEvent event) {
27                 onEquipDone(event);
28             }
29         });
30     }
31     inventoryInfoPane.getChildren().addAll(imageViewList);
    ...

```

After completely implement the latest update of Script 1.42, the application can perform with all the functionalities we have planned. The following subsection is the exercise questions that could help the audiences check their understanding of all the contents discussed in this chapter.

### 1.4.7 Exercise

1. Add one more item to the item list, its image size should be equal to that of the other images.
2. Create a new character type namely `BattleMage`, whose `fullHp` and `basedPower` are both 40. Note that you have to find your own image for the character and the image size must be equal to that of the other two character types.
3. Create one more type of status namely `SPD`. This is also a randomized status with its maximum value of 50. Please make sure that this status can be displayed on the application GUI.
4. If we click the *Generate Character* where there are some items being equipped, the status will be incorrect because it will show the re-randomized status which is not yet equipped by an equipment. Your task is to fix it.
5. Add a button to unequip the equipments from the `EquipmentPane`.
6. All the weapons have their own class determining by the `DamageType` enum. Our task is to allow only weapons with the same class as the character be equipped. Note for `BattleMage`, we allow it to equip any weapon but no armor can be equipped.

## Exception handling

**T**HERE is a classic adage that says *Anything that can go wrong will go wrong*, and it is well known as Murphy's law (Bloch 2003). In real-world software applications, Murphy's law can happen in many forms, such as network cut off, missing files, and attempting to compute what could not be computed. In such a case, the system itself may not be able to do anything and but crash, such as the blue screen problem usually presented in some Windows systems in the past. Java has a particular concept known as `Exception`, which provides us resources to deal with the possible crash. Of course, the `Exception` concept is not a silver bullet that automatically prevents all the possible problems leading to a program crash. In contrast, it provides us guidelines, methodology, and resources necessarily to plan for the possible problems systematically. In short, Java allows every method to have an alternative path if it cannot be completed in its intended usual way by making utilizing `Exception`. In Java programming and implementation, `Exception` events occur during the execution of applications and disrupt the normal flow. These events are such as an encounter of dividing by zero or missing file.

---

Bloch A (2003) Murphy's law. Penguin.

## 2.1 What to do with exceptions?

### 2.1.1 Catching exceptions

The logic of `Exception` can be viewed as a particular form of an if-statement, which is a control flow that allows us to have such a logic depicted in the comments part of Script 2.1. In the same script, the implementation of a typical `Exception` is shown below it from Line 8.

**Script 2.1: A basic idea of exceptions****ExceptionOverview.java**

```
+ 1  /*
+ 2      if(is no problem occurs) {
+ 3          normal flow
+ 4      } else {
+ 5          exception flow
+ 6      }
+ 7  */
+ 8  public class Launcher {
+ 9      public static void main(String[] args) {
+10          try {
+11              normalFlow();
+12          } catch (Exception e) {
+13              exceptionFlow();
+14          }
+15      }
+16  }
```

The control flow statement of Script 2.1 is known as a try-catch block. Its usage will tell the application what to do when an unusual problem happens. Without it, a crash will be otherwise introduced because the application does not know what it should do after the usual problem happens. This is, for example, the blue screen is shown up because the system does not know what to do with the problem just before the blue screen. Of course, the application causing the blue screen cannot know by itself, but the developer has to design and implement the guideline to tell the application how we want it to respond to the specific problems.

### 2.1.2 Try-catch block

Before we go further to discuss the design of an Exception handling method, it is better to know about the anatomy of an Exception first. Java has provided many predefined types of Exception for various possible problems, such as those related to IO, memory, and arithmetic expression. Also, Java lets us create our exception; however, this seems to be quite rare because what Java has provided is much more than sufficient. The main principle of Exception utilization is to attempt to detect the unusual problem when it occurs and handle it in the most sensible way possible. The successful detection and handling of the problem will make the program continue gracefully, rather than nothing but crash. Of course, the user will be much happier to see the program keeps continuing because crashing must be the last thing she or he may want to encounter. It is similar to how irritating when playing games that frequently crash.

**Script 2.2: An more concrete Exception example****ExceptionExample.java**

```
+ 1 public class FileReader {
+ 2     public String retrieve(String path) { ... }
+ 3 }
+ 4 public class Document {
+ 5     public void init() {
+ 6         FileReader fr = new FileReader();
+ 7         String content = fr.retrieve('~ /dummy.txt');
+ 8     }
+ 9 }
+10 public class Launcher {
+11     public static void main(String[] args) {
+12         Document d = new Document();
+13         d.init();
+14     }
+15 }
```

Script 2.2 illustrates a more concrete example considering stack calls. The `Launcher.main` method calls the constructor of the `Document` class at line 12 and then calls the `Document.init` method, through its instance, i.e., the `d` variable, at Line 13. We

call the code at Line 13 as a caller. The `Document.init` method, which later calls the constructor of the `FileReader` class at Line 6. At line 7, the `fr` variable calls the `FileReader.retrieve` method, which is a method of the `FileReader` class. Supposed that the `retrieve` method is being executed in regular operation, the `readFile` method will return a string to the `content` variable of the `Document` class at Line 7. However, alternatively, if the file in the path `~/dummy.txt` does not exist, it will cause a `FileNotFoundException` error. In this code fragment, where an exception handling is not implemented, what to be happened is that the `retrieve` method will immediately crash and terminate. In this case, there will be no string returning from the `fr.retrieve` method.

**Script 2.3: Adding a try-catch block****ExceptionExample.java**

```
1 public class FileReader {
- 2     public String retrieve(string path) { ... }
+ 3     public String retrieve(string path) {
+ 4         try {
+ 5             ...
+ 6         } catch (FileNotFoundException e) {
+ 7             System.out.println("File not found!");
+ 8         }
+ 9     }
10 }
11 public class Document {
12     public void init() {
13         FileReader fr = new FileReader();
14         String content = fr.retrieve('~/dummy.txt');
15     }
16 }
17 public class Test {
18     public static void main(String[] args) {
19         Document d = new Document();
20         d.init(); }
}
```

To react to an Exception such as `FileNotFoundException` by catching it, we have to consider two steps, namely, try and catch.

- **Try:** is a block statement, as shown in Script 2.1. If we know or see that the result of any operations in our program is likely to cause an exception, we surround the code with a try block.
- **Catch:** is a block statement following a try block. It is where we code one or more Exceptionhandlers. Note that a handler is caught in one catch block.

Script 2.3 shows an example after we attempt to handle a `FileNotFoundException` by adding an Exception handler to Script 2.2. The parameter `e` has the type of `FileNotFoundException`, and all the code fragments inside the block are to tell the system not to terminate but only inform the user about the missing file. What is instead happened in this script if the file `dummy.txt` is found to be missing is that the system will search backward through the called stack until it found an appropriate handler that can handle the `FileNotFoundException`. The application will continue with the code inside the catch block at Line 5 and does not crash.

### 2.1.3 The finally block

A finally block can be viewed as the default exit for a try-catch block. It is optional such that we can decide whether to implement a finally block or not. When the finally block is defined, it will always be executed no matter if the Exception occurs or not. For each try-catch block, there can be only one finally block. Even if it may not sound entirely useful, in fact, the finally block is practically useful by means that it can be a tool for the developer to explicitly define the cleanup code segment, which will always be executed. Thus, there will be no clean up code accidentally bypassed by a control element, such as the `continue` or `break` statement. Elsewhere, it is always recommended to use a finally block to guarantee the cleanup code's existence. In that case, a try block is needed and then followed by a finally block. Script 2.4 shows a modification example of Script 2.3 where a finally block is presented after the try-catch block.

Script 2.4: Adding a *finally* block

ExceptionExample.java

```
1 public class FileReader {
2     public String retrieve(String path) {
3         try {
4             ...
5         } catch (FileNotFoundException e) {
6             System.out.println("File not found!");
7         }
8     } finally {
9         System.out.println("Read file done!");
10    }
11 }
12 public class Document {
13     public void init() {
14         FileReader fr = new FileReader();
15         String content = fr.retrieve('~\\dummy.txt');
16     }
17 }
18 public class Test {
19     public static void main(String[] args) {
20         Document d = new Document();
21         d.init(); }
22 }
```

### 2.1.4 Throwing exceptions

Aside from catching Exceptions, we can also choose to delay their catching by passing the Exception back to its caller method. The term *throwing* is used for this passing action. In other words, the term *throwing* an Exception means passing it back to its caller method and leave the method to handle it. For example, we can decide not to implement any try-catch block in the retrieve method of the FileReader class in Script 2.4, but let the retrieve method throw the entire Exception object back to the Document class' init method. This can be done by appending the code: `throws FileNotFoundException` in the `readFile` method' declaration, and moving all the try-catch block to the body of the `init` method in the Document class. This modification is presented in Script 2.5.



The case we decide to handle the Exception immediately, i.e., putting the try-catch block right at the method where the Exception may occur, is formally called *handle* the Exception. On the other hand, the case we decide to delay and throw the Exception back to the caller method is called *propagate* the Exception.

Script 2.5: An more concrete Exception example

ExceptionExample.java

```
1 public class FileReader {
- 2     public String retrieve(String path) {
+ 3     public String retrieve(String path) throws FileNotFoundException {
- 4         try {
5             ...
- 6         } catch (FileNotFoundException e) {
- 7             System.out.println("File not found!");
- 8         } finally {
- 9             System.out.println("Read file done!");
-10        }
-11    }
12 }
13 public class Document {
14     public void init() {
15         FileReader fr = new FileReader();
+16         try {
17             String content = fr.retrieve('~\\dummy.txt');
+18         } catch (FileNotFoundException e) {
+19             System.out.println("File not found!");
+20         } finally {
+21             System.out.println("Read file done!");
+22        }
23    }
24 }
25 public class Test {
26     public static void main(String[] args) {
27         Document d = new Document();
28         d.init(); }
29 }
```

## 2.2 Categorization

Exceptions can be classified into two types: *checked* and *unchecked* Exceptions. The main difference between the two types is that the *checked* Exceptions are checked at the compile time whereas the *unchecked* Exceptions are checked at runtime. As the name implied, a Java compiler can help us check whether there are missing Exception handlers for the *checked* exceptions. The missing of these handlers will make the compiler generated a compilation error and prevent the application from building successfully. Hence, *checked* Exceptions are rarely missed or unhandled.

On the other hand, a compiler is unable to know whether an *unchecked* exception will be raised or not because it does not always happen. For example, an `ArithmeticException` and `ArrayIndexOutOfBoundsException` do not occur every time we execute the application but it depends on the behavioral changes at runtime. Even if many *unchecked* exceptions sound like it will never happen, as stated in the Murphy's law, it can happen. Therefore, even though the compiler does not check these exceptions for us; we do have to prepare for their handling carefully.

## 2.3 Exceptions handling

In the design phase, we have to plan ahead for all the possible types of Exceptions and decide where and how to handle each of which in the most sensible way. This activity is a kind of risk management, where there are several principles for its design as follows:

- If the Exception is more likely to be caused by user error, then it must be better to give the user a chance to correct it;
- Take a sensible default action;
- Show an error message to facilitate further actions.
- Display pop-up dialog to facilitate a quick action that can prevent any further exceptions due to misunderstanding.

The benefits of considering to handle `Exception`, as opposed to letting the system crash every time it encounters an error, are that we will not forget to prepare the flow for common failure. Furthermore, the predefined `Exception` objects in Java provide us a high-level summary of `Exception`, including a stack trace for every time an `Exception` is encountered. This stack trace is nowadays a useful material for collaboratively troubleshooting a problem in online communication forums, such as the [Reddit](https://www.reddit.com) (2020) and [StackOverflow](https://www.stackoverflow.com) (2020) websites. Further, after we practice `Exception` handling well, we will develop a kind of consistent logical thought where we will more frequently think about separating normal flows from exceptional flows. This is an essential fundamental for future courses and the real-world practices in software development.

---

[Reddit](https://www.reddit.com) (2020) [Reddit](https://www.reddit.com): the front page of the internet. Retrieved May, 2020, from <https://www.reddit.com>

[StackOverflow](https://www.stackoverflow.com) (2020) [Stack Overflow](https://www.stackoverflow.com) - Where Developers Learn, Share, & Build Careers, Retrieved May, 2020, from <https://www.stackoverflow.com>

## 2.4 Case study

This chapter will implement an application that visualizes the historical currency exchange rate records in line graph format. Figure 2.1 shows the GUI of the final product of this application. Specifically, the application has five initial requirements as follows:

1. The user shall be able to view the exchange rate data on the main GUI screen;
2. The user shall be able see the historical exchange rate data in the line chart format, where the range of the date shall be 14 days;
3. The data shall be the most up to date;
4. The user shall be able to change target currency as to show the exchange rate between the selected currency and Thai Baht.
5. The user shall be notified when the exchange rate has become lower than that of she or he is expecting.

As shown in Figure 2.1, there are three buttons in this application: *Refresh*, *Change*, and *Watch*. The *Refresh* button will reinitialize the application by enforcing the data retrieval process and redrawing all the application's visual elements. The *Change* button will let the user change the target currency, and finally, the *Watch* button will let the user enter the target exchange rate in which the user may want to monitor it. Before continue further, let the name of the project of this case be currency\_watcher and the project structure be as follows:

```
src
└─ controller
```

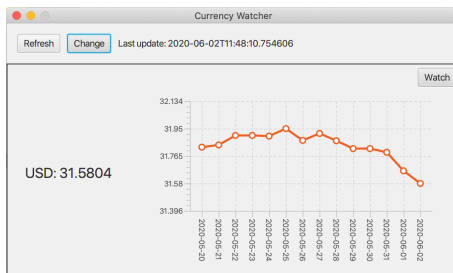
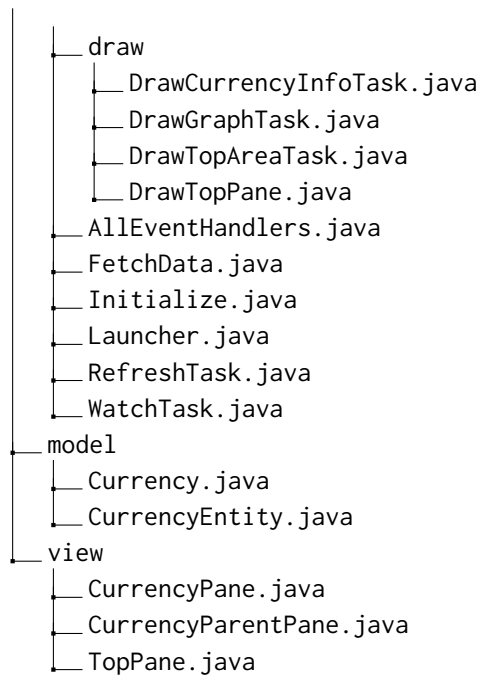


Figure 2.1: The GUI of the case study of Chapter 2



### 2.4.1 Looking for the data supplier

The requirements state the need for the information source to provide a real-time currency exchange rate information. Thus, we have to search for an online service that supplies accurate information at a reasonable price. The Reddit and StackOverflow websites are the places where software developers commonly select to investigate before

any other places because they are where an abundance of references and benchmarks have been made available. In this case study, let us examine the top comment of the url <https://stackoverflow.com/questions/3139879/how-do-i-get-currency-exchange-rates-via-an-api-such-as-google-finance>, which shows a very comprehensive benchmark between several well known data providers. In particular, let us see what we can do with the currency exchange rate data provided by <https://free.currencyconverterapi.com/>. After we land on the webpage and scroll through it, we will see the usage limit for a free account type, as shown in Figure 2.2. The usage seems to be almost perfect for us except that this API allows us to query only up to 8 days per query, where as our requirements say that we have to provide the exchange rate records up to 14 days. Thus, if this API is the only choice, we may have to do some trick to get the entire range of the exchange rate information of 14 days. Let us discuss this later.

#### Free Forex API Rate Limits

Currency Pairs per Request: 2  
Number of Requests per Hour: 100  
Date Range in History: 8 Days  
Allowed Back in History: 1 Year(s)

**Figure 2.2: Free usage limit of free.currconv.com.**

This API only allows a registered user to access it, so we do have to register for its service. Once the registration is completed, the API key will be sent to the email we used for registration. Then, let us query for some data. For example, if we put the query [https://free.currconv.com/api/v7/convert?q=USD\\_THB,THB\\_USD&compact=ultra&date=2019-05-30&endDate=2019-05-31&apiKey=XXXXX](https://free.currconv.com/api/v7/convert?q=USD_THB,THB_USD&compact=ultra&date=2019-05-30&endDate=2019-05-31&apiKey=XXXXX), on a web browser where XXXXX is the API key string, we will get `{"USD_THB":{"2020-05-30":31.815038,"2020-05-31":31.790099,"2020-06-01":31.665499},"THB_USD":{"2020-05-30":0.031432,"2020-05-31":0.031456,"2020-06-01":0.03158}}` as a return result in JSON format.

### 2.4.2 Extracting information from the API

The next step is to create three initial packages under the `src` folder, namely `controller`, `model`, and `view`. The first two classes to be implemented are for connecting our application with the API to retrieve the currency exchange rate data. Mainly, the two classes are the `CurrencyEntity` and `FetchData` classes. Their implementation are presented in Script 2.6 and Script 2.7, respectively. The `CurrencyEntity` class is a data structure for the currency entity, where the exchange rate and the associated date are bound together. The `FetchData` class has only one method named `fetch_range` which has two arguments: `src` and `N`. The `src` variable is the target currency that we will query for its exchange rate into Thai Baht. The `N` variable is the number of days before the current date. It will be used for querying the date range from the API. This `fetch_range` method begins with setting up the date range for querying through the API by marking the current date as the end day. This setup is shown in Line 5 and Line 6 of Script 2.7. Then, the API query string is assembled into the API url at Line 8. At Line 9, the method prepares an `ArrayList` as the query result container to be returned to the method's caller.

#### Script 2.6: `/src/model/CurrencyEntity.java`

```
+ 1 //Imports are omitted
+ 2 public class CurrencyEntity {
+ 3     private Double rate;
+ 4     private String date;
+ 5     public CurrencyEntity(Double rate, String date) {
+ 6         this.rate = rate;
+ 7         this.date = date;
+ 8     }
+ 9     public Double getRate() {
+10         return rate;
+11     }
+12     public String getTimestamp() {
+13         return date;
+14     }
+15     public String toString() {
+16         return String.format("%s %.4f", date, rate);
+17     }
+18 }
```

**Script 2.7: /src/controller/FetchData.java**

```
+ 1 //Imports are omitted
+ 2 public class FetchData {
+ 3     private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern
+ 4         ("yyyy-MM-dd");
+ 5     public static ArrayList<CurrencyEntity> fetch_range(String src, int N) {
+ 6         String dateEnd = LocalDate.now().format(formatter);
+ 7         String dateStart = LocalDate.now().minusDays(N).format(formatter);
+ 8         String apiKey = "XXXXXXX";
+ 9         String url_str = String.format("https://free.currconv.com/api/v7/convert?q
+ 10             =%s_THB&compact=ultra&date=%s&endDate=%s&apiKey=%s", src, dateStart,
+ 11             dateEnd, apiKey);
+ 12         ArrayList<CurrencyEntity> histList = new ArrayList<>();
+ 13         return histList;
+ 14     }
+ 15 }
```

The data returned from the API is in JSON format and we have to parse it for ease of future utilization. Based on the lecturer's opinion, two Java libraries named *JSON in Java* and *Apache Commons IO* are of the most straightforward approaches to parse any data into JSON. The two libraries can be retrieved from <https://mvnrepository.com/artifact/org.json/json/20200518> and <https://mvnrepository.com/artifact/commons-io/commons-io/2.7>, respectively. We can integrate the two libraries into our project by downloading their JAR version and adding the two downloaded JAR files to the list of project dependencies. This will make the application ready to fetch the currency exchange rate data from the API. Script 2.8 shows the updated FetchData class. The code fragment at Line 11 of the script attempts to connect to the API as if we pass the url to a web browser. The method `IOUtils.toString` attempts to parse the JSON formatted HTML output into a raw string. Between Line 12 and 18 of the script, the while-loop iteratively extracts the exchange rates and its associated dates into <key, value> pairs, where the key is a date and the value is the exchange rate. After all the <key, value> pairs have been added to the histlist variable, the code fragments between Line 19 and Line 24 attempt to ensure that the dates are ordered by sorting the histlist variable in ascending order. This sorting is necessary because JSON is an unordered data structure type. Finally, the method returns the histlist variable to the method caller at Line 25.



**Script 2.8: /src/controller/FetchData.java**

```
1  //Imports are omitted
2  public class FetchData {
3      ...
10     ArrayList<CurrencyEntity> histList = new ArrayList<>();
+ 11     String retrievedJson = IOUtils.toString(new URL(url_str), Charset.
        defaultCharset() );
+ 12     JSONObject jsonObj = new JSONObject(retrievedJson).getJSONObject(String.
        format("%s_THB",src));
13     Iterator keysToCopyIterator = jsonObj.keys();
+ 14     while(keysToCopyIterator.hasNext()) {
+ 15         String key = (String) keysToCopyIterator.next();
+ 16         Double rate = Double.parseDouble(jsonObj.get(key).toString());
+ 17         histList.add(new CurrencyEntity(rate, key));
+ 18     }
+ 19     histList.sort( new Comparator<CurrencyEntity>() {
+ 20         @Override
+ 21         public int compare(CurrencyEntity o1, CurrencyEntity o2) {
+ 22             return o1.getTimestamp().compareTo(o2.getTimestamp());
+ 23         }
+ 24     });
25     return histList;
26 }
27 }
```

### 2.4.3 Adding Exception handlers

If we observe Script 2.8 in the IDE, we will see that a compile error is detected at Line 11. This is an example of a *checked* exception, in which the compiler can suggest the missing Exceptionhandling methods for us. In this case, the possible problems that may cause the application not exhibited the normal flow are the validity of the url format and the validity of output variable's format. Script 2.9 shows an example of a valid try-catch block, which can pass the compiler check. Note that there are two catch blocks in this example. Each of which separately handles the url formatting problem and the output formatting problem.

Script 2.10 shows an example Launcher class of the application. Once the application is launched, there should be no error and the terminal should show the output in the same format as [2020-05-27 31.9170, 2020-05-28 31.8680, 2020-05-29 31.8150, 2020-05-30 31.8150, 2020-05-31 31.7901, 2020-06-01 31.6655]

**Script 2.9: /src/controller/FetchData.java**

```
1 //Imports are omitted
2 public class FetchData {
    ...
10     ArrayList<CurrencyEntity> histList = new ArrayList<>();
+ 11     String retrievedJson = null;
+ 12     try {
13         retrievedJson = IOUtils.toString(new URL(url_str),Charset.defaultCharset());
+ 14     } catch (MalformedURLException e) {
+ 15         System.out.println("Encountered a Malformed Url exception");
+ 16     } catch (IOException e) {
+ 17         System.out.println("Encounter an IO exception");
+ 18     }
19     JSONObject jsonObj = new JSONObject(retrievedJson).getJSONObject(String.format(
        "%s_THB",src));
20     Iterator keysToCopyIterator = jsonObj.keys();
    ...
10 }
```

**Script 2.10: /src/controller/Launcher.java**

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     private static Stage primaryStage;
+ 4     @Override
+ 5     public void start(Stage primaryStage) {
+ 6         this.primaryStage = primaryStage;
+ 7         this.primaryStage.setTitle("Currency Watcher");
+ 8         this.primaryStage.setResizable(false);
+ 9         System.out.println(FetchData.fetch_range("USD",6));
+ 10        this.primaryStage.show();
+ 11    }
+ 12 }
```

### 2.4.4 The view components

At the first launch, the application should be able to connect to the data supplier and retrieve the exchange rate from it. Next, we will implement all the essential components for the application GUI by beginning with the model components. Script 2.11 creates a class named `Currency` under the `model` package. This `Currency` class stores the information about a particular currency, such as the currency short code, the state saying whether the currency is being watched or not, and the watch rate. At Line 5 of the script, the `historical` variable is declared to be the variable that later stores the data retrieved from the `fetch_range` method, so it has the same type as the return type of the `fetch_range` method. Note that all the Getter and Setter methods are omitted in this handout due to space problems.

Script 2.11: `/src/model/Currency.java`

```
+ 1 //Imports are omitted
+ 2 public class Currency {
+ 3     private String shortCode;
+ 4     private CurrencyEntity current;
+ 5     private ArrayList<CurrencyEntity> historical;
+ 6     private Boolean isWatch;
+ 7     private Double watchRate;
+ 8     public Currency(String shortCode) {
+ 9         this.shortCode = shortCode;
+10         this.isWatch = false;
+11         this.watchRate = 0.0;
+12     }
+13 }
```

The next step is to create the view components for the `Currency` object. In Figure 2.1, we may see that the application has two separated segments: the top pane and the body pane, which are named `TopPane` and `CurrencyPane`, respectively. Script 2.12 shows the implementation of the `TopPane` class. The pane has two buttons in which their functionalities will be implemented later. The only point worth noting here is the presence of the `refreshPane` method at Line 17. This may make us imply that with the Refresh button is clicked, the information related to the timestamp will be updated on the pane.

**Script 2.12: /src/view/TopPane.java**

```
+ 1 //Imports are omitted
+ 2 public class TopPane extends FlowPane {
+ 3     private Button refresh;
+ 4     private Button change;
+ 5     private Label update;
+ 6     public TopPane() {
+ 7         this.setPadding(new Insets(10));
+ 8         this.setHgap(10);
+ 9         this.setPrefSize(640,20);
+10         change = new Button("Change");
+11         refresh = new Button("Refresh");
+12         update = new Label();
+13         refreshPane();
+14         this.getChildren().addAll(refresh,change,update);
+15     }
+16     public void refreshPane() {
+17         update.setText(String.format("Last update: %s",LocalDateTime.now().toString()));
+18     }
+19 }
```

The CurrencyPane class extends the BorderPane class, a pane type that allows us to compose several panes in it, i.e., the top, left, and body areas. These components will be a composition of the currency information area, the graph area, and the watch button, respectively. They all are implemented separately as three different panes named InfoPane, GraphPane, and TopArea, respectively. Script 2.13 illustrates the implementation of the CurrencyPane class.

**Script 2.13: /src/view/CurrencyPane.java**

```

+ 1 //Imports are omitted
+ 2 public class CurrencyPane extends BorderPane {
+ 3     private Currency currency;
+ 4     private Button watch;
+ 5     public CurrencyPane(Currency currency) {
+ 6         this.watch = new Button("Watch");
+ 7         this.setPadding(new Insets(0));
+ 8         this.setPrefSize(640,300);
+ 9         this.setStyle("-fx-border-color: black");
+10         this.refreshPane(currency);
+11     }
+12     public void refreshPane(Currency currency) {
+13         this.currency = currency;
+14         Pane currencyInfo = genInfoPane();
+15         Pane topArea = genTopArea();
+16         this.setTop(topArea);
+17         this.setLeft(currencyInfo);
+18     }
+19     private Pane genInfoPane() {
+20         VBox currencyInfoPane = new VBox(10);
+21         currencyInfoPane.setPadding(new Insets(5, 25, 5, 25));
+22         currencyInfoPane.setAlignment(Pos.CENTER);
+23         Label exchangeString = new Label("");
+24         Label watchString = new Label("");
+25         exchangeString.setStyle("-fx-font-size: 20;");
+26         watchString.setStyle("-fx-font-size: 14;");
+27         if (this.currency != null) {
+28             exchangeString.setText(String.format("%s: %.4f",currency.getShortCode(),
+29                 currency.getCurrent().getRate()));
+30             if(this.currency.getWatch() == true) {
+31                 watchString.setText(String.format("(Watch @%.4f)",this.currency.
+32                     getWatchRate()));
+33             }
+34         }
+35         currencyInfoPane.getChildren().addAll(exchangeString,watchString);
+36         return currencyInfoPane;
+37     }
+38     private HBox genTopArea() {
+39         HBox topArea = new HBox(10);
+40         topArea.setPadding(new Insets(5));
+41         topArea.getChildren().addAll(watch);
+42         ((HBox) topArea).setAlignment(Pos.CENTER_RIGHT);
+43         return topArea;
+44     }
+45 }

```

### 2.4.5 The controller components

Next, we modify the Launcher class and let it render all the view components implemented up to this step. Script 2.14 presents the updated Launcher class. The Initialize instance presented at Line 16 creates an instance of the Currency class and initializes them throughout the application. The implementation of the Initialize class is presented in Script 2.15. At Line 16 of Script 2.14, the `initMainPane` method is called for initializing the `TopPane` and `CurrencyPane` instances. Up to here, only the implementation for the `GraphPane` component remains.

Script 2.14: `/src/controller/Launcher.java`

```
1  //Imports are omitted
2  public class Launcher extends Application {
3      private static Stage primaryStage;
+ 4      private static Scene mainScene;
+ 5      private static FlowPane mainPane;
+ 6      private static TopPane topPane;
+ 7      private static CurrencyPane currencyPane;
+ 8      private static Currency currency;
9      @Override
10     public void start(Stage primaryStage) {
11         this.primaryStage = primaryStage;
12         this.primaryStage.setTitle("Currency Watcher");
13         this.primaryStage.setResizable(false);
- 14         System.out.println(FetchData.fetch_range("USD",6));
+ 15         Initialize.initialize_app();
+ 16         initMainPane();
+ 17         mainScene = new Scene(mainPane);
+ 18         this.primaryStage.setScene(mainScene);
19         this.primaryStage.show();
20     }
+ 21     public void initMainPane() {
+ 22         mainPane = new FlowPane();
+ 23         topPane = new TopPane();
+ 24         currencyPane = new CurrencyPane(this.currency);
+ 25         mainPane.getChildren().add(topPane);
+ 26         mainPane.getChildren().add(currencyPane);
+ 27     }
28 }
```

**Script 2.15: /src/controller/Initialize.java**

```
+ 1 //Imports are omitted
+ 2 public class Initialize {
+ 3     public static void initialize_app() {
+ 4         Currency c = new Currency("USD");
+ 5         ArrayList<CurrencyEntity> c_list = FetchData.fetch_range(c.getShortCode());
+ 6         c.setHistorical(c_list);
+ 7         c.setCurrent(c_list.get(c_list.size() -1 ));
+ 8         Launcher.setCurrency(c);
+ 9     }
+10 }
```

Script 2.16 illustrates a class named `DrawGraphTask` that draws a `VBox` visual element on the `CurrencyPane`. This class implements the Java `Callable` interface, which is a technique that allows the class to be executed as a background thread running, which is the same thread where the application launcher is running. The fundamental about background processes and threads will be thoroughly discussed in the next chapter. The `call` method is the entry point for the class that will be executed on the background. Between Line 20 and Line 24 of the script, the method iteratively examines all the pairs of currency exchange and the time stamp being stored in the currency instance's historical variable. Then, it put these key and value pairs on the graph based on the coordinates  $(x, y)$  at Line 29. Then, the graph visual element is added to the `GraphPane` at Line 31 and the `GraphPane` is returned to the method caller at Line 32. The `call` method in this `DrawGraphTask` class is executed by the `refreshPane` method in Script 2.17.

Script 2.17 shows an example of a *propagate* exception example. The `refreshPane` method appearing at Line 20 of the script was designed not to handle the exceptions immediately but to pass the `Exception` back to its caller.

**Script 2.16: /src/controller/DrawGraphTask.java**

```
+ 1 //Imports are omitted
+ 2 public class DrawGraphTask implements Callable<VBox> {
+ 3     Currency currency;
+ 4     public DrawGraphTask(Currency currency) {
+ 5         this.currency = currency;
+ 6     }
+ 7     @Override
+ 8     public VBox call() throws Exception {
+ 9         VBox graphPane = new VBox(10);
+10         graphPane.setPadding(new Insets(0, 25, 5, 25));
+11         CategoryAxis xAxis = new CategoryAxis();
+12         NumberAxis yAxis = new NumberAxis();
+13         yAxis.setAutoRanging(true);
+14         LineChart lineChart = new LineChart(xAxis,yAxis);
+15         lineChart.setLegendVisible(false);
+16         if (this.currency != null) {
+17             XYChart.Series series = new XYChart.Series();
+18             double min_y = Double.MAX_VALUE;
+19             double max_y = Double.MIN_VALUE;
+20             for (CurrencyEntity c : currency.getHistorical()) {
+21                 series.getData().add(new XYChart.Data(c.getTimestamp(),c.getRate()));
+22                 if(c.getRate() > max_y) max_y = c.getRate();
+23                 if(c.getRate() < min_y) min_y = c.getRate();
+24             }
+25             yAxis.setAutoRanging(false);
+26             yAxis.setLowerBound(min_y-(max_y-min_y)/2);
+27             yAxis.setUpperBound(max_y+(max_y-min_y)/2);
+28             yAxis.setTickUnit((max_y-min_y)/2);
+29             lineChart.getData().add(series);
+30         }
+31         graphPane.getChildren().add(lineChart);
+32         return graphPane;
+33     }
+34 }
```



**Script 2.17: /src/view/CurrencyPane.java**

```
1 //Imports are omitted
2 public class CurrencyPane extends BorderPane {
3     ...
4
5     public CurrencyPane(Currency currency) {
6         this.watch = new Button("Watch");
7         this.setPadding(new Insets(0));
8         this.setPrefSize(640,300);
9         this.setStyle("-fx-border-color: black");
10        try {
+ 11            this.refreshPane(currency);
12        } catch (ExecutionException e) {
+ 13            System.out.println("Encountered an execution exception.");
+ 14        } catch (InterruptedException e) {
+ 15            System.out.println("Encountered an interrupted exception.");
+ 16        }
+ 17    }
18 }
- 19 public void refreshPane(Currency currency) {
+ 20 public void refreshPane(Currency currency) throws ExecutionException,
    InterruptedException {
21     this.currency = currency;
22     Pane currencyInfo = genInfoPane();
+ 23     FutureTask futureTask = new FutureTask<VBox>(new DrawGraphTask(currency));
+ 24     ExecutorService executor = Executors.newSingleThreadExecutor();
+ 25     executor.execute(futureTask);
+ 26     VBox currencyGraph = (VBox) futureTask.get();
27     Pane topArea = genTopArea();
28     this.setTop(topArea);
29     this.setLeft(currencyInfo);
+ 30     this.setCenter(currencyGraph);
31 }
    ...
```

### 2.4.6 Event handlers

The current update of the application should make its GUI rendered the same as in Figure 2.1, but the historical exchange rate is only presented for seven days. Next, we will implement the functionalities for the buttons by beginning with the Refresh button. Script 2.18, Script 2.19, and Script 2.20 show minor changes to the Launcher, TopPane, and a new class named AllEventHandlers, which is constructed to handle all the events in this application.

#### Script 2.18: /src/controller/Launcher.java

```
1 //Imports are omitted
2 public class Launcher extends Application {
    ...
27     mainPane.getChildren().add(currencyPane);
28 }
+ 29 public static void refreshPane() {
+ 30     topPane.refreshPane();
+ 31     currencyPane.refreshPane(currency);
+ 32 }
33 }
```

#### Script 2.19: /src/view/TopPane.java

```
1 //Imports are omitted
2 public class TopPane extends FlowPane {
    ...
12     refresh = new Button("Refresh");
+ 13     refresh.setOnAction(new EventHandler<ActionEvent>() {
+ 14         @Override
+ 15         public void handle(ActionEvent event) {
+ 16             AllEventHandlers.onRefresh();
+ 17         }
+ 18     });
19     update = new Label();
20 }
    ...
```

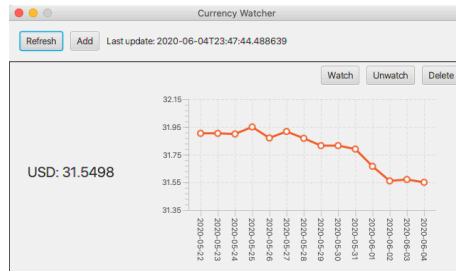
**Script 2.20: /src/controller/AllEventHandlers.java**

```

+ 1 //Imports are omitted
+ 2 public class AllEventHandlers {
+ 3     public static void onRefresh() {
+ 4         try {
+ 5             Launcher.refreshPane();
+ 6         } catch (Exception e) {
+ 7             e.printStackTrace();
+ 8         }
+ 9     }
+ 10 }

```

Suppose that we show the current application snapshot to our client and the client asked us to change some functionalities, such that in the current version, she or he can monitor only one single currency at a time; however, in real-world, the clients of the application of this kind are more likely to want to monitor more than one currency simultaneously. Instead of the *Change* button, the client asks us to rather replace it with an *Add* and a *Delete* buttons. For the other functionalities, all we have to do is to complete all the unfinished requirements. In the end, the application GUI shall be that same as that being shown in Figure 2.3.



**Figure 2.3: The updated GUI of the project**

### 2.4.7 Requirement change handling

The most significance in this change request is to handle multiple currency objects simultaneously, i.e., both models and views. Therefore, we have to change the data structure from an object into a list of objects. Furthermore, we also have to modify some contents in the list on the fly. This makes `ArrayList` be a more suitable choice for approaching the problem. In the previous chapter's case study, we have already converted an array to an `ArrayList` once. In this chapter, we will investigate a more sophisticated use case example of an `ArrayList`. Script 2.21 and Script 2.22 change the Launcher class's currency variable to be in the type `ArrayList<Currency>` and propagate the change to the Initialize class.

