1 глава

Установка Java

JavaFX, конечно, требует, чтобы у вас был установлен JDK. Получение необходимых зависимостей различается для разных версий Java.

JavaFX был представлен как часть выпуска Java 8. Однако позже он был удален из JDK и
перенесен в отдельный модуль в Java 11.

Это означает, что если вы ориентируетесь на Java 8-10, у вас есть все необходимые зависимости как часть вашего JDK уже. Ура! Если вы используете более новую версию Java — то есть 11+, вам нужно получить зависимости отдельно.

Maven

Получение и управление зависимостями вручную довольно неудобно, и в реальном приложении Вы бы редко так делали. Гораздо лучше использовать систему управления зависимостями, такую как Maven или Gradle. Таким образом, вы можете просто объявить, какие зависимости вы используете и какие версии, а система позаботится об остальном.

Архетип Maven

Конечно, вы можете настроить свой проект Maven вручную, с нуля. Тем не менее, вы можете предпочесть более удобный способ создания структуры и содержания базового проекта для вас с помощью Maven.

Maven имеет понятие архетипов, которое, по существу, означает, что вы можете создавать различные типы проекты из шаблона. Существуют многочисленные архетипы для различных типов проектов, и к счастью, есть пара для JavaFX. Архетип, который вы можете выбрать, зависит от того, какую версию Java вы используете.

Вы можете прочитать больше об архетипах Maven в следующем посте: Maven archetypes tutorial.

Архетип Java 8

Вы можете использовать com.zenjava:javafx-basic-archetype, или вы можете найти другие архетипы самостоятельно, если этот вам не подходит.

Вы можете легко сгенерировать проект из командной строки с помощью Maven, используя указанный выше архетип:

```
mvn archetype:generate -DarchetypeGroupId=com.zenjava -DarchetypeArtifactId=javafx-basic-archetype
```

Кроме того, вы можете создать новый проект Maven из архетипа прямо в вашей IDE.

Архетип Java 11

Для Java 11 вы можете использовать org.openjfx: javafx-archetype-simple.

Для создания проекта просто запустите:

```
mvn archetype:generate -DarchetypeGroupId=org.openjfx -DarchetypeArtifactId=javafx-archetype-simple
```

Ручная настройка Maven

Если вы хотите больше контроля, вы можете настроить свой проект Maven вручную, не генерируя его из архетипа.

Для начала вам понадобятся два компонента. Первый — Maven Plugin для Java FX.

Просто добавьте следующее в ваш pom.xml:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-maven-plugin</artifactId>
            <version>0.0.3</version>
            <configuration>
                <mainClass>com.example.App</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Обратите внимание, что

```
<mainclass>
```

должен указывать на ваш основной класс, который имеет метод main и расширяет javafx.application.Application. Мы расскажем об этом в следующей статье серии.

Вторая часть добавляет зависимость для элементов управления JavaFX:

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>11.0.2</version>
</dependency>
```

Gradle

В настоящее время Gradle не поддерживает создание проектов непосредственно из архетипов .

Вы можете использовать неофициальный плагин Gradle Archetype и использовать архетипы Maven, упомянутые выше.
Кроме того, вы можете создать свой проект с помощью Maven из архетипа, а затем преобразовать его в проект Gradle с помощью следующей команды в каталоге, содержащем ваш pom.xml:

```
gradle init
```

Ручная настройка Gradle

Как и в случае ручной настройки Maven, вам нужно добавить плагин JavaFX:

```
plugins {
    id 'application'
    id 'org.openjfx.javafxplugin' version '0.0.8'
}
```

И зависимость для элементов управления:

```
javafx {
    version = "11.0.2"
    modules = [ 'javafx.controls' ]
}
```

JavaFX SDK

Есть еще одна возможность использовать JavaFX локально. Вы можете скачать JavaFX SDK. Он содержит все необходимые библиотеки, которые вы можете затем связать с проектом в вашей IDE или добавить в classpath.

Это может быть полезно, когда вы не знакомы с Gradle или Maven и просто для локальной разработки.

При распределении вашего приложения это становится неудобным, так как вам нужно убедиться, что вы включили все необходимые зависимости.

С помощью этой опции вы можете сгенерировать проект без Maven/Gradle в вашей IDE, который содержит все необходимые файлы. В IntelliJ IDEA вы можете просто перейти на:

```
File → New → Project → JavaFX
```

2 глава

Структура приложения

Каждое приложение состоит из иерархии нескольких основных компонентов: Stages (окна), (сцены) и nodes (узлы). Давайте рассмотрим каждую из них.

Stage

Stage, по существу, представляет собой окно. Ваше приложение может иметь несколько компонент stage, но как минимум одна должна быть обязательно.

Scene

Scene отображает содержание сцены (stage). Каждая компонента stage может содержать несколько компонент scene, которые можно переключать. Представьте себе театральные подмостки, на которых сменяются несколько сцен во время спектакля.

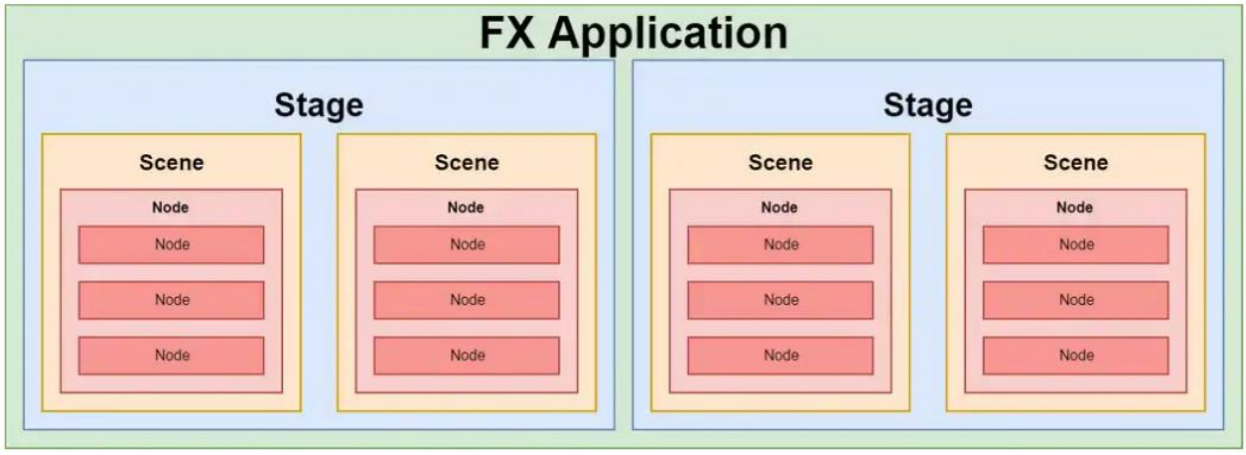
Node

Каждая компонента stage может содержать различные компоненты, называемые узлами (node). Узлы могут быть элементами управления, такими как кнопки или метки, или даже макетами (layout), которые могут содержать несколько вложенных компонентов. Каждая сцена (scene) может иметь один вложенный узел (node), но это может быть макет (layout), который может содержать несколько компонентов. Вложенность может быть многоуровневой — макеты могут содержать другие макеты и обычные компоненты.

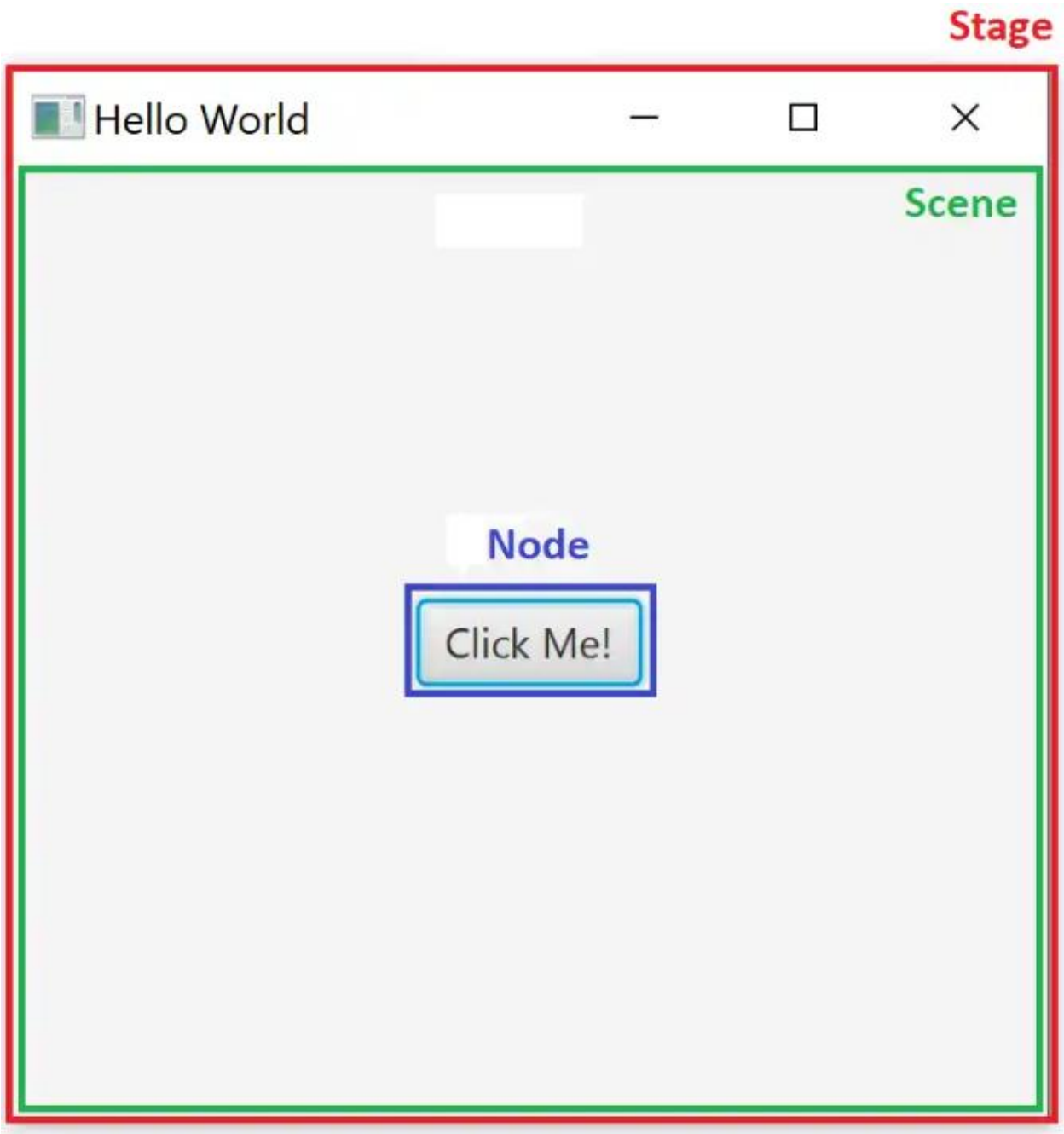
Резюме

Каждое приложение может иметь несколько окон (Stage). Каждая компонента Stage может переключать несколько Scene (сцен). Сцены содержат узлы (node) — макеты и обычные компоненты.

Эту иерархию можно визуализировать следующим образом:



Теперь давайте посмотрим на конкретный пример — реальное приложение.



Класс Application

Время начать программировать. Если вы следовали предыдущей статье, у вас уже готовы все необходимые зависимости.

Каждое приложение JavaFX должно иметь класс main, который расширяет класс:

```
javafx.application.Application
```

Кроме того, необходимо переопределить абстрактный метод из класса Application:

```
public void start(Stage primaryStage) throws Exception
```

Касс main выглядит примерно так:

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

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO implement me!
    }
}
```

Метод main

Для запуска JavaFx приложения не обязательно нужен метод main(). Вы можете упаковать исполняемый файл jar, используя JavaFX Packager Tool. Однако гораздо удобнее иметь метод main.