#### Script 2.21: /src/controller/Launcher.java

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     ...
4     private static CurrencyPane currencyPane;
5     private static Currency currency;
6     private static ArrayList<Currency> currencyList;
7     ...
8     public static void refreshPane() {
9         topPane.refreshPane();
10        currencyPane.refreshPane(currency);
11        currencyPane.refreshPane(currencyList);
12    }
13 }
14 }
```

#### Script 2.22: /src/controller/Initialize.java

```
1 //Imports are omitted
2 public class Initialize {
3     public static void initialize_app() {
4         ...
5         c.setCurrent(c_list.get(c_list.size() -1 ));
6         ArrayList<Currency> currencyList = new ArrayList<>();
7         currencyList.add(c);
8         Launcher.setCurrency(c);
9         Launcher.setCurrency(currencyList);
10    }
11 }
12 }
```

Apart from declaring the `ArrayList` element, we also have to concern about how to render or re-render multiple objects being stored in the `ArrayList` when needed. Notably, we have to assemble the three sub-components for each currency into one object view. Furthermore, we may have multiple `CurrencyPanes` when the `ArrayList` is storing more than one currency. Therefore, we need a dedicated controller component that can help us manage multiple currency objects at the same time. Script 2.23 and Script 2.24 shows the new class and the modified Launcher class, respectively.

**Script 2.23: /src/view/CurrencyParentPane.java**

```
+ 1 //Imports are omitted
+ 2 public class CurrencyParentPane extends FlowPane {
+ 3     public CurrencyParentPane(ArrayList<Currency> currencyList) throws
+ 4         ExecutionException, InterruptedException {
+ 5         this.setPadding(new Insets(0));
+ 6         refreshPane(currencyList);
+ 7     }
+ 8     public void refreshPane(ArrayList<Currency> currencyList) throws
+ 9         ExecutionException, InterruptedException {
+10         this.getChildren().clear();
+11         for(int i=0 ; i<currencyList.size() ; i++) {
+12             CurrencyPane cp = new CurrencyPane(currencyList.get(i));
+13             this.getChildren().add(cp);
+14         }
+15     }
+16 }
```

**Script 2.24: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     ...
4     private static TopPane topPane;
5     private static CurrencyPane currencyPane;
6     private static CurrencyParentPane currencyParentPane;
7     private static Currency currency;
8     ...
9     public void initMainPane() {
10         mainPane = new FlowPane();
11         topPane = new TopPane();
12         currencyPane = new CurrencyPane(this.currency);
13         currencyParentPane = new CurrencyParentPane(this.currencyList);
14         mainPane.getChildren().add(topPane);
15         mainPane.getChildren().add(currencyPane);
16     }
17     public static void refreshPane() {
18         topPane.refreshPane();
19         currencyPane.refreshPane(currencyList);
20         currencyParentPane.refreshPane(currencyList);
21     }
22 }
```

Next, we have to implement the *Add* and the *Delete* buttons. The *Add* button's functionality is more straightforward somewhat because we only need one button that let the application create a *Currency* object and adds the object to the *ArrayList*. However, the *Delete* button's functionality is more complicated because having only one *Delete* button in the application is not sufficient, as one *Delete* button is required for each *CurrencyPane*. Script 2.25 and Script 2.26 illustrate the modified *AllEventHandlers* and the *TopPane* classes, respectively.

**Script 2.25: /src/controller/AllEventHandlers.java**

```
1  //Imports are omitted
2  public class AllEventHandlers {
    ...
+11  public static void onAdd() {
+12      try {
+13          TextInputDialog dialog = new TextInputDialog();
+14          dialog.setTitle("Add Currency");
+15          dialog.setContentText("Currency code:");
+16          dialog.setHeaderText(null);
+17          dialog.setGraphic(null);
+18          Optional<String> code = dialog.showAndWait();
+19          if (code.isPresent()){
+20              ArrayList<Currency> currency_list = Launcher.getCurrency();
+21              Currency c = new Currency(code.get());
+22              ArrayList<CurrencyEntity> c_list = FetchData.fetch_range(c.getShortCode());
+23              c.setHistorical(c_list);
+24              c.setCurrent(c_list.get(c_list.size() -1 ));
+25              currency_list.add(c);
+26              Launcher.setCurrency(currency_list);
+27              Launcher.refreshPane();
+28          }
+29      } catch (InterruptedException e) {
+30          e.printStackTrace();
+31      } catch (ExecutionException e) {
+32          e.printStackTrace();
+33      }
+34  }
35 }
```

Script 2.26: /src/view/TopPane.java

```

1  //Imports are omitted
2  public class TopPane extends FlowPane {
    ...
12     refresh = new Button("Refresh");
- 13     change = new Button("Change");
+ 14     add = new Button("Add");
15     refresh.setOnAction(new EventHandler<ActionEvent>() {
16         @Override
17         public void handle(ActionEvent event) {
18             AllEventHandlers.onRefresh();
19         }
20     });
+ 21     add.setOnAction(new EventHandler<ActionEvent>() {
+ 22         @Override
+ 23         public void handle(ActionEvent event) {
+ 24             AllEventHandlers.onAdd();
+ 25         }
+ 26     });
27     update = new Label();
28     refreshPane()
- 29     this.getChildren().addAll(refresh, change, update);
+ 30     this.getChildren().addAll(refresh, add, update);
31 }
32 public void refreshPane() {
33     update.setText(String.format("Last update: %s", LocalDateTime.now().toString()));
34 }

```

A new event handling method for the delete functionality is created between Line 9 and Line 15 of Script 2.27. Its programming logics are implemented elsewhere on the 2.28 class between Line 35 and Line 55. The input of this method is the short currency code, and the method will look up in the `ArrayList` for the `Currency` object whose short code is equal to the search query. The technique uses an array index as a flag variable which is the same as what we applied in the earlier chapter's case study. As a continual to the delete functionality implementation, the very last button to be implemented is the *Watch* button. Furthermore, Script 2.29 and Script 2.30 modify the `CurrencyPane` and the `AllEventHandlers` classes, re-



spectively. The main objective of this modification is to enable the currency watch functionality. The watch rate should be presented under the current exchange rate if it is defined. In this example, the programming logic implementing the Watch button uses two variables named `isWatch` and `watchRate` to track the monitoring state.

**Script 2.27: /src/view/CurrencyPane.java**

```
1 //Imports are omitted
2
3 public class CurrencyPane extends BorderPane {
4     private Currency currency;
5     private Button watch;
+ 6     private Button delete;
7     public CurrencyPane(Currency currency) {
8         this.watch = new Button("Watch");
+ 9         this.delete = new Button("Delete");
+ 10        this.delete.setOnAction(new EventHandler<ActionEvent>() {
+ 11            @Override
+ 12            public void handle(ActionEvent event) {
+ 13                AllEventHandlers.onDelete(currency.getShortCode());
+ 14            }
+ 15        });
16        this.setPadding(new Insets(0));
17
18        ...
50    private HBox genTopArea() {
51        HBox topArea = new HBox(10);
52        topArea.setPadding(new Insets(5));
- 53        topArea.getChildren().addAll(watch);
+ 54        topArea.getChildren().addAll(watch, delete);
55        ((HBox) topArea).setAlignment(Pos.CENTER_RIGHT);
56        return topArea;
57    }
58 }
```

**Script 2.28: /src/controller/AllEventHandlers.java**

```
1 //Imports are omitted
2 public class AllEventHandlers {
3     ...
+ 35     public static void onDelete(String code) {
+ 36         try {
+ 37             ArrayList<Currency> currency_list = Launcher.getCurrency();
+ 38             int index = -1;
+ 39             for(int i=0 ; i<currency_list.size() ; i++) {
+ 40                 if (currency_list.get(i).getShortCode().equals(code) ) {
+ 41                     index = i;
+ 42                     break;
+ 43                 }
+ 44             }
+ 45             if (index != -1) {
+ 46                 currency_list.remove(index);
+ 47                 Launcher.setCurrency(currency_list);
+ 48                 Launcher.refreshPane();
+ 49             }
+ 50         } catch (InterruptedException e) {
+ 51             e.printStackTrace();
+ 52         } catch (ExecutionException e) {
+ 53             e.printStackTrace();
+ 54         }
+ 55     }
56 }
```

**Script 2.29: /src/view/CurrencyPane.java**

```
1 //Imports are omitted
2 public class CurrencyPane extends BorderPane {
3     private Currency currency;
+ 4     private Button watch;
+ 5     this.watch.setOnAction(new EventHandler<ActionEvent>() {
+ 6         @Override
+ 7         public void handle(ActionEvent event) {
+ 8             AllEventHandlers.onWatch(currency.getShortCode());
+ 9         }
+ 10    });
    ...
}
```

**Script 2.30: /src/controller/AllEventHandlers.java**

```
1  //Imports are omitted
2  public class AllEventHandlers {
    ...
+ 56  public static void onWatch(String code) {
+ 57      try {
+ 58          ArrayList<Currency> currency_list = Launcher.getCurrency();
+ 59          int index = -1;
+ 60          for(int i=0 ; i<currency_list.size() ; i++) {
+ 61              if (currency_list.get(i).getShortCode().equals(code) ) {
+ 62                  index = i;
+ 63                  break;
+ 64              }
+ 65          }
+ 66          if (index != -1) {
+ 67              TextInputDialog dialog = new TextInputDialog();
+ 68              dialog.setTitle("Add Watch");
+ 69              dialog.setContentText("Rate:");
+ 70              dialog.setHeaderText(null);
+ 71              dialog.setGraphic(null);
+ 72              Optional<String> retrievedRate = dialog.showAndWait();
+ 73              if (retrievedRate.isPresent()){
+ 74                  double rate = Double.parseDouble(retrievedRate.get());
+ 75                  currency_list.get(index).setWatch(true);
+ 76                  currency_list.get(index).setWatchRate(rate);
+ 77                  Launcher.setCurrency(currency_list);
+ 78                  Launcher.refreshPane();
+ 79              }
+ 80              Launcher.setCurrency(currency_list);
+ 81              Launcher.refreshPane();
+ 82          }
+ 83      } catch (InterruptedException e) {
+ 84          e.printStackTrace();
+ 85      } catch (ExecutionException e) {
+ 86          e.printStackTrace();
+ 87      }
+ 88  }
89 }
```

### 2.4.8 Background process

Aside from the *Watch* button, the watch functionality requires additional programming logic to refresh the application once after every specific interval of time. For example, if we want to check the watch rate every five seconds, we have to refresh the application every five seconds. In sum, what we have to implement are:

1. run a thread loop on the background;
2. let the thread sleep by default and wake up every five second to
  - refresh all the components of the application;
  - examine if the exchange rates have become lower than the rates set as watch rates and notify the user if they do.

Script 2.31 put the application into a sleep mode and wakes the application up every five seconds. After each wake-up, the class method will call the `refreshPane` method of the `Launcher` class to refresh the entire application. The method `Platform.runLater` is required here because the `refreshPane` method has to modify some Java FX view components during runtime. To apply this class in the application, we have to update the `Launcher` class following Script 2.32.

**Script 2.31: /src/controller/RefreshTask.java**

```
+ 1 //Imports are omitted
+ 2 public class RefreshTask extends Task<Void> {
+ 3     @Override
+ 4     protected Void call() throws Exception {
+ 5         for (;;) {
+ 6             try {
+ 7                 Thread.sleep((long) (60 * 1e3));
+ 8             } catch (InterruptedException e) {
+ 9                 System.out.println("Encountered an interrupted exception");
+10             }
+11             Platform.runLater(new Runnable() {
+12                 @Override
+13                 public void run() {
+14                     try {
+15                         Launcher.refreshPane();
+16                         ArrayList<Currency> allCurrency = Launcher.getCurrency();
+17                         String found = "";
+18                         for(int i=0 ; i<allCurrency.size() ; i++) {
+19                             if (allCurrency.get(i).getWatchRate() != 0 && allCurrency.get(i).
+20                                 getWatchRate() < allCurrency.get(i).getCurrent().getRate()) {
+21                                 if(found.equals(""))
+22                                     found = allCurrency.get(i).getShortCode();
+23                                 else
+24                                     found = found + " and " + allCurrency.get(i).getShortCode();
+25                             }
+26                             if (!found.equals("")) {
+27                                 Alert alert = new Alert(Alert.AlertType.WARNING);
+28                                 alert.setTitle(null);
+29                                 alert.setHeaderText(null);
+30                                 if(found.length()>3) {
+31                                     alert.setContentText(String.format("%s have become lower than
+32                                         the watch rate!", found));
+33                                 } else {
+34                                     alert.setContentText(String.format("%s has become lower than the
+35                                         watch rates!", found));
+36                                     alert.showAndWait();
+37                                 }
+38                             }
+39                         } catch (Exception e) {
+40                             e.printStackTrace();
+41                         }
+42                     }
+43                 }
+44             }
+45         }
+46     }
+47 }
```

**Script 2.32: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     private static Stage primaryStage;
4     private static Scene mainScene;
5     private static FlowPane mainPane;
6     private static TopPane topPane;
7     private static CurrencyPane currencyPane;
8     private static Currency currency;
9     @Override
10    public void start(Stage primaryStage) {
11        ...
20        this.primaryStage.show();
+ 21        RefreshTask r = new RefreshTask();
+ 22        Thread th = new Thread(r);
+ 23        th.setDaemon(true);
+ 24        th.start();
        ...
    }
```

There is still one problem regarding poor user experience such that whenever the watch notification is fired, the thread loop continues. This will cause duplicated alerts, leading to poor user experience. In turn, we have to make the loop pause and wait until the alert box is closed before resuming the background thread. What we have to do here is to separate the notification feature into a new class implementing the `JavaTask` class, in which its class method has to return something to its caller instead of pausing. The programming logic implementing the notification functionality is moved from the `RefreshTask` class to a new class named `WatchTask` as indicated in Script 2.33 and Script 2.34, respectively.

**Script 2.33: /src/controller/RefreshTask.java**

```

1  //Imports are omitted
2  public class RefreshTask extends Task<Void> {
    ...
14      public void run() {
15          try {
16              Launcher.refreshPane();
17              ArrayList<Currency> allCurrency = Launcher.getCurrency();
18              String found = "";
19              for(int i=0 ; i<allCurrency.size() ; i++) {
20                  if (allCurrency.get(i).getWatchRate() != 0 && allCurrency.get(i).
                      getWatchRate() < allCurrency.get(i).getCurrent().getRate()) {
21                      if(found.equals(""))
22                          found = allCurrency.get(i).getShortCode();
23                      else
24                          found = found + " and " + allCurrency.get(i).getShortCode();
25                  }
26              }
27              if (!found.equals("")) {
28                  Alert alert = new Alert(Alert.AlertType.WARNING);
29                  alert.setTitle(null);
30                  alert.setHeaderText(null);
31                  if(found.length() > 3) {
32                      alert.setContentText(String.format("%s have become lower than
                      the watch rate!", found));
33                  } else {
34                      alert.setContentText(String.format("%s has become lower than the
                      watch rates!", found));
35                      alert.showAndWait();
36                  }
37              }
38              } catch (Exception e) {
39                  e.printStackTrace();
40              }
41          }
42      });
+43      FutureTask futureTask = new FutureTask(new WatchTask());
+44      Platform.runLater(futureTask);
+45      try {
+46          futureTask.get();
+47      } catch (InterruptedException e) {
+48          System.out.println("Encountered an interrupted exception");
+49      } catch (ExecutionException e) {
+50          System.out.println("Encountered an execution exception");
+51      }
    ...

```

**Script 2.34: /src/controller/WatchTask.java**

```
+ 1 //Imports are omitted
+ 2 class WatchTask implements Callable<Void> {
+ 3     @Override
+ 4     public Void call() {
+ 5         ArrayList<Currency> allCurrency = Launcher.getCurrency();
+ 6         String found = "";
+ 7         for(int i=0 ; i<allCurrency.size() ; i++) {
+ 8             if (allCurrency.get(i).getWatchRate()!= 0 && allCurrency.get(i).
+ 9                 getWatchRate() < allCurrency.get(i).getCurrent().getRate()) {
+10                 if(found.equals("")) {
+11                     found = allCurrency.get(i).getShortCode();
+12                 } else {
+13                     found = found + " and " + allCurrency.get(i).getShortCode();
+14                 }
+15             }
+16             if (!found.equals("")) {
+17                 Alert alert = new Alert(Alert.AlertType.WARNING);
+18                 alert.setTitle(null);
+19                 alert.setHeaderText(null);
+20                 if(found.length()>3) {
+21                     alert.setContentText(String.format("%s have become lower than the watch
+22                         rate!", found));
+23                 } else {
+24                     alert.setContentText(String.format("%s has become lower than the watch
+25                         rates!", found));
+26                 }
+27                 alert.showAndWait();
+28             }
+29             return null;
+30         }
+31     }
+32 }
```

After Script 2.34 is completed, the application can almost fulfill the initial requirements. Several of which are remaining, and they all are intendedly left to be the exercise problems of this chapter. Precisely, the exercise tasks of the present chapter are as given below:



### 2.4.9 Exercise

1. Show the historical exchange rate up to 14 days as requested by the client.  
**Hint:** We have to call the API twice and concatenate the results.
2. Convert the two sub-panes in the `CurrencyPane` class into `Callable` objects. This is the same as how the class `DrawGraphTask` is implemented. Then, execute the two modified classes by using an `ExecutorService`.
3. Add an Unwatch button. This button should be located next to the watch button, i.e., to the right of the application window. After this Unwatch button is clicked, the watch rate should be disappeared and all the programming logic behind it should also be unset.
4. Let the user use either small or capital letters for the currency short code.
5. The application should be able to notify the user if the input currency short code is invalid and let the user re-enter the new one.

**Hint:** a try-catch-finally block should be useful. All the valid currency short code is available at: <https://free.currconv.com/api/v7/currencies?apiKey=XXXXXXXX>, where XXXXXXXX is the API key received from the provider by email at the beginning of the case study.

## Multithreaded programming

**M**ODERN commodity computers are equipped with multicore CPUs that enables them to execute multiple tasks concurrently. For example, simultaneously on the same computer, we can develop an application using an IDE, look up for some references on the Stack-Overflow website, and listen to music using a media player. We can also let these multicore CPUs collaboratively execute one complex task, such as graphic renderings and simulations. In terms of programming, enabling a program to harness the power of multicore CPUs is not fully automatic. We have to carefully design a concurrent program by ourselves to handle these powerful computing resources wisely to achieve the computational gain as desired.

### 3.1 Multithread and multiprocess

Up to now, we have practiced a lot in programming throughout several courses in the SE program. All the codes we have implemented appear to be in the form of an *input*  $\rightarrow$  *sequence of computations*  $\rightarrow$  *output* pattern. In this way, each application would use only one CPU core, and it is a kind of wasted resource utilization. The

programming paradigm that enables an application to utilize more than one computing core is known as concurrent programming. There are two basic units of execution: process and thread. In operation system terminology, a running program can be called a process, and of course, there can be more than one process running concurrently. A process owns a self-contained execution environment, and it is usually located at the level of operating systems. In terms of hardware utilization, each process has its registers, program counter, stack pointer, and memory space. Thus, multiple processes are executed as if they do not know each other.

Threads exist within a process. In brief, a thread is a paradigm that allows a single process to have multiple code segments running concurrently within the same context. Sometimes, threads are called lightweight processes or execution contexts. Concurrency can be achieved through either multiprocessing or multithreading. The main differences between the two are that the processes within a single program are allocated to the separated code and data space, and they will be executed isolatedly. On the other hand, threads within a single multithreaded program are allocated to shared code and data space. This makes multithreading take significantly less time and resource to start, due to that the operating system has less thing to do for its initiation.

The multiprocessing technique is widely used in computational-expensive tasks which demand the accelerated computing speed. In this case, the computer can allocate one or more dedicated CPU core and address space to carry out the task without preparing the resource for rapid synchronization. On the other hand, utilizing the multithreading technique, since all the resources are shared among threads, developers have to actively ensure that the resource changes during the computation shall not cause any readers–writers problems. This may sound like a real drawback for multithreading compared with multiprocessing and it will raise the question such that why do we need multithreads? In short, in practice, utilizing multithreads does not usually mainly aim at speed up. For example, user interfacing may best express the necessity for multithreads. With shared resources, multithreading allows different user interface elements to emit signals to control or interrupt other execution contexts. For example, in a video rendering application, once the execute button is clicked, the rendering task will be dispatched to a compute thread and executed on the background. If a cancel button is provided, the button press will attempt to interrupt the compute thread and stop the computing

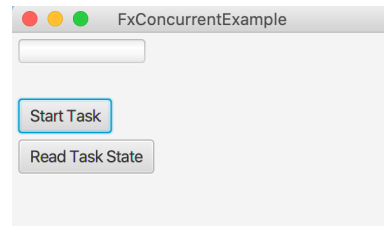
task. Suppose that this cancel button is not implemented as threads, even if there is a dedicated event handler for the button, the method will not be processed until the rendering is completed. Of course, this cancel button's presence will be utterly meaningless in such a case as it cannot immediately cancel the task. In contrast, if the event handler for this cancel button handles it in a separate thread, this thread can emit an interrupt signal to terminate the task on the other thread, which is what a cancel button should do. It should be noted that this course only put the main focus on multithreaded programming, and all its essence will be elaborated throughout this chapter.

## 3.2 Purpose

### 3.2.1 Multithreaded and GUI programming

Without multithreading, it would be impossible to build a GUI application since all the control and visual elements will be unresponsive. For example, suppose that the application implemented in Figure 3.1 has two buttons, and the button labeled as *Start Task* is bound with an event handler that computes a mathematical problem. If there is only one thread, after the *Start Task* button is clicked, the program will not lis-

ten or generate any other button's responses input until the computation is done. In brief, a GUI application should consists of at least two threads: (1) an event dispatch thread, which handles all GUI events, e.g., mouse events, keyboard events, and timer events, and (2) a thread for event listeners, which are method objects that will be invoked to respond to the GUI events. All the GUI activities need to be on the event dispatch thread. All these methods should not be time-consuming because they should be actively waiting for incoming signals to generate the response as promptly as possible. A failure to do it is very likely to make the user complain that the application is not sufficiently responsive enough.



**Figure 3.1: An example application with two buttons**

### 3.2.2 Multithreaded and accelerated computing

A common question one may be curious about the accelerated performance gain from implementing a multithreaded program is how significant the gain is. A short answer to the question is it depends (mainly on algorithm and resource). Based on the lecturer's personal experience, where his work is always computational resource demanding. In the experiment rig of his research in data science, e.g., in Phannachitta and Matsumoto (2019), thousands of experiments were carried out to compare the performance of numerous machine learning techniques over an empirical software engineering research problem named Software effort estimation. Utilizing a commodity computer with two CPU cores has taken the entire experiment up to nearly a month to complete. After he decided to invest more in the computing resource and replace the computational environment into the early 2010s render farm with a dual 8-core Intel XEON CPUs, the time required for completing the same experiments was decreased significantly into less than a week. More recently, with an availability of a cloud platform where many expensive computing resources are available on a rental basis, the lecturer extended the research study and experimented on a scale-larger experiment in the more recent study, i.e., Phannachitta (2020), where a larger number of experiments were conducted. This time, the Amazon Web Services (AWS) Cloud Credits for Research Program allowed him to access high-performance computing resources and carried out the experiment on a computing environment housed in with more modern and more expensive Intel XEON CPUs. Even if the lecturer selected the CPUs with the same number of computing cores, the experiments took even less time to complete. In Phannachitta (2020), the experiment took only a few days, although a more complicated experimental setup was undertaken. The lecturer also has a chance to examine the most expensive AWS computing instance at the time of writing, and its performance is awe-inspiring. In a big data project which was impossible to compute on a commodity computer due to that the computation is projected to complete in

---

Phannachitta P, Matsumoto K (2019) Model-based software effort estimation - a robust comparison of 14 algorithms widely used in the data science community. *Int. J. Innov. Comput. I.* 15(2), 569–589.

Phannachitta P (2020) On an Optimal Analogy-based Software Effort Estimation. *Inf. Softw. Technol.* 125, (to appear).

several months, the most potent ec2 instance named c5.24xlarge, it took only approximately a day to complete the task. Anyhow, the cost to operate such a machine seems to be overly expensive for any tasks in which it is remaining unclear whether it can generate as much higher profit as that of the rental cost of the computing instances. Exceptionally, at the time of writing, renting a c5.24xlarge instance on an on-demand basis demands be more than one million baht per year.

Even if the multiprocessing technique appears to be more suitable for a computational power-demanding task, we can also use the multithreading technique for the task. As preliminarily explained at the beginning of this chapter, it is difficult for the multithreading technique to achieve an ideal speed up, i.e., to obtain twice the performance gain from increasing the number of threads twice, is due to its inherent communication overhead. If we can ensure that our divided tasks do not require any communications during their execution, the ideal speed up may be achieved. This is similar to how we do a group project. The less we have meetings implies the more time we can spend on working. However, less meeting also has a drawback such that if a member makes some mistakes, no one will realize it until the very end of the project when all the pieces of work are assembled. Of course, everyone will realize at that time that the project cannot be completed and delivered in time. Following this analogy, we have to carefully decide about meeting time allocation which should not be too much and not too little. In terms of programming, this meeting between threads is called synchronization.

### Synchronization

The essential principle concerning synchronization is that any concurrent threads must be executed correctly without any interleaving of their instructions. To achieve that, the synchronization technique will let multiple threads safely coordinate the shared resources' access. For example, imagine that the two most essential ATM functionalities, i.e., `withdraw` and `deposit`, are implemented as depicted in Script 3.1 on different threads. Without any attempts to apply the synchronization technique, any two concurrent threads may withdraw the exact amount of money simultaneously, but it leaves the remaining amount of as if only one transaction has occurred. Anyone who experiences this situation may feel very lucky, but this can happen oppositely as well, such that any two deposit transactions can also count

only once.

#### Script 3.1: Withdraw and deposit example

```
1  int withdraw (int account, int amount) {  
2      double balance = readBalance(account);  
3      double balance = balance - amount;  
4      updateBalance(account, balance);  
5      return balance;  
6  }  
7  int deposit (int account, int amount) {  
8      double balance = readBalance(account);  
9      double balance = balance + amount;  
10     updateBalance(account, balance);  
11     return balance;  
12 }
```

The synchronized code fragments ensure that all the data objects must only be observed when it is in a consistent state. When one thread starts executing code in a synchronized method, it will see all the modifications on the same data object made by other threads. By the way, deciding how often should a task be synchronized is somewhat tricky. Too much synchronizations may cause the program to run slowly due to that all the threads will waste too much time waiting for the result from other threads, as worst still introduce a deadlock. On the other hand, too few synchronization may cause the program to deliver incorrect results due to one thread's changes may fail to propagate to other threads. In sum, it can be said that synchronized is the opposite of concurrent. Examples of the implementation of the synchronization technique will be provided later in this chapter.

### 3.3 The JavaFX concurrent framework

The framework is a kind of toolset providing the necessary interface, classes, and enum to support concurrent application development. Mainly, the framework provides four classes named Task, Service, ScheduledService, and WorkerState Event. The first three classes implement an interface named Worker in which an instance of

its type runs in a background thread. Also, an enum named `Worker.State` represents different states of a `Worker`, and it is observable from the application thread.

The differences between `Task`, `Service`, and `ScheduledService` are that an instance of a `Task` class cannot be reused, but that of the other two classes represent a reusable task. Also, the `ScheduledService` class is inherited from the `Service` class to represent a reusable `Task` scheduled to run repeatedly after a specified interval. Finally, an instance of a `WorkerStateEvent` class represents an event that occurs as a `Worker`'s state has been changed. We can add any event handlers that are triggered by the changes in the `Worker` states.

### 3.3.1 Components

#### The worker interface

A `Worker<V>` interface provides the specification for any tasks, where the generic parameter `V` is the data type of the result of the `Worker`. The state of a `Task` is observable and is published on the Java FX application thread. This makes it possible for a `Task` to communicate with the Java FX Scene graph that makes it fully flexible to control any visual elements from the background threads.

A `Worker` transitions through different states during its life cycle, ranging from its creation up to its end. The state of a worker is represented as a `Worker.State` enum. The particular predefined states are as follows: `Worker.State.READY`, `Worker.State.SCHEDULED`, `Worker.State.RUNNING`, `Worker.State.SUCCEEDED`, `Worker.State.FAILED`, and `Worker.State.CANCELLED`.

When a `Worker` is created, it is in the `Worker.State.READY` state. Right before it starts executing, it transitions to `Worker.State.SCHEDULED`. When it starts running, it will be turned into the `Worker.State.RUNNING` state. If the result of the completion determines a successful completion, its state will become `Worker.State.SUCCEEDED`. On the other hand, it will be in the `Worker.State.FAILED` state if an exception is thrown during its execution. The `Worker.State.CANCELLED` state represents the state that the `Worker` is intentionally terminated.



### The Task class

An instance of the `Task<V>` class is the place to implement our programming logic in which we want it to be executed in a background thread. Precisely, an abstract method named `call` is a particular method that will be executed in a background thread. The only thing we need to do is to override the `call` method can execute it.

A `Task` instance can also be monitored throughout the execution like a `Worker` instance, but it is properties rather than states. The main reason is that, compared with a `Worker` instance, a `Task` instance has more information involved as it progresses. For example, in practice, the completion percentage is a commonly-used message during the execution and the progress.

### The Service and ScheduledService class

The `Service<V>` class encapsulates the `Task<V>` class and makes it reusable, i.e., we can restart a `Service<V>` instance, which is otherwise impossible for the `Task<V>` class. The `ScheduledService<V>` class is a `Service<V>` instance that automatically restarts on schedule. We can configure it to restart whether after it finishes successfully or after it fails.

#### 3.3.2 Example

Scripts 3.2 illustrates an example of a multithreaded JavaFX program. Line 7 of the script shows the `call` method where all the code fragments between Line 8 and Line 14 will be executed in the background thread. This program's functionality is to increment a number from 1 to 50, once every second. The interval is controlled by the code `Thread.sleep(1000)` at Line 12. This line of code demands an Exception handler as described in the previous chapter. This example just uses `throws Exception` due to the problem of space limitation of the document page. In practice, the lecture suggests everyone to instead implement a `try-catch-finally` block for any Exception handling. Line 10 updates the progress bar as to provide the user the current status of the program execution, such that the progress bar will display that the task is being half-complete after the *start button* was clicked for 25 seconds. Line 11 shows the thread message on the GUI, where this example will let the message be the incrementing number. Outside the `for-loop`, the declaration of

**Script 3.2: FxConcurrentExample.java**

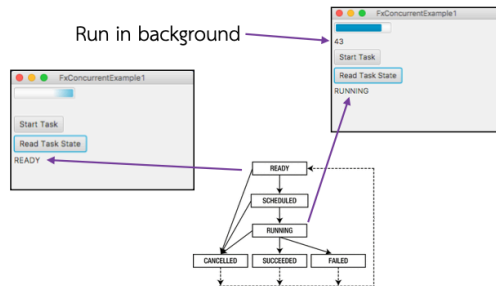
```
+ 1 //Imports are omitted
+ 2 public class FxConcurrentExample extends Application {
+ 3     @Override
+ 4     public void start(Stage primaryStage) {
+ 5         final Task task = new Task<Void>() {
+ 6             @Override
+ 7             protected Void call() throws Exception {
+ 8                 int max = 50;
+ 9                 for (int i = 1; i <= max; i++) {
+10                     updateProgress(i, max);
+11                     updateMessage(String.valueOf(i));
+12                     Thread.sleep(1000);
+13                 }
+14                 return null;
+15             }
+16         };
+17         ProgressBar pBar = new ProgressBar();
+18         pBar.setProgress(0);
+19         Label lblCount = new Label();
+20         final Label lblState = new Label();
+21         Button btnStart = new Button("Start Task");
+22         Button btnReadTaskState = new Button("Read Task State");
+23         VBox vBox = new VBox();
+24         vBox.setPadding(new Insets(5, 5, 5, 5));
+25         vBox.setSpacing(5);
+26         vBox.getChildren().addAll(pBar, lblCount, btnStart, btnReadTaskState,
+27                                 lblState);
+28         StackPane root = new StackPane();
+29         root.getChildren().add(vBox);
+30         Scene scene = new Scene(root, 300, 250);
+31         primaryStage.setTitle("FxConcurrentExample");
+32         primaryStage.setScene(scene);
+33         primaryStage.show();
+34     }
+35     public static void main(String[] args) { launch(args); }
```

this program's control elements are illustrated between Line 17 and Line 22, which consist of one progress bar, two labels, and two buttons. From Line 23, all the codes are to layout the GUI. Figure 3.1 demonstrates the compiled GUI of this example application.

**Script 3.3: FxConcurrentExample.java**

```
1 //Imports are omitted
2 public class FxConcurrentExample extends Application {
    ...
+20     pBar.setProgress(0);
+21     pBar.progressProperty().bind(task.progressProperty());
+22     Label lblCount = new Label();
+23     lblCount.textProperty().bind(task.messageProperty());
+24     final Label lblState = new Label();
+25     Button btnStart = new Button("Start Task");
+26     btnStart.setOnAction(new EventHandler<ActionEvent>() {
+27         @Override
+28         public void handle(ActionEvent t) {
+29             new Thread(task).start();
+30         }
+31     });
+32     Button btnReadTaskState = new Button("Read Task State");
+33     btnReadTaskState.setOnAction(new EventHandler<ActionEvent>() {
+34         @Override
+35         public void handle(ActionEvent t) {
+36             lblState.setText(task.getState().toString());
+37         }
+38     });
    ...
}
```

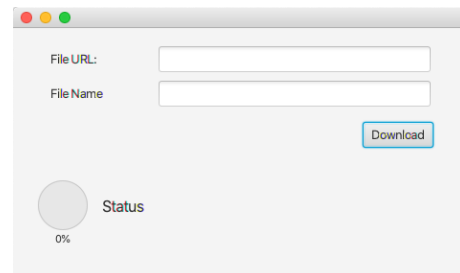
Script 3.3 adds four event handlers to the program. They all are as illustrated at Line 21, Line 23, Line 26, and Line 33 of the script, respectively. The first two event handlers synchronize the the status of the task to the progress bar and the message automatically. The other two event handlers are bound with the two buttons such that the thread, i.e., the code inside the `call` method, will be executed after the `btnStart` button is clicked. When the `btnReadTaskState` button is clicked, the status of the working thread will be retrieved and shown at the `lblState` element . Figure 3.2 shows a graphical explanation of this entire application.



**Figure 3.2: A graphical explanation of the Fx-ConcurrentExample**

### 3.4 A more real-world example project

Up to the current step, we have discussed two distinguish usage approaches of the multithreading technique, which are for GUI programming and for accelerated computing. The following example presents the two approaches in one single application. The materials used in this example project are available at [https://github.com/mhrimaz/JavaFXWorkshop\\_06](https://github.com/mhrimaz/JavaFXWorkshop_06) in which its GUI is rendered as shown in



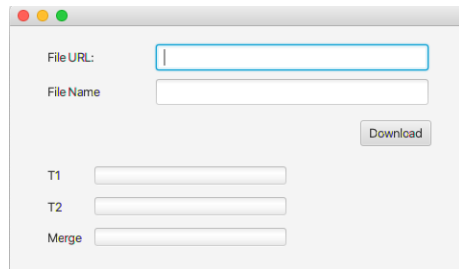
**Figure 3.3: An example download manager application**

Figure 3.3. It is the lecturer idea that this example project is a great example to demonstrate how we can add more functionalities to an existing project to improve it. The original version is a simple download manager, in which its vital design concept is to dispatch the download task to a background thread after the *Down-*

*load* button is clicked. Then, the main process observes the download progress and informs the user of the download status on the application GUI in real-time.

The original version of the application consists of three Java files, namely `Downloader.java`, `FXMLDocumentController.java`, and `JavaFXWorkshop_06.java`. The `Downloader` class opens the url connection for the input, reads the file content through an instance of the `ByteArrayOutputStream` object in to a buffer array, and writes the data from the buffer back to the local storage. The `FXMLDocumentController` class creates multiple control elements and binds them with their associated event handlers. Finally, the `JavaFXWorkshop_06` class is the application launcher.

Clearly, this example application shows a constructive example utilizing threads for GUI programming. The modification demonstrated in this section will make the application more efficient by enabling the multithreaded download functionality. It should be noted that a multithreaded downloader can be entirely embarrassingly parallelizable. This is because once a download thread know which part of the data it should handle, it can simply carry out its download task until the download is completed without any communication. Let this example employ two downloaders, and the final modified version of the application is illustrated in Figure 3.4. In terms of logic programming, we have to divide the download file by half into two chunks, and simultaneously download them using two downloaders. To yield the least amount of synchronization, the application will perform in steps as follows:



**Figure 3.4:** An example GUI of a multi-threaded downloader

1. Get the file size;
2. Divide the size by the number of downloaders;
3. Create a fixed thread pool and execute all the downloaders;
4. Synchronize the downloader only once when all the downloaders complete their task;

### 5. Merge all the pieces of data

Let us begin the implementation with a reconstruction of the folder structure to be as given below:

```
src
├── controller
│   ├── Launcher.java
│   └── MainController.java
├── model.java
│   ├── Downloader.java
│   └── Merger.java
└── view
    └── MainView.fxml
```

The files `FXMLDocumentController.java`, `JavaFXWorkshop_06.java`, and `FXMLDocument.fxml`, are renamed from the original version to `MainController`, `Launcher`, and `MainView.fxml`, respectively. The modification will transform the `Downloader` class into multipart downloaders and create a new `Merger` class to merge all the file parts after all of the file parts are completely downloaded.

Scripts 3.4 shows the modified `Downloader` class. The variable declaration up to Line 13 adds three new variables: `startByte`, `endByte`, and `doneSignal`. The `startByte` and `endByte` variables indicate the first and the last bytes that a partial-file downloader will be assigned to carry out. For example, we will implement two downloaders in this example, the value of the `startByte` variable of the first downloader will be zero, and that of the second downloader will be half of the file size plus one. The `doneSignal` variable has the type of `CountDownLatch`. It is one of the simplest approaches used for implementing synchronization, in which we need it to trigger the `Merger` class after all the download tasks are finished. The functionality of a `CountDownLatch` variable is the same as a latch used in physical hardware. A controller class that controls both the download and merge tasks will set the latch value as two, which is determined by the number of downloaders used in this example. After each downloader completes its task, the `doneSignal` variable will decrement by one. When the latch counter has reached zero, it will trigger the `Merger` instance by informing it that all the downloaded file parts are ready for merging.

**Script 3.4: /src/model/Downloader.java**