При этом приложение не только легче запустить, но в него можно как обычно передавать параметры командной строки.

Внутри метода main() приложение можно запустить используя метод:

```
Application.launch()
```

Легко заметить, что это статический метод в классе Application. Мы не указали основной класс, но JavaFX может определить это автоматически в зависимости от класса, который вызывает этот метод.

Настройка Stage

Теперь мы знаем как запустить наше приложение, используя метод main(). Тем не менее, ничего не произойдет, если мы сделаем это. Нужно окно, которое мы хотим показать. Окно называется stage, помните? На самом деле, у нас уже есть первичная stage, переданная в метод start в качестве входного параметра:

```
public void start (Stage primaryStage)
```

Мы можем использовать эту компоненту stage. Единственная проблема в том, что она скрыта по умолчанию. К счастью, мы можем легко показать ее, используя метод primaryStage.show():

```
@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.show();
}
```

Теперь, когда вы запустите приложение, вы должны увидеть следующее окно:

Не очень впечатляет, правда? Во-первых, давайте добавим хорошую подпись к нашему окну.

```
primaryStage.setTitle("Hello world Application");
```

Чтобы окно выглядело еще лучше, давайте добавим красивую иконку в верхнюю панель окна:

```
InputStream iconStream = getClass().getResourceAsStream("/icon.png");
Image image = new Image(iconStream);
primaryStage.getIcons().add(image);
```

Вы можете добавить несколько иконок, представляющих приложение. Точнее одну и ту же иконку разных размеров. Это позволит использовать иконку

походящего размера в зависимости от контекста приложения.

Теперь можно настроить свойства и поведение объекта Stage, например:

- Установить положение с помощью setX() и setY()
- Установить начальный размер с помощью setWidth() и setHeight()
- Ограничить максимальные размеры окна с помощью setMaxHeight() и setMaxWidth() или отключить изменение размера с помощью setResizable(false)
- Установить режим окно всегда сверху, используя setAlwaysOnTop()
- Установить полноэкранный режим, используя setFullScreen()
- И многое другое

Теперь у нас есть окно с таким фантастическим названием, но оно все еще пустое. Вы уже знаете, что нельзя добавлять компоненты непосредственно в Stage (окно). Вам нужна сцена (scene).

Однако для запуска конструктора сцены требуется указать дочерний узел. Давайте сначала создадим простую метку (label). Затем мы создаем сцену (scene) этой меткой (label) в качестве дочернего узла.

```
Label helloWorldLabel = new Label("Hello world!");
Scene primaryScene = new Scene(helloWorldLabel);
```

Вы заметили, что Scene допускает только один дочерний компонент. Что если нам нужно больше? Необходимо использовать layout (макет), который является компонентом, который может включать несколько дочерних элементов и размещать их на экране в зависимости от используемого layout (макета). Мы рассмотрим макеты позже в этой серии статей.

Чтобы сделать приложение немного более визуально привлекательным, давайте разместим метку в центре экрана.

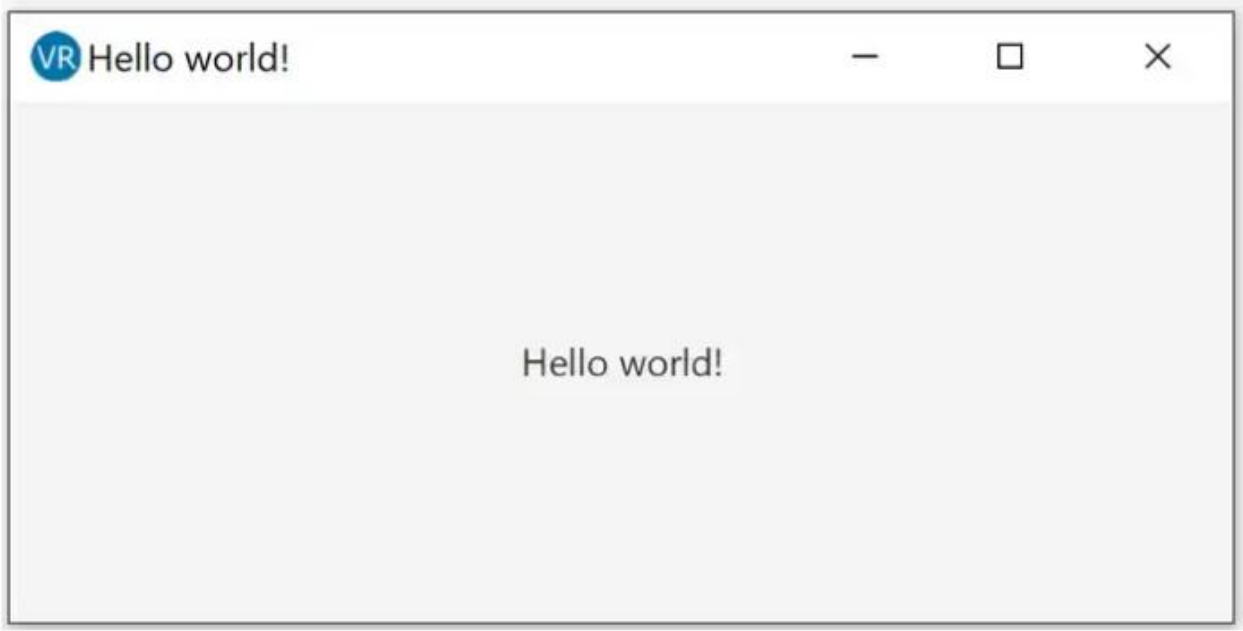
```
helloWorldLabel.setAlignment(Pos.CENTER);
```

Наконец, нам нужно установить Scene для Stage, которая у нас уже есть:

```
@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("Hello world Application");
    primaryStage.setWidth(300);
    primaryStage.setHeight(200);
```

```
    InputStream iconStream = getClass().getResourceAsStream("/icon.png");
    Image image = new Image(iconStream);
    primaryStage.getIcons().add(image);

    Label helloWorldLabel = new Label("Hello world!");
    helloWorldLabel.setAlignment(Pos.CENTER);
    Scene primaryScene = new Scene(helloWorldLabel);
    primaryStage.setScene(primaryScene);

    primaryStage.show();
}
```

Теперь наше окно содержит сцену с компонентом метка (label):



3 глава

Традиционный способ

В предыдущей статье мы создали простое приложение Hello World.

Просто напоминание — код выглядел так:

```
@Override
public void start(Stage primaryStage) throws Exception {
    primaryStage.setTitle("Hello world Application");
    primaryStage.setWidth(300);
    primaryStage.setHeight(200);

    InputStream iconStream = getClass().getResourceAsStream("/icon.png");
    Image image = new Image(iconStream);
    primaryStage.getIcons().add(image);

    Label helloWorldLabel = new Label("Hello world!");
```

```
    helloWorldLabel.setAlignment(Pos.CENTER);
    Scene primaryScene = new Scene(helloWorldLabel);
    primaryStage.setScene(primaryScene);

    primaryStage.show();
}
```

Как видите весь пользовательский интерфейс создан в Java коде.

Это очень простой пример, но по мере усложнения вашего приложения, когда приходится вводить несколько уровней вложенных макетов и множество компонентов, результирующий код может стать очень сложным для понимания. Однако это еще не все — в одном и том же классе присутствует код, который отвечает за структуру, визуальные эффекты и поведение одновременно.

У класса явно нет единой ответственности. Сравните это, например, с веб-интерфейсом, где каждая страница имеет четко разделенные задачи:

- HTML — это структура
- CSS — это визуальные эффекты
- JavaScript — это поведение

Представляем FXML

Очевидно, что иметь весь код в одном месте не очень хорошая идея. Вам нужно как-то структурировать его, чтобы его было легче понять и сделать более управляемым.

В действительности есть много шаблонов дизайна для этого. Как правило, в конечном итоге вы приходите к варианту «Model-View-Whatever» — это что-то вроде «Model View Controller», «Model View Presenter» или «Model View ViewModel».

Можно часами обсуждать плюсы и минусы разных вариантов — давайте не будем делать это здесь. Более важно то, что с JavaFx вы можете использовать любой из них.

Это возможно потому, что в дополнение к процедурной конструкции вашего пользовательского интерфейса вы можете использовать декларативную разметку XML.

Оказывается иерархическая структура XML — это отличный способ описать иерархию компонентов в пользовательском интерфейсе. HTML работает достаточно хорошо, верно?

Формат XML, специфичный для JavaFX, называется FXML. В нем вы можете определить все компоненты приложения и их свойства, а также связать их с контроллером, который отвечает за управление взаимодействиями.

Загрузка FXML файлов

Итак, как мы можем изменить наш метод запуска для работы с FXML?

```
FXMLLoader loader = new FXMLLoader();
URL xmlUrl = getClass().getResource("/mainScene.fxml");
loader.setLocation(xmlUrl);
Parent root = loader.load();

primaryStage.setScene(new Scene(root));
primaryStage.show();
```

Здесь *root* представляет корневой компонент вашего пользовательского интерфейса, остальные компоненты вложены в него.

Метод *load* имеет generic возвращаемое значение, поэтому вы можете указать конкретный тип, а не *Parent*. Далее, вы получаете доступ к компонентно-ориентированным методам. Однако, это делает ваш код более хрупким. Если вы измените тип корневого компонента в вашем FXML, приложение может перестать работать во время выполнения, но при этом во время компиляции не будет ошибок. Это происходит потому, что теперь есть несоответствие типа, объявленного в вашем FXML и в загрузчике Java FXML.

Создание FXML файла

Теперь мы знаем, как загрузить файл FXML, но нам все еще нужно его создать. Файл должен иметь расширение .fxml. В Maven проекте, вы можете поместить этот файл в папку ресурсов или FXMLLoader может загрузить его с внешнего URL-адреса.