```

1  //Imports are omitted
2  public class Downloader extends Task<Void> {
3      ...
+ 5      private long startByte, endByte;
+ 6      private CountDownLatch doneSignal;
- 7      public Downloader(URL url, String fileName) {
+ 8      public Downloader(URL url, String fileName, long startBit, long endBit,
          CountDownLatch doneSignal) {
9          ...
+11          this.startBit = startBit;
+12          this.endBit = endBit;
+13          this.doneSignal = doneSignal;
14      }
15      @Override
16      protected Void call() throws Exception {
17          try {
18              URLConnection openConnection = url.openConnection();
+19              openConnection.setRequestProperty("Range", "bytes="+ startByte+"-"+endByte);
+20              int fileSize = openConnection.getContentLength();
-21              ByteArrayOutputStream out;
-22              try (InputStream in = new BufferedInputStream(url.openStream())) {
-23                  out = new ByteArrayOutputStream();
+24                  OutputStream out = new FileOutputStream(this.fileName);
+25                  InputStream in = openConnection.getInputStream();
26                  byte[] buf = new byte[5120];
-27                  int n = 0;
28                  long downloaded = 0;
-29                  while (-1 != (n = in.read(buf))) {
+30                  while (downloaded < fileSize) {
+31                      int n = in.read(buf, 0, buff.length);
+32                      if (n != -1) {
33                          downloaded += n;
34                          out.write(buf, 0, n);
35                          updateProgress(downloaded, fileSize);
-36                          updateMessage("Downloaded : " + (downloaded/(1024F*1024F*8F))+ " MB");
37                      }
38                      out.close();
+39                      in.close();
40                  }
-41                  byte[] response = out.toByteArray();
-42                  try (FileOutputStream fos = new FileOutputStream(fileName))
-43                      fos.write(response);
44              } catch (Exception e) { }
+45              doneSignal.countDown();
46              return null;
47          }
48      }

```

Script 3.5 illustrates the code implementing the `Merger` class. Overall, the `Merger` class will keep waiting for its start signal, as implemented at Line 18 of the script. After the signal is arrived, it will read through the content for each downloaded file, in which the sequence number is determined by the number after the keyword `-part`, as shown at Line 22. Once a downloaded file is completely read and dump into the output file, it will be deleted. These steps will be repeated until all the downloaded parts are completely examined.

**Script 3.5: /src/model/Merger.java**

```
+ 1 //Imports are omitted
+ 2 public class Merger extends Task<Void> {
+ 3     private String fileName;
+ 4     private int totalParts;
+ 5     private CountDownLatch startSignal;
+ 6     public Merger(String fileName, int totalParts, CountDownLatch startSignal) {
+ 7         this.fileName = fileName;
+ 8         this.totalParts = totalParts;
+ 9         this.startSignal = startSignal;
+10     }
+11     @Override
+12     protected Void call() {
+13         byte[] buf = new byte[4096];
+14         OutputStream out = null;
+15         InputStream in = null;
+16         File f = null;
+17         try {
+18             startSignal.await();
+19             out = new FileOutputStream(this.fileName);
+20             for (int i = 0; i < this.totalParts; i++) {
+21                 int toWrite;
+22                 f = new File(this.fileName + "-part" + (i+1));
+23                 in = new FileInputStream(f);
+24                 while ((toWrite = in.read(buf)) != -1) {
+25                     out.write(buf, 0, toWrite);
+26                 }
+27                 f.delete();
+28                 updateProgress((i + 1), this.totalParts);
+29                 in.close();
+30             }
+31             out.close();
+32         } catch (Exception e) {
+33             e.printStackTrace();
+34         }
+35         return null;
+36     }
+37 }
```



Script 5.29 illustrates the application launcher of this application. It links between the application scene and the `fxml` file, loads all the visual elements, and loads all the code associated with `fxml` file to the application scene's root node. Script 3.7 shows the view component of the application. This file has the type of `fxml`, an XML formatted file commonly used to design a JavaFX application's GUI components. Java FX also provides a dedicated builder application named `SceneBuilder`, and it is already integrated into modern IDE tools, such as IntelliJ Idea (2020). The XML format is commonly used for representing and layouting GUI elements because it can merely make the layouting be in a hierarchy, such as that being used in HTML. The main controller of this application is illustrated in Script 3.8. Its first ten lines of code attempt to bind the all the variables with the visual components declared in the `fxml` file. Note that, the keyword used in binding these variables is indicated with an `fx:` prefix, such as those presented at Line 41, Line 42, and Line 44 of Script 3.7. At Line 15 of Script 3.8, the initial value of `CountDownLatch` is set as two. Then, at Line 17, a thread pool of the same size is created. This thread pool will let the two downloaders, which will be created later at Line 26, execute on the background threads as can be instructed by the code presence between Line 34 and Line 36. Finally, the thread pool will be shutdown after all the tasks are completed at Line 37. The final part of this application is the application launcher.

**Script 3.6: /src/controller/Launcher.java**

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) throws Exception {
+ 5         Parent root = FXMLLoader.load(getClass().getResource("../view/MainView.fxml"));
+ 6         Scene scene = new Scene(root);
+ 7         stage.setScene(scene);
+ 8         stage.show();
+ 9     }
+10     static void main(String[] args) { launch(args); }
+11 }
```

**Script 3.7: /src/view/MainView.fxml**

```

1 //Imports are omitted
- 2 <AnchorPane id="AnchorPane" prefHeight="261.0" prefWidth="474.0" xmlns:fx="http://
    javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8.0.65" fx:controller="
    FXMLDocumentController">
3 <AnchorPane id="AnchorPane" prefHeight="261.0" prefWidth="474.0" xmlns="http://
    javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1" fx:controller="
    controller.MainController">
    ...
19 </GridPane>
20 <Button layoutX="365.0" layoutY="98.0" mnemonicParsing="false" onAction="#
    handleDownloadAction" text="Download" />
- 21 <ProgressIndicator fx:id="progress" layoutX="25.0" layoutY="159.0" prefHeight=
    "72.0" prefWidth="53.0" progress="0.0" stylesheets="@style.css" />
- 22 <Label fx:id="status" layoutX="93.0" layoutY="176.0" prefHeight="19.0"
    prefWidth="345.0" text="Status" wrapText="true">
- 23 <font>
- 24 <Font name="Segoe UI Bold" size="16.0" />
- 25 </font>
- 26 </Label>
+ 27 <GridPane layoutX="39.0" layoutY="139.0" prefHeight="97.0" prefWidth="397.0">
+ 28 <columnConstraints>
+ 29 <ColumnConstraints hgrow="SOMETIMES" maxWidth="351.0" minWidth="10.0"
    prefWidth="49.0" />
+ 30 <ColumnConstraints hgrow="SOMETIMES" maxWidth="351.0" minWidth="10.0"
    prefWidth="206.0" />
+ 31 <ColumnConstraints hgrow="SOMETIMES" maxWidth="205.0" minWidth="10.0"
    prefWidth="142.0" />
+ 32 </columnConstraints>
+ 33 <rowConstraints>
+ 34 <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
+ 35 <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
+ 36 <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
+ 37 </rowConstraints>
+ 38 <children>
+ 39 <Label text="T1" GridPane.rowIndex="0" />
+ 40 <Label text="T2" GridPane.rowIndex="1" />
+ 41 <ProgressBar fx:id="thread1" prefWidth="200.0" progress="0.0" GridPane.
    columnIndex="1" />
+ 42 <ProgressBar fx:id="thread2" prefWidth="200.0" progress="0.0" GridPane.
    columnIndex="1" GridPane.rowIndex="1" />
+ 43 <Label text="Merge" GridPane.rowIndex="2" />
+ 44 <ProgressBar fx:id="merge_bar" prefWidth="200.0" progress="0.0" GridPane
    .columnIndex="1" GridPane.rowIndex="2" />
+ 45 </children>
+ 46 </GridPane>
+ 47 </children>
+ 48 </AnchorPane>

```

**Script 3.8: /src/controller/MainController.java**

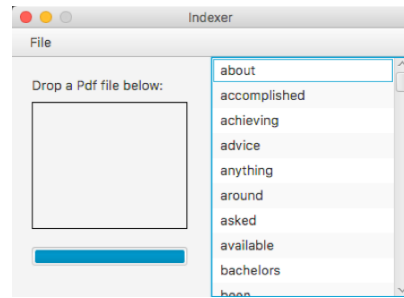
```
+ 1 //Imports are omitted
+ 2 public class MainController {
+ 3     @FXML
+ 4     private TextField urlField;
+ 5     @FXML
+ 6     private TextField fileField;
+ 7     @FXML
+ 8     private ProgressBar thread1, thread2, merge_bar;
+ 9
+ 10     long totalSizeOfFile;
+ 11     private static ExecutorService executor;
+ 12
+ 13     @FXML
+ 14     private void handleDownloadAction() {
+ 15         CountDownLatch countDownLatch = new CountDownLatch(2);
+ 16         try {
+ 17             executor = Executors.newFixedThreadPool(2);
+ 18
+ 19             URL url = new URL(urlField.getText());
+ 20             String filename = fileField.getText();
+ 21             URLConnection openConnection = url.openConnection();
+ 22
+ 23             totalSizeOfFile = openConnection.getContentLength();
+ 24             long baseLength = totalSizeOfFile/2;
+ 25
+ 26             Downloader downloader1 = new Downloader(url, filename+"-part1", 0,
+ 27                 baseLength, countDownLatch);
+ 28             Downloader downloader2 = new Downloader(url, filename+"-part2", (baseLength
+ 29                 +1), totalSizeOfFile, countDownLatch);
+ 30             Merger merger = new Merger(filename, 2, countDownLatch);
+ 31
+ 32             thread1.progressProperty().bind(downloader1.progressProperty());
+ 33             thread2.progressProperty().bind(downloader2.progressProperty());
+ 34             merge_bar.progressProperty().bind(merger.progressProperty());
+ 35
+ 36             executor.submit(downloader1);
+ 37             executor.submit(downloader2);
+ 38             executor.submit(merger);
+ 39             executor.shutdown();
+ 40         } catch (Exception e) {
+ 41             e.printStackTrace();
+ 42         }
+ 43     }
+ 44 }
```

### 3.5 Case study

This chapter implements a reverse index creation application. For example, if we drop a bunch of PDF files into the application's drop zone, it will extract all the words from all the PDF files, count each word frequencies, and create the reverse indexes of all words. These reversed indexes are sorted by the frequency count in descending order. These reversed indexes can be used for looking up for the documents containing a specific word. This reverse index is one of the essential

components of a search engine where any query we look up in the search engine can quickly bring us its associated webpages or documents. The sorted frequency count can imply how important the word is for a particular document.

In steps, this tutorial will begin with the interface. Then, we will implement the drop-file functionality and the indexing logic. Finally, we will accelerate the application with the multithreading technique. Figure 3.5 displays the rendered GUI of this application, and the file structure for this project is as follows:



**Figure 3.5: An example reverse index creation application**

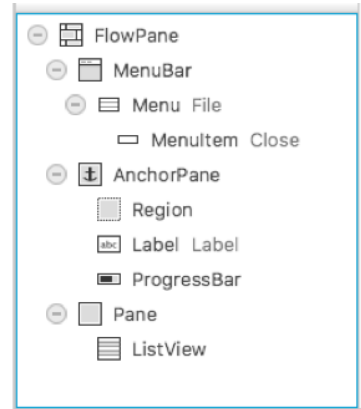
```

src
├── controller
│   ├── Launcher.java
│   ├── MainViewController.java
│   ├── WordMapMergeTask.java
│   └── WordMapPageTask.java
├── model
│   ├── FileFreq.java
│   └── PDFdocument.java
└── view
    └── MainView.fxml

```

### 3.5.1 Layouting the GUI using FXML

If we are using IntelliJ Idea as our IDE, we can design the GUI of the application by using a designing tool kit named SceneBuilder. For example, the GUI elements of this application is located in the file `mainView.fxml`. After we open it, we can click the tab named SceneBuilder at the bottom of the text editor section, and the IDE will bring us to the tool kit interface. Let us create the GUI layout for this application by dragging and dropping several Java FX components and constructing the GUI tree, as shown in Figure 3.6. We still have to configure the position and the properties for each component manually as follows:



**Figure 3.6: A GUI structure tree created by the Scene Builder toolkit**

- For the `FlowPane`, let all the configurable properties associating the size be `USE_COMPUTED_SIZE`;
- For the `MenuBar`, set its `Perf Width` be 400 and `Pref Height` be 20;
- For the `AnchorPane`, set its `Perf Width` be 200 and `Pref Height` be 240;
- For the `Label`, set its text as *"Drop PDF files below:"* then set its position for both `Layout X` and `Layout Y` as 20;
- For the `Region`, set its `Perf Width` be 150 and `Pref Height` be 120, then set its position for `Layout X` and `Layout Y` as 20 and 45, respectively. Finally set its `Style` as `-fx-border-color` with the property: `black`;
- For the `ProgressBar`, set its `Pref Width` as 150, then set its position for `Layout X` and `Layout Y` as 20, and 190, respectively;
- For the `Pane`, let all the properties associating with size be `USE_COMPUTED_SIZE`;
- Finally, for the `ListView`, set its `Perf Width` and `Height` be 200 and 240, respectively.

If we switch from the SceneBuilder back to Text editor mode, we will see that the code implementing all the Java FX GUI elements are represented in XML format. What next we have to do is to associate the id(s) of all the added control elements with Java FX nodes. This will later allow us to refer to them and control their behavior from other Java classes. Particularly, Script 3.9 illustrates four Java FX nodes where we associated their ids with the prefix fx: at Line 3, Line 18, Line 21, and Line 27, respectively. Later, we will refer to each of them as variable names: `controller.MainViewController`, `dropRegion`, `progressBar`, and `listView`, respectively, in other Java classes.

**Script 3.9: /src/view/mainView.java**

```

1 //Imports are omitted
- 2 <FlowPane xmlns="http://javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1">
+ 3 <FlowPane xmlns="http://javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="controller.MainViewController">
4     <children>
        ...
14     <AnchorPane prefHeight="240.0" prefWidth="200.0">
15         <children>
16             <Label layoutX="20.0" layoutY="20.0" text="Drop Pdf files below:" />
- 17             <Region layoutX="20.0" layoutY="45.0" prefHeight="120.0" prefWidth="150.0"
                style="-fx-border-color: black;">
+ 18             <Region fx:id="dropRegion" layoutX="20.0" layoutY="45.0" prefHeight="120.0"
                prefWidth="150.0" style="-fx-border-color: black;">
19             </Region>
- 20             <ProgressBar layoutX="20.0" layoutY="190.0" prefWidth="150.0" progress="0.0"
                />
+ 21             <ProgressBar fx:id="progressBar" layoutX="20.0" layoutY="190.0" prefWidth="
                150.0" progress="0.0" />
22         </children>
23     </AnchorPane>
24     <Pane prefHeight="200.0" prefWidth="200.0">
25         <children>
- 26             <ListView prefHeight="240.0" prefWidth="200.0" />
+ 27             <ListView fx:id="listView" prefHeight="240.0" prefWidth="200.0" />
28         </children>
29     </Pane>
30 </children>
31 </FlowPane>

```

### 3.5.2 The drop-file functionality

In the first chapter, we have implemented a drag and drop functionality, in which its small modification can turn it into a drop-file functionality. Script 3.10 shows the application launcher in which its structure is the same as in the previous example. Script 3.11 shows a controller for view elements that let the drop region on the left-hand side of the application window accept PDF files for their further processes. At Line 11 of the script, it checks whether the first file of all the files dropped into the dropRegion is in PDF format or not. If it is, the dragBoard will accept these files. Then in the setOnDragDropped method from Line 25 to Line 29, the for-loop will examine, one at the time, each PDF file and print out the file name on the terminal. Let us test whether the file dropped to the dropRegion will be processed further or not by implementing the application launcher and examining it.

Script 3.10: /src/controller/Launcher.java

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     @Override
+ 4     public void start(Stage primaryStage) throws Exception{
+ 5         Pane mainPane = FXMLLoader.load(getClass().getResource("../view/mainView.
+             fxml"));
+ 6         primaryStage.setTitle("Indexer");
+ 7         primaryStage.setScene(new Scene(mainPane));
+ 8         primaryStage.setResizable(false);
+ 9         primaryStage.show();
+10     }
+11     public static void main(String[] args) { launch(args); }
+12 }
```

**Script 3.11: /src/controller/MainViewController.java**

```
+ 1 //Imports are omitted
+ 2 public class MainViewController {
+ 3     @FXML
+ 4     private Region dropRegion;
+ 5     @FXML
+ 6     private ProgressBar progressBar;
+ 7     @FXML
+ 8     public void initialize() {
+ 9         dropRegion.setOnDragOver(event -> {
+10             Dragboard db = event.getDragboard();
+11             final boolean isAccepted = db.getFiles().get(0).getName().toLowerCase().
+                endsWith(".pdf");
+12             if (db.hasFiles() && isAccepted) {
+13                 event.acceptTransferModes(TransferMode.COPY);
+14             } else {
+15                 event.consume();
+16             }
+17         });
+18         dropRegion.setOnDragDropped(event -> {
+19             Dragboard db = event.getDragboard();
+20             boolean success = false;
+21             if (db.hasFiles()) {
+22                 success = true;
+23                 String filePath;
+24                 int total_files = db.getFiles().size();
+25                 for (int i = 0; i < total_files; i++) {
+26                     File file = db.getFiles().get(i);
+27                     filePath = file.getAbsolutePath();
+28                     System.out.println(filePath);
+29                 }
+30                 progressBar.setProgress(100);
+31             }
+32             event.setDropCompleted(success);
+33             event.consume();
+34         });
+35     }
+36 }
```



### 3.5.3 PDF file extraction

To generate the reverse indexes, the application has to extract the words from the files first. The toolset of choice that the lecturer has opted for this example is one of the most straightforward Java libraries named Fontbox. The library can be integrated to the application by including the files: fontbox-2.0.20.jar and pdfbox-2.0.20.jar from <https://pdfbox.apache.org/download.cgi>, together with its main dependency: commons-logging-1.2.jar, obtained from [http://commons.apache.org/proper/commons-logging/download\\_logging.cgi](http://commons.apache.org/proper/commons-logging/download_logging.cgi) to the project's external dependency list.

Script 3.12 and Script 3.13 depict two model classes. We will use them as the data structures for representing the PDF files' entity and the words extracted from the PDF files, respectively. Note that the two classes will be ready after adding the Getter and Setter methods to the classes.

#### Script 3.12: /src/model/FileFreq.java

```
+ 1 //Imports are omitted
+ 2 public class FileFreq {
+ 3     private String name;
+ 4     private String path;
+ 5     private Integer freq;
+ 6     public FileFreq(String name, String path, Integer freq) {
+ 7         this.name = name;
+ 8         this.path = path;
+ 9         this.freq = freq;
+10     }
+11     @Override
+12     public String toString() {
+13         return String.format("{s:%d}", name, freq);
+14     }
+15 }
```

**Script 3.13: /src/model/PDFdocument.java**

```
+ 1 //Imports are omitted
+ 2 public class PDFdocument {
+ 3     private String name;
+ 4     private String filePath;
+ 5     private PDDocument document;
+ 6     private LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets;
+ 7     public PDFdocument(String filePath) throws IOException {
+ 8         this.name = Paths.get(filePath).getFileName().toString();
+ 9         this.filePath = filePath;
+10         File pdfFile = new File(filePath);
+11         this.document = PDDocument.load(pdfFile);
+12     }
+13 }
```

After completing the view and the model classes, the next step is the application's essential logic. In this case, the programming logic is mainly the word extraction procedure, which comprises two main steps: (1) to extract all the words from the PDF files and count their frequencies, and (2) to merge the frequency count across multiple PDF files. The `WordMapPageTask` class handles the extraction and the frequency count functionalities. The `WordMapMergeTask` class handles the frequency merging functionality. The two classes are presented in Script 3.14 and Script 3.15, respectively.

The code appearing in these two classes used the *Stream* and *Lambda* techniques (Java.util.stream 2020), which can massively shorten the implementation codes. The general explanation of the `WordMapPageTask` class from Line 8 is as follows:

1. Split the one single big input string to be stream of words, where the tokenizer is a space;
2. For each word, delete all the non-Latin alphabet characters including the leading and trailing whitespace, and convert all the characters to lower cases;
3. Then, associate an integer value of one to all word;

---

Java.util.stream (2020) Package java.util.stream. Retrieved May, 2020, from <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/stream/package-summary.html>

**Script 3.14: /src/controller/WordMapPageTask.java**

```

+ 1 //Imports are omitted
+ 2 public class WordMapPageTask{
+ 3     private Map<String, FileFreq> wordCount;
+ 4     public WordMapPageTask(PDFdocument doc) throws IOException {
+ 5         PDFTextStripper reader = new PDFTextStripper();
+ 6         Pattern pattern = Pattern.compile(" ");
+ 7         String s = reader.getText(doc.getDocument());
+ 8         this.wordCount = pattern.splitAsStream(s)
+ 9             .map(word -> word.replaceAll("[^a-zA-Z]", "").toLowerCase().trim())
+10             .filter(word -> word.length() > 3)
+11             .map(word -> new AbstractMap.SimpleEntry<>(word, 1))
+12             .collect(toMap(e -> e.getKey(), e -> e.getValue(), (v1, v2) -> v1 + v2))
+13             .entrySet()
+14             .stream()
+15             .filter(e -> e.getValue() > 1)
+16             .collect(Collectors.toMap(e -> e.getKey(), e -> new FileFreq(doc.getName
+17                                     (), doc.getFilePath(), e.getValue())));
+18     }

```

4. Collect the frequency count for each word;
5. Recreate the stream again from the output of the previous step;
6. Select only word appeared more than one;
7. Make the final output, which is a map: (*word*  $\rightarrow$  a three-tuple  $\langle$ *filename, file path, and the frequency count* $\rangle$ );

For the WordMapMergeTask class, the code from Line 6 to Line 16 attempts to make a stream that runs through each word appearing in any PDF documents and applies a technique named *reduce* to aggregate only the frequency count of each word by making the summation of the values. From Line 17 to Line 21, the code attempts to sort the result products from produced by the *reduce* technique to generate the final result of the application. Note that at the current version, the order of the resulting product is still in alphabetically. We will modify it to be the order of frequency later.

**Script 3.15: /src/controller/WordMapMergeTask.java**

```

+ 1 //Imports are omitted
+ 2 public class WordMapMergeTask {
+ 3     private LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets;
+ 4     public WordMapMergeTask(Map<String, FileFreq>[] wordMap) {
+ 5         List<Map<String, FileFreq>> wordMapList = new ArrayList<>(Arrays.asList(
+ 6             wordMap));
+ 7         this.uniqueSets = wordMapList.stream()
+ 8             .flatMap(m -> m.entrySet().stream())
+ 9             .collect(Collectors.groupingBy(
+10                 e->e.getKey(),
+11                 Collector.of(
+12                     () -> new ArrayList<FileFreq>(),
+13                     (list, item) -> list.add(item.getValue()),
+14                     (current_list, new_items) -> {
+15                         current_list.addAll(new_items);
+16                         return current_list; })
+17             ))
+18             .entrySet()
+19             .stream()
+20             .sorted(Map.Entry.comparingByKey())
+21             .collect(Collectors.toMap(e->e.getKey(), e->e.getValue(),
+22                 (v1,v2)->v1, () -> new LinkedHashMap<>()));
+23     }
+24 }

```

It is the time to assemble all the model, view, and controller classes together since all the building blocks for the reverse index creation application are already prepared. Script 3.16 modifies the `MainViewController` by replacing the file name printing part with word extracting tasks and adding the word merging task at the end of the file. At Line 25 and Line 26, respectively, an array of `wordMapTaskList` instances and a `Map` data structure are created. These two variables will be used inside the for-loop between Line 27 and Line 40, where each PDF file, one at a time, will be parsed, and all the words inside it will be extracted. The product of the `getWordCount` method will be store in the `wordMap` variable in the form of a key-value pair `<String, FileFreq>`. This map array is then passed to be merged by a `wordMapMergeTask` class's instance at Line 42 of the script. The merged product

will then be sent to the GUI of the application to display itself to the user at Line 43 and Line 44 of the script.

At the current state of the application, all the functionality should be more than sufficient to fully our initial requirement. Anyhow, there are still some rooms for improvement. At least we are still able to improve the usability of the application. Several ideas include:

- adding a *Start Indexing* button since it is more usual to let the user drop several files from different paths and let the user start executing the task when she or he is ready;
- making the resultant list on the right hand side of the GUI window more easier to read;
- allowing the user click the resultant word term and frequency to open the associated PDF file;
- informing the user that the word extraction tasks are being processed on the background by using a loading screen.

To add a *Start Indexing* button, we also need to change several visual elements and the programming logic associated with it. For example, the static drop zone which is a simple `Region` at the current version, seems to be less useful than a list view that also informs the user what which files will be extracted after the button is clicked. Next, the progress bar appears to be more optional as space shall be better reserved for the button. Hence, let us begin with modifying the GUI elements by using the `SceneBuilder` feature. We need to change the `Region` under the `AnchorPane` to a `ListView`, and replace the `ProgressBar` with a `Button`. For the properties, we will set the `Pref Width` and `Pref Height` properties of the `ListView` to be 180 and 150, respectively, Also for the `Layout X` and `Layout Y`, let the values be 10 and 45, respectively. For the `Button`, simply label it as *Start Indexing* and let the button sit in the middle of the space below the `ListView`. After we finish layouting the GUI elements, using the text editor mode, we have to associate the `id(s)` of the two new visual elements of the `mainView.fxml`. Let the `id(s)` be `inputListView` and `startButton` for the new `ListView` and `Button`, respectively. Script 3.17 presents the updated `MainViewController` class after a `ListView` and a `Button` replaced the

two elements. In this update, the `setOnDragOver` handler is just changed to be associated with the `inputListView` as presented at Line 13 of the script. Next, the `setOnDragOver` handler will not extract the keyword yet, but the task will instead wait for another signal fired from the new button. Script 3.18 illustrates the event handler associated with the added `Button`. The `setOnAction` handler only reuses the codes executed after the drop-file in to the region before this very update. Specifically, the codes illustrated between Line 56 and Line 70 of the script iteratively examines each of the PDF files listed on the `inputListView` element and extracts all the words in each file. After the extraction is finished, it merges the frequency count back into one single data container and shows the user on the right-hand side of the GUI window.

**Script 3.16: /src/controller/MainViewController.java**

```

1  //Imports are omitted
2  public class MainViewController {
3      ...
18      dropRegion.setOnDragDropped(event -> {
19          Dragboard db = event.getDragboard();
20          boolean success = false;
21          if (db.hasFiles()) {
22              success = true;
23              String filePath;
24              int total_files = db.GetFiles().size();
+25          WordMapPageTask[] wordMapTaskList = new WordMapPageTask[total_files];
+26          Map<String, FileFreq>[] wordMap = new Map[total_files];
27          for (int i = 0; i < total_files; i++) {
-28              File file = db.GetFiles().get(i);
-29              filePath = file.getAbsolutePath();
-30              System.out.println(filePath);
+31              try {
+32                  File file = db.GetFiles().get(i);
+33                  filePath = file.getAbsolutePath();
+34                  PDFdocument p = new PDFdocument(filePath);
+35                  wordMapTaskList[i] = new WordMapPageTask(p);
+36                  wordMap[i] = wordMapTaskList[i].getWordCount();
+37                  progressBar.setProgress(i / wordMapTaskList.length * 100);
+38              } catch (IOException e) {
+39                  e.printStackTrace();
+40              }
+41          }
+42          WordMapMergeTask merger = new WordMapMergeTask(wordMap);
+43          LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets = merger.getUniqueSets
              ();
+44          listView.getItems().addAll(uniqueSets.entrySet());
45          progressBar.setProgress(100);
46      }
47      event.setDropCompleted(success);
48      event.consume();
49  });
50  }
51  }

```

**Script 3.17: /src/controller/MainViewController.java**

```

1  //Imports are omitted
2  public class MainViewController {
+ 3      LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets;
4      @FXML
- 5      private Region dropRegion;
+ 6      private ListView<String> inputListView;
7      @FXML
- 8      private ProgressBar progressBar;
+ 9      private Button startButton;
10     @FXML
11     public void initialize() {
- 12         dropRegion.setOnDragOver(event -> {
+ 13         inputListView.setOnDragOver(event -> {
14             ...
21         });
- 22         dropRegion.setOnDragDropped(event -> {
+ 23         inputListView.setOnDragDropped(event -> {
24             Dragboard db = event.getDragboard();
25             boolean success = false;
26             if (db.hasFiles()) {
27                 success = true;
28                 String filePath;
29                 int total_files = db.GetFiles().size();
- 30                 WordMapPageTask[] wordMapTaskList = new WordMapPageTask[total_files];
- 31                 Map<String, FileFreq>[] wordMap = new Map[total_files];
32                 for (int i = 0; i < total_files; i++) {
- 33                     try {
34                         File file = db.GetFiles().get(i);
35                         filePath = file.getAbsolutePath();
+ 36                         inputListView.getItems().add(file.getName());
- 37                         PDFdocument p = new PDFdocument(filePath);
- 38                         wordMapTaskList[i] = new WordMapPageTask(p);
- 39                         wordMap[i] = wordMapTaskList[i].getWordCount();
- 40                         progressBar.setProgress(i / wordMapTaskList.length * 100);
- 41                     } catch (IOException e) {
- 42                         e.printStackTrace();
- 43                     }
44                 }
- 45                 WordMapMergeTask merger = new WordMapMergeTask(wordMap);
- 46                 LinkedHashMap<String,ArrayList<FileFreq>> uniqueSets = merger.getUniqueSets
47                     ();
48                 listView.getItems().addAll(uniqueSets.entrySet());
49                 progressBar.setProgress(100);
50             }
51             ...

```



**Script 3.18: /src/controller/MainViewController.java**

```

1  //Imports are omitted
2  public class MainViewController {
3      ...
+50      startButton.setOnAction(event -> {
+51          List<String> inputListViewItems = inputListView.getItems();
+52          int total_files = inputListViewItems.size();
+53          WordMapPageTask[] wordMapTaskList = new WordMapPageTask[total_files];
+54          Map<String, FileFreq>[] wordMap = new Map[total_files];
+55
+56          for (int i = 0; i < total_files; i++) {
+57              try {
+58                  String filePath = inputListViewItems.get(i);
+59                  PDFdocument p = new PDFdocument(filePath);
+60                  wordMapTaskList[i] = new WordMapPageTask(p);
+61                  wordMap[i] = wordMapTaskList[i].getWordCount();
+62                  progressBar.setProgress(i / wordMapTaskList.length * 100);
+63              } catch (IOException e) {
+64                  e.printStackTrace();
+65              }
+66              WordMapMergeTask merger = new WordMapMergeTask(wordMap);
+67              uniqueSets = merger.getUniqueSets();
+68              listView.getItems().addAll(uniqueSets.keySet());
+69          }
+70      });
+71  }
+72  }

```

### 3.5.4 Map and ListView

The following functionality we will implement is to let the user click on the resulting extracted word list. The application will then respond to the mouse clicking by opening the PDF file associating the particular word. We have to modify two classes: the Launcher and the MainViewController classes in which their modification is illustrated in Script 3.19 and Script 3.20, respectively. In Script 3.19, the primaryStage variable is changed into a class variable. This will provide us more

flexibility to control any GUI element later. For the new variable `hs` with the type of `HostServices` added to the Script at Line 4, the variable will allow us to connect to the local machine from the application scene as we want to open a PDF file from inside the application. This `hs` variable is used at Line 80 of Script 3.20.

**Script 3.19: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
+ 3     public static Stage primaryStage;
+ 4     public static HostServices hs;
5     @Override
6     public void start(Stage primaryStage) throws Exception{
+ 7         this.primaryStage = primaryStage;
+ 8         hs = getHostServices();
9         Pane mainPane = FXMLLoader.load(getClass().getResource("../view/mainView.fxml"));
- 10        primaryStage.setTitle("Indexer");
- 11        primaryStage.setScene(new Scene(mainPane));
- 12        primaryStage.setResizable(false);
- 13        primaryStage.show();
+ 14        this.primaryStage.setTitle("Indexer");
+ 15        this.primaryStage.setScene(new Scene(mainPane));
+ 16        this.primaryStage.setResizable(false);
+ 17        this.primaryStage.show();
18    }
19    public static void main(String[] args) { launch(args); }
20 }
```

For Script 3.20, mainly, a new event handler, `setOnMouseClicked`, is implemented. The technique used to implement this functionality is called `Map`. It is one of the most powerful data structures commonly used for accelerating the computation related to searching. Using `Map` will allow us to show a more clean search result lists simply. We can present only what we want to present and simply hide other information in the class variable implemented as the value of the key-value pairs stored as a `Map`. Notably, in this example, we will let the word terms be the `Map` key, and all the corresponding information such as file path and the frequency count be the `Map` value. The for-loop between Line 74 and Line 77 of Script 3.20,

the file path is associated with the file entity, so when the user clicks a particular word, this path will be read, and the file stored in the path will be shown as a popup window, as described as a `setOnMouseClicked` event at Line 79.

#### Script 3.20: `/src/controller/MainViewController.java`

```

1  //Imports are omitted
2  public class MainViewController {
    ...
+ 70      listView.setOnMouseClicked(event -> {
+ 71          ArrayList<FileFreq> listOfLinks = uniqueSets.get(listView.getSelectionModel().
              getSelectedItem());
+ 72          ListView<FileFreq> popupListView = new ListView<>();
+ 73          LinkedHashMap<FileFreq,String> lookupTable = new LinkedHashMap<>();
+ 74          for (int i=0 ; i<listOfLinks.size() ; i++) {
+ 75              lookupTable.put(listOfLinks.get(i),listOfLinks.get(i).getPath());
+ 76              popupListView.getItems().add(listOfLinks.get(i));
+ 77          }
+ 78          popupListView.setPrefHeight(popupListView.getItems().size() * 28);
+ 79          popupListView.setOnMouseClicked(innerEvent -> {
+ 80              Launcher.hs.showDocument("file:///"+lookupTable.get(popupListView.
                  getSelectedItem().getSelectedItem()));
+ 81              popupListView.getScene().getWindow().hide();
+ 82          });
+ 83          Popup popup = new Popup();
+ 84          popup.getContent().add(popupListView);
+ 85          popup.show(Launcher.primaryStage);
+ 86      });
87  }
88  }
```

### 3.5.5 Multithreaded indexer

Typically, an indexing task is carried out over a huge amount of documents to be looked up, such as books in the whole library or web pages on the internet. Therefore, the problem demands exceptional computational power, wherein this example will harness the power of multithreaded programming and create multithreaded indexers. To transform any program into a Java FX multithreaded program, the very first step is to make the class implement the `Callable` interface and create a `call` method as the class's execution point. This will allow the class to be submitted as a background thread. Script 3.21 and Script 3.22 modify the `WordMapPageTask` and `WordMapMergeTask` classes, respectively. The tasks submission of these two classes is called in the event handler of the `startButton`, locating in the `MainViewController` class.

Script 3.23 presents the most recent update of `MainViewController` class. There are two service instances in the script: `ExecutorService` and `ExecutorCompletionService` declared at Line 51 and Line 52 of the script, respectively. We have already used an `ExecutorService` instance once in the earlier example. On the other hand, this is the first time we will implement an instance of the `ExecutorCompletionService` class. It is a kind of `Service` task that examines whether the `Service` instances being executed have already reached their completion state or not. Once these services are put to terminate, the `ExecutorCompletionService` will be able to collect the return results from the submitted task for us to process the results further. In this example, Line 63 of the script shows that, for each PDF file to be extracted, the `ExecutorCompletionService` instance associated with a `WordMapPageTask` instance is submitted for its execution. Their results will be retrieved inside the following for-loop between Line 68 and Line 75. Inside this for loop, a variable with the type `Future` will collect the results sent back from the submitted tasks. Note that a `Future` instance is a special kind of variable that can passively wait for the return results generated in the future, i.e., the results returned from the tasks executed in the different threads. Similarly, a `WordMapMergeTask` is submitted by an `Executor` at Line 78, and a `Future` instance will also retrieve its result at the same line of code. Overall, the modification of these three classes shows an approach to multithreaded programming in Java FX.