После создадания файла в его первой строке необходимо ввести декларацию XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Импорт

Прежде чем добавить отдельные компоненты в файл, необходимо убедиться, что они правильно распознаются. Для этого необходимо добавить операторы импорта. Это очень похоже на импорт в Java классах. Вы можете импортировать отдельные классы или использовать знаки подстановки как обычно. Давайте рассмотрим пример раздела импорта:

```
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
```

Хорошей новостью является то, что вместо добавления всех операторов импорта вручную, ваша IDE должна помочь вам добавить импорт аналогично добавлению их в классы Java.

Добавление компонентов

Теперь пришло время добавить некоторые компоненты. В предыдущей статье мы узнали, что каждая сцена может иметь только один дочерний компонент. Для начала давайте добавим простую метку (label):

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>

<!--Допускается только один компонент на корневом уровне-->
<Label>Hello World!</Label>
```

Конечно, метка в качестве корневого компонента — это не очень реалистичный пример. Обычно предпочтительнее использовать какой-то макет (layout), который является контейнером для нескольких компонентов и организует их расположение. Мы рассмотрим макеты позже в этой серии, а сейчас давайте просто воспользуемся простым VBox, который размещает свои дочерние элементы вертикально друг над другом.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Button?>

<VBox>
   <Label text="Hello world!"/>
   <Label text="This is a simple demo application."/>
   <Button text="Click me!"/>
</VBox>
```

FX Namespace

Существует пара элементов и атрибутов FXML, которые по умолчанию недоступны. Вам нужно добавить пространство имен (Namespace) FXML, чтобы

сделать их доступными. Его необходимо добавить к корневому компоненту:

```
<?xml version="1.0" encoding="UTF-8"?>
   ...

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

Теперь можно использовать новые элементы из пространства имен fx. Давайте попробуем добавить уникальные идентификаторы в наши компоненты:

```
<Label fx:id="mainTitle" text="Hello world!"/>
```

Атрибут *fx:id* является уникальным идентификатором компонента, который можно использовать для ссылки на компонент из других частей нашего FXML и даже из нашего контроллера.

Скрипты

Наше приложение пока статично. Есть несколько меток и кнопка, но приложение не делает ничего динамического.

Давайте отреагируем на нажатие нашей кнопки и изменим заголовок с «Click me!» на «Click me again!».

Первое, что нужно сделать, это добавить обработчик события onAction для нашей кнопки.

```
<Button fx:id="mainButton" text="Click me!" onAction="buttonClicked()"/>
```

Обратите внимание на fx:id, это идентификатор, который будет использоваться позже для ссылки на кнопку.

Теперь нужно предоставить функцию, которая будет вызвана для обработки события. Ее можно определить внутри тега fx:script. Важно то, что вы можете использовать различные языки для написания скрипта, JavaScript, Groovy или Clojure. Давайте посмотрим пример на JavaScript:

```
<fx:script>
function buttonClicked() {
    mainButton.setText("Click me again!");
}
</fx:script>
```

Заметьте, что мы ссылаемся на наш компонент Button с помощью идентификатора mainButton, который был объявлен так:

```
fx:id = "mainButton"
```

Также необходимо указать, какой язык сценариев вы используете в файле FXML:

```
<?language javascript?>
```

Давайте рассмотрим полный текст примера:

```
<?xml version="1.0" encoding="UTF-8"?>
<?language javascript?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Button?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml">
    <Label fx:id="mainTitle" text="Hello world!"/>
    <Label fx:id="subTitle" text="This is a simple demo application."/>
    <Button fx:id="mainButton" text="Click me!" onAction="buttonClicked()"/>
    <fx:script>
        function buttonClicked() {
            mainButton.setText("Click me again!")
        }
    </fx:script>
</VBox>
```

Должен ли я использовать это?

В приведенном выше примере показано, как ссылаться на компоненты с помощью *fx:id* и как добавить простое поведение с помощью скрипта на JavaScript. Неужели это то, что вы должны на самом деле делать?

Ответ — в большинстве случаев нет. Есть несколько проблем с таким подходом.

Причина, по которой введен FXML, была разделение интересов — чтобы отделить структуру и поведение пользовательского интерфейса. В этом скрипте снова вернулось поведение слитное со структурой пользовательского интерфейса. Более того, поскольку мы больше не работаем с кодом Java, а с XML, были утрачены все проверки кода во время компиляции и безопасность типов. Теперь все проблемы в приложении будут обнаружены во время выполнения, а не во время компиляции. Приложение стало очень хрупким и подверженным ошибкам.

Добавление контроллера

Итак, что можно сделать, чтобы получить четкое разделение интересов? Можно связать контроллер с нашим файлом FXML. Контроллер — это Java класс, который отвечает в приложении за обработку поведения и взаимодействия с пользователем. Таким образом можно вернуть безопасность типов и проверки времени компиляции.

Контроллер является POJO, он не должен расширять или реализовывать что-либо, а также не должен иметь никаких специальных аннотаций.

Как можно связать класс контроллера с нашим FXML? По существу, есть два варианта.

На Java

Вы можете создать экземпляр контроллера самостоятельно или использовать любые другие способы создания экземпляра, такие как инъекция зависимости. Затем просто загрузите вашим *FXMLLoader*.

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

В FXML

Вы можете указать класс вашего контроллера как атрибут *fx:controller*, который должен находиться в корневом компоненте.

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

Если вы объявляете свой класс Controller в FXML, он автоматически создается

для вас. Этот подход имеет одно ограничение — в контроллере нужно создать конструктор без аргументов, чтобы позволит легко создавать новый экземпляр класса Controller.

Для получения доступа к экземпляру контроллера, созданного автоматически, можно использовать загрузчик FXML:

```
FXMLLoader loader = new FXMLLoader();
loader.setLocation(getClass().getResource("/mainScene.fxml"));
MainSceneController controller = loader.getController();
```

Вызов методов контроллера

Теперь, когда имеется контроллер, можно удалить скрипт и реализовть логику нажатия кнопок прямо в контроллере:

```
public class MainSceneController {

    public void buttonClicked() {
        System.out.println("Button clicked!");
    }
}
```

Следующим шагом является регистрация вызова этого метода в качестве обработчика события *onAction* нашей кнопки. Чтобы ссылаться на методы из нашего контроллера, нам нужно использовать знак # перед именем метода:

```
<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml" fx:controller="com.vojtechruzicka.MainSceneController">
    <Label fx:id="mainTitle" text="Hello world!"/>
    <Label fx:id="subTitle" text="This is a simple demo application."/>
    <Button fx:id="mainButton" text="Click me!" onAction="#buttonClicked"/>
</VBox>
```

При нажатии на кнопку, она вызывает метод *MainSceneController.buttonClicked()*. Имейте в виду, что это работает, только если метод объявлен public. Если модификатор доступа более строгий, необходимо аннотировать метод аннотацией *@FXML*.

```
@FXML
private void buttonClicked() {
    System.out.println("Button clicked!");
```

```
}
```

Внедрение компонентов в контроллер

Пока что мы просто печатаем на консоль. Что если мы снова захотим изменить текст нашей кнопки на «*Click me again*»? Как мы можем получить ссылки на компоненты в нашем контроллере?

К счастью, это легко. Помните эти атрибуты *fx:id*?

```
<Button fx:id="mainButton" text="Click me!" onAction="#buttonClicked"/>
```

JavaFX пытается автоматически сопоставить компоненты с *fx:id* с полями определенным в вашем контроллере с тем же именем.

Предположим, у нас есть кнопка описанная выше с

```
fx:id="mainButton"
```

JavaFX пытается внедрить объект кнопки в ваш контроллер в поле с именем *mainButton*:

```
public class MainSceneController {

    //Вот внедренный компонент с fx:id = "mainButton"
    @FXML
    private Button mainButton;
}
```

Как и в предыдущих методах, ваши поля должны быть *public* или аннотированными *@FXML*.

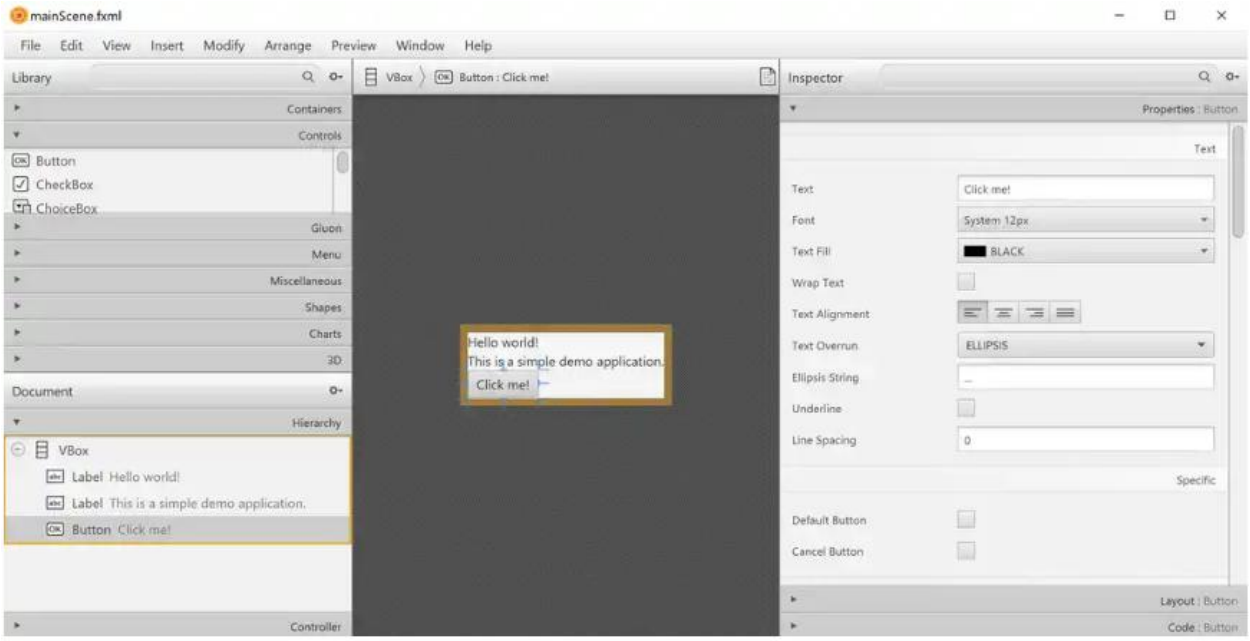
Теперь, когда у нас есть ссылка на нашу кнопку, можно легко изменить ее текст:

```
public class MainSceneController {

    @FXML
    private Button mainButton;