**Script 3.21: /src/controller/WordMapPageTask.java**

```
1 //Imports are omitted
- 2 public class WordMapPageTask{
+ 3 public class WordMapPageTask implements Callable<Map<String,FileFreq>> {
- 4     private Map<String, FileFreq> wordCount;
+ 5     private PDFdocument doc;
- 6     public WordMapPageTask(PDFdocument doc) throws IOException {
+ 7         this.doc = doc;
+ 8     }
+ 9     @Override
+ 10    public Map<String, FileFreq> call() throws Exception {
+ 11        Map<String, FileFreq> wordCount;
+ 12        PDFTextStripper reader = new PDFTextStripper();
+ 13        Pattern pattern = Pattern.compile(" ");
+ 14        String s = reader.getText(doc.getDocument());
- 15        this.wordCount = pattern.splitAsStream(s)
+ 16        wordCount = pattern.splitAsStream(s)
+ 17            .map(word -> word.replaceAll("[^a-zA-Z]", "").toLowerCase().trim())
+ 18            .filter(word -> word.length() > 3)
+ 19            .map(word -> new AbstractMap.SimpleEntry<>(word, 1))
+ 20            .collect(toMap(e -> e.getKey(), e -> e.getValue(), (v1, v2) -> v1 + v2))
+ 21            .entrySet()
+ 22            .stream()
+ 23            .filter(e -> e.getValue() > 1)
+ 24            .collect(Collectors.toMap(e -> e.getKey(), e-> new FileFreq(doc.getName
+ 25                (),doc.getFilePath(),e.getValue()))));
+ 26        return wordCount;
+ 27    }
}
```

**Script 3.22: /src/controller/WordMapMergeTask.java**

```

1  //Imports are omitted
- 2  public class WordMapMergeTask {
+ 3  public class WordMapMergeTask implements Callable<LinkedHashMap<String, ArrayList<
      FileFreq>>> {
- 4      private LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets;
+ 5      private Map<String, FileFreq>[] wordMap;
      6      public WordMapMergeTask(Map<String, FileFreq>[] wordMap) {
+ 7          this.wordMap = wordMap;
+ 8      }
+ 9      @Override
+ 10     public LinkedHashMap<String, ArrayList<FileFreq>> call() throws Exception {
+ 11         LinkedHashMap<String, ArrayList<FileFreq>> uniqueSets;
+ 12         List<Map<String, FileFreq>> wordMapList = new ArrayList<>(Arrays.asList(wordMap))
            ;
- 13         this.uniqueSets = wordMapList.stream()
+ 14         uniqueSets = wordMapList.stream()
            .flatMap(m -> m.entrySet().stream())
            .collect(Collectors.groupingBy(
+ 17                 e->e.getKey(),
+ 18                 Collector.of(
+ 19                     () -> new ArrayList<FileFreq>(),
+ 20                     (list, item) -> list.add(item.getValue()),
+ 21                     (current_list, new_items) -> {
+ 22                         current_list.addAll(new_items);
+ 23                         return current_list; })
            ))
            .entrySet()
            .stream()
            .sorted(Map.Entry.comparingByKey())
            .collect(Collectors.toMap(e->e.getKey(), e->e.getValue(),
+ 29                 (v1,v2)->v1, () -> new LinkedHashMap<>()));
+ 30         return uniqueSets;
+ 31     }
+ 32 }

```

**Script 3.23: /src/controller/MainViewController.java**

```

1  //Imports are omitted
2  public class MainViewController {
3      ...
4
50     startButton.setOnAction(event -> {
+ 51         ExecutorService executor = Executors.newFixedThreadPool(4);
+ 52         final ExecutorCompletionService<Map<String, FileFreq>> completionService = new
            ExecutorCompletionService<>(executor);
53         List<String> inputListViewItems = inputListView.getItems();
54         int total_files = inputListViewItems.size();
- 55         WordMapPageTask[] wordMapTaskList = new WordMapPageTask[total_files];
56         Map<String, FileFreq>[] wordMap = new Map[total_files];
57         for (int i = 0; i < total_files; i++) {
58             try {
59                 String filePath = inputListViewItems.get(i);
60                 PDFdocument p = new PDFdocument(filePath);
- 61                 wordMapTaskList[i] = new WordMapPageTask(p);
- 62                 wordMap[i] = wordMapTaskList[i].getWordCount();
+ 63                 completionService.submit(new WordMapPageTask(p));
64             } catch (IOException e) {
65                 e.printStackTrace();
66             }
67         }
+ 68         for (int i = 0; i < total_files; i++) {
+ 69             try {
+ 70                 Future<Map<String, FileFreq>> future = completionService.take();
+ 71                 wordMap[i] = future.get();
+ 72             } catch (Exception e) {
+ 73                 e.printStackTrace();
+ 74             }
+ 75         }
+ 76         try {
77             WordMapMergeTask merger = new WordMapMergeTask(wordMap);
+ 78             Future<LinkedHashMap<String, ArrayList<FileFreq>>> future = executor.submit(
                merger);
- 79             uniqueSets = merger.getUniqueSets();
+ 80             uniqueSets = future.get();
81             listView.getItems().addAll(uniqueSets.keySet());
+ 82         } catch (Exception e) {
+ 83             e.printStackTrace();
+ 84         } finally {
+ 85             executor.shutdown();
+ 86         }
87     });
88     ...

```

### 3.5.6 Adding a loading widget

It is widely recommended to provide feedback to the user that the computing is being carried out for any time-consuming tasks, such as this reverse indexes computing. At the beginning of the case study, we implemented a progress bar but took it out to save some space on the GUI window. Anyhow, having such a widget has more benefit than a drawback. For this example, let us bring it back in the form of a loading widget, which does not consume any space on the GUI window, and it is one of the standard techniques used in web applications or games. In particular, we will place a loading spinner inside the *startButton* and let it execute when the term extraction and term merging tasks are executing. It is important to note that Java FX event handlers are natively executed in a synchronous manner. Thus, typically, we should not be unable to modify any GUI elements during the execution of any event handling methods. For example, any code fragments attempting to modify a GUI element will be executed at the end of the particular method. Hence, the loading widget does require a special kind of technique to enable an asynchronous computation. One of the most straightforward approaches is to create an asynchronous block that wraps all the time-consuming code fragments by a *Task* class and uses a thread signal to simulate an asynchronous behavior. Script 3.24 presents the latest modification of the *MainViewController* class by adding the loading widget to the application. In the script, the code fragments added between Line 51 and Line 58 implements a *call* method and move all the body of the event handler's current implementation into the *call* method. Besides, the latest updated *MainViewController* class adds a *ProgressIndicator* instance, which is the visual element of the loading widget, and associating it with the GUI element. The code fragment added between Line 97 and Line 99 of the script declares an event handler for the loading widget that communicates with the application scene and tells it whether to present the loading widget. Finally, the code fragments added between Line 100 and Line 102 create a thread for the loading widget and run in the background thread.

Up to the current point of implementation, the application can let the user drop PDF files to it, and it will list paths to all the files to inform the user. It will then let the user decide when to start the term extraction and frequency count task by pressing the *Start Indexing* button. After the button is clicked, all the applica-



**Script 3.24: /src/controller/MainViewController.java**

```

1  //Imports are omitted
2  public class MainViewController {
    ...
50      startButton.setOnAction(event -> {
+ 51          Parent bgRoot = Launcher.primaryStage.getScene().getRoot();
+ 52          Task<Void> processTask = new Task<Void>() {
+ 53              @Override
+ 54              public Void call() throws IOException {
+ 55                  ProgressIndicator pi = new ProgressIndicator();
+ 56                  VBox box = new VBox(pi);
+ 57                  box.setAlignment(Pos.CENTER);
+ 58                  Launcher.primaryStage.getScene().setRoot(box);
59                  ExecutorService executor = Executors.newFixedThreadPool(4);
60                  ...
84                  try {
85                      WordMapMergeTask merger = new WordMapMergeTask(wordMap);
86                      Future<LinkedHashMap<String, ArrayList<FileFreq>>> future = executor.
                          submit(merger);
87                      uniqueSets = merger.getUniqueSets();
88                      uniqueSets = future.get();
89                      listView.getItems().addAll(uniqueSets.keySet());
90                  } catch (Exception e) {
91                      e.printStackTrace();
92                  } finally {
93                      executor.shutdown();
94                  }
95              }
96          };
+ 97          processTask.setOnSucceeded( e -> {
+ 98              Launcher.primaryStage.getScene().setRoot(bgRoot);
+ 99          });
+ 100          Thread thread = new Thread(processTask);
+ 101          thread.setDaemon(true);
+ 102          thread.start();
103      });
    ...

```

tion's tasks will be executed on the background threads. During the computation, a loading widget will inform the user that the application is still running. Once the extraction tasks are completed, the results will be shown on the application's right-hand side. The user can also interact with the result list by clicking it, and a pop-up window will show the PDF file associated with the term to the user.

### 3.5.7 Exercise

1. The result entries shown in the `ListView` on the left-hand side of the application window are sorted alphabetically. Our task is to let them ordered in terms of their total frequency, where the word with the highest frequency count should come first.
2. For the word appearing on many PDF files, show the frequency counts separately.
3. Let the drop-file `ListView` present only file names, as the entire file parts seem to be too long and somewhat too difficult to read.
4. Add a menu bar on the top of the application window. At least, we should be able to close the program after clicking on the menu item *File* and then *Close*.
5. Once the user clicks a term presented on the right-handed `ListView`, she or he cannot cancel the selection. Our task here is to create another event handler that accepts a keystroke *esc* to close the pop-up window.

# CHAPTER 4

## Debugging

**F**AULT and error are inevitable during program development and production. A method may return a wrong value, or the application may crash due to some exceptions. There are numerous possible causes of errors, such as data input error, incorrect parameter naming, the incorrect type of variable declaration, incorrect operator utilization, incorrect operand utilization, incorrect computational implementation, incorrect algorithm selection, and many others. In brief, we call the conditions causing the software application to fail to perform its required function as a fault and the difference between actual output and expected output as an error. Some places in our code must have problems when a fault or an error occurs, and we have to find it. In general, the action of investigating, localizing, and fixing these problems is called debugging.

### 4.1 Debugging methodologies

There are numerous kinds of faults and errors that we have to deal with as long as we are coding. Of course, some of them that can be localized by the IDE is easy to handle by observing the stack trace. These errors are mostly syntax errors and

compile errors. On the other hand, those related to programming logic and algorithms are more tricky to investigate, localize, and fix. Since, in most cases, we can only examine the unexpected behaviors by scrutinizing the input and output pairs to see the suspected pairs. After we can identify what appears to be incorrect, we have breakdown the components and examined each module by ourselves. Hence, in practice, debugging has a wide range of toolsets and methodologies available. In this material, we will discuss three categories widely used in practice.

#### 4.1.1 Probing

Probing is not a kind of common term, but it is the lecturer's choice to denounce the technique out of the category of debugging. Mainly, probing is to observe the particular behavior of a variable or a method by printing some values on the screen. For example, if we want to know the exact value of a variable `x`, we add the code snippet `System.out.println(x)` to the program around the line of code we want to examine it. Then, we may add other code fragments and see how the change propagates to this `x`. Of course, this is the most common way we may see in practice as it is super easy to start and tend to be helpful if the developer is familiar with the context. This may be the reason some developers call the action of adding the `System.out.println()` code fragment to the program as a debugging methodology. However, if we know what exactly a proper debugging is, we may hesitate to call this action as debugging any more.

The possible drawbacks to observe the application behavior with `System.out.println()` code snippet include that it is not easy to customize the probing at all. That it, it is particularly difficult to swap the points of investigation from a variable to the other variable and then swap back. Furthermore, it is considered tedious when we want to observe multiple variables at the same time. After the problem we have investigated is fixed, we also have to remove all the related code fragments used in probing from the program as keeping it the in program may generate other unexpected behaviors. In short, probing is cheap to start, but it appears to be very costly at its end.

### 4.1.2 Logging

Logging is supposed to be the more recommended method of choice as an upgrade from probing. In the current practical standard, It is one of the most effective practice applied to our codes for facilitating future support. Logging is the process of writing log messages during the execution of a program to a central place, which can be the terminal, a flat-file, an email, or even a database. Each entry in a log record has essential information, including a timestamp, meta information, and a message we planned ahead as it is a piece of useful information.

This logging action allows us to report and persist info, error, and warning messages later retrieved and analyzed. These achieved messages are considered more and more useful and software development assets in recent days. In short, logging allows anyone to read the log entry to understand how the application behaved in production. For a longer elaboration, the logging assets are used in the mining software repository process that analyzes the data generated during the software development to uncover interesting and actionable information about software systems and projects. For example, these software assets are used in many open-source software projects to maintain a sufficiently high collaboration to maintain project quality. The mining software repository process is significant support that makes us see many exceptionally high-quality open-source software such as Mozilla Firefox, Eclipse, and LibreOffice.

In Java, logging is commonly implemented through logging APIs, such as the Log4j (2018) or the Slf4j (2019) libraries. The following list is the essential components of logging:

- Logging frameworks;
- Level;
- Logger;
- Appender;
- Layout;
- Configuration;

---

Log4j (2020). Log4j – Apache Log4j 2. Retrieved June, 2020, from <https://logging.apache.org/log4j>

Slf4j (2020). Slf4j. Retrieved June, 2020, from <http://www.slf4j.org>

### Logging frameworks

Logging activities in Java require at least one logging framework to organize it. Java provides a built-in framework in the `java.util.logging` package. It is a framework that provides the necessary components to create and store log messages. These components include objects, methods, and other configurations. However, it seems to be much less popular as compared with alternative third-party frameworks, such as `Log4j` and `Slf4j`.

Recently, in practice, binding `Log4j` with `Slf4j` is the de-facto standard for Java developers, as it makes the logging activities ultimately flexible since almost every possible configurations that a developer may need for their program, have already made available.

### Logging levels

The Logging level is simply the categorized labels of a log entry, where the categorization is based on the level of urgency. Since urgent messages generated during the production may often mean that a too unexpected behavior has occurred, the possible consequences can be that we are losing some reputations or money because of it. Thus, clearly defined labels are essential such that they will help anyone who reads the log entries to be able to quickly separate the kinds of information by the level of urgency. For example, a log entry stating that it is related to a severe error should be caught by every readers' eye. On the other hand, only some readers may be interested in the log entry supplying trivial information. Typically, there is a set of typical levels that are widely applied in many programming languages. The following list presents the typical log levels sorted in the order from most severe to least severe:

- **FATAL** represents the most dreadful situations that may occur in our application. Such a fatal situation can be any situation where the application or the entire system stop its service. By the level of urgency, anyone who is its handling charge of it would have to take any action as immediately as possible.

- **ERROR** is a somewhat less serious issue than *FATAL*, but it still requires quick action. The level of damage or urgency may not as catastrophic as that of *FATAL* since, in most cases, the application may still work. However, some undesirable output may generate, such that users are affected without having a way to work around the issue by themselves. More often, the only use for the *ERROR* level within an application is when a valuable business use case cannot be completed. It is important to note that we should not assign this log level to too many log entries as it would add noise to the logging and reduce the level of significance of the more proper *ERROR* log entries.
- **WARN** represents potentially harmful situations, where the application still can overcome the problem by itself, or the fixes can wait for longer than that of *ERROR*. An example situation that perfectly fits the *WARN* level is the first attempt to reconnect to a data provider, where its second attempt shall be better an *ERROR* log.
- **INFO** represents routine application operations. This information is more useful during development, and it is also used in production to track what is happening in the application. For example, *INFO* level logs include user logged in entires or the state change of a flag variable.
- **DEBUG** represents the most granular diagnostic information for debugging purposes. Mostly, logs set as *DEBUG* level logs are implemented to provide detailed diagnostic information for fellow developers.
- **TRACE** represents an even finer grain than that of *DEBUG*. Practically, the logs of this level are often merged with the *DEBUG* level logs.

Two remaining typical log levels indicate turning off all the logs and log everything. They are *OFF* and *ALL*, respectively. How each log level works is the next question. During runtime, the application code will make logging requests, which will have a level. The logging framework has a log level configured as a threshold level. If the request level is at the configured level or higher, it gets logged to the configured target. Otherwise, the logging system will simply deny it. Particularly, the following rank order for the log levels explained above: *ALL* < *TRACE* < *DEBUG* < *INFO* < *WARN* < *ERROR* < *FATAL* < *OFF*, where the < sign indicate the

log level on the right-hand side is at the level above that on the left-hand side. For example, if the logging framework level is set to *ERROR*, all the log requests with *ERROR* and *FATAL* levels are accepted, whereas the other will be denied.

### Loggers

Loggers are responsible for capturing log events, e.g., those that were not denied due to log levels configured and passing the log events to the appropriate Appender. They give the programmer runtime control on which statements are sent to the selected Appender or not.

### Appender

Appenders are responsible for recording log events to one or more output destinations. Aside from presenting the log message on the terminal or dump to a file, there are many more possibilities to present or store the log entries. The somewhat lengthy list below shows the widely used Appenders:

- ***ConsoleAppender*** appends the log events to the terminal, i.e., bypassing the formatted log entries to `System.out` or `System.err`. Note that `System.out` is a default target, where for a more refined purpose of bug triaging, `System.err` can also be used. A *ConsoleAppender* is considered the most frequently used appender. It is very similar to how a developer interprets the probing technique with a higher number of predefined elements, such as timestamps. Anyhow, compared with many other appender options, a *ConsoleAppender* seems to be among the least useful appenders when considering its applicability in production. This is because any messages presenting at the terminal during production can be nothing but waste. Thus, in most cases, the choice of *ConsoleAppender* used during the development will be later changed to other appenders when the application is going to become its production state. Modern logging frameworks have provided an effortless way to alter the appender anytime, by simply changing the appender name in a textual configuration file.



- ***FileAppender*** appends the log events to a file at the predefined location. A *FileAppender* is one of the most widely used in production, as opposed to a *ConsoleAppender*, where at least the log messages appended to file can be retrieved and trace anytime after the error. For example, if a user reported a software defect where we are more available to fix it many hours later, we definitely need to consult the timestamp that should indicate the time when problem may occur.
- ***RollingFileAppender*** is a variation of the *FileAppender*, where it creates a new log file anytime the predefined conditions are met. It is widely used in mature software systems where the software maintainers may find it is more useful to organize the log files better to retrieve some records in the future.
- ***DailyRollingFileAppender*** is a variation of the *RollingFileAppender*, where it creates a new log file every day. The use of a *DailyRollingFileAppender* is more generic than the *RollingFileAppender* in which the rolling conditions have to be predefined.
- ***SMTPAppender*** composes the log events as an email and sent it out when a specific logging event occurs. Typically it is used on the log events defined as *ERROR* or *FATAL* level logs. More recently, the idea of utilizing an *SMTPAppender* is more adapted to more recent technology. For example, software developers are keener to implement a custom appender that sends urgent log messages to their preferred channel, such as an instant messaging application.
- ***JDBCAppender*** appends the log events to a database. This option maximizes the possibility that the software maintainers will analyze the log entries in the future.

### Layout

An appender uses a layout to format a log event into a form that meets the needs of whatever will be consuming the log event. Modern logging frameworks mainly provide Layouts for plain text, HTML, XML, , and other logs. In practice, the log reader is not only willing to see just the log message sent by the logger, but also

much other information that may assist them in their bug or defect triaging. Utilizing modern logging frameworks, this supportive information is automatically combined with the log message before injecting it into a log appender.

Since there are several logging frameworks nowadays, there are various standards to define the layout pattern. One of the most common layouts used in Java is named `PatternLayout`. It let us decide and determine what we may need from the logging messages in the future and implement the log format. An example `PatternLayout` reads in the format like `%d{ABSOLUTE}}%d{yyyy/MM/dd HH:mm:ss} [%-6p] %c{1} -%m%n` where it will convert each piece of information back to log entries like `2020/07/01 08:00:00 [DEBUG ] HelloWorld - Debug Message Logged !!`. We have to initially know before further examining the `PatternLayout` is the meaning of the special notation like `%-6p`. The list of notations given below are of the essential ones:

- `%d{ABSOLUTE}` is for formatting the time stamp where the commonly configured `%d{ABSOLUTE}` is in the form of `%d{yyyy/MM/dd HH:mm:ss}` which represents year, month, day, time in hour, time in minute, and time in second. Changes in these characters will change how the timestamp is presented as the output.
- `%5p` where 5 can be any number, presents the log level in the log entry, where the number like 5 here forces the field's width to be five characters.
- `%t` displays the name of the thread that generates this log message.
- `%c{1}` displays the class name of the package. The number one shown in this example indicates the hierarchical level of the class to be displayed.
- `%M` represents the method name.
- `%L` decides whether the line number should also be displayed or not.
- `%m` indicates the human-understandable string that will be shown as the log message.
- `%n` indicates line breakpoint.

## Configuration

Similar to layouts, the vast amount of existing logging frameworks generates numerous configuration formats. Let us discuss the one used in the Log4j framework as it can be considered one of the most common frameworks used in practice. To configure the Log4j logging framework, we need to create a file named `log4j.properties` in the resource folder of the Java project. For example, the folder under `project/classes` directory. Script 4.1 illustrates an example `log4j.properties` file. In this example, the logging level's default value is set as `DEBUG`, and two appenders: `ConsoleAppender` and `RollingFileAppender`, are working simultaneously. Both appenders use the same layout as shown on Line 7 and Line 14 of the script.

**Script 4.1: An example Log4j configuration**

**log4j.properties**

```
+ 1 # Root logger option
+ 2 log4j.rootLogger=DEBUG, stdout, file
+ 3 # Redirect log messages to console
+ 4 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
+ 5 log4j.appender.stdout.Target=System.out
+ 6 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
+ 7 log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c
  {1}:%L - %m%n
+ 8 # Redirect log messages to a log file, support file rolling.
+ 9 log4j.appender.file=org.apache.log4j.RollingFileAppender
+10 log4j.appender.file.File=C:\\log4j-application.log
+11 log4j.appender.file.MaxFileSize=5MB
+12 log4j.appender.file.MaxBackupIndex=10
+13 log4j.appender.file.layout=org.apache.log4j.PatternLayout
+14 log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%
  L - %m%n
```

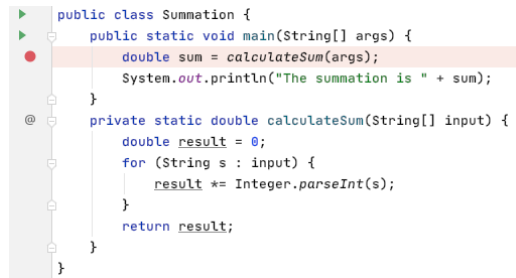
During the program execution, when the line of code with log requests are executed, the logger will consult the configuration file to emit the log messages formatted as described in the `PatternLayout` to all the appenders. More detailed examples of logging in Java will be provided later in the case study section at the end of this chapter.

### 4.1.3 Debugging

Debugging is the process of detecting and correcting errors in a program on the fly. Different from other techniques discussed earlier in this chapter, one that requires a debugger is considered a more complicated case in which an IDE or a stack trace cannot merely help us. In particular, these case is more likely to be related to logic or algorithms such that if it happens early in the program, it may not manifest itself until much later when the implementation of many other components is completed and severely affected by it. Often, it is a real challenge to overcome these kinds of problems.

The critical feature of debugging is figuring out the program state anytime during its runtime by pausing the execution at any line of code of the program and analyzing the variable states. In other words, during the program execution, the debugger allows us to thoroughly examine any variables to see how they are changed from a line of code to another line of code. In this way, it is considered more comfortable to see whether the change in any suspected variable behaves unexpectedly and introduces a bug at the end.

Let us see how a debugger works by walking through the following simple scenario, which begins by examining the source code fragment in Script 4.2. In the script, the program is supposed to calculate the summation of all values passed as command-line arguments. After compilation,

The image shows a snippet of Java code in an IDE. The code defines a class 'Summation' with a 'main' method and a 'calculateSum' method. A red dot, representing a breakpoint, is placed on the line 'double sum = calculateSum(args);' in the 'main' method. The 'calculateSum' method takes an array of strings and returns a double sum. The 'main' method prints the result. The IDE interface includes a gutter on the left with various icons, and the code is color-coded (keywords in blue, variables in black, etc.).

```
public class Summation {  
    public static void main(String[] args) {  
        double sum = calculateSum(args);  
        System.out.println("The summation is " + sum);  
    }  
    private static double calculateSum(String[] input) {  
        double result = 0;  
        for (String s : input) {  
            result += Integer.parseInt(s);  
        }  
        return result;  
    }  
}
```

Figure 4.1: Defining a breakpoint

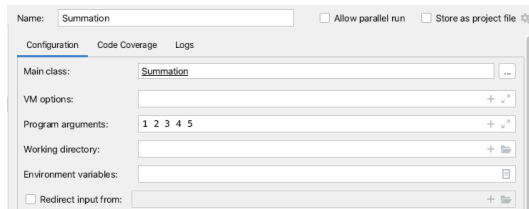
this IDE can run the programming without any error shown up. However, the result calculated by the program is incorrect. For example, if we pass 1 2 3 4 5, the expected output should be 15. However, this program instead generates 0 as the output, which is incorrect. In this kind of situation, the first thing we may have to do is think about which part of the program is where the error may come from. Of course, it should not be any lines of code related to the view component because it does not carry out any computation. Most likely in this program, the problem

shall come from the `calculateSum` method. Since, this error is not a compile error, we have to examine it at runtime.

**Script 4.2: An example Java class to debug****summation.java**

```
+ 1 public class Summation {
+ 2     public static void main(String[] args) {
+ 3         double sum = calculateSum(args);
+ 4         System.out.println("The summation is " + sum);
+ 5     }
+ 6     private static double calculateSum(String[] input) {
+ 7         double result = 0;
+ 8         for (String s : input) {
+ 9             result *= Integer.parseInt(s);
+10         }
+11         return result;
+12     }
+13 }
```

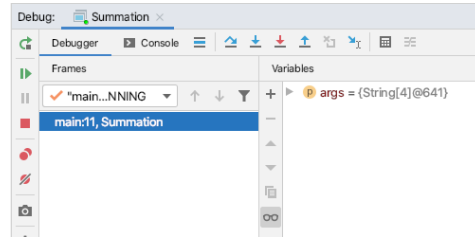
Using a debugger, we can set a breakpoint at any line of code to represent the place to pause the execution and examine the current values of all the declared variables. Typically, we pause the execution once before the suspected line of code. Let this example illustrates how it can be achieved in the IntelliJ Idea IDE. Initially, we set the first breakpoint at Line 3 of the Script 4.2 by clicking in front of the line of code until the IDE shows a red circle symbol in front of the line of code as shown in Figure 4.1, then run the program in debug mode.



**Figure 4.2: Defining the program arguments for debugging**

To run the program in IntelliJ Idea IDE debug mode of the IntelliJ Idea IDE, we have to access the feature by selecting Run and then Edit Configurations at the main menu. Then we have to enter the program arguments which we get the incorrect answer, in this case, it is 1 2 3 4 5. The graphical explanation of this step is illustrated in Figure 4.2.

The *Debug* execution mode is under the *Run* button we always use to run the program in the IDE. After we click the execution button, it will bring up the debugging session where the program is executed up to the line of code where the breakpoint is defined. The Debug tool window presenting the



**Figure 4.3:** An example Debug tool windows of the IntelliJ Idea IDE

current debugging process is shown in Figure 4.3. As we may see in the tool window, the `calculateSum` method has not been called yet, so all its local variables are not presented in the tool window yet. From this step, we can get the most out of the Debug tool by letting it execute the code line by line, and at the same time, show us the current state of all the variables already declared and defined before the particular line of code. To do that, we have to click the button named *Step into* which is represented by a down arrow symbol in blue color. After clicking the step into button a few more times, we will set that it can examine each iteration of the for loop, and we shall see that the result variable always holds the value 0.0 in all the iterations. Thus, this gives us an important clue where the error might relate to the result variable's initial value at Line 7 or the operand at Line 9. If we think more carefully, we will figure out that the multiplying operand in Line 9 is the root cause of all the problems here. Hence, after we change the `*=` operand into `+=` and execute the program again, we will see that the problem is already gone as the output is not equal to 10.

The *Step into* function examines all the lines of code associated with the line of code it starts. For example, if we click the *Step into* button at the line having a method call, the debugger will go through all the lines of the called method one by one. However, suppose that we only want to examine the final result of the method. In that case, we can use another functionality named *Step over*, which can be launched in by the IntelliJ Idea IDE by clicking the button on the left-hand side of the *Step into*. The debugger will execute the method and show us the program state after completing the method call.

The last but not least feature of the IntelliJ Idea debugger is the *Evaluate expression* functionality. Without this functionality, we may have to modify the code

when we want to undertake a what-if analysis. On the other hand, the *Evaluate expression* functionality provides us with the tool to do it without modifying any code. To bring up the application windows, we have to click the second button on the right-hand side of the debugger tool's icon bar. After clicking the button, an application window will be presented. Here, we can add any code fragment to it and it lets any changes made in this window propagate to the program under debugging. With the availability of this debugging toolset, the lecturer cannot find any excuses for anyone to prefer the probing technique over the debugging technique.

## 4.2 Additional topic: Sprite animation

It is the lecturer thought that a case study to showcase the logging and debugging activities might require a scenario with a lot of state changing. Among many possible choices, the lecturer opted for a simple game with sprite animation implemented. In this scenario, there is numerous state changes and it is challenging to ensure the consistency and correctness for all the states during the program execution. Thus, before we begin the case study of this chapter, let us examine some essential backgrounds on games and sprite animations first. Basically, a game can be viewed as a composition of logic loops and some calculations. Generally, two loops are implemented. One of which takes care of logic computations and the other one takes care of visual effect. The main difference between the two loops is that the two loops refresh themselves at different frequencies. Compared with a logic loop, a drawing loop refreshes itself much more frequently, e.g., 60 times per second or slightly higher. In contrast, a logic loop may only need to refresh itself 5 to 10 times per second because the drawing loop has the primary responsibility to make every visual effect as much smooth as possible, whereas the logic computation does not require such a rapid refresh update.

Implementing any animation-related components is deviated somewhat from how we have coded since drawing shall be synchronized with the human eye. Too slow or too fast in some graphic movement is undesirable. For example, if we implement a shape object's movement following the code snippet in Script 4.3, despite no error, it is considered not an animation since we cannot see the move-

ment due to the movement is too fast. All we have to do here is to control the refresh rates to be the closest to human convenience. Script shows 4.4 an updated `redrawShape.java`, where a thread sleep is added to the drawing loop. The number  $1000/60$  represents the refresh rate, i.e., each time lasts  $1000/60$  milliseconds.

**Script 4.3: Drawing objects which seems to be too fast****redrawObject.java**

```
+ 1 for (int i = 0; i < 1000; i++) {  
+ 2     x++;  
+ 3     y++;  
+ 4     c.redraw();  
+ 5 }
```

**Script 4.4: Adding a thread sleep to implement the movement****redrawShape.java**

```
1 for (int i = 0; i < 1000; i++) {  
2     x++;  
3     y++;  
4     c.redraw();  
+ 5     try {  
+ 6         Thread.sleep(1000/60);  
+ 7     } catch (Exception e) {  
+ 8         e.printStackTrace();  
+ 9     }  
10 }
```

In programming terms, animation is also referred to as a time-based motion that draws an image, erases it, and redraws it in a new coordinate in every single iteration of the drawing loop, with the refresh rate closest to human convenience as possible. It should be noted that the refresh rate must not be too fast or too slow, since too slow will make the user experience lagging, whereas too fast will make the user experience teleportation, rather than movement. To say in broad, animation can be viewed as the time-based alteration of graphical objects through different states, locations, sizes, and orientations (Haase and Guy 2007). The essen-

---

Haase C, Guy R (2007) *Filthy rich clients: Developing animated and graphical effects for desktop Java applications*. Pearson Education.



tial components required for implementing basic animation are drawing, clocks, movement, collision handling, and interpolation.

Drawing in JavaFX is to cast every object we want to implement motion on it into its accepted type. Notably, we have to extend the object to `javafx.scene.layout.Pane` and use the object as a JavaFX node. Preliminarily, the only thing we may need to note is about coding the coordinate system. Unlike conventional coordinate systems where the coordinate  $(0, 0)$  is usually at the bottom left of the arrangement, the coordinate  $(0, 0)$  in Java is at the object's top left.

A clock is a metaphor used for synchronizing different timeframes. Anyone who is familiar with high-end digital audio systems may also be familiar with this clock. A digital audio system often comprises many devices, such as a wireless receiver, a digital to analog converter, an audio-video receiver, several amplifiers, and many others, depends on how seriously the system owner is. These devices nowadays have to be more involved with digitalized signals. Without a proper synchronization of these signals, the output audiophile sound quality can be drastically degraded due to signal interference. Thus, in many expensive systems, the system owner usually decides to add another device called a Master clock to the system. This device does nothing but generates the reference signal to synchronize all the signals generated from each arbitrary device in the entire system. Likewise, in animation, a for loop is usually implemented to mimic the master clock's functionality. In brief, the purpose of clocks in animation is to let all the visual components synchronized with only one reference. This course lets us synchronize the two main programming loops, i.e., the drawing loop and the logic loop, with one master clock that ticks every millisecond.

The movement considers the essential ingredients required for implementing a movable object with the main goal that the movement must be at the highest level of realistic possible. One of the most accepted methodologies implementing movement is to apply some principles in basic physics mechanics, where a moveable object comprises a position, a velocity, and an acceleration at all times. Without the need to control all of them, applying the principles allows to simply alter the acceleration and the change will propagate to the velocity and the position automatically. Furthermore, it shall be even more realistic if we are able to add many other physics principles to the movement, such as force and momentums. For example, to implement the movement of a game character, we let the character move

towards the direction defined by keyboard events. We let the key becomes the force that pushes the character to the direction. Then, the character moves with its pre-defined acceleration values until it reaches its maximum speed limit. The releases of the key are to stop adding the movement force and the character shall slow down until the speed reach zero. Of cause, we are not the ones who control the position or the velocity of the game character directly, but it is the physics mechanic formulas that do it.

Collision is to mimic how physical objects collided with each other. Since all the objects in a game are virtual elements, we have to define the behavior what shall they react to each other when a movement make some of them collided with the other objects. Clearly, this is the component that requires the most various mathematical works among the other five components. Common responses to a collision between game objects are such as preventing further movement, health point reduction, damage rendering, death scene rendering, and score calculation.

Before we discuss interpolation, let us do a quick summarization of the above four components once. Script 4.5 and Script 4.6 present the structures of the logic loop and the drawing loop, respectively. Between Line 5 and Line 6 of the two scripts defines the refresh rate, where the Logic loop refreshes 10 times per second and the Drawing loop refreshes 60 times per second. For the Logic loop, Line of 12 and Line 13 of Script 4.5 attempts to update the game object once per clock tick, i.e., every 100 milliseconds. Then, it checks where there any collisions have occurred and waits for the next clock tick. For the drawing loop, on the other hand, the clock once every 62.5 milliseconds. In each iteration, it first checks for any possible drawing collisions and redraws the game object based on the results of the collisions or other input signals.

The final component essential for implementing animation is an interpolation, which is a technique that alters a set of figures fast enough to fool the human brain to believe that the object is moving. The set of figures is usually composed as one image file called a sprite sheet, for better organization and making the program more



Figure 4.4: An example sprite sheet

Script 4.5: An example logic loop structure

logicLoop.java

```
+ 1 //Imports are omitted
+ 2 public class LogicLoop {
+ 3     public LogicLoop(Platform platform) {
+ 4         this.platform = platform;
+ 5         frameRate = 10;
+ 6         interval = 1000.0f / frameRate;
+ 7         running = true;
+ 8     }
+ 9     @Override
+10     public void run() {
+11         while (running) {
+12             update(platform.getCharacter());
+13             checkCollisions(platform.getCharacter());
+14             try {
+15                 Thread.sleep((long) interval);
+16             } catch (InterruptedException e) {
+17                 e.printStackTrace();
+18             }
+19         }
+20     }
+21 }
```

simple. An example sprite sheet is shown in Figure 4.4 (Armstead n.d.). To utilize the sprite sheet, we have to create a viewport, a kind of sliding window that slides through every single image in the sheet once every clock tick. For example, at the beginning of the program execution, only the Megaman image on the sprite's top left is shown. After 1000/60 seconds have passed, the next Megaman image on the same row of the sprite sheet will be shown in the application instead. The action of rendering a subfigure one by one and returning to the first one after the entire sheet is rendered is called interpolation. This interpolation will make moving objects more realistic.

---

Armstead L (n.d.) Exemplo De Spritesheet - Sprite Sheet Megaman Png, Transparent Png. Retrieved June, 2020, from [https://www.kindpng.com/imgv/mwToi\\_exemplo-de-spritesheet-sprite-sheet-megaman-png-transparent/](https://www.kindpng.com/imgv/mwToi_exemplo-de-spritesheet-sprite-sheet-megaman-png-transparent/)

Script 4.6: An example drawing loop structure

drawingLoop.java

```
+ 1 //Imports are omitted
+ 2 public class DrawingLoop {
+ 3     public DrawingLoop(Platform platform) {
+ 4         this.platform = platform;
+ 5         frameRate = 60;
+ 6         interval = 1000.0f / frameRate;
+ 7         running = true;
+ 8     }
+ 9     @Override
+10     public void run() {
+11         while (running) {
+12             checkDrawCollisions(platform.getCharacter());
+13             paint(platform.getCharacter());
+14             try {
+15                 Thread.sleep((long) interval);
+16             } catch (InterruptedException e) {
+17                 e.printStackTrace();
+18             }
+19         }
+20     }
+21 }
```

Script 4.7 illustrates a code fragment using a viewport to implement the interpolation functionality. The `tick` method selects the subfigure to render during the current clock tick, where the image sequence is indicated by the `curIndex` variable. The `columns` and `rows` variables represent the number of columns and rows of the sprite sheet. The `interpolate` method sets the actual coordinate of the viewport, where the `x` and `y` represent the origin coordinate of the viewport. For example, the values of `x` and `y` variables for the first subfigure are both 0. After a clock tick, the viewport will move to the right, and the value of `x` will become the width of the subfigure. It is important to note that all the subfigures a sprite sheet must be in the same size. Table 4.1 shows the relationship between the `curIndex`, `curColumnIndex`, and the `curRowIndex` variables, given the size of the sprite sheet of Figure 4.4 of five columns and two rows.