    @FXML
    private void buttonClicked() {
```

```
        mainButton.setText("Click me again!");
    }
}
```

Scene Builder

Написание вашей структуры GUI в XML может быть более естественным, чем в Java (особенно если вы знакомы с HTML). Тем не менее, до сих пор это не очень удобно. Хорошей новостью является то, что существует официальный инструмент под названием Scene Builder, который поможет вам в создании пользовательского интерфейса. В двух словах, это графический редактор для вашего графического интерфейса.



В редакторе имеется три основных области:

1. В левой части отображаются доступные компоненты, которые можно перетащить в среднюю часть. Она также содержит иерархию всех компонентов в вашем пользовательском интерфейсе, поэтому вы можете легко перемещаться по ней.
2. Средняя часть — это ваше приложение, отображаемое на основе вашего файла FXML.
3. Справа находится инспектор текущих компонентов. Здесь вы можете редактировать различные свойства выбранного текущего компонента. Любой компонент, выбранный в средней части иерархии, отображается в инспекторе.

Standalone

Scene Builder можно загрузить как отдельное приложение, которое можно использовать для редактирования FXML файлов.

Интеграция с IntelliJ IDEA

В качестве альтернативы, Scene Builder предлагает интеграцию с IDE.

В IntelliJ IDEA вы можете нажать правой кнопкой мыши на любом FXML файле и затем выбрать опцию меню «Открыть» в SceneBuilder.

В качестве альтернативы, IntelliJ IDEA интегрирует SceneBuilder непосредственно в IDE. Если вы откроете файл FXML в IDEA, в нижней части экрана появятся две вкладки

- Текст
- SceneBuilder

Для каждого файла FXML вы можете легко переключаться между редактированием файла FXML напрямую или через SceneBuilder.

В IntelliJ IDEA можто настроить расположение исполняемого файла SceneBuilder:

Settings → Languages & Frameworks → JavaFX → Path to SceneBuilder

4 глава

Макеты

Макет (Layout) является контейнером для компонентов. Макеты полезны тем, что вы можете позиционировать этот контейнер как целое независимо от того, какие компоненты находятся внутри. Более того, каждая сцена может содержать только один компонент, поэтому вам нужен макет в качестве корневого компонента для вашей сцены, чтобы вы могли разместить все необходимые компоненты сцены. Конечно, одного макета обычно недостаточно, но вы можете поместить один макет внутрь другого.

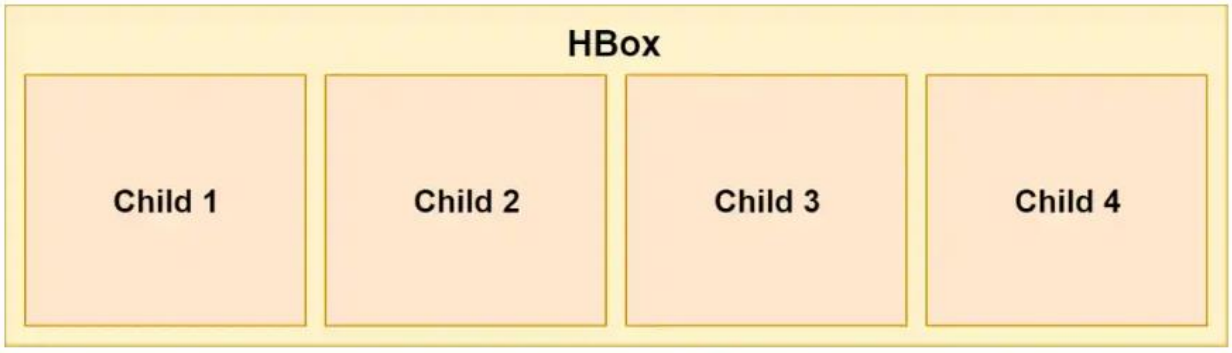
В дополнение к этому макеты также организуют и размещают ваши компоненты внутри себя. В зависимости от используемого макета дочерние компоненты могут быть расположены:

- Один за другим по горизонтали
- Один за другим по вертикали
- Друг над другом как стек
- В сетке

Есть еще много вариантов. Важно то, что макет автоматически обновляет положение своих дочерних элементов при изменении его размера. Таким образом, можно иметь согласованный макет, даже при изменении размера окна приложения пользователем.

HBox

Это один из самых простых среди имеющихся макетов. Он просто помещает все предметы по горизонтали в ряд, один за другим, слева направо.



В FXML вы можете использовать HBox следующим образом:

```
<HBox>
   <Button>1</Button>
   <Button>2</Button>
   <Button>3</Button>
   <Button>4</Button>
</HBox>
```

В Java вы можете использовать этот код:

```
HBox hbox = new HBox();
Button btn1 = new Button("1");
Button btn2 = new Button("2");
Button btn3 = new Button("3");
Button btn4 = new Button("4");
hbox.getChildren().addAll(btn1, btn2, btn3, btn4);
```

Spacing

Наши элементы теперь аккуратно разложены в ряд, один за другим:

Однако такой вариант не очень хорош, так как элементы расположены друг за другом без промежутков. К счастью, мы можем определить промежуток между компонентами, используя свойство spacing HBox:

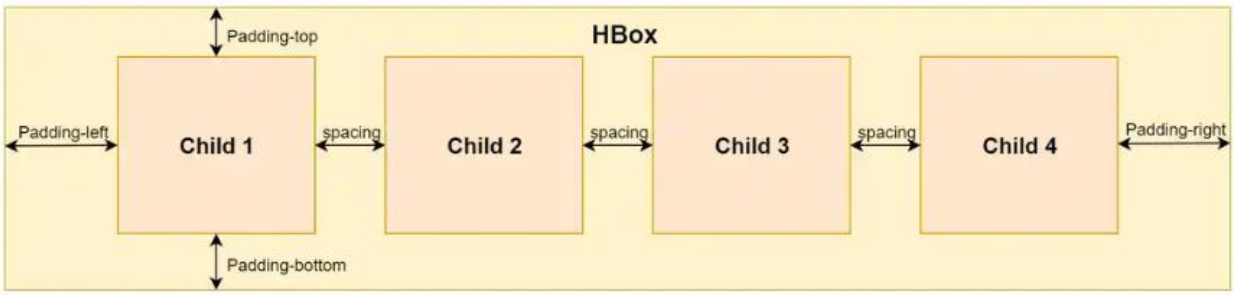
```
<HBox spacing="10">
    ...
</HBox>
```

Или в Java с помощью *setSpacing()*:

```
HBox hbox = new HBox();
hbox.setSpacing(10);
```

Padding

Элементы теперь расположены правильно, однако между элементами и самим HBox по-прежнему нет отступов. Может быть полезно добавить Padding (заполнение) в наш HBox:



Вы можете указать каждую область для заполнения отдельно — верхняя, нижняя, левая и правая.

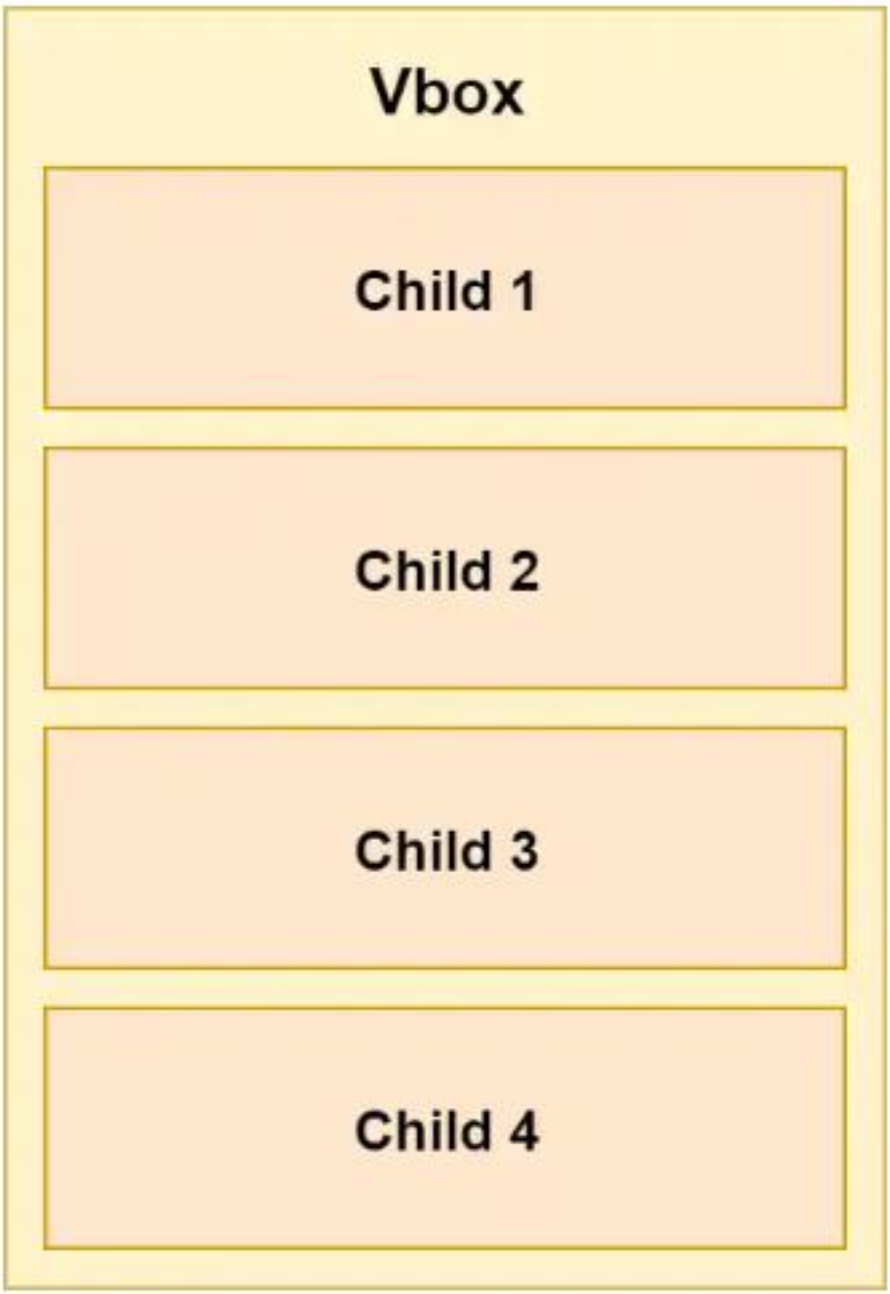
```
<HBox>
   <padding>
      <Insets top="10" bottom="10" left="10" right="10"/>
   </padding>
   ...
</HBox>
```

То же самое может быть сделано в Java:

```
HBox hbox = new HBox();
hbox.setPadding(new Insets(10, 10, 10, 10));
```

VBox

VBox очень похож на HBox, но вместо того, чтобы отображать внутренние компоненты по горизонтали друг за другом, он отображает их вертикально в столбец:



Вы по-прежнему можете устанавливать свойства spacing и padding так же, как в HBox.

В коде VBox используется точно так же, как HBox, только имя другое:

```
<VBox spacing="10">
   <padding>
      <Insets top="10" bottom="10" left="10" right="10"/>
   </padding>
   <Button>1</Button>
   <Button>2</Button>
   <Button>3</Button>
   <Button>4</Button>
</VBox>
```

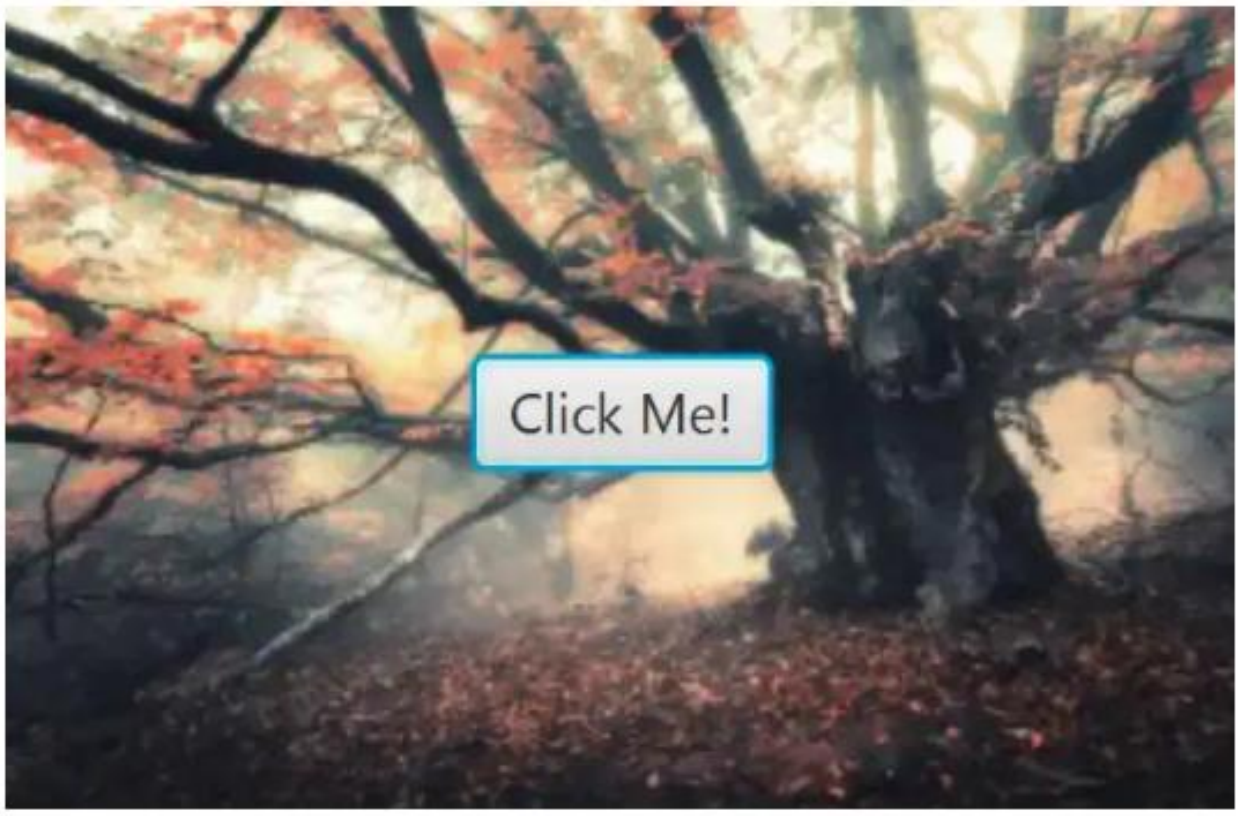
И на Java:

```
VBox vbox = new VBox();
vbox.setPadding(new Insets(10, 10, 10, 10));
vbox.setSpacing(10);
Button btn1 = new Button("1");
Button btn2 = new Button("2");
Button btn3 = new Button("3");
Button btn4 = new Button("4");
vbox.getChildren().addAll(btn1, btn2, btn3, btn4);
```

StackPane

Этот макет полезен для размещения его компонентов друг над другом. Порядок вставки определяет порядок элементов. Это означает, что первый элемент находится внизу, следующий — сверху, и так далее.

Это может быть полезно, например, для того, чтобы в макете показать рисунок и поверх него отобразить какой-либо текст или кнопку.



В следующем примере используется StackPane в FXML:

```
<StackPane>
```

```
    <ImageView>
        <Image url="/image.jpg"/>
    </ImageView>
    <Button>Click Me!</Button>
</StackPane>
```

Тот же пример на Java:

```
StackPane stackPane = new StackPane();
Image image = new Image(getClass().getResourceAsStream("/image.jpg"));
ImageView imageView = new ImageView(image);
Button btn = new Button("Click Me!");
stackPane.getChildren().addAll(imageView, btn);
```

Выравнивание элементов

Вы можете установить выравнивание элементов в стеке, чтобы лучше организовать их расположение:

```
<StackPane alignment="BOTTOM_CENTER">
 ...
</StackPane>
```

Конечно, вы можете сделать то же самое в Java:

```
StackPane stackPane = new StackPane();
stackPane.setAlignment(Pos.BOTTOM_CENTER);
```

Margin

Если вы хотите еще более детально контролировать расположение элементов, вы можете установить поля (margin) для отдельных элементов в стеке:

```
<StackPane alignment="BOTTOM_CENTER">
    <ImageView>
        <Image url="/image.jpg"/>
    </ImageView>
    <Button>
        <StackPane.margin>
            <Insets bottom="10"/>
        </StackPane.margin>
```

```
        Click Me!
    </Button>
</StackPane>
```

Или на Java:

```
StackPane stackPane = new StackPane();
Button btn = new Button("Click Me!");
stackPane.getChildren().add(btn);
StackPane.setMargin(btn, new Insets(0,0,10,0));
```

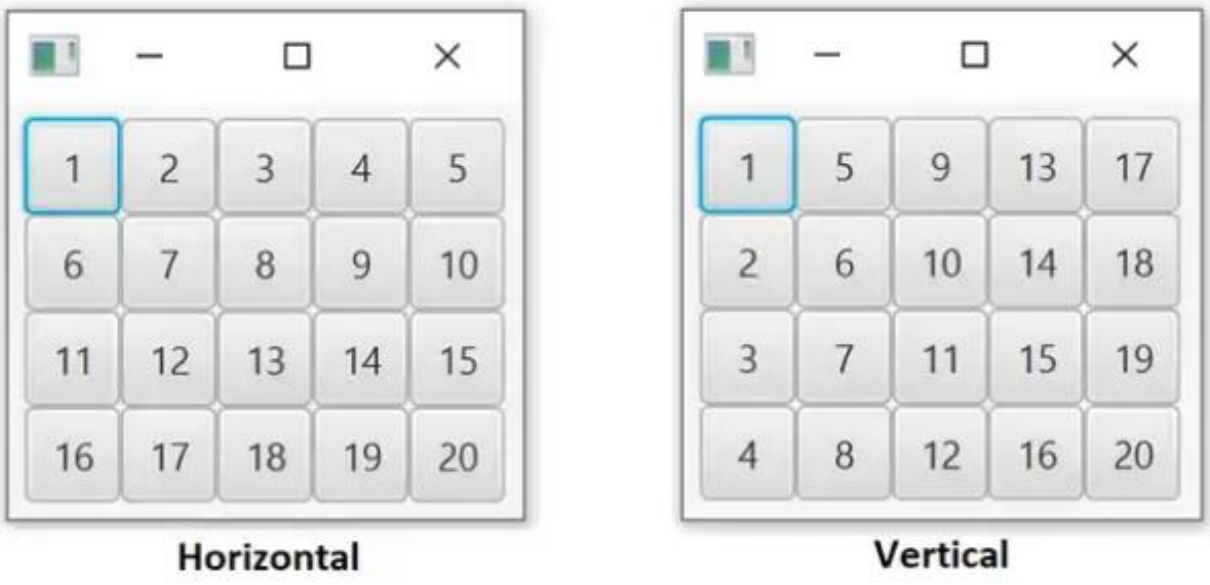
FlowPane

Панель Flow может работать в двух режимах — горизонтальном (по умолчанию) или вертикальном.

В горизонтальном режиме элементы отображаются горизонтально, один за другим, как в HBox. Разница в том, что, когда горизонтального пространства больше нет, оно переносится в следующий ряд под первым и продолжается снова. Таким образом, может быть много строк, а не только один, как в HBox.

Вертикальный режим очень похож, но (подобно VBox) он отображает элементы вертикально, сверху вниз. Когда места больше нет, он добавляет еще один столбец и продолжает.

Следующий рисунок иллюстрирует эти два режима:



Заметим, что элементы не обязательно должны иметь такой же размер, как на изображении выше.

Обратите внимание, как пересчитывается положение компонентов, если вы измените размер контейнера:

Вы можете установить внутреннее заполнение этого макета так же, как для HBox и VBox. Однако использование свойства spacing немного отличается. Вместо одного свойства для spacing, вам нужно иметь отдельные горизонтальные и вертикальные свойства spacing, так как элементы могут отображаться в нескольких строках / столбцах. Для горизонтального свойства spacing используйте *hgap*, для вертикального — *vgap*.

```
FlowPane flowPane = new FlowPane();
flowPane.setOrientation(Orientation.VERTICAL);
flowPane.setVgap(10);
flowPane.setHgap(10);
flowPane.getChildren().addAll(...);
```

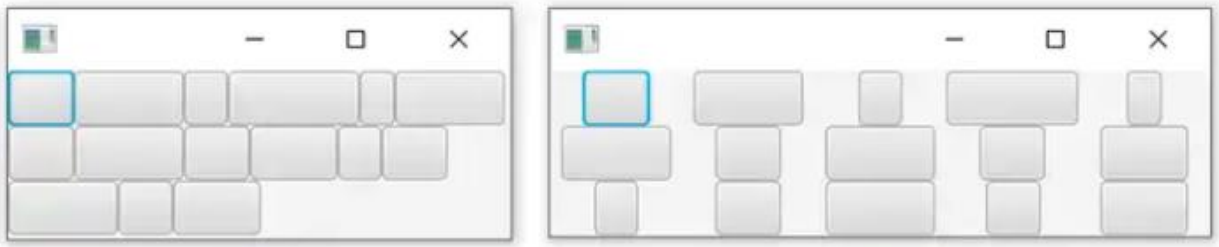
Пример FlowPane в FXML:

```
<FlowPane hgap="10" vgap="10" orientation="VERTICAL">
  ...
</FlowPane>
```

TilePane

Этот макет очень похож на FlowPane. Его способ отображения компонент практически идентичен. Вы все еще можете использовать горизонтальный или вертикальный режим и определять *vgap* и *hgap*.

Одно из важных различий заключается в размерах клеток. FlowPane назначает только пространство, необходимое для каждого компонента. TilePane, с другой стороны, делает размер всех ячеек одинаковым в зависимости от самого большого элемента. Таким образом, все элементы управления хорошо выровнены в строках / столбцах.



На изображении выше одинаковые компоненты размещены одинаковым образом, но вы легко можете заметить разницу.

FlowPane размещает элементы управления один за другим, без лишних интервалов
TilePane помещает элементы управления в ячейки одинакового размера на основе самого большого элемента.

Создание TilePane ничем не отличается от FlowPane, за исключением названия.