Script 4.7: An example implementation of interpolation

interpolation.java

```
+ 1 public void tick() {
+ 2     curColumnIndex = curIndex % columns;
+ 3     curRowIndex = curIndex / columns;
+ 4     curIndex = (curIndex+1) / (columns * rows);
+ 5     interpolate();
+ 6 }
+ 7 protected void interpolate() {
+ 8     final int x = curColumnIndex*width+offsetX;
+ 9     final int y = curRowIndex*height+offsetY;
+10     this.setViewport(new Rectangle2D(x, y, width, height));
+11 }
```

Table 4.1: The relationship between the curIndex, curColumnIndex, and the curRowIndex variables given the size of the sprite sheet of Figure 4.4

curIndex	curColumnIndex	curRowIndex
0	0	0
1	1	0
2	2	0
3	3	0
4	4	0
5	0	1
6	1	1
7	2	1
8	3	1
9	4	1

### 4.3 Case study

This chapter will build a simple 2D platformer game in which that game character can be moved by using a keyboard. Let the name of the project of this case be Platformer, and its assets are made available at <http://myweb.cmu.ac.th/passakorn.p/953233/materials/chapter04.zip>. The files used in this project are structured as follows:

```
src
├── assets
├── controller
│   ├── DrawingLoop.java
│   ├── GameLoop.java
│   └── Launcher.java
├── model
│   ├── AnimatedSprite.java
│   ├── Character.java
│   └── Keys.java
└── view
    └── Platform.java
```

The implementation of all the components of this case study is considered very complicated due to numerous flag variables are to be used. Therefore, this is the programming situation in which the lecture believes that it is one of the best situations to learn to let the debugger assist our application development. During any application development step in this case study, the lecturer highly recommends everyone practice troubleshooting with a debugger whenever a development problem is encountered.

Let us begin with the Character class by following Script 4.8. This class consists of three groups of variables. The first group is the size of character image defined by the CHARACTER\_WIDTH and CHARACTER\_HEIGHT variables. The next group is for the character image, its container, and the character's current position on the screen. These are defined by the characterImg and imageView, x and y variables, respectively. The last group is the keyboard codes used in moving the character, which are de-

finied by the leftKey, rightKey, and upKey, respectively. The constructor of the class simply sets the position of the object instance, load the image visually representing the character, and binds the assigned keyboard keys that are used for moving the character image on the screen.

**Script 4.8: /src/model/Character.java**

```
+ 1 //Imports are omitted
+ 2 public class Character extends Pane {
+ 3     public static final int CHARACTER_WIDTH = 64;
+ 4     public static final int CHARACTER_HEIGHT = 64;
+ 5     private Image characterImg;
+ 6     private ImageView imageView;
+ 7     private int x;
+ 8     private int y;
+ 9     private KeyCode leftKey;
+10     private KeyCode rightKey;
+11     private KeyCode upKey;
+12     public Character(int x, int y, KeyCode leftKey, KeyCode rightKey, KeyCode
        upKey) {
+13         this.x = x;
+14         this.y = y;
+15         this.setTranslateX(x);
+16         this.setTranslateY(y);
+17         this.characterImg = new Image(getClass().getResourceAsStream("/assets/
            StillMario.png"));
+18         this.imageView = new ImageView(characterImg);
+19         this.leftKey = leftKey;
+20         this.rightKey = rightKey;
+21         this.upKey = upKey;
+22         this.getChildren().addAll(this.imageView);
+23     }
+24 }
```

On continual from the Character class, we implement the Platform class to represent the game scene. The class is illustrated in Script 5.23, and it also contains three groups of variables. The first group consists of constant variables: WIDTH, HEIGHT, and GROUND. The WIDTH and HEIGHT variables define the size of the game

screen in terms of pixels. The `GROUND` variable represents the background image's virtual ground, where the value is set as 300, meaning that the virtual ground is set to be at 300 pixels from the top of the background image. The second group of variables in this class has only one variable named `platformImg`. It is the background image of the game. Finally, the third group of variables also has one variable named `character`. It is an instance of the `Character` we implemented in Script 4.8.

**Script 4.9: /src/view/Platform.java**

```
+ 1 //Imports are omitted
+ 2 public class Platform extends Pane {
+ 3     public static final int WIDTH = 800;
+ 4     public static final int HEIGHT = 400;
+ 5     public final static int GROUND = 300;
+ 6     private Image platformImg;
+ 7     private Character character;
+ 8     public Platform() {
+ 9         platformImg = new Image(getClass().getResourceAsStream("/assets/Background.
+ 10             png"));
+ 11         ImageView backgroundImg = new ImageView(platformImg);
+ 12         backgroundImg.setFitHeight(HEIGHT);
+ 13         backgroundImg.setFitWidth(WIDTH);
+ 14         character = new Character(30, 30, KeyCode.A, KeyCode.D, KeyCode.W);
+ 15         getChildren().addAll(backgroundImg, character);
+ 16     }
+ 17     public Character getCharacter() { return character; }
```

Next is the `Launcher` class of the application, in which we only create a simple JavaFX launcher class with an instance of the platform class in it. Script 4.10 illustrates the initial version of this `Launcher` class. After all the three classes have been implemented, we will be able to run it, and the application will be rendered as a still image shown in Figure 4.5.



**Script 4.10: /src/controller/Launcher.java**

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     public static void main(String[] args) { launch(args); }
+ 4     @Override
+ 5     public void start(Stage primaryStage) {
+ 6         Platform platform = new Platform();
+ 7         Scene scene = new Scene(platform,platform.WIDTH,platform.HEIGHT);
+ 8         primaryStage.setTitle("platformer");
+ 9         primaryStage.setScene(scene);
+10         primaryStage.show();
+11     }
+12 }
```

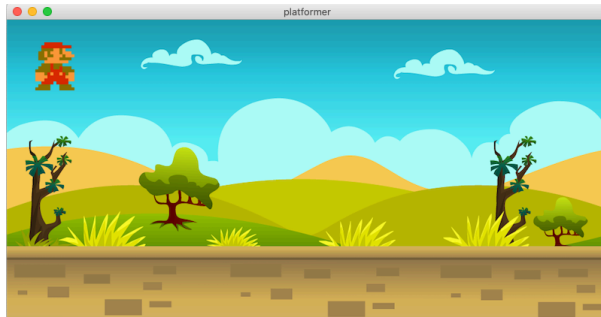


Figure 4.5: The initialized GUI of the case study of Chapter 4

### 4.3.1 Game loop

Before examining the implementation with the separated logic loop and drawing loop discussed earlier in this chapter, let us examine a game with one single game loop first. Script 5.28 illustrates a class name `GameLoop` which implements the `Runnable` interface. Its main body is an infinite loop running on a background thread that refreshes itself every 1/60 seconds. In the `run` method, we call the `update`, `checkCollisions`, and `paint` methods, in sequence. To examine how `GameLoop` works, let us modify the `Character`, the `GameLoop`, and the `Launcher` classes following Script 5.10, Script 5.15, and Script 4.14, respectively. The updated `Character` class in 5.10 implements how character falls to the virtual ground. Line 15 and Line 16 of the script creates two variables named `yVelocity` and `isFalling`. The `yVelocity` variable determines the falling speed, which we will later improve the mechanism to let this velocity controlled by gravity instead of a constant falling speed. The `isFalling` variable is a flag variable determining the falling state of the character. There are three methods added to the current updated of the class. The `moveY` method simply replaces the coordinate value on the y axis of the character, and the new coordinate will be redrawn on the next clock tick. The `checkReachFloor` method checks whether the character should no longer fall since it has already reached the virtual ground. Finally, the `repaint` method is the method that redraws the character inside the platform. The three updated methods in the `Character` are controlled by the `GameLoop` controller, where each method is called at Line 17, Line 20, and Line 23 of Script 5.15, respectively. Finally, this update is completed by initializing an instance of the `GameLoop` class, and run it on a background thread. These are shown at Line 9 and Line 14, of the updated `Launcher` class in Script 4.14. If we compile and run the application now, we will be able to see the character image falling towards the ground, and it will stop when the bottom border of the image has reached the virtual ground.

**Script 4.11: /src/controller/GameLoop.java**

```
+ 1 //Imports are omitted
+ 2 public class GameLoop implements Runnable {
+ 3     private Platform platform;
+ 4     private int frameRate;
+ 5     private float interval;
+ 6     private boolean running;
+ 7     public GameLoop(Platform platform) {
+ 8         this.platform = platform;
+ 9         frameRate = 60;
+10         interval = 1000.0f / frameRate;
+11         running = true;
+12     }
+13     private void update(Character character) {
+14     }
+15     private void checkCollisions(Character character) {
+16     }
+17     private void paint(Character character) {
+18     }
+19     @Override
+20     public void run() {
+21         while (running) {
+22             float time = System.currentTimeMillis();
+23             update(platform.getCharacter());
+24             checkCollisions(platform.getCharacter());
+25             paint(platform.getCharacter());
+26             time = System.currentTimeMillis() - time;
+27             if (time < interval) {
+28                 try {
+29                     Thread.sleep((long) (interval - time));
+30                 } catch (InterruptedException e) {
+31                     e.printStackTrace();
+32                 }
+33             } else {
+34                 try {
+35                     Thread.sleep((long) (interval - (interval % time)));
+36                 } catch (InterruptedException e) {
+37                     e.printStackTrace();
+38                 }
+39             }
+40         }
+41     }
+42 }
```

**Script 4.12: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
3     ...
+15     int yVelocity = 3;
+16     boolean isFalling = true;
17     public Character(int x, int y, KeyCode leftKey, KeyCode rightKey, KeyCode
        upKey) {
18         this.x = x;
        ...
27     }
+28     public void moveY() {
+29         if(isFalling) {
+30             y = y + yVelocity;
+31         }
+32     }
+33     public void checkReachFloor() {
+34         if(isFalling && y >= Platform.GROUND - CHARACTER_HEIGHT) {
+35             isFalling = false;
+36         }
+37     }
+38     public void repaint() {
+39         setTranslateX(x);
+40         setTranslateY(y);
+41     }
42 }
```

**Script 4.13: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
3     ...
16     private void update(Character character) {
+17         character.moveY();
18     }
19     private void checkCollisions(Character character) {
+20         character.checkReachFloor();
21     }
22     private void paint(Character character) {
+23         character.repaint();
24     }
    ...
}
```

**Script 4.14: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     public static void main(String[] args) { launch(args); }
4     @Override
5     public void start(Stage primaryStage) {
6         Platform platform = new Platform();
+ 7         GameLoop gameLoop = new GameLoop(platform);
8         Scene scene = new Scene(platform,platform.WIDTH,platform.HEIGHT);
9         primaryStage.setTitle("platformer");
10        primaryStage.setScene(scene);
11        primaryStage.show();
+ 12        (new Thread(gameLoop)).start();
13    }
14 }
```

**4.3.2 Keys**

Let us use the keyboard keys to control the character movement. For example, for the character we are implementing, let the keys *a*, *w*, and *d* control its movement, where the key *a* will move the character to the left, *w* is to jump, and *d* to move the character to the right. To bind these keys in the application, we have to implement some event objects and event handlers. Then we have to bind each key with the particular event it is meant to control. Script 4.15 illustrates a new class named `Keys`, which acts as a broker who checks the keys being pressed during a particular clock tick to let the application respond to all the keys correctly. The `Keys` class is designed to be independent of the `Character` class because, in a local co-op setup, the players will use different keys to control different characters. Thus, a separated class design will provide better modularity to the implementation of the control activities. Note that a `HashMap` is used in this `Keys` class because, often in gaming, combinations of keys are pressed simultaneously to create more control possibilities with a composite input signal.

To integrate the `Keys` class to the application, let us modify the `Platform`, the `Launcher`, the `GameLoop` classes, and the `Character` classes, following Script 5.13, Script 4.17, Script 4.18, and Script 5.11, respectively. Three lines of code modified

**Script 4.15: /src/model/Keys.java**

```
+ 1 //Imports are omitted
+ 2 public class Keys {
+ 3     private HashMap<KeyCode, Boolean> keys;
+ 4     public Keys() {
+ 5         keys = new HashMap<>();
+ 6     }
+ 7     public void add(KeyCode key) {
+ 8         keys.put(key, true);
+ 9     }
+10     public void remove(KeyCode key) {
+11         keys.put(key, false);
+12     }
+13     public boolean isPressed(KeyCode key) {
+14         return keys.containsKey(key);
+15     }
+16 }
```

in 5.13 are at Line 11, Line 14, and Line 21. They all are to integrate the Keys class with the application's main view component, which is the Platform class. Next, Line 11 and Line 12 of the updated Launcher class in Script 4.17 are anonymous classes that pass the keyboard pressing or releasing signal the application can capture to the Keys class's instance in the Platform class. These keyboard signals will be added to the HashMap if the signal is a key pressing, or removed from the HashMap if it is a key releasing. The changes in the updated GameLoop class are in the update method on Line 16. In this updated script, four conditional statements have been added. They are to create how the application should respond to the user input signals, which intend to move the character to the left, to the right, and to stop the character, respectively. Finally, the modified Character class in Script 5.11 simply adds the programming logic to control the character movement to the left, right, and to stop the character. Note that, we can stop the character on the y axis because an external force, not us, controls it. After we add the Getter methods for the leftKey, rightKey, and upKey variables of the Character class, we should be able to move the Mario character using the *a*, *w*, and *d* keys.

**Script 4.16: /src/view/Platform.java**

```
1 //Imports are omitted
2 public class Platform extends Pane {
3     ...
10    private Character character;
+ 11    private Keys keys;
12    public Platform() {
+ 13        keys = new Keys();
14        platformImg = new Image(getClass().getResourceAsStream("/assets/Background.
        png"));
15        ImageView backgroundImg = new ImageView(platformImg);
16        ...
19    }
20    public Character getCharacter() { return character; }
+ 21    public Keys getKeys() { return keys; }
22 }
```

**Script 4.17: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     public static void main(String[] args) { launch(args); }
4     @Override
5     public void start(Stage primaryStage) {
6         Platform platform = new Platform();
7         GameLoop gameLoop = new GameLoop(platform);
8         Scene scene = new Scene(platform,platform.WIDTH,platform.HEIGHT);
+ 9         scene.setOnKeyPressed(event-> platform.getKeys().add(event.getCode()));
+ 10        scene.setOnKeyReleased(event -> platform.getKeys().remove(event.getCode()));
11        ...
12    }
```

**Script 4.18: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
    ...
16     private void update(Character character) {
+ 17         if (platform.getKeys().isPressed(character.getLeftKey())) {
+ 18             character.setScaleX(-1);
+ 19             character.moveLeft();
+ 20         }
+ 21         if (platform.getKeys().isPressed(character.getRightKey())) {
+ 22             character.setScaleX(1);
+ 23             character.moveRight();
+ 24         }
+ 25         if (!platform.getKeys().isPressed(character.getLeftKey()) && !platform.
            getKeys().isPressed(character.getRightKey())) {
+ 26             character.stop();
+ 27         }
+ 28         if (platform.getKeys().isPressed(character.getLeftKey()) && platform.
            getKeys().isPressed(character.getRightKey())) {
+ 29             character.stop();
+ 30         }
31         character.moveY();
32     }
    ...
}
```

**Script 4.19: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
    ...
+ 15     int xVelocity = 5;
16     int yVelocity = 3;
17     boolean isFalling = true;
    ...
+ 28     public void moveLeft() {
+ 29         x = x-xVelocity;
+ 30     }
+ 31     public void moveRight() {
+ 32         x = x+xVelocity;
+ 33     }
+ 34     public void stop() { xVelocity = 0; }
    ...
}
```



### 4.3.3 Collision checking

The first collision we will check in this application is the check whether the character is hitting the game border or not. Only two things to be updated to enable this collision checking functionality are in the `GameLoop` and `Character` classes. They all are presented in Script 4.20 and Script 5.14, respectively. The update in the `GameLoop` class just adds a method call to check whether the character is hitting the game border or not. This method is implemented in `Character` class at Line 34 of its current update in Script 5.14. The programming logic to check this condition is to simply check whether the modified position of the `x` variable due to its movement, makes its value go below 0 above the width of the background image or not. If any of which does, this method simply alters the value of `x` to become the boundary value, e.g., the `x` at the application border, before the `GameLoop` calls the repaint method to redraw the image.

#### Script 4.20: /src/controller/GameLoop.java

```

1 //Imports are omitted
2 public class GameLoop implements Runnable {
    ...
19     private void checkCollisions(Character character) {
+ 20         character.checkReachGameWall();
21         character.checkReachFloor();
22     }
    ...

```

#### Script 4.21: /src/model/Character.java

```

1 //Imports are omitted
2 public class Character extends Pane {
    ...
+ 34     public void checkReachGameWall() {
+ 35         if(x <= 0) {
+ 36             x = 0;
+ 37         } else if( x+getWidth() >= Platform.WIDTH) {
+ 38             x = Platform.WIDTH-(int)getWidth();
+ 39         }
+ 40     }
41     public void checkReachFloor() {
42         if(isFalling && y >= Platform.GROUND - CHARACTER_HEIGHT) {
43             isFalling = false;
44         }
45     }

```

#### 4.3.4 Jumping

Pressing the key that moves the character up, e.g., *w*, should result in exerting the force to send it up from the virtual ground. After a short while, when the force is no longer higher than that of the gravity, it should move down until it hit the ground. The action of moving down is not at a constant speed because gravity is not velocity, but it is an acceleration. Thus, the speed from the highest place the character stops moving should be increasing until it arrives at the ground. By the way, mimicking this jumping mechanism may be too complicated to be a good starting point. Thus, let us examine a step at a time by trying to understand how jumping works by simplifying the jumping by representing it with only force, velocity, and location, first. Let us begin with the `Character` class. As shown in Script 4.22, we add three new variables named `canJump`, `isJumping`, and `highestJump`. The first two variables are flag variables indicating the state whether the character can jump or not, and whether the character is jumping or not, respectively. The two flags are essential because, without them, we will not be able to prevent the player from spamming the up button to let the character do the double or triple jumps. Thus, when we set the `canJump` state to be `false`, no matter how many times the player hits the *w* button, the character will not jump again until it falls back to the ground. The `highestJump` variable indicates the height in terms of pixels above the virtual ground the character will reach until it falls back. In this current update, we add the condition that the character is moving up in the `moveY` method between Line 39 and Line 41 of the script. Then we add two new methods named `jump` and `checkReachHighest` between Line 50 and Line 63 of the script. These two methods simply play with the two flag variables such that the `jump` method will be executed right after the player presses the *w* button only if the `canJump` state says that it can jump, i.e., the value is `true`. Then, in the `checkReachHighest`, it will check whether the character has reached the maximum vertical height it can reach or not. If it does, it should start to fall down by letting the `isJumping` state become `false` and `isFalling` state become `true`. Then to allow the character to jump again, the `checkReachFloor` method will be the method that checks if the character has already arrived at the ground after falling. Once it has hit the virtual ground, the `canJump` state will be set to `true` again.

**Script 4.22: /src/model/Character.java**

```
1  //Imports are omitted
2  public class Character extends Pane {
    ...
16     int yVelocity = 3;
17     boolean isFalling = true;
+ 18     boolean canJump = false;
+ 19     boolean isJumping = false;
+ 20     int highestJump = 100;
    ...
35     public void moveY() {
36         if(falling) {
37             y = y + yVelocity;
- 38         }
+ 39     } else if(isJumping) {
+ 40         y = y - yVelocity;
+ 41     }
42 }
43 public void checkReachGameWall() {
44     if(x <= 0) {
45         x = 0;
46     } else if( x+getWidth() >= Platform.WIDTH) {
47         x = Platform.WIDTH-(int)getWidth();
48     }
49 }
+ 50 public void jump() {
+ 51     if (canJump) {
+ 52         yVelocity = 5;
+ 53         canJump = false;
+ 54         isJumping = true;
+ 55         isFalling = false;
+ 56     }
+ 57 }
+ 58 public void checkReachHighest () {
+ 59     if(isJumping && y <= Platform.GROUND - CHARACTER_HEIGHT - highestJump) {
+ 60         isJumping = false;
+ 61         isFalling = true;
+ 62     }
+ 63 }
64 public void checkReachFloor() {
65     if(isFalling && y >= Platform.GROUND - CHARACTER_HEIGHT) {
66         isFalling = false;
+ 67         canJump = true;
68     }
69 }
```

Some minor updates are required to integrate the updated Character class into the application via the GameLoop class. That is, to execute the jump functionality, the GameLoop has to check whether the up key is being pressed or not. If so, the jump method will be executed. Then, inside the checkCollisions, the checkReachHighest class method is called at Line 38, meaning that it will be checked once per clock tick after the GameLoop has checked whether the game character has collided with the game border or not.

**Script 4.23: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
3     ...
16 private void update(Character character) {
17     ...
28     if (platform.getKeys().isPressed(character.getLeftKey()) && platform.
29         getKeys().isPressed(character.getRightKey()) ) {
30         character.stop();
31     }
+ 31     if (platform.getKeys().isPressed(character.getUpKey())) {
+ 32         character.jump();
+ 33     }
34     character.moveY();
35 }
36 private void checkCollisions(Character character) {
37     character.checkReachGameWall();
+ 38     character.checkReachHighest();
39     character.checkReachFloor();
40 }
    ...
```

Up to the current implementation, the fundamentals of a platform game is readily presented. We can control the character movement using a keyboard, and the application can respond to all the keys in both direct and indirect ways, e.g., moving left and right, or jumping. It seems to be the perfect time for us to move further to more advanced topics discussed earlier in the chapter: multiple game loops, interpolation, acceleration, and logging. Let us begin with the double game loops by implementing the `DrawingLoop` class by separating all the functionalities related to drawing from the `GameLoop` class. Particularly, Script 4.24 illustrates the `DrawingLoop` class, where the methods it takes from the `GameLoop` class are those related to collision detection and graphic rendering. For the original `GameLoop` class, Script 4.25 illustrates its modification where all the functionalities related to drawing have been moved to the `DrawingLoop` class. The refresh rate of the loop is decreased to 1000/10 seconds.

Recall what we have discussed earlier in this chapter's body, the drawing loop must only observe the changes in the variable as to render the consequence of the changes on the screen. Thus, there should be no programming logic computation carried out in any method bound to the drawing loop. This requires some workouts on the `Character` class at its `moveLeft` and `moveRight` method. In particular, Script 4.26 provides the details on this modification. The main change in this update is to compute the movement on the  $x$  axis in the same way as that of the  $y$  axis. That is, aside from modifying the location directly, we let some external forces determine it. Hence, it will be much more flexible for us to upgrade the engine used to simulate the movement in the future because the calculations for the character's position on both  $x$  and  $y$  axes are independent of the character rendering. At Line 38, Line 39, Line 43, and Line 44 of Script 4.26, the value of  $x$  is no longer modified by the `moveLeft` and `moveRight` method anymore. However, instead, we set the flag variable to control these movements, and let a new separated method named `moveX` handle everything about rendering instead. Now, the movement on both axes are controlled by for flag variables: `isMoveLeft`, `isMoveRight`, `isFalling`, and `isJumping`, which is considered complicated. The lecturer believes that is one of the situations where a debugger can greatly assist us. To integrate all the recent updates to the application, let us slightly modify the `Launcher` class following Script 4.27 to register the `DrawingLoop` class to the application and launch it.

**Script 4.24: /src/model/DrawingLoop.java**

```
+ 1 //Imports are omitted
+ 2 public class DrawingLoop implements Runnable {
+ 3     private Platform platform;
+ 4     private int frameRate;
+ 5     private float interval;
+ 6     private boolean running;
+ 7     public DrawingLoop(Platform platform) {
+ 8         this.platform = platform;
+ 9         frameRate = 60;
+10         interval = 1000.0f / frameRate; // 1000 ms = 1 second
+11         running = true;
+12     }
+13     private void checkDrawCollisions(Character character) {
+14         character.checkReachGameWall();
+15         character.checkReachHighest();
+16         character.checkReachFloor();
+17     }
+18     private void paint(Character character) {
+19         character.repaint();
+20     }
+21     @Override
+22     public void run() {
+23         while (running) {
+24             float time = System.currentTimeMillis();
+25             checkDrawCollisions(platform.getCharacter());
+26             paint(platform.getCharacter());
+27             time = System.currentTimeMillis() - time;
+28             if (time < interval) {
+29                 try {
+30                     Thread.sleep((long) (interval - time));
+31                 } catch (InterruptedException e) {
+32                 }
+33             } else {
+34                 try {
+35                     Thread.sleep((long) (interval - (interval % time)));
+36                 } catch (InterruptedException e) {
+37                 }
+38             }
+39         }
+40     }
+41 }
```

**Script 4.25: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
3     ...
4
5     public GameLoop(Platform platform) {
6         this.platform = platform;
7         frameRate = 60;
8         frameRate = 10;
9         interval = 1000.0f / frameRate;
10        running = true;
11    }
12    private void update(Character character) {
13        ...
14        if (platform.getKeys().isPressed(character.getLeftKey()) && platform.
15            getKeys().isPressed(character.getRightKey()) ) {
16            character.stop();
17        }
18        if (platform.getKeys().isPressed(character.getUpKey())) {
19            character.jump();
20        }
21        character.moveY();
22    }
23    private void checkCollisions(Character character) {
24        character.checkReachGameWall();
25        character.checkReachHighest();
26        character.checkReachFloor();
27    }
28    private void paint(Character character) {
29        character.repaint();
30    }
31    @Override
32    public void run() {
33        while (running) {
34            float time = System.currentTimeMillis();
35            update(platform.getCharacter());
36            checkCollisions(platform.getCharacter());
37            paint(platform.getCharacter());
38            time = System.currentTimeMillis() - time;
39            ...
40        }
41    }
42 }
```

**Script 4.26: /src/model/Character.java**

```
1  //Imports are omitted
2  public class Character extends Pane {
    ...
18     int yVelocity = 3;
+19     boolean isMoveLeft = false;
+20     boolean isMoveRight = false;
21     boolean isFalling = true;
    ...
36     public void moveLeft() {
-37         x = x-xVelocity;
+38         isMoveLeft = true;
+39         isMoveRight = false;
40     }
41     public void moveRight() {
-42         x = x+xVelocity;
+43         isMoveLeft = true;
+44         isMoveRight = false;
45     }
-46     public void stop() { xVelocity = 0; }
+47     public void stop() {
+48         isMoveLeft = false;
+49         isMoveRight = false;
+50     }
+51     public void moveX() {
+52         setTranslateX(x);
+53         if(isMoveLeft) { x = x - xVelocity; }
+54         if(isMoveRight) { x = x + xVelocity; }
+55     }
56     public void moveY() {
+57         setTranslateY(y);
58         if(isFalling) { y = y + yVelocity; }
59         else if(isJumping) { y = y - yVelocity; }
60     }
    ...
84     public void repaint() {
-85         setTranslateX(x);
-86         setTranslateY(y);
+87         moveX();
+88         moveY();
89     }
    ...
}
```



**Script 4.27: /src/controller/Launcher.java**

```
1 //Imports are omitted
2 public class Launcher extends Application {
3     public static void main(String[] args) { launch(args); }
4     @Override
5     public void start(Stage primaryStage) {
6         Platform platform = new Platform();
7         GameLoop gameLoop = new GameLoop(platform);
+ 8         DrawingLoop drawingLoop = new DrawingLoop(platform);
          ...
20         (new Thread(gameLoop)).start();
+ 21         (new Thread(drawingLoop)).start();
          ...
```

### 4.3.5 Interpolation

Interpolation is a technique that makes any game objects movement more realistic. Suppose that we closely examine the sprite sheet, i.e., the file name `MarioSheet.png` in the assets folder, we will see that if we create a rectangle image view and use it to show one Mario character image at a time, and let the image view interpolates between images fast enough, we will see the Mario character moving realistically. This is how animation works. Let us begin with a new class named `AnimatedSprite` in which its implementation is illustrated in Script 4.28. The essential methods of the class are `tick` and `interpolate` that utilizes a viewport to, one at a time, select a subfigure to render in the application, where the alternation is made once per clock tick. The `curIndex` variable indicates the image sequence. The `columns` and `rows` variables represent the number of columns and rows of the sprite sheet.

Since the functionality of the `AnimatedSprite` directly interacts with the character image, the `Character` and `platform` class also need another update to utilize the interpolation functionality. Script 4.29 and Script 4.30 illustrate the associated changes due to the interpolation functionality.

**Script 4.28: /src/model/AnimatedSprite.java**

```
+ 1 //Imports are omitted
+ 2 public class AnimatedSprite extends ImageView {
+ 3     int count, columns, rows, offsetX, offsetY, width, height, curIndex,
+ 4         curColumnIndex = 0, curRowIndex = 0;
+ 5     public AnimatedSprite(Image image, int count, int columns, int offsetX, int
+ 6         offsetY, int width, int height) {
+ 7         this.setImage(image);
+ 8         this.count = count;
+ 9         this.columns = columns;
+10         this.rows = rows;
+11         this.offsetX = offsetX;
+12         this.offsetY = offsetY;
+13         this.width = width;
+14         this.height = height;
+15         this.setViewport(new Rectangle2D(offsetX, offsetY, width, height));
+16     }
+17     public void tick() {
+18         curColumnIndex = curIndex % columns;
+19         curRowIndex = curIndex / columns;
+20         curIndex = (curIndex+1) / (columns * rows);
+21         interpolate();
+22     }
+23     protected void interpolate() {
+24         final int x = curColumnIndex * width + offsetX;
+25         final int y = curRowIndex * height + offsetY;
+26         this.setViewport(new Rectangle2D(x, y, width, height));
+27     }
+28 }
```

**Script 4.29: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
- 3     public static final int CHARACTER_WIDTH = 64;
+ 4     public static final int CHARACTER_WIDTH = 32;
5     public static final int CHARACTER_HEIGHT = 64;
6     private Image characterImg;
- 7     private ImageView imageView;
+ 8     private AnimatedSprite imageView;
    ...
- 28     public Character(int x, int y, KeyCode leftKey, KeyCode rightKey, KeyCode
        upKey) {
+ 29     public Character(int x, int y, int offsetX, int offsetY, KeyCode leftKey,
        KeyCode rightKey, KeyCode upKey) {
30         this.x = x;
31         this.y = y;
32         this.setTranslateX(x);
33         this.setTranslateY(y);
- 34         this.characterImg = new Image(getClass().getResourceAsStream("/assets/
            StillMario.png"));
+ 35         this.characterImg = new Image(getClass().getResourceAsStream("/assets/
            MarioSheet.png"));
- 36         this.imageView = new ImageView(characterImg);
+ 37         this.imageView = new AnimatedSprite(characterImg,4,4,1,offsetX,offsetY
            ,16,32);
38         this.imageView.setFitWidth(CHARACTER_WIDTH);
39         this.imageView.setFitHeight(CHARACTER_HEIGHT);
40         this.leftKey = leftKey;
41         this.rightKey = rightKey;
42         this.upKey = upKey;
43         this.getChildren().addAll(this.imageView);
44     }
    ...
```

**Script 4.30: /src/view/Platform.java**

```
1 //Imports are omitted
2 public class Platform extends Pane {
    ...
13 public Platform()
    ...
18 backgroundImg.setFitWidth(WIDTH);
- 19 character = new Character(30, 30, KeyCode.A, KeyCode.D, KeyCode.W);
+ 20 character = new Character(30, 30, 0, 0, KeyCode.A, KeyCode.D, KeyCode.W);
21 getChildren().addAll(backgroundImg, character);
22 }
    ...
```

After updating the model and view components, it is time to update the controller component. Since we have done everything by means of modularity maximization, only updating the update method of the GameLoop class is sufficiently enough for the controller. Particularly, Script 4.31 presents the latest update of the GameLoop class.

**Script 4.31: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
    ...
16 private void update(Character character) {
    ...
+ 28 if (platform.getKeys().isPressed(character.getLeftKey()) || platform.
    getKeys().isPressed(character.getRightKey())) {
+ 29 character.getImageView().tick();
+ 30 }
31 if (platform.getKeys().isPressed(character.getUpKey())) {
32 character.jump();
33 }
34 }
    ...
```

#### 4.3.6 Acceleration

We shall now see that the Mario character run more smoothly due to the implementation of the interpolation functionality. Overall, we may see that this platformer game has significantly improved its look and feel then that of the first time we launched the application. Anyhow, as we may still have a concern. The way the character run and jump is still a long way from what we can call realistic movement. One of the main problems is that we still let the character move at a constant velocity. For the next update, we will let all the movement be subject to the acceleration where that on the  $x$  axis will be subject to the input force, and that on the  $y$  axis will be subject to both input force and the gravity. Let us try to implement a simplified acceleration and examine how much it can make the character movement more realistic than based only on the velocity.

Owing to the design that maximizes the modularity, the implementation of acceleration only requires an update of the Character class. Anyhow, due to the problem of space, the current update of the Character class is spitted into two scripts, Script 4.32 and Script 4.33, where Script 4.32 presents the changes of the variable declaration and assignment, and the updated `moveX` and `moveY` methods. Then, the updates in the `jump`, `checkReachHighest`, and `checkReachFloor` methods are presented in Script 4.33. Basically, these changes implement the projectile motion principle by implementing the acceleration in both  $x$  and  $y$  axes. The key differences between the motion on the two axes are that on the  $x$  axis, the movement starts with its velocity equal to 0. It keeps increasing until it reaches the maximum possible velocity. On the other hand, the movement on the  $y$  axis starts with the maximum possible velocity. Then it continually reduces until it reaches zero, which will make the object be at the maximum  $y$  position. Then it will start falling with increased velocity until it reaches the virtual ground.

**Script 4.32: /src/model/Character.java**

```
1  //Imports are omitted
2  public class Character extends Pane {
    ...
- 17  int xVelocity = 5;
- 18  int yVelocity = 3;
+ 19  int xVelocity = 0;
+ 20  int yVelocity = 0;
+ 21  int xAcceleration = 1;
+ 22  int yAcceleration = 1;
+ 23  int xMaxVelocity = 7;
+ 24  int yMaxVelocity = 17;
- 25  int highestJump = 100;
    ...
51  public void moveX() {
52      if(isMoveLeft) {
+ 53          xVelocity = xVelocity>=xMaxVelocity? xMaxVelocity : xVelocity+xAcceleration;
54          x = x - xVelocity;
55      }
56      if(isMoveRight) {
+ 57          xVelocity = xVelocity>=xMaxVelocity? xMaxVelocity : xVelocity+xAcceleration;
58          x = x + xVelocity;
59      }
60  }
61  public void moveY() {
62      setTranslateY(y);
63      if(falling) {
+ 64          yVelocity = yVelocity >= yMaxVelocity? yMaxVelocity : yVelocity+yAcceleration;
65          y = y + yVelocity;
66      }
67      else if(jumping) {
+ 68          yVelocity = yVelocity <= 0 ? 0 : yVelocity-yAcceleration;
69          y = y - yVelocity;
70      }
71  }
    ...
}
```

**Script 4.33: /src/model/Character.java**

```
1  //Imports are omitted
2  public class Character extends Pane {
    ...
72  public void jump() {
73      if (canJump) {
74          canJump = false;
75          isJumping = true;
76          isFalling = false;
- 77          yVelocity = 5;
+ 78          yVelocity = yMaxVelocity;
79      }
80  }
81  public void checkReachHighest() {
- 82      if(isJumping && y <= Platform.GROUND - CHARACTER_HEIGHT - highestJump) {
+ 83      if(isJumping && yVelocity <= 0) {
84          isJumping = false;
85          isFalling = true;
+ 86          yVelocity = 0;
87      }
88  }
    ...
100 public void checkReachFloor() {
101     if(falling && y >= Platform.GROUND - CHARACTER_HEIGHT) {
102         isFalling = false;
103         canJump = true;
+ 104         yVelocity = 0;
105     }
106 }
    ...
```

### 4.3.7 Logging

We must have been intensely using the debugging to make us here. Now, it is time to examine some alternative techniques, which are probing and logging. Script 4.34 and Script 4.35 show an example of the probing technique on the `Character` and the `GameLoop` classes, respectively. We added one method named `trace` to the `Character` class to print the current coordinate the object and the velocity in both axes on the terminal. For the `GameLoop` class, we added code fragments that calls the `trace` method to observe the response generated from by the `trace` method when we move the game character to the left or the right.

#### Script 4.34: `/src/model/Character.java`

```
1 //Imports are omitted
2 public class Character extends Pane {
    ...
+ 120     public void trace() {
+ 121         System.out.println(String.format("x:%d y:%d vx:%d vy:%d",x,y,xVelocity,
+ 122                                         yVelocity));
123     }
```

Alternatively, probing is widely criticized for being no longer used, and it is suggested to be replaced its utilization anywhere in the code with logging. In general, logging is used as a technique that allows us to observe particular variables for tracing the program state or troubleshooting. To start playing with a Java logger, let us download and integrate the `Slf4j` library with our project. Note that at the time of writing, the latest stable version of the `Slf4j` library is 1.7.30. The bare minimum requirement for the `Slf4j` library requires two files named `slf4j-api-1.7.30.jar` (2019) and `slf4j-simple-1.7.30.jar` (2019). After we download the two files, we have to include them as the project's external libraries. Then, we shall replace the

`slf4j-api-1.7.30.jar` (2019) Retrieved July, 2020, from <https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.30/slf4j-api-1.7.30.jar>  
`slf4j-simple-1.7.30.jar` (2019) Retrieved July, 2020, from <https://repo1.maven.org/maven2/org/slf4j/slf4j-simple/1.7.30/slf4j-simple-1.7.30.jar>



**Script 4.35: /src/controller/GameLoop.java**

```
1 //Imports are omitted
2 public class GameLoop implements Runnable {
3     ...
20 private void update(Character character) {
21     if (platform.getKeys().isPressed(character.getLeftKey())) {
22         character.setScaleX(-1);
23         character.moveLeft();
+ 24     platform.getCharacter().trace();
25     }
26     if (platform.getKeys().isPressed(character.getRightKey())) {
27         character.setScaleX(1);
28         character.moveRight();
+ 29     platform.getCharacter().trace();
30     }
31     ...
}
```

code fragments to implement the probing technique with this logger. Script 4.36 instructs the only changes to be made in the Character class, where the logger instance is initiated at Line 3 of the script, and its usage replaces the use of the probing technique at Line 121 and Line 122 of the script.

**Script 4.36: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
+ 3     Logger logger = LoggerFactory.getLogger(Character.class);
4     ...
120 public void trace() {
- 121     System.out.println(String.format("x:%d y:%d vx:%d vy:%d",x,y,xVelocity,
122                                     yVelocity));
+ 122     logger.info("x:{} y:{} vx:{} vy:{}",x,y,xVelocity,yVelocity);
123 }
124 }
```

To control the logging behavior with a better-detailed configuration, we integrate the `Slf4j` library with the other logging framework named `Log4j`. The integration starts with downloading the essential assets, `apache-log4j-2.13.3-bin.zip` (2020). Note that the version 2.13.3 is the latest stable version at the time of writing. Once the file is downloaded and extracted, we have to include the files `log4j-core-2.13.3.jar`, `log4j-api-2.13.3.jar`, and `log4j-slf4j-impl-2.13.3.jar` to the dependencies list of our project. We also have to remove `slf4j-simple-1.7.30.jar` from the dependencies to avoid conflicts with the recently added `Log4j` library. After the external libraries are completely setup, let us create a configuration file named `log4j2.properties` in the project's `src` folder. The content of the file is as presented in Script 4.37. This configuration lets the appender be the only console, and the root log level is equal to `INFO`. The pattern to be presented lists the log level, the date format from year to a millisecond, the thread name, the class name, the log message, and the line break. After we compile and run the application, we should see the changes in the log pattern. With this improved flexibility and modularity promoted by the `Log4j` library, we can simply change the log pattern, such as presenting in Script 4.38.