```
<TilePane vgap="10" hgap="10" orientation="VERTICAL" >
```

То же самое на Java:

```
TilePane tilePane = new TilePane();
tilePane.setVgap(10);
tilePane.setHgap(10);
tilePane.setOrientation(Orientation.VERTICAL);
```

5 глава

AnchorPane

AnchorPane — интересный и мощный макет. Это позволяет вам определять точки привязки (якоря) к компонентам внутри макета. Существует 4 типа якорей (anchor):

- top (верх)

- bottom (низ)
- left (лево)
- right (право)

Каждый компонент может иметь любую комбинацию якорей. От нуля до всех четырех.

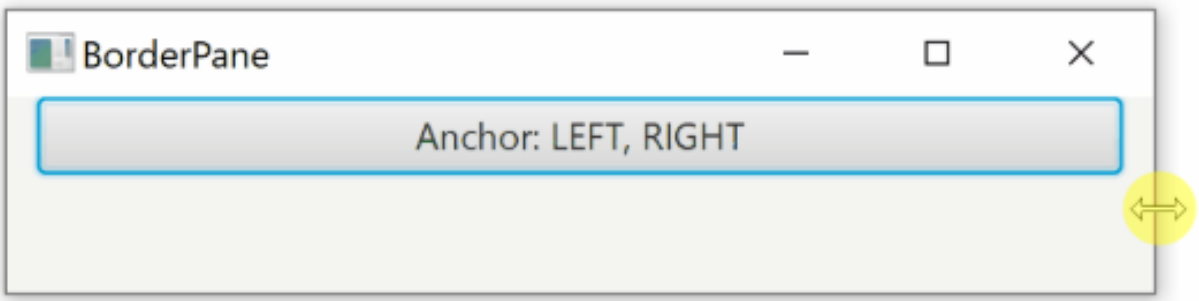
Привязка (anchoring) компонента означает, что он сохраняет определенное расстояние от определенного края макета (например, TOP). Это расстояние сохраняется даже при изменении размера макета.

Например: *anchorRight*=10 означает, что компонент будет сохранять расстояние 10 от правого края макета.

Вы можете указать две точки привязки (anchor), которые не находятся в противоположных направлениях, чтобы привязать ваш компонент к определенному углу макета.

Привязка *TOP = 10*, *RIGHT = 10* означает, что компонент будет оставаться в верхнем правом углу макета на расстоянии 10 от обоих краев.

В приведенном выше примере размер каждого компонента остается неизменным при изменении размера окна. Однако, если вы определяете точки привязки в противоположных направлениях, вы можете увеличивать / уменьшать свой компонент при изменении размера окна.



Вы можете использовать различные комбинации привязки, например:

- LEFT + RIGHT изменяет размер по горизонтали
- TOP + BOTTOM изменяет размеры по вертикали
- Указание всех 4-х якорей означает как горизонтальное, так и вертикальное изменение размеров компонента

Указать точки привязки в FXML легко. В следующем примере есть все четыре якоря, но вы можете включить только те, которые вы хотите, или не включать ни одного.

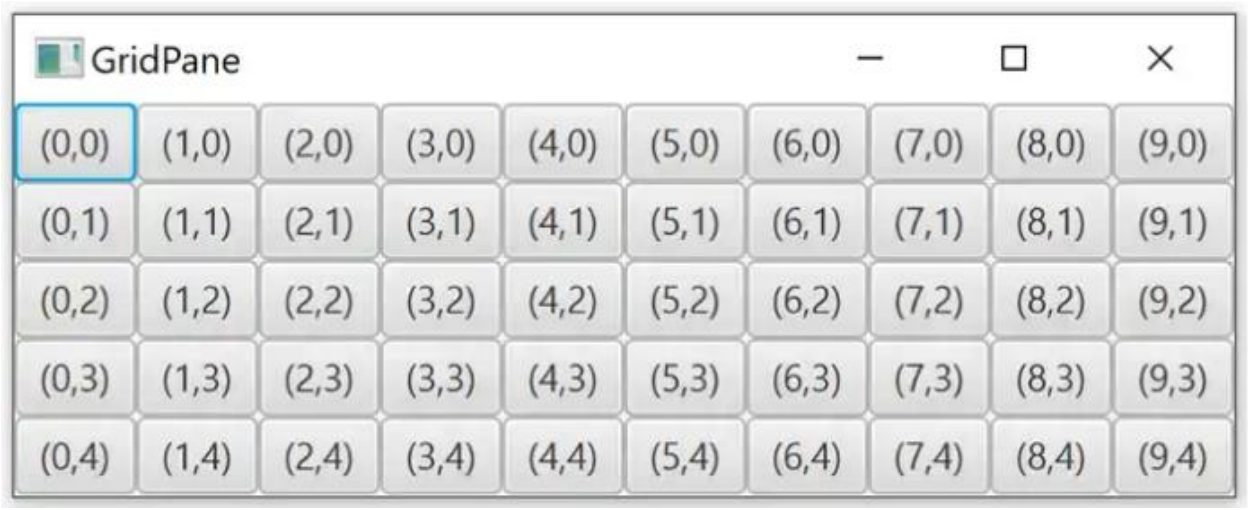
```
<AnchorPane>
   <Button AnchorPane.topAnchor="10"
           AnchorPane.leftAnchor="10"
           AnchorPane.rightAnchor="10"
           AnchorPane.bottomAnchor="10">I am fully anchored!</Button>
</AnchorPane>
```

Теперь давайте рассмотрим, как реализуется привязка в Java:

```
AnchorPane anchorPane = new AnchorPane();
Button button = new Button("I am fully anchored!");
AnchorPane.setTopAnchor(button, 10d);
AnchorPane.setBottomAnchor(button, 10d);
AnchorPane.setLeftAnchor(button, 10d);
AnchorPane.setRightAnchor(button, 10d);
anchorPane.getChildren().add(button);
```

GridPane

GridPane — это макет, который позволяет вам расположить ваши компоненты в виде таблицы. В отличие от TilePane, которая добавляет компоненты один за другим, здесь вам при добавлении каждого нового компонента нужно указать координаты его местоположения в вашей таблице.




```
<GridPane hgap="10" vgap="10">
   <Label GridPane.rowIndex="0" GridPane.columnIndex="0">First</Label>
   ...
</GridPane>
```

В Java при добавлении нового компонента мы сначала указываем параметр *ColumnIndex(x)* и затем *RowIndex(y)*.

```
   GridPane grid = new GridPane();
   grid.add(new Label("Hello!"), columnIndex, rowIndex);
```

Spacing

По умолчанию ячейки таблицы не имеют промежутков. Компоненты находятся рядом друг с другом, без интервалов. Интервал может быть определен отдельно для строк и столбцов, то есть горизонтально и вертикально.

- *hgap* устанавливает горизонтальный интервал (между столбцами)
- *vgap* устанавливает вертикальный интервал (между рядами)

```
<GridPane hgap="10" vgap="10">
   ...
</GridPane>
```

Интервал (Spacing), определенный в Java:

```
GridPane grid = new GridPane();
```

```
grid.setHgap(10);
grid.setVgap(10);
```

Интервал (Spacing) для нескольких ячеек

Компоненты в GridPane могут занимать несколько строк и / или столбцов. Компонент с rowspan расширяется до основания от своей исходной ячейки. Компонент с colspan расширяется вправо.

```
<GridPane>
    <Label GridPane.columnSpan="2"
           GridPane.rowSpan="2"
           GridPane.rowIndex="0"
           GridPane.columnIndex="0">
        Foo!
    </Label>
</GridPane>
```

В Java возможны два способа установки параметров rowSpan и columnSpan. Вы можете установить их непосредственно при добавлении компонента в таблицу:

```
grid.add(component, columnIndex, rowIndex, columnSpan, rowSpan);
```

Или через GridPane:

```
GridPane.setColumnSpan(component, columnSpan);
GridPane.setRowSpan(component, rowSpan);
```

Sizing (Определение размеров)

Хотя в первоначальном примере все ячейки были одинакового размера, это не обязательно должно быть так. Размеры ячеек таблицы определяются следующим образом:

- Высота каждой строки равна наибольшему элементу в строке
- Ширина каждого столбца равна самому широкому элементу в столбце

Ограничения столбцов и строк

Как уже упоминалось, по умолчанию размеры столбцов и строк основаны на компонентах внутри панели. К счастью, есть возможность лучше контролировать

размеры отдельных столбцов и строк.

Для этого используются классы *ColumnContstraints* и *RowConstraints*.

По сути у вас есть два варианта. Либо задайте процент доступного пространства для отдельных строк и столбцов, либо задайте предпочтительную ширину / высоту. В последнем случае вы также можете определить предпочтительное поведение при изменении размеров столбцов и строк.

Процент

Это довольно просто. Вы можете установить процент от доступного пространства, которое будет занято данной строкой или столбцом. При изменении размера макета строки и столбцы также изменяются в соответствии с новым размером.

```
<GridPane>
   <columnConstraints>
      <ColumnConstraints percentWidth="50" />
      <ColumnConstraints percentWidth="50" />
   </columnConstraints>
   <rowConstraints>
      <RowConstraints percentHeight="50" />
      <RowConstraints percentHeight="50" />
   </rowConstraints>
   ...
</GridPane>
```

Тот же пример на Java:

```
GridPane gridPane = new GridPane();

ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth(50);
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth(50);
gridPane.getColumnConstraints().addAll(col1, col2);

RowConstraints row1 = new RowConstraints();
row1.setPercentHeight(50);
RowConstraints row2 = new RowConstraints();
row2.setPercentHeight(50);
gridPane.getRowConstraints().addAll(row1, row2);
```

Абсолютный размер

Вместо определения размера в процентах вы можете определить предпочтительный и минимальный размер. Кроме того, вы можете указать, как строка / столбец должны вести себя при изменении размера макета. Столбцы

используют свойство *hgrow*, а строки имеют свойство *vgrow*.

Эти свойства могут иметь три разных значения.

- *NEVER* (НИКОГДА): никогда не увеличивается и не уменьшается при изменении размера макета. Значение по умолчанию.
- *ALWAYS* (ВСЕГДА): при изменении размера макета все элементы с этим значением либо растягиваются, чтобы заполнить доступное пространство, либо сокращаются.
- SOMETIMES (ИНОГДА): Размер этих элементов изменяется только в том случае, если нет других элементов.