**Script 4.37: log4j2.properties**

```
+ 1 name = PropertiesConfig
+ 2 property.filename = logs
+ 3 appenders = console
+ 4
+ 5 appender.console.type = Console
+ 6 appender.console.name = STDOUT
+ 7 appender.console.layout.type = PatternLayout
+ 8 appender.console.layout.pattern = [%5p] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} -
    %msg%n
+ 9
+ 10 rootLogger.level = info
+ 11 rootLogger.appenderRefs = stdout
+ 12 rootLogger.appenderRef.stdout.ref = STDOUT
```

---

`apache-log4j-2.13.3-bin.zip` (2020) Retrieved July, 2020, from <https://www.apache.org/dyn/closer.lua/logging/log4j/2.13.3/apache-log4j-2.13.3-bin.zip>

**Script 4.38: log4j2.properties**

```
1 name = PropertiesConfig
2 property.filename = logs
3 appenders = console
4
5 appender.console.type = Console
6 appender.console.name = STDOUT
7 appender.console.layout.type = PatternLayout
- 8 appender.console.layout.pattern = [%5p] %d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %c{1} -
    %msg%n
+ 9 appender.console.layout.pattern = [%5p] %d{yyyy-MM-dd HH:mm:ss} [time %r millisec
    ] [%t] %c{1} - %msg%n
...

```

**4.3.8 Exercise**

1. Add one more character to the application. Let the keyboard keys *left*, *up*, and *right* control its movement.
2. Let the newly added character has a different movement speed for both x and y axes.
3. Interpolate the newly added character with the Megaman sprite sheet presented earlier in this chapter.
4. Use the *FileAppender* in parallel with the *ConsoleAppender* to store the log entries in a file.
5. Modify the *PatternLayout* of the log message to add the line number, the method name, and change the date format to show the day of the week and the information shown at the current log pattern version.

# CHAPTER 5

## Unit test

**S**OFTWARE testing is one of the most crucial software processes that ensure the software under development can be delivered with high quality. Basically, testing is carried out to evaluate and verify that a software product or application does what it is required to do. The benefits of testing include preventing bugs, reducing maintenance costs, and improving the software's overall performance.

Software testing is to examine if the software can perform without any mistakes after it is intentionally encountered a situation that is likely to make it failed. Such a situation is introduced to the system in terms of test cases, carefully designed to observe the software's behavior after executing it. In general, test cases are designed to cover all the possible problems during its production state. Common software problems are, such as incorrect calculation, incorrect data edits and ineffective data edits, incorrect matching and merging of data, data searches that yield incorrect results, incorrect processing of data relationship, incorrect coding and implementation of business rules, inadequate software performance, and many more.

## 5.1 Basic software testing

According to Kaner (2009), Software testing is defined as an empirical technical investigation conducted to provide stakeholders with information about the product's quality or service under test. There are three keywords in the statement and all of which have a broad meaning. Empirical means to derive from experiment, experience, and observation. Technical means of having a particular skill or practical knowledge, and finally, investigation means a detailed inquiry or systematic examination. In sum, based on evidence, software testing requires particular technical skills and practical knowledge to provide important assessments of the quality of a software project.

### 5.1.1 Why is software testing important?

As a software application gets more features and supports more platforms, it becomes increasingly difficult to make it bug-free or defect-free. A bug or defect in any form can degrade the software and worst cause serious malfunctions. There are numerous severe cases in the history ranging from losing a vast amount of profits to injuries or fatalities. For example, five nuclear power plants in the United States were temporarily shut down because of a fault in the simulation program used to design a nuclear reactor to withstand earthquakes (Lin 1985). The cause of the problem was the use of an arithmetic sum of a set of numbers instead of their absolute values. An F-18 fighting aircraft has been crashed because of a missing condition in the control flow, i.e., if ... then ... without the else clause that was thought could not possibly arise (Neumann 1986). A spacecraft named Ariane 5 was exploded at its launch due to an integer overflow problem (Le 1997). For a

---

Kaner C (2009) Software Testing as a Quality Improvement Activity. *IEEE Computer Society Webinar Series*, 1-86.

Lin H (1985) The development of software for ballistic-missile defense. *Scientific American*, 253(6), 46-53.

Neumann PG (1986) Risks to the public in computer systems. *ACM SIGSOFT Softw. Eng. Notes*, 11(2), 2-14.

Le LG (1997) An analysis of the Ariane 5 flight 501 failure-a system engineering perspective *In Proc. of Intl. Conf. and Workshop on Engineering of Computer-Based Systems*, 339-346.

more recent case, Toyota has to recall over 4 million cars due to accidents resulting from the unintended and uncontrolled acceleration of its cars (Austen-Smith et al. 2017). From these cases, it can be said that a more proper test can prevent them all from occurring. It has been a kind of long debate on how much effort and cost should be spent on testing. A vague but very valid answer is that it depends on how much the companies can save if the worst defect or fault is later introduced, or in short, it depends on the risk to impact ratio. Some consequences may be too hard to accept, such as frustrated and lost customers up to brand's reputation.

There is an inconvenient truth about software testing stating that *Bugs found later cost more to fix*. That is, it is observed that the cost to fix a bug increases exponentially from a software development phase to the following phase. For instance, suppose a software requirement bug can be fixed within one person-hour at the requirement phase slip until it is completely built. The cost to fix it will include all the cost wasted for implementing it plus the cost required to develop software according to the fixed requirement. What can be considered waste includes development time, meeting time, testing time, bug fixing time, and many more. Thus, this is why it can be roughly said that the cost to fix a bug increases exponentially from a software development phase to the following phase.

### 5.1.2 What can be the sources of bugs?

We have already acknowledged that the earlier we can capture and fix a bug, the lower the cost is required to fix it. According to a survey study by Perry and Steig (1993), the bug occurs at the coding and testing phases are roughly accounted for only 50 percent of the total population. This means that earlier phase bugs are not less significant in practice, but its impact is much greater if it slips to a later phase.

Starting from the requirement phase, bugs introduced in this phase include problems originating during the requirements specification phase of development. Mostly, they are erroneous, incomplete, and inconsistent requirements. All of which come from inadequate carefulness spent at the time of requirement gathering.

---

Austen-Smith D, Diermeier D, Zemel E (2017) Unintended acceleration: Toyota's recall crisis. *Kellogg School of Management Cases.*, 1-16.

Perry DE, Stieg CS (1993) Software faults in evolving a large, real-time system: a case study. *In proc. of the European Software Engineering Conference*, 48-67.

ering and requirement elicitation. For the design phase, the problem comes from the architectural and design phases of development. A famous case of the design phase flaw is that researchers have found a fundamental flaw in Bluetooth-enabled devices that the way it communicates to applications makes it vulnerable to hacking (Zuo et al., 2019). For coding and testing phases, the problems originate during the coding phases and those originating in the construction or provision of the testing environment, respectively.

### 5.1.3 Common test methods in different software development phases

The V-Model (Java T Point, 2020), shown in Figure 5.1, is one of the most well-known software testing graphical explanation diagrams. From the figure, there are two main phases described as the verification phase and validation phase. Verification checks whether specified requirements meet. Mainly we carry out various static tests to ensure the correctness of the software components. On the other hand, validation checks the executing code to see whether the software meets the customer expectation and requirement. In brief, we may have heard that the question to ask when carrying out verification is *Are we building the product right?* and that of during validation is *Are we developing the right product?* The items on the V model's left-hand side present the product or activity determining each primary software development phase, such as requirement specifications and designs. The right-hand side names the number of proper test methods for each activity on the model's left-hand side. Some explanation for the entries on the left-hand side of the model as follows:

- **Business requirement specification** attempts to completely synchronize the understanding of the product requirements between the software developer teams and the customers. In this phase, clear communication is mostly the leading focus to ensure that the development teams have fully understood the customer's exceptions and the exact requirements.

---

Zuo C, Wen H, Lin Z, Zhang Y (2019) Automatic fingerprinting of vulnerable iot devices with static uuids from mobile apps. *In Proc. of the ACM SIGSAC Conf. on Computer and Communications Security*, 1469-1483.

Java T Point (2020) V-Model. Retrieved June, 2020, from <https://www.javatpoint.com/software-engineering-v-model>

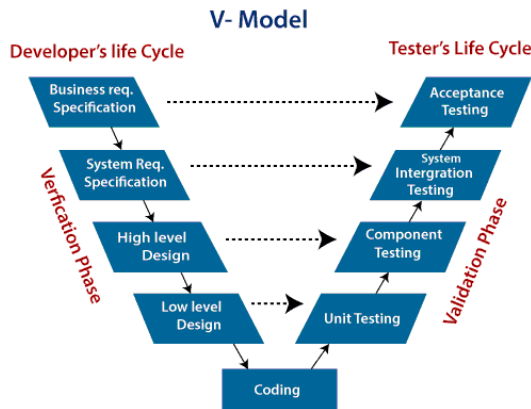


Figure 5.1: The V model of software testing (Java T Point, 2020)

- **System requirement specification** attempts to describe the features and the behaviors of the system or the software application to be developed. The list must completely define the intended functionality required by the customers to satisfy their software users.
- **High level design** is the overall system design. It covers everything at the level of system architecture and database. In terms of programming logic, high-level design describes the relationship between different modules, components, and system functions. Furthermore, in terms of data, high-level design describes data flow, data structures, and the system's database.
- **Low level design** can be viewed as a detailed high-level design, where actual programming for every software application component is defined. Thus, class diagram defining variables and methods are designed in this phase.
- **Coding** is all about implementation up to building the system or software application.



The right-hand side of the *V-model* presents the Validation phase where all the items are the name of effective test methods used in software development:

- **Unit testing** is a test method developed during the low-level design phase to eliminate or unit level errors. In general, a unit is the smallest entity that can independently exist in a software application. Thus, unit levels are commonly referred to as method levels, and tests in these levels are commonly checked to see whether the parameters being passed between methods behaved as correctly as what they are intended to do. In sum, unit testing verifies that the smallest entity of the software application can function correctly when isolated from the other parts of the application.
- **Component testing** is also known as integration testing. It is the test planned to validate the correctness of the product generated at the high-level design phase. Testing at this level validates whether the integration of several units of the same type or that of the same use case still perform correctly as what they are intended to do.
- **System testing** attempts to ensure that expectations from an application developer are met. This testing is carried out by the customer's business team to see whether the system or the software application can fulfill their requirements. The test covers document checking up to testing the actual software.
- **Acceptance testing** has the same goals as that of *System testing* but the test is carried out through the real users or a user group sampling from the real users. Not only functional problems *Acceptance testing* can address, but it can also reveal the compatibility problems with the different systems, which is available within the user atmosphere. Somehow, it can also discover non-functional problems such as performance and scalability problems as the devices the users use in the real world are not the same as that used to develop the software application.

#### 5.1.4 Testing objectives

*To find bugs as early in the software development processes as possible and ensure all of the found bugs are fixed* are the ultimate software testing goals in general. To achieve them, a software tester often has to design and create test cases, which are programs isolated from the main application. These programs are designed and executed with the intent of (1) finding errors and bugs, (2) checking whether the system meets the requirements and it can be executed successfully in the Intended environment, (3) examining whether the system is already *fit for its purpose*, and (4) ensuring that the system does what it is expected to do. The following list is the classified common testing objectives:

- **Alpha testing** is to let software developers test the built software application in the real-world environment to ensure whether the software application can perform without any mistake after its launch.
- **Beta testing** is almost identical to *Alpha testing* with the only difference that **Beta testing** is carried out by volunteers, who commonly do not have any background on software development.
- **Conformance testing** is to test whether the software's behavior is the same as the plan or not.
- **Performance testing** examines whether the software's response rate under different situations is acceptable by the user or not.
- **Recovery testing** examines whether the software application can recover itself after an unexpected system fault or system shutdown. This test is mostly performed in the system intended to deploy in difficult situations, such as disasters.
- **Stress testing** exerts full loads on the software application to determine the level of load in which the software application will not perform anymore.
- **Usability testing** quantifies how simple and easy the software application can be used to improve the usability level continually.

## 5.2 Unit test

This course will focus only on *unit testing* as it is the fundamental of other testing methods. To explain what a software *unit testing* is, the lecturer sees that it is much simpler to explain as if a software application is an electronic device. For instance, suppose that we can only hear the sound from one channel of a stereo system, and we want to examine which part of the stereo system is causing the problem. What we can do is isolate each significant component of the system, i.e., the speakers, the amplifier, and the source, and probe each component systematically. The probing at this level is analogous to a software *integration testing*. Then, suppose that, we suspect that the amplifier is the most likely the broken component, we may use a multimeter to probe the voltage between the two ends of each particular part to find which part does not behave as what it should be. A test in this way might take time, but it can potentially help us gradually isolate the healthy parts from the malfunctions. The probing at this fine-grain level is analogous to *unit testing*.

Transferring the method used in probing the physical electronic devices to a software application, software *unit testing* is a piece of code that exercises a small and specific area of functionality, e.g., a particular method in a particular context. Hence, if the test can indicate that a particular code fragment works as we expected, we can proceed to assemble and test the entire system later without any hesitation that the code we tested will behave differently from our expectations. Besides, one of the biggest challenges in isolating each part when carrying out *unit testing* is that after parts are disassembled and isolated, there are still possible dependencies, such as files and database entries, used in multiple methods of the same class. This is similar to how we probe the amplifier, where both the source's signal and the power current still run between the device components. Typically, a tester will have to create a dummy object that substitutes and imitates a real object within a testing environment. For example, when we test a method running an `HttpRequest` to retrieve data from an external API, we may be willing to check if the tests execute predictably or not. Hence, it is much more convenient to set up a controlled environment by running a dummy request that allows us to entirely modify the request components and observe its consequences in a more predictable way.

### 5.2.1 Objective

A well-designed *unit testing* allows us to code with more confidence. We may all know that the larger the software size, the higher possibility a bug will show up. Such increased probability is due to both a larger number of functionalities and the interfaces wiring each method together. Furthermore, we should be able to acknowledge a bug's presence and localize it as soon as possible. If we design the test properly, we are likely to plan ahead for the *unit test* methods when we are carrying out the low-level design. In this case, the total number of bugs introduced during coding phases and later should be lessened. This inherently helps us lessen the number of difficult debugging problems we have to carry out.

### 5.2.2 JUnit

JUnit (2020) is the most commonly used framework to undertake *unit testing* in Java. Many toolsets are already made available in its library to help a tester to perform unit testing systematically and smoothly. Basically, for a given class `Foo`, we create a class `FooTest` to test the `Foo` class, and this `FooTest` class contains various test cases to test its methods at the unit level. Each method looks for particular results and determines whether it passes or fails. JUnit provides an `assert` statement to control the way we write the *unit test* codes. The idea behind using an `assert` statement is that we put an assertion call in the body of our test method with a condition we expect it to be true. For instance, if we expect that an empty `ArrayList` size should be equal to two after we add two list members to it. So, we add this size condition to the `assert` method and examine it. If what we expected to be true does not hold the boolean value of true, the test will fail.

Script 5.1 illustrates an example JUnit test class structure. JUnit uses annotations, e.g., `@Test`, to identify test methods. When a method with an `@Test` annotation is executed under test, the JUnit framework will execute all these methods and examine all the `assert` methods inside them.

---

JUnit (2020) JUnit. Retrieved June, 2020, from <https://junit.org/junit4/>

Script 5.1: An example template of a JUnit test class

nameTest.java

```
+ 1 import org.junit.*;
+ 2 import static org.junit.Assert.*;
+ 3 public class nameTest {
+ 4     @Test
+ 5     public void name() {
+ 6         // a test case method ...
+ 7     }
+ 8 }
```

### Assert methods

The assert methods are a family of static methods that JUnit has provided its user to test for specific conditions. These methods' name typically starts with the keyword `assert`, followed by the condition it is meant to test. Under test, an assert method will compare the actual value being stored in the variable we passed to the method and the predefined expected value. The comparison result will determine whether the test is passed or failed. Particularly, if the test condition is not met, the method will say that the test case is failed by throwing an `AssertionException` back to the method caller. The followings are the list of the available assertion methods:

- **`assertTrue("message", condition)`**: The test will fail if the boolean value evaluated from the condition statement is false. If fail, the string passed as message will be shown. Note that the message parameter can also be omitted.
- **`assertFalse("message", condition)`**: The test will fail if the boolean value evaluated from the condition statement is true.
- **`assertEquals("message", expected, actual, tolerance)`**: The test will fail if the expected value and the actual value are not equal. The tolerance parameter is the number of decimal points in which the two values must be the same. Note that the tolerance parameter can also be omitted.
- **`assertSame("message", expected, actual)`**: The test will fail if the two variables refer to different objects.

- **assertNotSame("message", expected, actual):** The test will fail if the two variables refer to the same objects.
- **assertNull("message", object):** The test will fail if the object is not null.
- **assertNotNull("message", object):** The test will fail if the object is null.
- **fail("message"):** To let the current test to fail immediately. This is a particularly unique tool used to check that a specific part of the code is not reached or flag a failing test before the test code is implemented.

Script 5.2: A JUnit test class with two test methods

ArrayListMethodTest.java

```
+ 1 import org.junit.*;
+ 2 import static org.junit.Assert.*;
+ 3 public class ArrayListMethodTest {
+ 4     @Test
+ 5     public void memberShouldBeRetrievedInOrder() {
+ 6         ArrayList list = new ArrayList();
+ 7         list.add(42);
+ 8         list.add(-3);
+ 9         list.add(15);
+10         assertEquals(42, list.get(0));
+11         assertEquals(-3, list.get(1));
+12         assertEquals(15, list.get(2));
+13     }
+14     @Test
+15     public void listShouldBeEmpty() {
+16         ArrayList list = new ArrayList();
+17         list.add(123);
+18         list.remove(0);
+19         assertTrue(list.isEmpty());
+20     }
+21 }
```

Script 5.2 illustrates an example JUnit test class with two test methods. The first test method, named `memberShouldBeRetrievedInOrder` presented at Line 5 of the script, has three assert statements. Before it tests any assertions, it pre-allocated an `ArrayList` of type `integer` and added three integer values to the array. The first assertion statement, presented at Line 10, tests whether the first value retrieved from the `ArrayList` by using the `get` will be a value equal to 42 or not. The other two assertion methods also test for equal conditions for the value -3 and 15, respectively. The other test method in this class test whether its `ArrayList` is empty after all its values have been retrieved.

### 5.2.3 JUnit naming conventions

JUnit has potential naming conventions. A widely-used naming method for test classes is to let the keyword `Test` be the class's postfix to be tested. For example, the class name of the test class in Script 5.3 is `DateObjectTest`. For each test case, as a general rule, a test case name should be self-explained as anyone should understand what the test case is attempting to do when she or he reads the test case name for the first time. Unlike the naming conventions used in naming other Java methods, it is widely suggested to give the test case methods very long descriptive names. Three commonly seen conventions are:

- to give a simple but really long method name, such as `newArrayListsHaveNoElement`;
- to use the word “should” in the test method name, such as the `memberShouldBeRetrievedInOrder` method shown in Script 5.2;
- to name all the test methods in the form “Given[ExplainTheInput]When[WhatIsDone]Then[WhatIsTheExpectedResult]”, such as the method named `GivenDateObjectWhenDayIsAddedThenDateShouldBeAdvanced` presented in Script 5.3.

All these naming conventions can provide a hint of what should happen if the test method is executed. Thus, any testers are no longer needed to read the actual implementation.

### 5.2.4 JUnit structure conventions

Alongside the naming conventions, JUnit also has some conventions for implementing assertion methods in JUnit test codes. All of these are for better code readability, which promotes better maintainability in the long term. For example, it is widely suggested to put the expected values on the left side of the parameter list of any assertion methods, such as `assertEquals(2020, d.getYear());`; is more suggested over `assertEquals(d.getYear(), 2050);`. Furthermore, all the implemented assertion methods are widely recommended to have messages explaining what is being checked. For the case when both expected values and the actual values are not single values, it is suggested to let the two be an object, and we let the assertion method to test it compare them as objects. Script 5.3 provides a summary of these conventions.

**Script 5.3: A well-structured JUnit test class****DateObjectTest.java**

```
+ 1 import org.junit.*;
+ 2 import static org.junit.Assert.*;
+ 3 public class DateObjectTest {
+ 4     @Test
+ 5     public void GivenDateObjectWhenDayIsAddedThenDateShouldBeAdvanced() {
+ 6         Date d = new Date(2020, 1, 15);
+ 7         d.addDays(14);
+ 8         Date expected = new Date(2020, 1, 29);
+ 9         assertEquals("date after +14 days", expected, d);
+10     }
+11 }
```

### 5.2.5 Unit testing tips

In fact, we cannot test every possible input or input parameter value, especially when the input is a real number. So, we have to think of a limited set of test cases that potentially expose the largest number of bugs. For example, Script 5.4 illustrates a test class that checks for all the possible cases; in other words, it has many redundant assertion checks. In other words, it has many redundant assertion



checks. A standard solution to address this problem is to think about boundary cases. For example, if the range of the numbers to be tested can be both negative and positive values, the boundary cases which are more likely to introduce bugs than others are the minimum, -1, 0, 1, and the maximum because these numbers tend to correspond to more distinct behaviors. Therefore, instead of testing for numerous possible numbers, perhaps a reduced set of the minimum, -1, 0, 1, and the maximum are sufficiently enough for this case. Other cases that are likely to introduce bugs are the edge of an array or a collection. This is a typical case for one who frequently switches between programming languages, in which some languages the array indexing starts at 1, not 0, like many others. Besides, the variation of the representation of empty and nonexisting values is also the case that frequently introduces bugs, for example, 0, -1, null, nan, none, or an empty list or array are all used by different developers to represent emptiness or a not yet assigned value. For the case illustrated in Script 5.4, it must be more than enough to test the dates at the beginning of some months and some dates at the end of the other months.

Another commonly suggested tip is about the trustworthiness of a particular test case. To say whether a test case is self-contained or not, a big test case is suggested to be broken down into many small test cases, where if possible, only one assert statement per test case is considered the best practice. This ensures that one thing is tested at a time per test method. Furthermore, tests should avoid programming logics, such as an if, else, switch, and for-loop, and a try-catch-finally block. If an exception is an objective of a test case, using a particular test method for exception testing over a try-catch-finally block. This exception test method will let the test fail if the exception is not thrown. Script 5.5 illustrates an example test method test for expected errors, where Line 4 of the script lets this method test for an `ArrayIndexOutOfBoundsException`.

### 5.2.6 JUnit test suites

Generally, we have many test classes in one single application. JUnit provides a test suite tool to ease the test on many classes by let us combine all the test classes into a test suite. Running a test suite will execute all the test classes we declared in that test suite. In other words, bundling many test classes into a test suite is a simple way to run all of the application's tests at once. Scripts 5.6 illustrates an

Script 5.4: A unit test method with redundant assertion checks

DateTest.java

```
+ 1 import org.junit.*;
+ 2 import static org.junit.Assert.*;
+ 3 public class DateTest {
+ 4     // test every day of the year
+ 5     @Test(timeout = 10000)
+ 6     public void tortureTest() {
+ 7         Date date = new Date(2050, 1, 1);
+ 8         int month = 1;
+ 9         int day = 1;
+10         for (int i = 1; i < 365; i++) {
+11             date.addDays(1);
+12             if (day < DAYS_PER_MONTH[month]) {
+13                 day++;
+14             } else {
+15                 month++;
+16                 day=1;
+17             }
+18             assertEquals(new Date(2050, month, day), date);
+19         }
+20     }
+21     private static final int[] DAYS_PER_MONTH = {
+22         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
+23     };
+24 }
```

Script 5.5: An example exception testing

ArrayTest.java

```
+ 1 import org.junit.*;
+ 2 import static org.junit.Assert.*;
+ 3 public class ArrayTest {
+ 4     @Test(expected = ArrayIndexOutOfBoundsException.class)
+ 5     public void exceptionShouldBeThrownWhenTryingToAccessEmptyList() {
+ 6         ArrayIntList list = new ArrayIntList();
+ 7         list.get(4);
+ 8     }
+ 9 }
```

example JUnit test suite. This suite contains five test classes, i.e., `WeekdayTest`, `TimeTest`, and `CourseTest`. If we want to add another test class, we can add it to the `@Suite.SuiteClasses` statement.

**Script 5.6: An example test suite****AllTests.java**

```
+ 1 import org.junit.runner.*;
+ 2 import org.junit.runners.*;
+ 3 @RunWith(Suite.class)
+ 4 @Suite.SuiteClasses({
+ 5     WeekdayTest.class,
+ 6     TimeTest.class,
+ 7     CourseTest.class
+ 8 })
+ 9 public class AllTests {
+10
+11 }
```

### 5.2.7 Reflection

Reflection is an instrumental technique allowing us to examine or modify the run-time behavior of applications. Basically, we can inspect the types or properties of an object at runtime. Also, if needed, we can modify the behavior of the object as well. One of the most common use cases of reflection in Java is to test a private method, in which we cannot directly call it by the object's instance. Without reflection, the only approach to handle such a situation is to make the method public and test it. In this way, we have to change it back after the test is completed. Of course, this is not a good idea at all. In contrast, using reflection, we can simply take full control of the class and run a private method as if it is a public method. In this way, we do not need to modify anything before and after the test. Precisely, reflection enables us to do the following things at run time:

- Examine an object's class structure and behavior;
- Construct an object for a particular class;

- Examine a class's field and method;
- Invoke any method of an object;
- Change the accessibility flag of Constructor, Method, and Field.

Script 5.7 shows the use of reflection to get the class name from an object on the fly. Then, Script 5.8 presents an update of Script 5.7 and shows how we can invoke a method on an unknown object. In particular, Script 5.7 has two classes named `ReflectionExample` and `Foo`. At Line 4 of the script, a `Foo` class instance is created in the `ReflectionExample` as a variable name `f`. Then, at Line 5 of the script, the command `f.getClass().getName()` retrieves the information of the entire class of the variable `f` and then presents its name. The output will be the package name, followed by `.Foo`. For Script 5.8, the code fragments between Line 6 and Line 13 create a variable `method` that retrieved a method instance with the name `print` from the variable `f`. Since this `print` method is a private method, the code at line 9 changes its accessibility to be the same as that of a public method and calls it with an invokes the command, as presented at Line 10. In this way, we can directly call any private as if it is a public method. This technique is widely used in *unit testing*.

Script 5.7: An example usage of Java reflection

ReflectionExample.java

```
+ 1 import java.lang.reflect.Method;
+ 2 public class ReflectionExample {
+ 3     public static void main(String[] args){
+ 4         Foo f = new Foo();
+ 5         System.out.println(f.getClass().getName());
+ 6     }
+ 7 }
+ 8 class Foo {
+ 9     private void print() {
+10         System.out.println("abc");
+11     }
+12 }
```

**Script 5.8: Invoking a method using Java reflection****ReflectionExample.java**

```
+ 1 import java.lang.reflect.Method;
+ 2 public class ReflectionExample {
+ 3     public static void main(String[] args){
+ 4         Foo f = new Foo();
- 5         System.out.println(f.getClass().getName());
+ 6         Method method;
+ 7         try {
+ 8             method = f.getClass().getMethod("print", new Class<?>[0]);
+ 9             method.setAccessible(true);
+10             method.invoke(f);
+11         } catch (Exception e) {
+12             e.printStackTrace();
+13         }
+14     }
+15 }
+16 class Foo {
+17     private void print() {
+18         System.out.println("abc");
+19     }
+20 }
```

## 5.3 Test-driven development (TDD)

TDD is one of the most referenced practices in recent days. It is where we write a test before we write code. Thus, we are more likely to design and write just enough production code to pass the test, leading to a more robust implementation and a better design. In steps, TDD is performed in the following orders:

1. Broken down the tasks into small tasks and prioritize them;
2. Select a task to be completed, and write a test for the task;
3. Run all the tests to verify that the new test fails, at least the new test case;
4. Write minimal production code to complete the task;
5. Run all the tests again to verify that all the tests app this time;
6. Repeat from the first step until all the tasks are completed.

Designing and writing test cases before code requires all the team members to think ahead and consider the concrete test cases that will say the requirement is fulfilled after a particular new test case is passed. Along with the test cases, what also needs to consider includes the design of the solution, how the information will flow, the possible outputs of the code, and exceptional scenarios that might occur.

### 5.3.1 Benefits

With TDD, developers can instantly know whether the newly written production code works when its implementation is completed. This is owing to that its associated test codes are always available. Furthermore, the available test codes are not only that of the newly written production code, but also that of all the code implemented before it. These resources will enable the developers to execute a regression test anytime. Thus the developers can ensure that the newly added code does not interfere with the already implemented code. In other words, TDD encourages the developers to decompose the problem into manageable and formalized programming tasks, and it provides contests in which low-level design decisions are made by focusing on writing only the code necessary to pass tests.

Since the scope of a single test is limited, when the test fails, rework is much easier as compared with that of a fail test case of a cluttered and complicated task. Furthermore, TDD will create a growing collection of automated test cases as its byproducts. This collection can be executed anytime to verify the entire software application's correctness whenever a change is introduced to the system.

In summary, a developer team that actively follows TDD's principle can potentially achieve a better code and application quality, increased productivity, and a collection of test codes with a significantly higher level of test coverage.

## 5.4 Case study -- unit testing

There are two case studies in this chapter. One is a *unit test* showcase, and the other one is a TDD showcase. The first case study is on continuity from the previous chapter's case study, where three more functionalities are added to the application to complete it as a game. Since each of us may develop the previous chapter exercises solution differently, let us start with the same code snapshot, provided at <http://myweb.cmu.ac.th/passakorn.p/953233/materials/chapter05.zip>. This project still has the same structure as what we completed in the previous chapter as follows:

```
src
├── assets
├── controller
│   ├── DrawingLoop.java
│   ├── GameLoop.java
│   └── Launcher.java
├── model
│   ├── AnimatedSprite.java
│   ├── Character.java
│   └── Keys.java
└── view
    ├── Platform.java
    └── Score.java
```

Before starting the implementation, we have to add all the Log4j and Slf4j jar files to the project dependency list, as we did for the case study in the previous chapter. It is still highly recommended to use a debugger to troubleshoot whenever the application does not behave as expected.

After we configure the project from the provided source code and execute it, we should see the platformer game with two Mario characters falling from the top left and the top right area of the game screen, respectively. The two characters can be moved using different key sets, but the collision event between the two characters has not been implemented yet. This functionality is the first thing we are going to examine in this chapter.

### 5.4.1 Character collision

Script 5.9 presents an updated `DrawingLoop` class by adding the code fragments implementing the character collision checking. Particularly, the nested `for`-loop from Line 31 of the script examines all the two characters' state, whether the graphical representation `viewports` of the two characters are overlapped or not. Particularly, the code fragment `cA.getBoundsInParent().intersects(cB.getBoundsInParent())` at Line 34 of the script is the check condition. When the condition is met, the instance representing the two characters will consult their handling method named `collided` for the handling solution. The methods calling are presented at Line 35 and Line 36 of the script, respectively.

The `collided` method is a class method of the `Character` class, in which its implementation is illustrated in Script 5.10. This collision checking implements two sets of programming logic to handle the collision differently between the collision on the *x* axis and that of the *y* axis. These programming logics are shown between Line 125 and Line 138 of Script 5.10. Particularly, the collision detected on the *x* axis simply sends the two characters back to the coordinate right before they have collided. In other words, this handling solution does not allow the characters to run through other characters anymore. For the collision detected on the *y* axis, it checks which character is the one who stomps over the other character or the one who was stomped. Hence, the application can generate different handling responses between the two situations. For example, later in this case study, we will let the character who stomps over the other character earn the score. On the other



hand, we will render a simple dead scene for the other and respawn it at its origin coordinates.

The respawn method presented between Line 139 and Line 149 of the script creates a method to reset all the character variables set to respawn. All the other variables added to this update are the required variables required to respawn to its origin coordinates. For example, the variables `startX` and `startY` are necessary because, wherever the character is stomped it will be moving back to its origin coordinates by assigning the values of `startX` and `startY` to the variables `x` and `y`, respectively.