```
<GridPane>
   <columnConstraints>
      <ColumnConstraints minWidth="50" prefWidth="100" />
      <ColumnConstraints minWidth="50" prefWidth="100" hgrow="SOMETIMES" />
   </columnConstraints>
   <rowConstraints>
      <RowConstraints minHeight="50" prefHeight="100" />
      <RowConstraints minHeight="50" prefHeight="100" vgrow="SOMETIMES" />
   </rowConstraints>
   ...
</GridPane>
```

Тот же пример на Java:

```
 GridPane gridPane = new GridPane();

ColumnConstraints col1 = new ColumnConstraints();
col1.setMinWidth(50);
col1.setPrefWidth(100);
ColumnConstraints col2 = new ColumnConstraints();
col2.setMinWidth(50);
col2.setPrefWidth(100);
col2.setHgrow(Priority.SOMETIMES);
gridPane.getColumnConstraints().addAll(col1, col2);

RowConstraints row1 = new RowConstraints();
row1.setMinHeight(50);
row1.setPrefHeight(100);
RowConstraints row2 = new RowConstraints();
row2.setMinHeight(50);
row2.setPrefHeight(100);
row2.setVgrow(Priority.SOMETIMES);
gridPane.getRowConstraints().addAll(row1, row2);
```

Вы также можете указать параметры *maxHeight* и *maxWidth* (максимальную высоту и штрину) для отдельных строк и столбцов.

BorderPane

BorderPane — это макет с пятью разделами:

- Top (Верх)
- Bottom (Низ)
- Right (Право)
- Left (Лево)
- Center (Центр)

Вы можете назначить компоненты отдельным разделам BorderPane:

```
<BorderPane>
   <top>
      <Label>TOP</Label>
   </top>
   <bottom>
      <Label>BOTTOM</Label>
   </bottom>
   <left>
      <Label>LEFT</Label>
   </left>
   <right>
      <Label>RIGHT</Label>
   </right>
   <center>
      <Label>CENTER</Label>
   </center>
</BorderPane>
```

Теперь тот же пример на Java:

```
Label top = new Label("TOP");
Label bottom = new Label("BOTTOM");
Label left = new Label("LEFT");
Label right = new Label("RIGHT");
Label center = new Label("CENTER");

BorderPane borderPane = new BorderPane();
borderPane.setTop(top);
borderPane.setBottom(bottom);
borderPane.setLeft(left);
borderPane.setRight(right);
borderPane.setCenter(center);
```

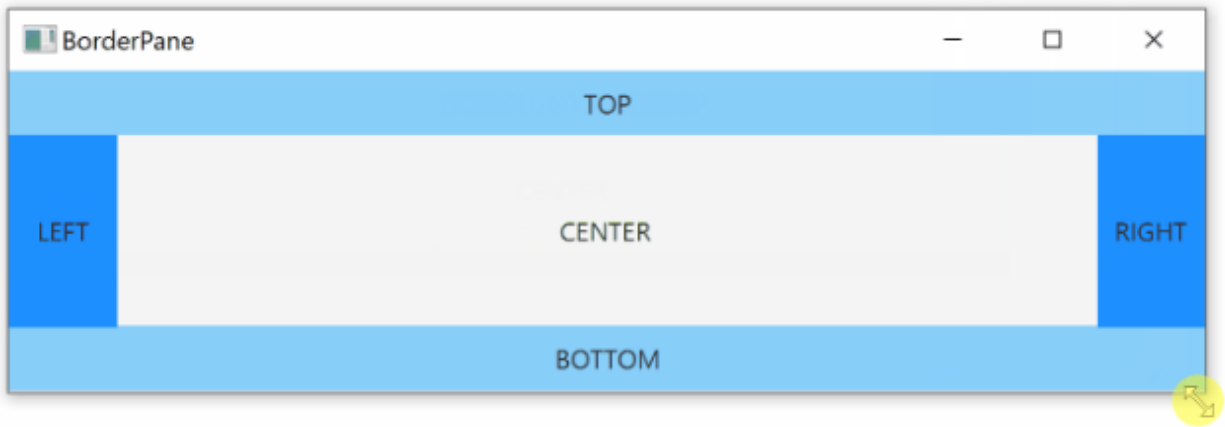
Sizing (Определение размеров)

Все регионы, кроме центрального (*center*), имеют фиксированный размер. Центр (center) тогда заполняет остальное пространство.

Верхняя (*Top*) и нижняя (*Bottom*) области растягиваются по всему доступному горизонтальному пространству. Их высота зависит от высоты компонента внутри.

Слева и справа заполнить все доступное вертикальное пространство (кроме того, что занимают сверху и снизу). Их ширина зависит от ширины компонента внутри.

Центр имеет динамический размер и заполняет остальное пространство, не занимаемое другими секциями. Давайте посмотрим на пример:



6 глава

Разделение визуальных элементов

В предыдущей статье о FXML мы узнали, как JavaFX обеспечивает четкое разделение задач путем разделения кода пользовательского интерфейса на две части. Компоненты и их свойства объявлены в файле FXML, а логика взаимодействия четко выделена в контроллер.

Кроме этого, есть и третья часть, язык FXML, управляющий только компонентами вашего приложения, их свойствами и тем, как они вложены друг в друга. Он не определяет визуальные элементы компонента, а именно: шрифты, цвета, фоны, отступы. В целом вы можете достичь этого в FXML, но не стоит этого делать. Вместо этого визуальные элементы должны быть четко определены в таблицах стилей CSS.

Таким образом, ваш дизайн становится независимым и может быть легко заменен или изменен без ущерба для остальной части приложения. Вы можете даже просто реализовать несколько тем, которые можно переключать по желанию пользователя.

CSS

Вы, вероятно, знакомы с CSS (Cascading Style Sheets — Каскадные таблицы стилей), используемыми для стилизации HTML-страниц в Интернете. Похожий подход реализован в JavaFX, несмотря на то что JavaFX использует набор своих собственных пользовательских свойств.

Давайте рассмотрим пример:

```
.button {
    -fx-font-size: 15px;
}
```

Здесь использованы две основные концепции. Первая — это селектор — *.button*. Он определяет, к каким компонентам должен применяться стиль. В этом примере стиль применяется для всех кнопок.

Вторая часть — это фактические свойства стиля, которые будут применены ко всем компонентам, которые соответствуют нашему селектору. Свойства — это все, что расположено внутри фигурных скобок.

Каждое свойство имеет определенное значение. В нашем примере есть свойство *-fx-font-size*, которое определяет, насколько большим будет текст. В примере указано значение *15px*, но это значение может быть любым другим.

Подводя итог — мы создали правило, которое гласит — все кнопки везде должны иметь текст размером 15 пикселей.

Selectors (Селекторы)

Теперь давайте рассмотрим подробнее, как работают селекторы в JavaFX. Это происходит почти так же, как в обычном CSS.

Class (Класс)

Класс в CSS представляет несколько похожих элементов. Например, кнопки или флажки. Селектор, который должен применяться ко всем элементам одного класса, начинается с точки ".", сопровождаемый непосредственно именем класса. Соглашение об именовании классов состоит в том, чтобы разделять отдельные слова с помощью символа "-". Следующий селектор применяется ко всем элементам с классом **label**.

```
.label {
    // Some properties
}
```

Built-in classes (Встроенные классы)

Хорошая новость заключается в том, что все встроенные компоненты JavaFX (такие как Label или Button) уже имеют предопределенный класс. Если вы хотите настроить стиль всех меток в своем приложении, вам не нужно добавлять какие-либо пользовательские классы для каждой из ваших меток. Каждая метка по-умолчанию имеет класс *label*.

Легко определить имя класса из компонента:

- Возьмите имя Java класса компонента — например. *Label*
- Сделайте имя строчным
- Если он состоит из нескольких слов, разделите их с помощью символа "-"

Некоторые примеры:

- Метка → метка
- CheckBox → check-box

При использовании таких классов, как селекторы, не забудьте добавить ".". Это означает, что селектором для класса *label* является *.label*.

Custom classes (Пользовательские классы)

Если встроенных классов недостаточно, вы можете добавить свои собственные классы к своим компонентам. Вы можете использовать несколько классов, разделенных запятой:

```
<Label styleClass="my-label,other-class">I am a simple label</Label>
```

Или на Java:

```
Label label = new Label("I am a simple label");
label.getStyleClass().addAll("my-label", "other-class");
```

Добавление классов таким способом не удаляет класс компонента по умолчанию (в данном случае *label*).

Существует один специальный класс, называемый *root*. Он является корневым компонентом вашей сцены. Вы можете использовать его, чтобы стилизовать все внутри вашей сцены (например, установить глобальный шрифт). Это похоже на использование селектора тегов body в HTML.

ID

Другим способом выбора компонентов в CSS является использование идентификатора компонента (ID). Он является уникальным идентификатором компонента. В отличие от классов, которые могут быть назначены нескольким компонентам, идентификатор должен быть уникальным в сцене.

В то время как, для указания класса используют символ "." перед именем в их селекторах, идентификаторы помечаются символом "#".

```
#my-component {
  ...
}
```

В FXML вы можете использовать *fx:id* для установки CSS-идентификатора компонента.

```
<Label fx:id="foo">I am a simple label</Label>
```

Однако есть одна оговорка. Этот же идентификатор используется для ссылки на объект компонента, объявленный в вашем контроллере с тем же именем. Так как идентификатор и имя поля в контроллере должны совпадать, *fx:id* должен учитывать ограничение именования Java для имен полей. Хотя соглашение об именах CSS определяет отдельные слова, разделенные символом "-", это недопустимый символ для имен полей Java. Поэтому для *fx:id* с несколькими словами вам нужно использовать другое соглашение об именах, такое как

CamelCase, или использовать подчеркивание.

```
<!--  This is not valid  -->
<Label fx:id="my-label">I am a simple label</Label>
<!--  This is valid  -->
<Label fx:id="my_label">I am a simple label</Label>
<Label fx:id="MyLabel">I am a simple label</Label>
```

В Java вы можете просто вызвать метод *setId()* вашего компонента.

```
Label label = new Label("I am a simple label");
label.setId("foo");
```

Properties (Свойства)

Хотя CSS, используемый в JavaFX, очень похож на оригинальный веб-CSS, есть одно большое отличие. Имена свойств различны, и есть много новых свойств, специфичных для JavaFX. Они имеют префикс *-fx-*.

Вот некоторые примеры:

- *-fx-background-color*: Цвет фона
- *-fx-text-fill*: Цвет текста
- *-fx-font-size*: Размер текста

Вы можете найти список всех свойств в официальном руководстве по дизайну.

Псевдоклассы

В дополнение к обычным классам, которые отмечают определенные компоненты, существуют так называемые псевдоклассы, которые обозначают состояние компонента. Это может быть, например, класс для маркировки того, что компонент имеет фокус или на нем находится курсор мыши.

Есть множество встроенных псевдоклассов. Давайте посмотрим на кнопку. Существует несколько псевдоклассов, которые вы можете использовать, например:

- *hover*: мышь над кнопкой
- *focused*: кнопка имеет фокус
- *disabled*: кнопка отключена

- *pressed*: кнопка нажата

Псевдоклассы начинаются с символа ":" (например, *:hover*) в селекторах CSS. Вам, конечно, нужно указать, к какому компоненту относится ваш псевдокласс — например, *button:hover*. В следующем примере показан селектор, который применяется ко всем кнопкам, имеющим фокус:

```
.button:focused {
    -fx-background-color: red;
}
```

В отличие от CSS, который имеет только базовые псевдоклассы для состояний, таких как *focus*и *hover*, JavaFX имеет специфичные для компонента псевдоклассы, которые относятся к различным состояниям или свойствам компонентов.

Например:

- Полосы прокрутки (Scrollbars) имеют псевдоклассы *horizontal* и *vertical*
- Элементы (Cells) имеют псевдоклассы *odd* и *even*
- TitledPane имеет псевдоклассы *expanded* и *collapsed*

Пользовательские псевдоклассы

В дополнение ко встроенным псевдоклассам, вы можете определять и использовать свои собственные псевдоклассы.

Давайте создадим нашу собственную метку (наследуя от класса Label). У него будет новое логическое свойство, называемое *shiny*. В таком случае, мы хотим, чтобы у нашей метки был псевдокласс *shiny*.

Поскольку у метки есть псевдокласс *shiny*, мы можем установить фон метки *gold*:

```
.shiny-label:shiny {
    -fx-background-color: gold;
}
```

Теперь создадим сам класс.