**Script 5.9: /src/controller/DrawingLoop.java**

```
1 //Imports are omitted
2 public class DrawingLoop implements Runnable {
3     ...
25 private void checkDrawCollisions(ArrayList<Character> characterList) {
26     for (Character character : characterList ) {
27         character.checkReachGameWall();
28         character.checkReachHighest();
29         character.checkReachFloor();
30     }
+ 31     for (Character cA : characterList) {
+ 32         for (Character cB : characterList) {
+ 33             if( cA != cB) {
+ 34                 if (cA.getBoundsInParent().intersects(cB.getBoundsInParent())) {
+ 35                     cA.collided(cB);
+ 36                     cB.collided(cA);
+ 37                     return;
+ 38                 }
+ 39             }
+ 40         }
+ 41     }
42 }
```

**Script 5.10: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
3     ...
4
5     private int x;
6     private int y;
7 + 10     private int startX;
8 + 11     private int startY;
9 + 12     private int offsetX;
10 + 13     private int offsetY;
11     ...
12
13     public Character(int x, int y, int offsetX, int offsetY, KeyCode leftKey,
14         KeyCode rightKey, KeyCode upKey) {
15 + 21         this.startX = x;
16 + 22         this.startY = y;
17 + 23         this.offsetX = offsetX;
18 + 24         this.offsetY = offsetY;
19     ...
20
21     public void collided(Character c) {
22 + 125         if (isMoveLeft) {
23 + 126             x = c.getX() + CHARACTER_WIDTH + 1;
24 + 127             stop();
25 + 128         } else if (isMoveRight) {
26 + 129             x = c.getX() - CHARACTER_WIDTH - 1;
27 + 130             stop();
28 + 131         }
29 + 132     }
30 + 133     if(y < Platform.GROUND - CHARACTER_HEIGHT) {
31 + 134         if( falling && y < c.getY()) {
32 + 135             c.respawn();
33 + 136         }
34 + 137     }
35 + 138 }
36 + 139 public void respawn() {
37 + 140     x = startX;
38 + 141     y = startY;
39 + 142     imageView.setFitWidth(CHARACTER_WIDTH);
40 + 143     imageView.setFitHeight(CHARACTER_HEIGHT);
41 + 144     isMoveLeft = false;
42 + 145     isMoveRight = false;
43 + 146     falling = true;
44 + 147     canJump = false;
45 + 148     jumping = false;
46 + 149 }
47 150 }
```

### 5.4.2 Death scene rendering

After executing the application, we will see that it has become much more like a game now. Perhaps, a few brushing up on the application will complete it. For example, in the next update, we will implement a simple animation for the death scene because it appears too fast to respawn a character after it was stomped. Script 5.11 modifies the `collided` method of the `Character` class and add one more method named `collapsed` to the class. Particularly, after a character is stomped by the other one, the character will be collapsed in the same way we see in classic Mario games. After half a second, it will respawn at its origin coordinates.

Script 5.11: `/src/model/Character.java`

```
1 //Imports are omitted
2 public class Character extends Pane {
    ...
125     public void collided(Character c) {
        ...
133         if(y < Platform.GROUND - CHARACTER_HEIGHT) {
- 134             if( falling && y < c.getY()) {
+ 135             if( falling && y < c.getY() && Math.abs(y-c.getY())<=CHARACTER_HEIGHT+1) {
+ 136                 y = Platform.GROUND - CHARACTER_HEIGHT - 5;
+ 137                 repaint();
+ 138                 c.collapsed();
139                 c.respawn();
140             }
141         }
142     }
+ 143     public void collapsed() throws InterruptedException {
+ 144         imageView.setFitHeight(5);
+ 145         y = Platform.GROUND - 5;
+ 146         repaint();
+ 147         TimeUnit.MILLISECONDS.sleep(500);
+ 148     }
    ...
}
```

### 5.4.3 Scoreboard

The last functionality to be implemented for this platformer game is the scoreboard. Each character has its score starting from zero. This score will increment by one every time after the character can stomp over the other. Unlike the implementation for the death scene in which we can complete it only by modifying the Character class, we have to add one more view class to display the two players' scores. This score will be later added to the Platform class. Script 5.12 implements the scoreboard to the app, in which it has to display the score on the application scene.

For the programming logics, Script 5.13, Script 5.14, and Script 5.15 illustrate the modifications of the Platform class, the Character class, and the the GameLoop classes, respectively. For the Platform class, the code fragments modified in Script 5.13 only add the Score class instances to the game platform. Then, the updated Character class adds a score variable as a new class variable with an initial value of 0. Then, whenever a character can stomp over the other, this score variable will increment by one. Next, the GameLoop class is the controller for this functionality. Once every clock ticks, it will synchronize the scores shown on the game scene with that of being stored to the Character instance. Once all these implementations are updated, this platformer game can be considered complete.

#### Script 5.12: /src/view/Score.java

```
+ 1 //Imports are omitted
+ 2 public class Score extends Pane {
+ 3     Label point;
+ 4     public Score(int x, int y) {
+ 5         point = new Label("");
+ 6         setTranslateX(x);
+ 7         setTranslateY(y);
+ 8         point.setFont(Font.font("Verdana", FontWeight.BOLD, 30));
+ 9         point.setTextFill(Color.web("#FFF"));
+10         getChildren().addAll(point);
+11     }
+12     public void setPoint(int score) {
+13         this.point.setText(Integer.toString(score));
+14     }
+15 }
```

**Script 5.13: /src/view/Platform.java**

```
1 //Imports are omitted
2 public class Platform extends Pane {
    ...
8     private Image platformImg;
- 9     private ArrayList<Character> characterList = new ArrayList();
+ 10     private ArrayList<Character> characterList;
+ 11     private ArrayList<Score> scoreList;
12     private Keys keys;
13     public Platform() {
14         characterList = new ArrayList<>();
+ 15         scoreList = new ArrayList();
+ 16         keys = new Keys();
    ...
25     getChildren().addAll(characterList);
+ 26     getChildren().addAll(scoreList);
    ...
}
```

**Script 5.14: /src/model/Character.java**

```
1 //Imports are omitted
2 public class Character extends Pane {
    ...
+ 20     private int score=0;
21     private KeyCode leftKey;
    ...
125     public void collided(Character c) {
        ...
133         if(y < Platform.GROUND - CHARACTER_HEIGHT) {
134             if( falling && y < c.getY() && Math.abs(y-c.getY())<=CHARACTER_HEIGHT+1) {
+ 135                 score++;
136                 y = Platform.GROUND - CHARACTER_HEIGHT - 5;
137                 repaint();
138                 c.collapsed();
139                 c.respawn();
140             }
141         }
142     }
    ...
}
```

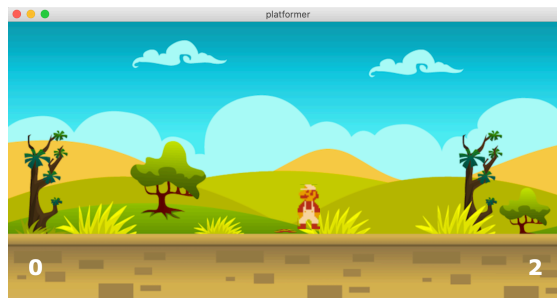
**Script 5.15: /src/controller/GameLoop.java**

```

1  //Imports are omitted
2  public class GameLoop implements Runnable {
    ...
+ 50  private void updateScore(ArrayList<Score> scoreList, ArrayList<Character>
      characterList) {
+ 51      javafx.application.Platform.runLater(() -> {
+ 52          for (int i=0 ; i<scoreList.size() ; i++) {
+ 53              scoreList.get(i).setPoint(characterList.get(i).getScore());
+ 54          }
+ 55      });
+ 56  }
57  @Override
58  public void run() {
59      while (running) {
60          float time = System.currentTimeMillis();
61          update(platform.getCharacterList());
+ 62          updateScore(platform.getScoreList(),platform.getCharacterList());
      ...

```

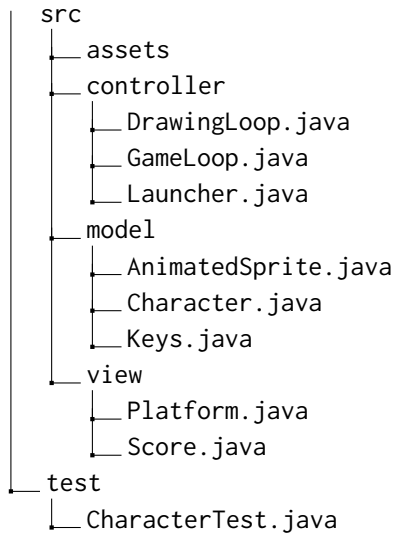
Figure 5.2 shows the final version of this platformer game case study. For anyone who is still facing an error, the possible problems are (1) the missing Getter methods, (2) the missing Exception handling methods, and (3) the missing dependency files.



**Figure 5.2: The final version of this platformer game case study.**

### 5.4.4 Unit testing

Now it is time to showcase the *unit testing* for this platformer game. First of all, let us create a file structure as given below:



For one who uses the IntelliJ Idea IDE, the test code's root folder can be annotated by right-clicking on the test folder and select Test Sources Root. Three jar dependencies are required for running JUnit in the JavaFX environment. They are junit-4.13.jar (2020), hamcrest-core-1.3.jar (2012), and mvvmfx-testing-utils-1.7.0.jar (2018). The first two jar files are the libraries provided by JUnit itself. For the mvvmfx-testing-utils-1.7.0.jar, it is a powerful helper library that can help us eliminating the unnecessary steps required for testing a JavaFX project, such as loading all the JavaFX dependencies to the test environment before the test.

---

junit-4.13.jar 2020. Retrieved July, 2020, from <https://search.maven.org/remotecontent?filepath=junit/junit/4.13/junit-4.13.jar>

hamcrest-core-1.3.jar (2012) Retrieved July, 2020, from <https://search.maven.org/remotecontent?filepath=org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar>

mvvmfx-testing-utils-1.7.0.jar (2018) Retrieved July, 2020, from <https://github.com/sialcasa/mvvmFX/releases/download/mvvmfx-1.7.0/mvvmfx-testing-utils-1.7.0.jar>

**Script 5.16: /test/CharacterTest.java**

```
+ 1 import model.Character;
+ 2 import javafx.scene.input.KeyCode;
+ 3 import org.junit.Before;
+ 4 import org.junit.Test;
+ 5 import org.junit.runner.RunWith;
+ 6 import static org.junit.Assert.assertEquals;
+ 7 import de.saxsys.mvvmfx.testingutils.jfxrunner.JfxRunner;
+ 8
+ 9 @RunWith(JfxRunner.class)
+10 public class CharacterTest {
+11     private Character floatingCharacter;
+12     @Before
+13     public void setup() {
+14         floatingCharacter = new Character(30, 30, 0, 0, KeyCode.A, KeyCode.D,
+15             KeyCode.W);
+16     }
+17     @Test
+18     public void characterInitialValuesShouldMatchConstructorArguments() {
+19         assertEquals("Initial x", 30, floatingCharacter.getX(), 0);
+20         assertEquals("Initial y", 30, floatingCharacter.getY(), 0);
+21         assertEquals("Offset x", 0, floatingCharacter.getOffsetX(), 0.0);
+22         assertEquals("Offset y", 0, floatingCharacter.getOffsetY(), 0.0);
+23         assertEquals("Left key", KeyCode.A, floatingCharacter.getLeftKey());
+24         assertEquals("Right key", KeyCode.D, floatingCharacter.getRightKey());
+25         assertEquals("Up key", KeyCode.W, floatingCharacter.getUpKey());
+26     }
+27 }
```

Script 5.16 shows an example *unit test* case for the `Character` class. Following the naming convention, let the class `CharacterTest` be the test class for its testing. The list of import libraries is presented in this script because the automatic import feature may not work correctly in some environments. For anyone who experiences an import error, kindly following the example provided in the script. The `@RunWith(JfxRunner.class)` annotation at Line 9 of the script declares that all the code fragments implementing the class will be executed with the resources provided by the `JfxRunner` class. The `@Before` annotation at Line 12 flags a method



called before executing all the *unit test* cases. This is also a practically useful tool that allows us to prepare some commonly used resources such as initializing an object instance before running any tests requiring it. In this example, the method `setup` creates a `Character` instance that will be later used in the test. The only test case in this script is presented between Line 17 and Line 25 of the script. It just checks whether all the argument values passing to the constructor of the `Character` class have been correctly assigned to the right variables. Now, we should be able to run and see that the test is passed.

#### 5.4.5 Unit testing with reflection

Next, this tutorial will demonstrate the test for character movements. The main difficulty for testing these functionalities is that we have to test private methods that we cannot call them directly from the object instance. As discussed earlier in the chapter, the conventional solutions to address this situation is to use the *reflection* technique. Script 5.17 presents an updated `CharacterTest` class to test whether it can perform correctly after pressing the key meant to move it to the left. Mainly, this update involves with many classes, since the key pressing is made at a controller. Then, the consequence of the key pressing will propagate to the associated model, and finally, the view components. Therefore, many more variables are associated with this test case than the previous one. Between Line 5 and Line 9 of the script, the class instances are created. Then, they all are initialized in the `setup` method between Line 13 and Line 27. The *reflection* technique is presented in the `setup` method between Line 19 and Line 22 to provide access to the private methods of the `GameLoop` and the `DrawingLoop` classes. Finally, the body of this test case is illustrated between Line 48 and Line 56 of the script. We manually invoke the pressing of the keyboard key `A` and force the `GameLoop` to refresh itself once, followed by the `DrawingLoop`. This mimics one single clock tick for each loop. Then three assertion checks are made to observe the changes of the controller component, followed by the model component, and finally, the view component.

**Script 5.17: /test/CharacterTest.java**

```

1  //Imports are omitted
2  @RunWith(JfxRunner.class)
3  public class CharacterTest {
4      private Character floatingCharacter;
+ 5      private ArrayList<Character> characterListUnderTest;
+ 6      private Platform platformUnderTest;
+ 7      private GameLoop gameLoopUnderTest;
+ 8      private DrawingLoop drawingLoopUnderTest;
+ 9      private Method updateMethod;
10     @Before
11     public void setup() {
12         floatingCharacter = new Character(30, 30, 0, 0, KeyCode.A, KeyCode.D, KeyCode.W);
+ 13         characterListUnderTest = new ArrayList<Character>();
+ 14         characterListUnderTest.add(floatingCharacter);
+ 15         platformUnderTest = new Platform();
+ 16         gameLoopUnderTest = new GameLoop(platformUnderTest);
+ 17         drawingLoopUnderTest = new DrawingLoop(platformUnderTest);
+ 18         try {
+ 19             updateMethod = GameLoop.class.getDeclaredMethod("update", ArrayList.class);
+ 20             redrawMethod = DrawingLoop.class.getDeclaredMethod("paint", ArrayList.class);
+ 21             updateMethod.setAccessible(true);
+ 22             redrawMethod.setAccessible(true);
+ 23         } catch (NoSuchMethodException e) {
+ 24             e.printStackTrace();
+ 25             updateMethod = null;
+ 26             redrawMethod = null;
+ 27         }
28     }
29     ...
+ 48     @Test
+ 49     public void characterShouldMoveToTheLeftAfterTheLeftKeyIsPressed() throws
        IllegalAccessException, InvocationTargetException {
+ 50         platformUnderTest.getKeys().add(KeyCode.A);
+ 51         updateMethod.invoke(gameLoopUnderTest, characterListUnderTest);
+ 52         redrawMethod.invoke(drawingLoopUnderTest, characterListUnderTest);
+ 53         assertTrue("Controller: Left key pressing is acknowledged", platformUnderTest.
            getKeys().isPressed(KeyCode.A));
+ 54         assertTrue("Model: Character moving left state is set", characterListUnderTest.
            get(0).isMoveLeft());
+ 55         assertTrue("View: Character is moving left", characterListUnderTest.get(0).getX()
            < startX);
+ 56     }
57 }

```

#### 5.4.6 Exercise

1. Add a test case to test the consequences after pressing the key *D* on the keyboard.
2. Add a test case to test the consequence after pressing the key *W* on the keyboard. Let the character is being on the ground.
3. Add a test case to test the consequence after pressing the key *W* on the keyboard. Let the character is not being on the ground.
4. Add a test case to test the consequences as the character is hitting a border.
5. Add a test case to test the consequences as the character has collided with the other character.
6. Add a test case to test the consequences as the character stomps the other.
7. Add a test case to test the consequences as the character is stomped by the other.

## 5.5 Case study -- TDD

The second case study of this chapter showcases a software development example following the TDD principle. Specifically, we will implement a simple variation of the well-known snake game. As a beginning step, let us create the below project structure:

```
src
├── controller
│   ├── GameLoop.java
│   └── Launcher.java
├── model
│   ├── Direction.java
│   ├── Food.java
│   └── Snake.java
└── view
    └── Platform.java
test
├── GameLoopTest.java
├── JUnitTestSuite.java
└── SnakeTest.java
```

Following the steps to perform TDD discussed earlier in the chapter, let us begin the development of this snake game by breaking down its essential tasks into smaller tasks and prioritizing them. As an example, the following list demonstrates one possible approach for the snake game. Note that the tasks have not been prioritized yet.

- The game is real-time. In one clock tick, the snake moves toward the direction it is heading to;
- The snake can move in 4 directions: up, down, left, right;
- The snake cannot suddenly turn to the direction which is opposite to what it is heading to;

- The snake can collide with food;
- The snake can grow after colliding with food;
- Food must be respawned after it was collided;
- The snake can die if it collides with its body;
- The snake can also die if it collides with the game wall.

#### 5.5.1 The snake's head, body, and tail

The SnakeTest is the test class for the Snake class and it is illustrated in Script 5.18. It should be noted that the codes are still unable to compile and run yet because they all are still the template to help us design the game, and we will implement the production code to pass these tests later. At this step, the followings are what to be tested:

- The Snake instance is created at a particular coordinate, such as (0, 0);
- The direction of the snake character, i.e., up, down, left, right, is set;
- Up, down, left, and right keys should be implemented using enum, e.g., Direction.Down
- Updating the snake character should follow a clock tick, so in this variation of the game, we will let the snake move one pixel towards the direction it is heading to.

Script 5.19 presents an enum representing the four directions the snake character can move. These Direction enums are used in the Snake class. The current version of the Snake class implements only to pass the currently available test cases. Thus, only a constructor and the other three methods called in the test are presented. A snake character is represented by an ArrayList of 2D coordinates where the coordinate being stored in the ArrayList are the coordinates the snake body is present. The movement is made only at the head of the snake character, so an additional variable representing the snake character's head is required alongside

**Script 5.18: /test/SnakeTest.java**

```
+ 1 //Imports are omitted
+ 2 @RunWith(JfxRunner.class)
+ 3 public class SnakeTest {
+ 4     private Snake snake;
+ 5     @Before
+ 6     public void setup() {
+ 7         snake = new Snake(new Point2D(0, 0));
+ 8     }
+ 9     @Test
+10     public void snakeShouldBeSpawnAtTheCoordinateItWasCreated() {
+11         assertEquals(snake.getHead(), new Point2D(0, 0));
+12     }
+13     @Test
+14     public void snakeShouldMoveDownwardIfItIsHeadingDownward() {
+15         snake.setCurrentDirection(Direction.DOWN);
+16         snake.update();
+17         assertEquals(snake.getHead(), new Point2D(0, 1));
+18     }
+19 }
```

the body `ArrayList`. Apart from the head, the tail coordinate also requires a dedicated variable. This variable will be used to determine which coordinate should be removed from the body `ArrayList` to represent the snake's movement. The programming logic used in the update method is to (1) determine the coordinates the snake character is moving to, (2) remove the coordinates currently representing the snake character's tail, and (3) add the current head to the body `ArrayList` after the movement. The other two methods are simply the Getter of the variables used in the `SnakeTest` class.

**Script 5.19: /src/model/Direction.java**

```
+ 1 //Imports are omitted
+ 2 public enum Direction {
+ 3     RIGHT(new Point2D(1, 0)),
+ 4     LEFT(new Point2D(-1, 0)),
+ 5     UP(new Point2D(0, -1)),
+ 6     DOWN(new Point2D(0, 1));
+ 7     public final Point2D current;
+ 8     Direction(Point2D current) { this.current = current; }
+ 9 }
```

**Script 5.20: /src/model/Snake.java**

```
+ 1 //Imports are omitted
+ 2 public class Snake {
+ 3     private Direction direction;
+ 4     private Point2D head;
+ 5     private Point2D prev_tail;
+ 6     private ArrayList<Point2D> body;
+ 7     public Snake(Point2D position) {
+ 8         direction = Direction.DOWN;
+ 9         body = new ArrayList<>();
+10         this.head = position;
+11         this.body.add(this.head);
+12     }
+13     public void update() {
+14         head = head.add(direction.current);
+15         prev_tail = body.remove(body.size() - 1);
+16         body.add(0, head);
+17     }
+18     public void setCurrentDirection(Direction direction) { this.direction =
+19         direction; }
+19     public Direction getCurrentDirection() { return this.direction; }
+20     public Point2D getHead() { return head; }
+21 }
```

### 5.5.2 Food colliding and snake growing

Script 5.25 adds four test cases to the SnakeTest class. They are to test whether the snake character acknowledges its collision with food. Then, the food should respawn on a different coordinate. Next, the snake character's length should grow by one and the head should move forward. Finally, the head on the last clock tick should become the body on the current tick. Specifically, four methods are added to the script in this update: `collisionFlagShouldRaiseIfSnakeCollideWithFood`, `foodShouldRespawnOnDifferentCoordinates`, `snakeGrowthShouldAddItsLengthByOne`, and `bodyOfGrownSnakeShouldContainPreviousHead`, respectively.

#### Script 5.21: /test/SnakeTest.java

```
1 //Imports are omitted
2 @RunWith(JfxRunner.class)
3 public class SnakeTest {
4     ...
+ 27     @Test
+ 28     public void collisionFlagShouldRaiseIfSnakeCollideWithFood() {
+ 29         Food food = new Food(new Point2D(0,0));
+ 30         assertTrue(snake.isCollidingWith(food));
+ 31     }
+ 32     @Test
+ 33     public void foodShouldRespawnOnDifferentCoordinates() {
+ 34         Food food = new Food(new Point2D(0,0));
+ 35         food.respawn();
+ 36         assertNotSame(food.getPosition(),new Point2D(0,0));
+ 37     }
+ 38     @Test
+ 39     public void snakeGrowthShouldAddItsLengthByOne() {
+ 40         snake.grow();
+ 41         assertEquals(snake.getLength(),2);
+ 42     }
+ 43     @Test
+ 44     public void bodyOfGrownSnakeShouldContainPreviousHead() {
+ 45         Point2D cur_head = snake.getHead();
+ 46         snake.update();
+ 47         snake.grow();
+ 48         assertTrue(snake.getBody().contains(cur_head));
+ 49     }
50 }
```



Two more classes are required to develop a minimum production code to past these test cases. The first class is the food model, and the other class is for the main view of the application. Let the names of the two classes be `Food` and `Platform`, respectively. Also, the `Snake` class needs implementation for the methods called in the updated `SnakeTest` class. Mainly, Script 5.22 presents the `Food` class with two main variables of the type `Point2D` and `Random`, representing the food instance coordinates, and a randomizer used in spawning the food elsewhere after the snake character consumes it. Script 5.23 implements the game scene of a fixed size of 30 multiplied by 30 tiles, where each tile has the size of 10 pixels multiplied by 10 pixels. Finally, Script 5.24 illustrates the updated `Snake` class by implementing four new methods named `isCollidingWith`, `grow`, `getLength`, and `getBody`. The `isCollidingWith` method checks whether the head of the snake in the current clock tick is being collided with any other entities or not. The `grow` method increases the length of the snake by one at its tail. The `getLength` method returns the current size of the class variable `body`. Finally, `getBody` method returns the entire snake body as a list. After these updates, our collection of the *unit test* cases will grow to six test cases.

**Script 5.22: /src/model/Food.java**

```
+ 1 //Imports are omitted
+ 2 public class Food {
+ 3     private Point2D position;
+ 4     private Random rn;
+ 5     public Food(Point2D position) {
+ 6         this.rn = new Random();
+ 7         this.position = position;
+ 8     }
+ 9     public Food() {
+10         this.rn = new Random();
+11         respawn();
+12     }
+13     public void respawn() {
+14         Point2D prev_position = this.position;
+15         do {
+16             this.position = new Point2D(rn.nextInt(Platform.WIDTH), rn.nextInt(
+17                 Platform.HEIGHT));
+18         } while(prev_position == this.position);
+19     }
+20     public Point2D getPosition() { return position; }
+21 }
```

**Script 5.23: /src/view/Platform.java**

```
+ 1 //Imports are omitted
+ 2 public class Platform extends Pane {
+ 3     public static final int WIDTH = 30;
+ 4     public static final int HEIGHT = 30;
+ 5     public static final int TILE_SIZE = 10;
+ 6     private Canvas canvas;
+ 7     private KeyCode key;
+ 8
+ 9     public Platform() {
+10         this.setHeight(TILE_SIZE * HEIGHT);
+11         this.setWidth(TILE_SIZE * WIDTH);
+12         canvas = new Canvas(TILE_SIZE * WIDTH, TILE_SIZE * HEIGHT);
+13         this.getChildren().add(canvas);
+14     }
+15     public void render(Snake snake, Food food) {
+16         GraphicsContext gc = canvas.getGraphicsContext2D();
+17         gc.clearRect(0,0,WIDTH*TILE_SIZE,HEIGHT*TILE_SIZE);
+18         gc.setFill(Color.BLUE);
+19         snake.getBody().forEach(p -> {
+20             gc.fillRect(p.getX() * TILE_SIZE, p.getY() * TILE_SIZE, TILE_SIZE,
+21                         TILE_SIZE);
+22         });
+23         gc.setFill(Color.RED);
+24         gc.fillRect(food.getPosition().getX() * TILE_SIZE, food.getPosition().getY()
+25                     * TILE_SIZE, TILE_SIZE, TILE_SIZE);
+26     }
+27     public KeyCode getKey() { return key; }
+28     public void setKey(KeyCode key) { this.key = key; }
+29 }
```

**Script 5.24: /src/model/Snake.java**

```
1 //Imports are omitted
2 public class Snake {
3     ...
18     public void setCurrentDirection(Direction direction) { this.direction = direction; }
19     public Direction getCurrentDirection() { return this.direction; }
20     public Point2D getHead() { return head; }
+ 21     public boolean isCollidingWith(Food food) { return head.equals(food.getPosition()); }
+ 22     public void grow() { body.add(prev_tail); }
+ 23     public int getLength() { return body.size(); }
+ 24     public List<Point2D> getBody() { return body; }
25 }
```

**5.5.3 Getting the snake killed**

According to the initial requirements, two conditions that kill the snake character are when the snake collides with the game border and itself. Script 5.25 covers the two cases by implementing two test methods named `snakeWillDieIfItGoesBeyondTheGameBorder` and `snakeWillDieIfItHitsItsBody` in the `SnakeTest` class. For the production code, one new method named `isDead` is implemented and called by the `Snake` instance, as presented in Script 5.26. The `snakeWillDieIfItGoesBeyondTheGameBorder` method simply places the snake character near the border and set the head towards it. After an update, the snake character will literally hit the game border, and the test case checks whether its state is set to dead or not. For the `snakeWillDieIfItHitsItsBody` test case, initially, the test case grows the snake body to the size that it can hit itself, Then, the test turns the snake 360 degrees to make it hit itself literally, and the test case checks whether its state is set to dead or not. Once all the updates are made completely, it is time examine whether the application can now pass all the eight test cases or not.

**Script 5.25: /test/SnakeTest.java**

```
1 //Imports are omitted
2 @RunWith(JfxRunner.class)
3 public class SnakeTest {
4     ...
+ 49     @Test
+ 50     public void snakeWillDieIfItGoesBeyondTheGameBorder() {
+ 51         snake = new Snake(new Point2D(30,30));
+ 52         snake.setCurrentDirection(Direction.RIGHT);
+ 53         snake.update();
+ 54         assertTrue(snake.isDead());
+ 55     }
+ 56     @Test
+ 57     public void snakeWillDieIfItHitsItsBody() {
+ 58         snake = new Snake(new Point2D(0,0));
+ 59         snake.setCurrentDirection(Direction.DOWN);
+ 60         snake.update();
+ 61         snake.grow();
+ 62         snake.setCurrentDirection(Direction.LEFT);
+ 63         snake.update();
+ 64         snake.grow();
+ 65         snake.setCurrentDirection(Direction.UP);
+ 66         snake.update();
+ 67         snake.grow();
+ 68         snake.setCurrentDirection(Direction.RIGHT);
+ 69         snake.update();
+ 70         snake.grow();
+ 71         assertTrue(snake.isDead());
+ 72     }
73 }
```

**Script 5.26: /src/model/Snake.java**

```
1 //Imports are omitted
2 public class Snake {
3     ...
+ 18     public boolean isDead() {
+ 19         boolean isOutOfBounds = head.getX() < 0 || head.getY() < 0 || head.getX() >
           Platform.WIDTH || head.getY() > Platform.HEIGHT;
+ 20         boolean isHitBody = body.lastIndexOf(head) > 0;
+ 21         return isOutOfBounds || isHitBody;
+ 22     }
+ 23     ...
}
```

#### 5.5.4 The game loop

Script 5.27 illustrates the `GameLoopTest` class, the test class of the `GameLoop` class. This `GameLoop` class will be implemented following the flow implemented for the `GameLoop` in the previous case study, which consists of three main methods: `update`, `collisionChecking`, and `redraw`. All these methods are private methods where we have to use the *reflection* technique to test them. In this class, another specific technique is used in the method named `clockTickHelper`. The method is also common for *unit testing* known as a helper method, which is used to bundle other methods often called together during the test. In this case, a clock tick is meant to invoke all the three main methods of the `GameLoop` class sequentially. Thus, bundling them together will allow us to organize test codes better and maximize the class's maintainability. The two test methods presenting in the script are the test methods for the two remaining requirements about the real-time stuff, and the other one saying the snake cannot suddenly turn to the direction which is opposite to what it is heading to. These two test cases are implemented as the methods `snakeShouldAdvanceAfterAClockTick` and `snakeCannotDirectlyTurnToTheOppositeDirection`, respectively.

The associated class to be implemented is the `GameLoop` class. Scrutinizing the test class in Script 5.27, we shall see that the `GameLoop` should contain the instances of the `Platform`, `Snake`, and `Food` classes, as hinted at the Line 9 of the script. Next, aside from the `update`, `checkCollision`, and `redraw` methods, the `Getter` methods are also needed for the `Platform` and `Snake` classes because they are called in both `snakeShouldAdvanceAfterAClockTick` and `snakeCannotDirectlyTurnToTheOppositeDirection` methods. Particularly, Script 5.28 illustrates the `GameLoop` class's implementation that can pass all the test cases of the `GameLoopTest` class.

The most noteworthy part of the `GameLoop` class is its `update` method, which implements the programming logic for the requirement saying the snake cannot suddenly turn to the direction opposite to what it is heading to. The method checks whether the key pressed is in the opposite direction of that the snake heading direction or not. We proceed with the change in direction for the snake character only if it is not the opposite. Otherwise, the snake character will continue moving as if nothing happens. All the other implementations in the class follow the same programming logic sets implemented for the previous case study's platformer game.

**Script 5.27: /test/GameLoopTest.java**

```

+ 1 //Imports are omitted
+ 2 public class GameLoopTest {
+ 3     private GameLoop gameLoopUnderTest;
+ 4     private Method update;
+ 5     private Method collision;
+ 6     private Method redraw;
+ 7     @Before
+ 8     public void init() throws NoSuchMethodException {
+ 9         gameLoopUnderTest = new GameLoop(new Platform(), new Snake(new Point2D(0, 0))
+ 10             , new Food());
+ 11         update = GameLoop.class.getDeclaredMethod("update");
+ 12         update.setAccessible(true);
+ 13         collision = GameLoop.class.getDeclaredMethod("checkCollision");
+ 14         collision.setAccessible(true);
+ 15         redraw = GameLoop.class.getDeclaredMethod("redraw");
+ 16         redraw.setAccessible(true);
+ 17     }
+ 18     private void clockTickHelper() throws InvocationTargetException,
+ 19         IllegalAccessException {
+ 20         update.invoke(gameLoopUnderTest);
+ 21         collision.invoke(gameLoopUnderTest);
+ 22         redraw.invoke(gameLoopUnderTest);
+ 23     }
+ 24     @Test
+ 25     public void testClockTick() throws InvocationTargetException,
+ 26         IllegalAccessException {
+ 27         gameLoopUnderTest = new GameLoop(new Platform(), new Snake(new Point2D(0,0)),
+ 28             new Food());
+ 29         clockTickHelper();
+ 30         assertEquals(gameLoopUnderTest.getSnake().getHead(), new Point2D(0,1));
+ 31     }
+ 32     @Test
+ 33     public void testNoBack() throws InvocationTargetException,
+ 34         IllegalAccessException {
+ 35         gameLoopUnderTest = new GameLoop(new Platform(), new Snake(new Point2D(0,0)),
+ 36             new Food());
+ 37         gameLoopUnderTest.getPlatform().setKey(KeyCode.DOWN);
+ 38         clockTickHelper();
+ 39         assertEquals(gameLoopUnderTest.getSnake().getHead(), new Point2D(0,1));
+ 40         gameLoopUnderTest.getPlatform().setKey(KeyCode.DOWN);
+ 41         clockTickHelper();
+ 42         assertEquals(gameLoopUnderTest.getSnake().getHead(), new Point2D(0,2));
+ 43         gameLoopUnderTest.getPlatform().setKey(KeyCode.UP);
+ 44         clockTickHelper();
+ 45         assertEquals(gameLoopUnderTest.getSnake().getHead(), new Point2D(0,3));
+ 46     }
+ 47 }

```

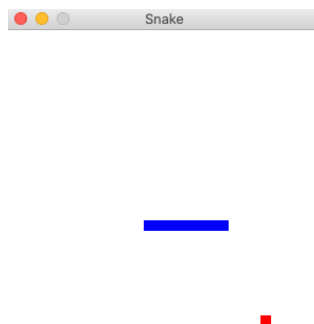
**Script 5.28: /src/controller/GameLoop.java**

```
+ 1 //Imports are omitted
+ 2 public class GameLoop implements Runnable {
+ 3     private Platform platform;
+ 4     private Snake snake;
+ 5     private Food food;
+ 6     private float interval = 1000.0f / 10;
+ 7     private boolean running;
+ 8     public GameLoop(Platform platform, Snake snake, Food food) {
+ 9         this.snake = snake;
+10         this.platform = platform;
+11         this.food = food;
+12         running = true;
+13     }
+14     private void update() {
+15         KeyCode cur_key = platform.getKey();
+16         Direction cur_direction = snake.getCurrentDirection();
+17         if (cur_key == KeyCode.UP && cur_direction != Direction.DOWN)
+18             snake.setCurrentDirection(Direction.UP);
+19         else if (cur_key == KeyCode.DOWN && cur_direction != Direction.UP)
+20             snake.setCurrentDirection(Direction.DOWN);
+21         else if (cur_key == KeyCode.LEFT && cur_direction != Direction.RIGHT)
+22             snake.setCurrentDirection(Direction.LEFT);
+23         else if (cur_key == KeyCode.RIGHT && cur_direction != Direction.LEFT)
+24             snake.setCurrentDirection(Direction.RIGHT);
+25         snake.update();
+26     }
+27     private void checkCollision() {
+28         if (snake.isCollidingWith(food)) {
+29             snake.grow();
+30             food.respawn();
+31         }
+32         if (snake.isDead()) { running = false; }
+33     }
+34     private void redraw() { platform.render(snake, food); }
+35     @Override
+36     public void run() {
+37         while (running) {
+38             update();
+39             checkCollision();
+40             redraw();
+41             try {
+42                 Thread.sleep((long)interval);
+43             } catch (InterruptedException e) {
+44                 e.printStackTrace();
+45             }
+46         }
+47     }
+48 }
```

We shall see the passing test results in our growing test collections. Now it is time to assemble all the building blocks by implementing the Launcher class. Script 5.29 illustrates an example Launcher class of this application. Figure 5.3 shows an example of the rendered game window after playing it for a few seconds.

**Script 5.29: /src/controller/Launcher.java**

```
+ 1 //Imports are omitted
+ 2 public class Launcher extends Application {
+ 3     public static void main(String[] args) { launch(args); }
+ 4     @Override
+ 5     public void start(Stage primaryStage) {
+ 6         Platform platform = new Platform();
+ 7         Snake snake = new Snake(new Point2D(platform.WIDTH/2,platform.HEIGHT/2));
+ 8         Food food = new Food();
+ 9         GameLoop gameLoop = new GameLoop(platform,snake,food);
+10         Scene scene = new Scene(platform,platform.WIDTH*platform.TILE_SIZE,platform
+11             .HEIGHT * platform.TILE_SIZE);
+12         scene.setOnKeyPressed(event-> platform.setKey(event.getCode()));
+13         scene.setOnKeyReleased(event -> platform.setKey(null));
+14         primaryStage.setTitle("Snake");
+15         primaryStage.setScene(scene);
+16         primaryStage.setResizable(false);
+17         primaryStage.show();
+18         (new Thread(gameLoop)).start();
+19     }
+20 }
```



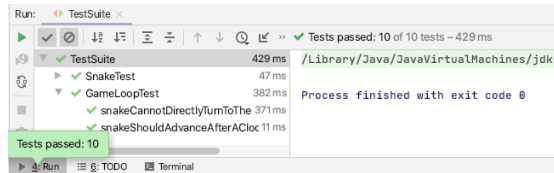
**Figure 5.3: The rendered snake game window.**



### 5.5.5 The test suite

The last thing this tutorial will instruct is to bundle the two test classes into a test suite. Script 5.30 illustrates a class name `TestSuite` in which the execution of this class will run all the test cases in the `SnakeTest` and

`GameLoopTest` classes. Figure 5.4 shows example test results generated from the execution of the `TestSuite` class in the IntelliJ Idea IDE.



**Figure 5.4: An example test results generated from a test suite.**

#### Script 5.30: `/test/TestSuite.java`

```
+ 1 import org.junit.runner.RunWith;
+ 2 import org.junit.runners.Suite;
+ 3
+ 4 @RunWith(Suite.class)
+ 5 @Suite.SuiteClasses({SnakeTest.class, GameLoopTest.class})
+ 6
+ 7 public class TestSuite { }
```

### 5.5.6 Exercise

1. Create a popup window to say that the game is over after the snake character is dead.
2. Demonstrate the TDD principle's use to create a score functionality that eating one food worth one point.
3. Demonstrate the TDD principle's use to add special foods to the game. This food is randomly generated, and eating this food will worth five points. Let the color representing this food be green.

## Build management system

**S**OFTWARE build management system is an essential component of Software configuration management (SCM), which is a crucial supportive element in the software development process. SCM primary responsibilities are to handle changes, corrections, extensions, and adaptations made to the software application during its developments. In practice, SCM is driven by the advancement of toolkits, such as Git (2020), Maven (2020), and Docker (2020). SCM key functionalities are those related to build and release management, including workspace, versioning, building, release management, dependency management, repository management, and change management.

In this course, we will put the main focus on software building and dependency management. For the other elements, they all will be gradually introduced in higher-level courses of the SE program.

---

Git (2020) git –distributed-is-the-new-centralized. Retrieved August, 2020, from <https://git-scm.com/>

Maven (2020) Apache Maven Project. Retrieved August, 2020, from <https://maven.apache.org/>

Docker (2020) Docker. Retrieved August, 2020, from <https://www.docker.com/>

## 6.1 Built tool and dependency management

### 6.1.1 Build tool

A typical build tool is a composite product primarily used to compile and make a runnable software application from source codes. A build tool consists of two main components: the configuration and the construction components, a.k.a., a build script and an executable that processes the build script, respectively. All these components are platform free such that it should be able to execute on any operating systems without additional efforts to configure it. Specifically, the followings are the main tasks for a build tool:

- recompile only the portions of software application that have been modified;
- properly handle dependencies of the software application;
- allow the user to select compiling profiles, e.g., development or production;
- minimize the effort required to adapt to changes regarding new product requirements;
- make everything automated and remove all manual efforts as possible.

### 6.1.2 Dependency management

Before we discuss dependency management, we may have to know what is meant by dependency first. The notion of dependency starts from the fact that it is tough to find any working software projects in practice existing in isolation. Such that, each task carried out in a project by a module or component is more likely to rely on an existing module or component developed elsewhere. For example, we included external libraries to our projects in the previous chapters' case studies, such as the module used for extracting a PDF file in the third chapter and that of for handling logging in the fourth chapter. We call the libraries added to the project as the dependencies of the project.

Dependencies required careful management because they all can be easily messed up. The problem is due to that, in practice, we are not the only ones who have to

manage these dependencies when including them into our project manually, but the dependencies we included in our project also have their dependencies to manage as well. These dependencies often conflict with each other in production, such that the libraries we manually included in our application may cause the other dependencies to fail to compile.

A typical dependency management tool has its primary responsibility to identify software components' dependencies and resolve dependency conflicts if they exist. A dependency management tool's particular requirement is that they do have to carry out all their task in an automated manner. This is because manual dependency management is nothing but a big area of pain in software development. For example, we had to physically download all the libraries from online repositories, manually importing the downloaded files to the dependency list of the project, and check whether they all work or not for all the projects we implemented throughout this course. These are not an easy task at all for a large software project with a vast amount of libraries to be included. Overall, a dependency management tool has to automatically search and pinpoint a unique set of dependencies accepted by all the modules or components of the software application, making the application compiled and run successfully.

More modern build tools include dependency management as one of their main functionalities. Besides, as to supply tasks commonly done with coding, modern build tools also offer fully managed and automated pipelines that allow us to compile, test, package, document, and deploy our software project with the most hassle-free experience.

## 6.2 Java build management system

There are three most well know build management systems for Java language, which are Apache ANT+Ivy (2017), Maven, and Gradle (2020). ANT stands for Another Neat Tool, and it is known as the first build tool for Java created in 1999. ANT was designed to be compact, extensible, and operating system independent. At the time of its release, it is well accepted as an easy to learn toolset, and it can effectively enable its users to completely control the entire build process, which was otherwise impossible before its existence. One of the main reasons ANT was mentioned as an easy-to-learn use toolset is that a collection of xml files specifies it.. However, these become a drawback later as the XML trend was faded, and more compact notations such as domain-specific language (DSL) and JSON have become more mainstream. Incidentally, what makes ANT less popular was not to blame the choice of using the markup language; however, it was widely said that ANT is not scalable because it does not provide the predefined set of build cycle as well as the crucial dependency management functionality. Later, Apache introduces Ivy as a vital partnership to ANT as it gives the dependency management required to more extensive, more complex projects. However, it appears to be too late for this collaboration, since the introduction of Apache Maven around the same time has quickly gained the reputation, and it has long been the Java standard build tool up to the present days.

Maven is a word in the Yiddish language, meaning accumulator of knowledge. Even if a Maven script is also represented in xml format, which is the same as that of ANT, Maven is considered the current most popular build tool today. Maven's two killing features, making it won against all its competitors very soon after its release are (1) its provided perfect dependency management functionality and (2) its design concept aiming to standardize the build process. In short, Maven simplifies the whole process of software packaging and adding necessary dependencies to a project. Before the release of Maven, Java developers typically used ANT, and they criticized ANT over many redundant steps they have to repeat in all their projects. Thus, many build activities were consolidated into the dedicated concept of a build

---

Apache ANT+Ivy (2017) The Apache ANT Project. Retrieved August, 2020, from <https://ant.apache.org/ivy/>

Gradle (2020) Gradle Build Tool. Retrieved August, 2020, from <https://gradle.org/>

lifecycle proposed by Maven. In brief, a build lifecycle is composed of predefined activities commonly done with coding. For example, a simple lifecycle may contain a *compile* phase followed by a *package* phase, where source codes are compiled into binary files and bundled into a deliverable format, respectively. More details about Maven build lifecycle will be elaborated later in this chapter.

Last but not least is Gradle. It was released in 2007, and it can be considered the most modern build tool among the three build tool discussing here. Gradle is designed around multi-project environments, where more recent projects are more geared towards this direction. Different from that of ANT and Maven, Gradle selects DSL for its representation. This makes the build script based on Gradle significantly more compact and more straightforward than those based on XML syntaxes.

In this course, it is the lecturer's choice to select Maven to showcases the Java build management system components. The main reason is owing to its rich structured components are most suitable for one who is getting to scrutinize the Java build management system for her or his very first time.

### 6.2.1 Apache Maven

There are two areas of Maven that are best known to its users. Remarkably, they are a complete build lifecycle and dependency management. Maven users have a broad agreement that both build and dependency management can be done with Maven with the most hassle-free experience as if it is only a matter of copy and paste. For example, in dependency management, without the hassle of downloading and importing all the dependencies to our project classpath, we only need to copy the dependency coordinates from the Maven repository and paste it in the Maven configuration file in our project and let Maven automatically handle rest for us.

### Maven build lifecycle

Maven build lifecycle is defined as sequential phases representing stages in the life-cycle. Each phase contains goals it has to fulfill as to say the task carried out at the particular build stage is completed. For example, the *package* phase's goal is to bundle all the compiled code into a jar file successfully. Maven provides 23 default build lifecycle phases in total; however, only a subset is needed for a particular software project. The following lists the typical phases along with their brief descriptions:

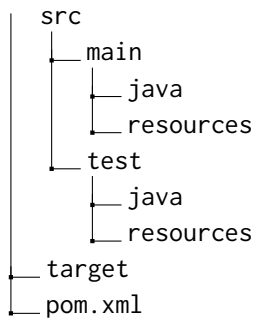
- ***clean*** removes all files generated by the previous build;
- ***validate*** checks whether the project is correct and all necessary information is available;
- ***process-resources*** copies and processes the resources into the destination directory, ready for packaging;
- ***compile*** compiles the source code of the project;
- ***process-test-resources*** copies and processes resources into the test destination directory;
- ***test-compile*** compiles the test source code into the test destination directory;
- ***test*** executes the unit tests;
- ***package*** takes the compiled code and bundles it into the deliverable format, such as a JAR;
- ***verify*** carries out an integration test;
- ***install*** installs the deliverables in the local repository;
- ***deploy*** copies or uploads and installs the deliverables to the remote repository.

In practice, a Maven user selects a subset of the build phases for specific deliverables, such as a jar file. The selected subset will be executed sequentially in order to generate the build results desired by the user. For example, without explicit customization, Maven selects the sequence: *validate*, *compile*, *test*, *verify*, *install*, and

*deploy* phases for a typical Java project by default. However, if we have a clear deliverable of choice, we can select the phase subset that is more suitable for our target deliverable. For example, the common build phase subset used in packaging a typical Java project into a jar file (Mcintosh et al. 2012) comprises *process-resources*, *compile*, *process-test-resources*, *test-compile*, *test*, *package*, *install*, and *deploy* phases.

### Maven standard directory

Maven defines a standard project directory layout to enable automated project-build infrastructure. Typically, without Maven, most software projects are more likely to have a different directory layout. These differences typically make it difficult for newcomer developers as it would take time for them to understand the project components and get acquainted with the project. However, with Maven, the project directory layout has a standard to follow. When joining a new project, a Maven user who has already familiar with this standardized structure can easily understand the project layout and know exactly where to find each particular file. Also, since the standardized project directory layout is decided outside IDE so that Maven projects can be shared easily between IDEs without compatibility issues. Specifically, a newly initialized Maven project has the following project layout:





There are only two subdirectories: `src` and `target`, where the `src` directory contains all the project source codes and other materials for building the project. The `src` directory has two subdirectories: `main` and `test`. These two directories have their subdirectories in the same structure, a directory for Java codes, and another directory for assets. The subdirectories of the Java codes directory is a usual package hierarchy. The other directory for assets is named `resources`, in which all files inside it will later be copied to the target's classpath. From this Maven standard project directory, what we have to discuss further is the `pom.xml` file. It is the fundamental unit of work in Maven, where we configure it to tell Maven what it has to handle.

### Project object model (POM)

POM is an xml file located at the root directory of a Maven Java project. A POM file contains information about the project and configuration used for building the project. When attempting to build a project, Maven will read the POM file of the project at the project root directory to properly configure the project building following the developers' intent. The information commonly specified in the POM file are project properties, project dependencies, plugins, goals, build profiles, project version, and project descriptions.

Before creating a POM file, we have to define the project group, project name, and project version. Maven defines the terms for these identities as `groupId`, `artifactId`, and `version`, respectively. These identities plus a specified Maven protocol version are the minimum requirements of a POM file. Particularly, an example of the POM file with its minimum requirement is illustrated in Script 6.1. In this script, the project notation is `com.mycompany.sample-group:myHelloWorld:1.0`.

**Script 6.1: An example pom file with its minimum requirement.**

**pom.xml**

```
+ 1 <project>
+ 2   <modelVersion>4.0.0</modelVersion>
+ 3   <groupId>com.mycompany.sample-group</groupId>
+ 4   <artifactId>myHelloWorld</artifactId>
+ 5   <version>1.0</version>
+ 6 </project>
```

The following lists detailed explanations for each tag in Script 6.1:

- `<project>` is the root tag of the pom.
- `<modelVersion>` is the protocol version of Maven. It should be set to 4.0.0.
- `<groupId>` is the id of a particular project's group. It must be unique among organizations or internal projects.
- `<artifactId>` is the id of the project. Typically, it is the name of the project.
- `<version>` is the version of the project. It is used along with the `groupId` within an artifact's repository to separate versions from each other. Note that it is a convention to add the word `-SNAPSHOT` after the version name to indicate that it is in its development phase.

**Script 6.2: An example pom file with properties and dependencies.**

**pom.xml**

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.mycompany.sample-group</groupId>
4   <artifactId>myHelloWorld</artifactId>
5   <version>1.0</version>
+ 6   <packaging>jar</packaging>
+ 7   <properties>
+ 8     <maven.compiler.source>14</maven.compiler.source>
+ 9     <maven.compiler.target>14</maven.compiler.target>
+10   </properties>
+11   <dependencies>
+12     <dependency>
+13       <groupId>junit</groupId>
+14       <artifactId>junit</artifactId>
+15       <version>4.13</version>
+16       <scope>test</scope>
+17     </dependency>
+18   </dependencies>
19 </project>
```

Script 6.2 adds additional information regarding project packaging, project properties, and project dependencies. As an example, the project is decided to be packaged into jar format. Using the tags `<maven.compiler.source>` and `<maven.compiler.target>`, we can specify the JDK versions used to compile the source codes and speculate the Java version the compile classes must be compatible with, respectively. In this example, both are set to the JDK version 14. All the dependencies that will be handled by Maven are listed under the `<dependencies>` tag. They all must also be identified following the `<groupId>:<artifactId>:<version>` notation as that used for identifying the current project. In this example, we include the JUnit library version 4.13, which is the same version used in the earlier case studies. Also, at Line 16, we specify that this JUnit library should be available only for the *test* goal of the build lifecycle.

A common question raised regarding dependency management with Maven is how we can know what to be put to the `pom.xml` file to specify the dependencies required for our project. The simplest way is to compose a query with the *the dependency name* followed by the word `pom`

in the Google search engine and follow the link bringing us to a web page under the `https://mvnrepository.com/` domain. For example, Figure 6.1 and Figure 6.2 illustrate an example querying and the resulting web page, respectively. The phrases to be added to the `pom` file is at the bottom of Figure 6.2 under the Maven tag. Script 6.3 shows the updated `pom` file that has the Log4j library included in the project dependency list.

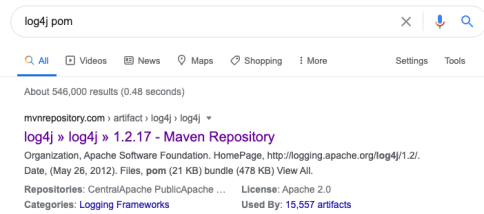


Figure 6.1: Querying for a Maven dependency

## Script 6.3: An example pom file with properties and dependencies.

pom.xml

```

1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.mycompany.sample-group</groupId>
4   <artifactId>myHelloWorld</artifactId>
5   <version>1.0</version>
6   <packaging>jar</packaging>
7   <properties>
8     <maven.compiler.source>14</maven.compiler.source>
9     <maven.compiler.target>14</maven.compiler.target>
10  </properties>
11  <dependencies>
12    <dependency>
13      <groupId>junit</groupId>
14      <artifactId>junit</artifactId>
15      <version>4.13</version>
16      <scope>test</scope>
17    </dependency>
+ 18    <dependency>
+ 19      <groupId>log4j</groupId>
+ 20      <artifactId>log4j</artifactId>
+ 21      <version>1.2.17</version>
+ 22    </dependency>
23  </dependencies>
24 </project>

```

Home » log4j » log4j » 1.2.17

 **Apache Log4j » 1.2.17**  
Apache Log4j 1.2

License	Apache 2.0
Categories	Logging Frameworks
Organization	Apache Software Foundation
HomePage	<a href="http://logging.apache.org/log4j/1.2/">http://logging.apache.org/log4j/1.2/</a>
Date	(May 26, 2012)
Files	<a href="#">pom (21 KB)</a> <a href="#">bundle (478 KB)</a> <a href="#">View All</a>
Repositories	<a href="#">Central</a> <a href="#">Apache Public</a> <a href="#">Apache Releases</a> <a href="#">BeDataDriven</a> <a href="#">Redhat GA</a> <a href="#">Sonatype</a> <a href="#">Spring Plugins</a>
Used By	15,618 artifacts

Maven [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```

<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>

```

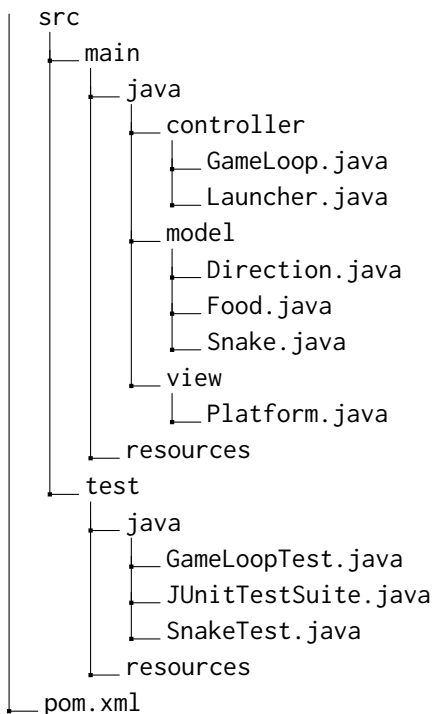
Figure 6.2: An example querying results

## 6.3 Case study

This is the closing case study of the course. It is to showcase how to build and package a Java project effectively. This case study's project configuration is different from that of all the earlier case studies. In the past, we had to create a JavaFX project in the IntelliJ Idea IDE manually and tediously configure the dependencies and virtual machine parameters. In contrast, we will configure this project as a Maven Java project and configure everything through the `pom.xml` file.

### 6.3.1 A Maven Project

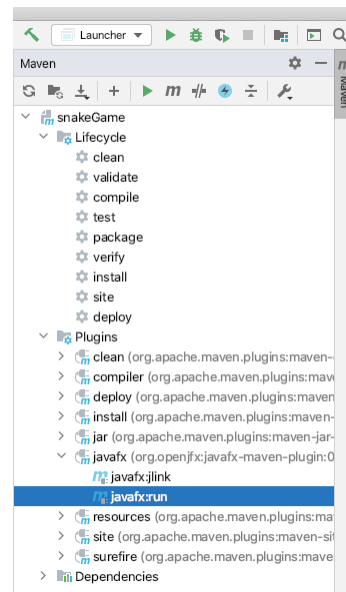
As an example, we will configure the completed Snake game implemented in the previous case study as a Maven project. First of all, let us create a Maven project in an IntelliJ Idea IDE by selecting the *create a new project* menu, then Maven, let the project name be `snakeGame`, and click *Finish*. The IDE will initialize a mint Maven project with the minimum configurations in its `pom.xml` file. Next, let us copy all the files under the `src` and `test` directories of the previous case study to the `src/main/java` and `src/test/java` directories of this Maven project, respectively. The directory structure of the current project will be as follows:



If we examine the files we added to the project by copy and paste, we will see that nearly all the libraries to be imported are still missing. This also includes the JavaFX libraries themselves. The instruction demonstrating how to configure Maven for a JavaFX project is available in the JavaFX (2020) website. It is significantly more effortless than to directly configure it as a regular Java project in the IDE. Specifically, let us follow the instruction provided at <https://openjfx.io/openjfx-docs/#maven> webpage and configured the `pom.xml` file following Script 6.4.

In the script, three components have been configured. The first one is the JDK version for both source and target. The next component is a dedicated plugin that takes care of all the environment configuration. Finally, the last component is all the dependencies associated with JavaFX. Note that at Line 18 of the script, we have to specify the application launcher to let JavaFX know the project's entry point. Note that for the project that also uses the JavaFX-FXML library, we also have to include it to the dependency list alongside JavaFX-Control presented in this script. To force the Maven update, we have to right-click on the `pom.xml` file, locate the word *Maven*, and click *Reimport*. After the progress bar at the bottom of the IDE filled up, we should see that all the missing JavaFX libraries have now been resolved. Thus, the IDE should now be only saying that the missing libraries still exist in for the test classes.

Scrutinizing the `GameLoopTest`, `SnakeTest`, and `TestSuite`, we will see the missing imported libraries, which are those regarding two libraries: JUnit and `Mvvmfx.testingutils`. Following the approach discussed earlier in this chapter, what needs to be included in the `pom.xml` can be easily retrieved using a search engine. From the querying results supplied by the <https://mvnrepository.com/> website, we can update the `pom.xml` to be like Script 6.4. After we reimport the `pom.xml` file, we will see that all the missing libraries have all been resolved. To run the applica-



**Figure 6.3: The Maven menu in IntelliJ Idea**

**Script 6.4: /pom.xml**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
5         org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7     <groupId>org.example</groupId>
8     <artifactId>snakeGame</artifactId>
9     <version>1.0-SNAPSHOT</version>
+ 9 <packaging>jar</packaging>
+ 10 <properties>
+ 11     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
+ 12     <maven.compiler.source>14</maven.compiler.source>
+ 13     <maven.compiler.target>14</maven.compiler.target>
+ 14 </properties>
+ 15 <dependencies>
+ 16     <dependency>
+ 17         <groupId>org.openjfx</groupId>
+ 18         <artifactId>javafx-controls</artifactId>
+ 19         <version>14</version>
+ 20     </dependency>
+ 21 </dependencies>
+ 22 <build>
+ 23     <plugins>
+ 24         <plugin>
+ 25             <groupId>org.openjfx</groupId>
+ 26             <artifactId>javafx-maven-plugin</artifactId>
+ 27             <version>0.0.4</version>
+ 28             <configuration>
+ 29                 <mainClass>controller.Launcher</mainClass>
+ 30             </configuration>
+ 31         </plugin>
+ 32     </plugins>
+ 33 </build>
34 </project>
```

tion, we will no longer use the run button of the IDE anymore. In turn, we will run it through Maven by clicking the Maven tab on the right most of the IntelliJ Idea IDE, as shown in Figure 6.3. Then, all we need to do is to execute the application is to locate the word *Plugins*, then *javafx*, and double click *javafx:run* menu item. In this way, a Maven project is IDE independent, and together with that, the `pom.xml` is reusable, using Maven is considered the most hassle-free experience for the project configuration.

**Script 6.5: /pom.xml**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
5         org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7     <groupId>org.example</groupId>
8     <artifactId>snakeGame</artifactId>
9     <version>1.0-SNAPSHOT</version>
10    <packaging>jar</packaging>
11    ...
+ 21    <dependency>
+ 22        <groupId>junit</groupId>
+ 23        <artifactId>junit</artifactId>
+ 24        <version>4.13</version>
+ 25        <scope>test</scope>
+ 26    </dependency>
+ 27    <dependency>
+ 28        <groupId>de.saxsys</groupId>
+ 29        <artifactId>mvvmfx-testing-utils</artifactId>
+ 30        <version>1.8.0</version>
+ 31        <scope>test</scope>
+ 32    </dependency>
33 </dependencies>
...
```



### 6.3.2 Testing with Maven

Test is a defined Maven lifecycle as shown in the Maven menu in Figure 6.3. Executing it can be as simple as double-clicking the menu item and Maven will execute all the test codes under the `test/java` directory for us. However, if we try executing the *test* lifecycle, an error is rather turned out. Maven provides a thorough detail of its activities at the bottom area of the IDE. If we click the word *test*, we will be able to see the potential cause of the problem, which says that it was due to that a class named `javafx/embed/swing/JFXPanel` is missing from the classpath. From the hint, since the missing class is related to JavaFX, it should be handled by importing the associated class to the `pom.xml` file. Checking all the possible relevant classes in <https://mvnrepository.com/>, the available artifacts under the `org.openjfx` group are `javafx-controls`, `javafx-graphics`, `javafx-base`, `javafx-fxml`, `javafx-web`, `javafx-swing`, and `javafx-media`. Given that the error logs we generated by Maven telling us about the class named `Swing`, what worth trying the most is to add the entry shown in <https://mvnrepository.com/artifact/org.openjfx/javafx-swing/14> to the `pom.xml` file. Script 6.6 shows the resulting `pom.xml` from this trial. Let us refresh the `pom.xml` file and execute the *test* lifecycle again. This time we shall see that all the *unit test* cases have been passed.

#### Script 6.6: /pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
        org/xsd/maven-4.0.0.xsd">
    ...
27     <dependency>
28         <groupId>de.saxsys</groupId>
29         <artifactId>mvvmfx-testing-utils</artifactId>
30         <version>1.8.0</version>
31         <scope>test</scope>
32     </dependency>
+ 33     <dependency>
+ 34         <groupId>org.openjfx</groupId>
+ 35         <artifactId>javafx-swing</artifactId>
+ 36         <version>14</version>
+ 37     </dependency>
38 </dependencies>
    ...

```

### 6.3.3 Packaging with Maven

There are two things to consider when attempting to package an executable Java project: (1) we have to include all the dependencies into the jar file, and (2) we have to tell the jar file which is the main class that will be called when a user double clicks the jar file. Before we further configure the `pom.xml` file, let us ensure that more configuration is needed by simply building the project by running the package lifecycle. As a result, we will see the Maven attempting to compile, test, and package a jar file. The build output will be stored at the target folder. Specifically, the build output of this project is the `target/snakeGame-1.0-SNAPSHOT.jar` file. Typically, we should be able to execute the project on any computer that already has a Java virtual machine installed in it. However, the build output instead shows a popup window saying that *The Java JAR file snakeGame-1.0-SNAPSHOT.jar could not be launched.*, and it suggests us to *Check the Console for possible error messages.* Following the suggestion, let us open a terminal or a console of the operating system and locate the target folder under the project folder and run the command `java -jar snakeGame-1.0-SNAPSHOT.jar`. The result of the command execution will be *no main manifest attribute, in snakeGame-1.0-SNAPSHOT.jar*, which is the actual problem preventing the jar file from executing.

The configurations for both dependency packaging and main class annotation can be done by using Maven plugin named `maven-shade-plugin`. Specifically, the configured plugin is shown in Script 6.7. Anyhow, after we rebuild and attempt to execute the jar file, we will run into another error stating that *Error: JavaFX runtime components are missing, and are required to run this application.* This time, the error is pretty beyond those common ones. We therefore have to look up for an external source to help us overcoming the error. By copying and pasting the entire error message to a search engine and then examining several StackOverflow pages, the one appearing to be the most relevant is <https://stackoverflow.com/questions/52653836>. The answer says that it is a glitch of the plugin itself, and a quick patch for the problem is to extract the Launcher class. Let us follow this StackOverflow answer and create a new class named `JarLauncher` with the content shown in Script 6.8. Then we modify the `pom.xml` file to rather launch this new `JarLauncher` class than the `Launcher` class. This latest modified `pom.xml` file is shown in Script 6.9. After all these fixes, we should now be able to launch the game by double-clicking

the snakeGame-1.0-SNAPSHOT.jar file or by using command `java -jar snakeGame-1.0-SNAPSHOT.jar` in the target folder under the project folder.

#### Script 6.7: /pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
    ...
38  </dependencies>
39  <build>
40    <plugins>
41      <plugin>
42        <groupId>org.openjfx</groupId>
43        <artifactId>javafx-maven-plugin</artifactId>
44        <version>0.0.4</version>
45        <configuration>
46          <mainClass>controller.Launcher</mainClass>
47        </configuration>
48      </plugin>
+ 49    <plugin>
+ 50      <groupId>org.apache.maven.plugins</groupId>
+ 51      <artifactId>maven-shade-plugin</artifactId>
+ 52      <version>3.2.4</version>
+ 53      <executions>
+ 54        <execution>
+ 55          <phase>package</phase>
+ 56          <goals>
+ 57            <goal>shade</goal>
+ 58          </goals>
+ 59          <configuration>
+ 60            <transformers>
+ 61              <transformer implementation="org.apache.maven.plugins.shade
+ 62                .resource.ManifestResourceTransformer">
+ 63                <mainClass>controller.Launcher</mainClass>
+ 64              </transformer>
+ 65            </transformers>
+ 66          </configuration>
+ 67        </execution>
+ 68      </executions>
+ 69    </plugin>
70  </build>
71 </project>

```

**Script 6.8: /src/controller/JarLauncher.java**

```
+ 1 //Imports are omitted
+ 2 public class JarLauncher {
+ 3     public static void main(String[] args) {
+ 4         Launcher.main(args);
+ 5     }
+ 6 }
```

**Script 6.9: /pom.xml**

```
1  <?xml version="1.0" encoding="UTF-8"?>
    ...
38  </dependencies>
39  <build>
40      <plugins>
41          <plugin>
42              <groupId>org.openjfx</groupId>
43              <artifactId>javafx-maven-plugin</artifactId>
44              <version>0.0.4</version>
45              <configuration>
- 46                  <mainClass>controller.Launcher</mainClass>
+ 47                  <mainClass>controller.JarLauncher</mainClass>
48              </configuration>
49          </plugin>
50          <plugin>
51              <groupId>org.apache.maven.plugins</groupId>
52              <artifactId>maven-shade-plugin</artifactId>
    ...
60          <configuration>
61              <transformers>
62                  <transformer implementation="org.apache.maven.plugins.shade
                    .resource.ManifestResourceTransformer">
- 63                      <mainClass>controller.Launcher</mainClass>
+ 64                      <mainClass>controller.JarLauncher</mainClass>
65                  </transformer>
    ...
```

**6.3.4 Exercise**

1. Using Maven to build an executable jar file of the case study of the first chapter.
2. Using Maven to build an executable jar file of the project of the second chapter's case study.
3. Use Maven to build an executable jar file of the case study of the third chapter.
4. Use Maven to build an executable jar file of the first case study of the fifth chapter.

# Appendix

**T**HIS chapter provides a tutorial of how to prepare the computing environment for coding throughout this handout. Specifically, the tutorial includes installing the current version of the Java SE development kits (JDK), IntelliJ Idea IDE, and JavaFX, at the time of writing, and how to configure them to work collaboratively.

## Installation

### JDK

The official JDK provided by Oracle is the recommended JDK alternative. It is made available on the Oracle website (<https://www.oracle.com/java/technologies/javase-jdk14-downloads.html>). All we have to do is to locate the word JDK Download and lookup for the operating system, which is the most closely related to what is being installed on the computer. For example, for one who is using a MacOS, select `jdk-14.0.2-osx-x64_bin.dmg` and download it. Incidentally, since the Oracle-provided JDK is stringent upon its licensing, we have to register an Oracle account before we can download it. Once the download is complete, double click the downloaded file and install it.

### IntelliJ Idea

The IntelliJ Idea IDE is used in the demonstration throughout this course. JetBrains provides it in at the url: <https://www.jetbrains.com/idea/>. Before download-

ing and installing it, the lecturer highly recommends everyone apply for a student license first through <https://www.jetbrains.com/community/education/#students>. At the time of writing, the license permits the license holder to access the ultimate version of the IDE, which has more rich in features provided. For more details on the feature differences, check the webpage: [https://www.jetbrains.com/idea/features/editions\\_comparison\\_matrix.html](https://www.jetbrains.com/idea/features/editions_comparison_matrix.html). Once more, after the installer is completely downloaded, install it.

## JavaFX

In the past JavaFX was bundled with JDK, until the release JDK 11 in 2018. Decided by Oracle, JavaFX was separated into one single project named OpenJDK. Oracle gave the main reason for the separation as to accelerate the pace of its development. The official site of the OpenJFX is <https://openjfx.io/>, where the entire SDK can be downloaded from the web page. At the time of writing, the latest released JavaFX version is also 14. Note that the downloaded JavaFX SDK does not contain an installer. All we have to do is to download it and store it somewhere we can easily retrieve it in the future. For example, simply store the downloaded file at home folder and extract it.

## Configuration

The configuration is somewhat tedious since the version that JavaFX is no longer part of the JDK. That is, we have to configure many components the same way we configure external libraries. The following in this section provides an example walkthrough to show how to configure a JavaFX project in the IntelliJ Idea IDE.

Let us begin by creating a new JavaFX project by opening the IntelliJ Idea IDE, locate *create a new project*, and select JavaFX. Importantly, we have to select the same JDK version as what we have just downloaded, e.g., an example shown in Figure 6.4. Let us name the

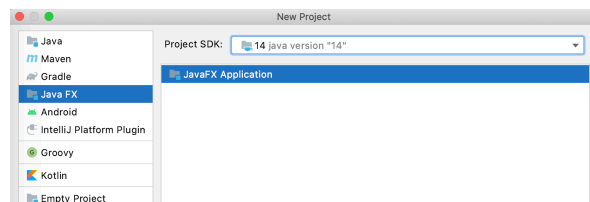


Figure 6.4: Selecting the JDK version

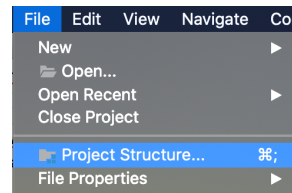
project HelloWorld and click *Finish*. We will see that the IDE has initialized a Java project structured as given below:

```

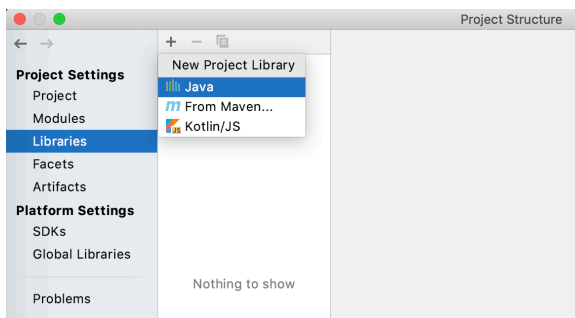
| out
|_ src
   |_ Controller.java
   |_ Main.Java
   |_ sample.fxml

```

If we examine the Main class, we will see that all the libraries are still missing, and this makes the IDE unable to recognize many classes referred to in the file. Thus, the next thing we have to do is to include all the necessary JavaFX libraries to the project dependency list. This can be done by using the *Project Structure* configurator, whose access is located at a menu item under *File* in the menu bar, i.e., as shown in Figure 6.5. There is also a shortcut to access it via its icon on the top right of the IDE toolbar. On the tab on the left-hand side, select *Libraries*, then click the plus sign, select *Java*, and locate to the folder containing the extracted the JavaFX SDK files.



**Figure 6.5: Project Structure menu item**



**Figure 6.6: Adding JavaFX dependencies**

the libraries. However, the IDE is still unable to run the program because the Java launcher requires an additional virtual machine configuration, which can be done in the following steps:

Then we select the *lib* folder as shown in Figure 6.6, click *Open*, and select *Ok*. This will add the *lib* folder to Class, Sources, and Native Library Locations attributes. After we click ok, the IDE will start indexing the dependencies. Once it is finished, we will see that the IDE can now recognize all



1. define a global variable named `PART_TO_FX` in the IDE Preferences configurator, which can be accessed by selecting the menu item named *Preferences* in the menu bar and locating the menu tab *Part Variables* on the left-hand side. Then in the *Add Variable* pop up, put the string `PART_TO_FX` for *Name*, and the full path to the `javafx-sdk-14.0.2.1/lib` folder for *Values*. Figure 6.7 shows an example of this step.

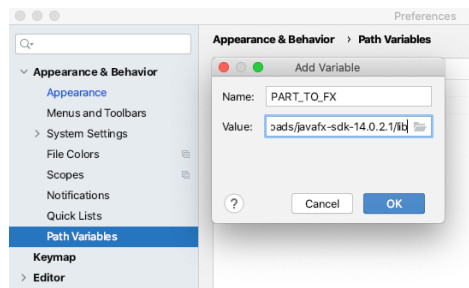


Figure 6.7: Configure the `PATH_TO_FX` global variable

2. configure the virtual machine parameter in the *Run/Debug Configurations* configurator by selecting the menu item named *Run* in the menu bar and choosing *Edit Configurations*. Then put the string `--module-path ${PATH_TO_FX} --add-modules javafx.controls,javafx.fxml` into the text-box next to the *VM options* label. Figure 6.8 shows an example of this step.

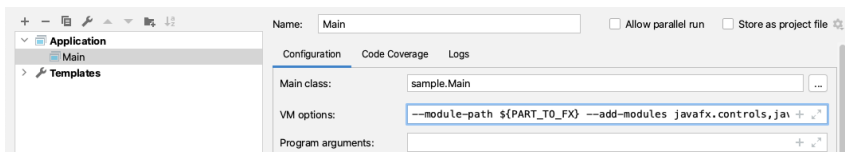


Figure 6.8: Configure the VM options

Once the above steps are completed, we should be able to run the application. However, for a MacOS user, the first time the IntelliJ Idea IDE executes a JavaFX project, the operating system will prevent the application from launching because

libraries may be subject to some security threats. This can be simply fixed by running the command `xattr -d com.apple.quarantine XXXXX/javafx-sdk-14.0.2.1/lib/*` in the terminal, where XXXXX is the parent directory of the `javafx-sdk-14.0.2.1` folder. Besides, it should also be noted that we have to configure the VM options for every JavaFX project up to Chapter 5 of this handout.

# Bibliography

- Apache ANT+Ivy (2017) The Apache ANT Project. Retrieved August, 2020, from <https://ant.apache.org/ivy/>
- apache-log4j-2.13.3-bin.zip (2020) Retrieved July, 2020, from <https://www.apache.org/dyn/closer.lua/logging/log4j/2.13.3/apache-log4j-2.13.3-bin.zip>
- Arduino-er (2015) Java/JavaFX/jSSC control Arduino + 8x8 LED Matrix. Retrieved June, 2020, from <http://arduino-er.blogspot.com/2015/09/javajavafxjssc-control-arduino-8x8-led.html>
- Armstead L (n.d.) Exemplo De Spritesheet - Sprite Sheet Megaman Png, Transparent Png. Retrieved June, 2020, from [https://www.kindpng.com/imgv/mwToi\\_exemplo-de-spritesheet-sprite-sheet-megaman-png-transparent/](https://www.kindpng.com/imgv/mwToi_exemplo-de-spritesheet-sprite-sheet-megaman-png-transparent/)
- Austen-Smith D, Diermeier D, Zemel E (2017) Unintended acceleration: Toyota's recall crisis. *Kellogg School of Management Cases.*, 1-16.
- Bloch A (2003) Murphy's law. Penguin.
- Docker (2020) Docker. Retrieved August, 2020, from <https://www.docker.com/>
- Git (2020) git –distributed-is-the-new-centralized. Retrieved August, 2020, from <https://git-scm.com/>
- Gradle (2020) Gradle Build Tool. Retrieved August, 2020, from <https://gradle.org/>
- Haase C, Guy R (2007) Filthy rich clients: Developing animated and graphical effects for desktop Java applications. Pearson Education.
- hamcrest-core-1.3.jar (2012) Retrieved July, 2020, from <https://search.maven.org/remotecontent?filepath=org/hamcrest/hamcrest-core/1.3/hamcrest-core-1.3.jar>
- IntelliJ Idea (2020) IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. Retrieved May, 2020, from <https://www.jetbrains.com/idea/>

- Java.awt (2018) Package java.awt. Retrieved May, 2020, from <https://docs.oracle.com/javase/10/docs/api/java/awt/package-summary.html>
- Java.util.stream (2020) Package java.util.stream. Retrieved May, 2020, from <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/stream/package-summary.html>
- JavaFX 14 (2020) Overview (JavaFX 14). Retrieved May, 2020, from <https://openjfx.io/javadoc/14/index.html>
- Javax.swing (2018) Package javax.swing. Retrieved May, 2020, from <https://docs.oracle.com/javase/10/docs/api/javax/swing/package-summary.htm>
- JavaFX (2020) JavaFX 14. Retrieved May, 2020, from <https://openjfx.io>
- Java T Point (2020) V-Model. Retrieved June, 2020, from <https://www.javatpoint.com/software-engineering-v-model>
- JUnit (2020) JUnit. Retrieved June, 2020, from <https://junit.org/junit4/junit-4.13.jar>
- junit-4.13.jar (2020) Retrieved July, 2020, from <https://search.maven.org/remotecontent?filepath=junit/junit/4.13/junit-4.13.jar>
- Kaner C (2009) Software Testing as a Quality Improvement Activity. *IEEE Computer Society Webinar Series*, 1-86.
- Le Lann G (1997) An analysis of the Ariane 5 flight 501 failure-a system engineering perspective *In Proc. of Intl. Conf. and Workshop on Engineering of Computer-Based Systems*, 339-346.
- Lin H (1985) The development of software for ballistic-missile defense. *Scientific American*, 253(6), 46-53.
- Log4j (2020) Log4j – Apache Log4j 2. Retrieved June, 2020, from <https://logging.apache.org/log4j>
- Maven (2020) Apache Maven Project. Retrieved August, 2020, from <https://maven.apache.org/>
- Mcintosh S, Adams B, Hassan AE (2012) The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5), 578-608.
- mvvmfx-testing-utils-1.7.0.jar (2018) Retrieved July, 2020, from <https://github.com/sialcasa/mvvmFX/releases/download/mvvmfx-1.7.0/mvvmfx-testing-utils-1.7.0.jar>

- Neumann PG (1986) Risks to the public in computer systems. *ACM SIGSOFT Softw. Eng. Notes*, 11(2), 2-14.
- ObservableList (2020) Interface ObservableList. Retrieved May, 2020, from <https://openjfx.io/javadoc/14/javafx.base/javafx/collections/ObservableList.html>
- Perry DE, Stieg CS. (1993). Software faults in evolving a large, real-time system: a case study. *In proc. of the European Software Engineering Conference*, 48-67.
- Phannachitta P (2020) On an Optimal Analogy-based Software Effort Estimation. *Inf. Softw. Technol.* 125, (to appear).
- Phannachitta P, Matsumoto K (2019). Model-based software effort estimation - a robust comparison of 14 algorithms widely used in the data science community. *Int. J. Innov. Comput. I.* 15(2), 569-589.
- Reddit (2020) Reddit: the front page of the internet. Retrieved May, 2020, from <https://www.reddit.com>
- Slf4j (2020) Simple Logging Facade for Java. Retrieved June, 2020, from <http://www.slf4j.org>
- slf4j-api-1.7.30.jar (2019) Retrieved July, 2020, from <https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.30/slf4j-api-1.7.30.jar>
- slf4j-simple-1.7.30.jar (2019) Retrieved July, 2020, from <https://repo1.maven.org/maven2/org/slf4j/slf4j-simple/1.7.30/slf4j-simple-1.7.30.jar>
- StackOverflow (2020) Stack Overflow - Where Developers Learn, Share, & Build Careers, Retrieved May, 2020, from <https://www.stackoverflow.com>
- Wong S (2017) Java GUI Programming Primer. Retrieved May, 2020, from <https://www.clear.rice.edu/comp310/JavaResources/GUI/>
- Zuo C, Wen H, Lin Z, Zhang Y (2019) Automatic fingerprinting of vulnerable iot devices with static uuids from mobile apps. *In Proc. of the ACM SIGSAC Conf. on Computer and Communications Security*, 1469-1483.