```
public class ShinyLabel extends Label {
    private BooleanProperty shiny;

    public ShinyLabel() {
        getStyleClass().add("shiny-label");
```

```
        shiny = new SimpleBooleanProperty(false);
        shiny.addListener(e -> {
            pseudoClassStateChanged(PseudoClass.getPseudoClass("shiny"), shiny.get());
        });
    }

    public boolean isShiny() {
        return shiny.get();
    }

    public void setShiny(boolean shiny) {
        this.shiny.set(shiny);
    }
}
```

Здесь есть несколько важных частей:

1. У нас есть логическое свойство *BooleanProperty* вместо обычного *boolean*. Это означает, что объект *shiny* является observable (наблюдаемым), и мы можем отслеживать (слушать) изменения в его значении.
2. Мы регистрируем listener (слушатель), который будет вызываться каждый раз, когда значение объекта *shiny* изменяется с использованием *shiny.addListener()*.
3. Когда значение *shiny* изменяется, мы добавляем/удаляем псевдокласс *shiny* в зависимости от текущего значения *pseudoClassStateChanged(PseudoClass.getPseudoClass(«shiny»), shiny.get())*.
4. Мы добавляем пользовательский класс для всех меток *shiny-label*, вместо того чтобы иметь только класс *label*, унаследованный от родителя. Таким образом, мы можем выбирать только метки *shiny*.

Таблица стилей по умолчанию

Даже если вы сами не предоставляете никаких стилей, каждое приложение JavaFX уже имеет некоторые визуальные стили. Существует таблица стилей по умолчанию, которая применяется к каждому приложению. Она называется *modena* (начиная с JavaFX 8, ранее она называлась *caspian*).

Эту таблицу стилей можно найти в файле:

jfxrt.jar\com\sun\javafx\scene\control\skin\modena\modena.css

Или вы можете найти файл здесь. В том же каталоге есть множество изображений, используемых таблицей стилей.

Эта таблица стилей предоставляет стили по умолчанию, но имеет самый низкий приоритет по сравнению с другими типами таблиц стилей, поэтому вы можете легко ее переопределить.

Scene stylesheet (Таблица стилей сцены)

В дополнение к таблице стилей по умолчанию, упомянутой выше, вы, конечно, можете предоставить свою собственную. Самый высокий уровень, на котором вы можете применить стилизацию — это вся сцена. Вы можете реализовать это в вашем FXML:

```
<BorderPane xmlns="http://javafx.com/javafx"
        xmlns:fx="http://javafx.com/fxml"
        stylesheets="styles.css"
        ...
        >
  ...
</BorderPane>
```

Или в вашем Java коде:

```
String stylesheet = getClass().getResource("/styles.css").toExternalForm();
scene.getStylesheets().add(stylesheet);
```

Обратите внимание на вызов *toExternalForm()*. Scene ожидает получить содержимое таблицы стилей в виде строки, а не файла, поэтому нам нужно предоставить содержимое нашей таблицы стилей в виде строки.

Parent stylesheet (Родительская таблица стилей)

В дополнение к таблице стилей для всей сцены, иногда бывает полезно иметь стили на уровне макета. То есть — для отдельного контейнера, такого как VBox, HBox или GridPane. Общим родителем всех макетов является родительский класс, который определяет методы для обработки таблиц стилей на уровне макета. Эти стили применяются только для компонентов в данном макете, а не для всей сцены. Стиль на уровне макета имеет приоритет над стилем на уровне сцены.

```
<HBox stylesheets="styles.css">
   ...
</HBox>
```

В Java вам нужно загрузить содержимое таблицы стилей самостоятельно, так же как и ранее для сцены:

```
HBox box = new HBox();
String stylesheet = getClass().getResource("/styles.css").toExternalForm();
box.getStylesheets().add(stylesheet);
```

Inline styles (Встроенные стили)

Пока что мы рассмотрели только случаи назначения внешней таблицы стилей для всей сцены или макета. Но можно задать индивидуальные свойства стиля на уровне компонента.

Здесь вам не нужно беспокоиться о селекторе, так как все свойства установлены для определенного компонента.

Вы можете указать несколько свойств, разделенных точкой с запятой:

```
<Label style="-fx-background-color: blue; -fx-text-fill: white">
  I'm feeling blue.
</Label>
```

В Java вы можете использовать метод *setStyle()*:

```
Label label = new Label("I'm feeling blue.");
label.setStyle("-fx-background-color: blue; -fx-text-fill: white");
```

Стили на уровне компонента имеют приоритет как над стилями сцены, так и над родительскими стилями на уровне макета.

Почему нужно их избегать

Стилизация на уровне компонентов может быть удобной, но это быстрое и «грязное» решение. Вы отказываетесь от основного преимущества CSS, которое заключается в отделении стилей от компонентов. Теперь вы жестко привязываете свои визуальные элементы непосредственно к компонентам. Вы больше не сможете легко переключать свои таблицы стилей, когда это необходимо, вы не можете менять темы.

Более того, у вас больше нет единого центрального места, где определяется ваш стиль. Когда вам нужно что-то изменить в наборе похожих компонентов, вам нужно изменить каждый из компонентов по отдельности, а не редактировать только одно место во внешней таблице стилей. Поэтому следует избегать встроенных стилей компонентов.

Stylesheet priorities (Приоритеты таблиц стилей)

Вы можете обеспечить стилизацию на нескольких уровнях — сцена, родительский, встроенные стили, и также есть таблица стилей модены по умолчанию. Если вы изменяете одно и то же свойство одного и того же компонента на нескольких уровнях, JavaFX имеет настройку приоритета, которая определяет, какие стили следует использовать. Список приоритетов — от высшего к низшему:

1. Inline styles (Встроенные стили)
2. Parent styles (Родительские стили)
3. Scene styles (Стили сцены)
4. Default styles (Стили по умолчанию)

Это означает, что, если вы установите цвет фона определенной метки как на встроенном, так и на уровне сцены, JavaFX будет использовать значение, установленное во встроенных стилях, поскольку оно имеет более высокий приоритет.

7 глава

JavaFX & Spring

В настоящее время Java разработчики, редко работают просто с Java. В большинстве проектов они обычно используют Spring Framework или Spring Boot. У этого подхода много преимуществ, так как эти проекты включают в себя много полезных функций.

Однако, когда вы разрабатываете приложения JavaFX, нет простого способа интегрировать его со Spring. Интеграция не работает «из коробки», так как приложения JavaFX имеют свой собственный жизненный цикл и обеспечивают создание экземпляров контроллеров.

Для этого поста есть репозиторий с примером, где вы можете посмотреть окончательный код проекта.

JavaFX-Weaver

JavaFX-Weaver — это проект Рене Гилена (Rene Gielen), целью которого является интеграция Spring и JavaFX. Настройка, к счастью, не так сложна.

Начнем

Давайте попробуем с самым простым проектом Spring Boot, и попробуем интегрировать в него JavaFX. Вы можете создать новый проект с помощью Spring Initializr. Вам не нужно добавлять какие-либо зависимости. Если вы хотите избежать настройки JavaFX в качестве зависимости, выберите версию более раннюю чем Java 11, поскольку она по-прежнему содержит JavaFX как часть JDK.

Добавление контроллера

Нам понадобится контроллер и сопутствующее представление .fxml, чтобы можно было проверить, правильно ли работает наше приложение как с Spring, так и с JavaFX. Давайте сначала создадим контроллер и пока оставим его пустым.

```
public class MyController {
}
```

Добавление представления

Теперь нам нужен файл *.fxml*, который будет использоваться с нашим контроллером в качестве представления. Поместим его в папку ресурсов. **Для правильной работы интеграции необходимо создать ее в папке ресурсов, но в структуре каталогов, соответствующей пакету, в котором находится наш контроллер.**

Например, предположим, что наш контроллер находится в пакете *com.vojtechruzicka.javafxweaverexample*. Файл *.fxml* должен быть размещен именно здесь:

src\main\resources\com\vojtechruzicka\javafxweaverexample

Давайте назовем наш файл *main-scene.fxml*.

Зависимости

Если вы используете обычный Spring, настройка немного отличается, но для Spring Boot вам просто нужно добавить следующую зависимость в ваш файл *pom.xml*:

```
<dependency>
    <groupId>net.rgielen</groupId>
    <artifactId>javafx-weaver-spring-boot-starter</artifactId>
    <version>1.3.0</version>
</dependency>
```

Или это, если вы используете Gradle:

implementation 'javafx-weaver-spring-boot-starter:1.3.0'

Класс Spring Boot Application

Когда мы генерировали наше базовое приложение Spring Boot, для нас также был создан основной класс приложения. Это класс, аннотированный *@SpringBootApplication*, используется в качестве точки входа для запуска всего приложения.

Но подождите! JavaFX также имеет свой основной класс приложения, который используется в качестве точки входа для запуска приложений JavaFX.

Это сбивает с толку. Итак, какой из них нужно использовать для запуска нашего приложения, как Spring Boot, так и JavaFX?

Мы все еще будем использовать наше *@SpringBootApplication* с небольшой модификацией. Вместо непосредственного запуска приложения Spring, мы будем использовать его для запуска нашего приложения JavaFX. Приложение JavaFX будет отвечать за правильный запуск контекста приложения Spring и интеграцию всего вместе с помощью *JavaFX Weaver*.

Сначала нам нужно убедиться, что приложение Spring Boot запускает наше приложение JavaFX.

```
import javafx.application.Application;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootExampleApplication {

    public static void main(String[] args) {
        // This is how normal Spring Boot app would be launched
        // SpringApplication.run(SpringBootExampleApplication.class, args);

        // JavaFxApplication doesn't exist yet,
        // we'll create it in the next step
        Application.launch(JavaFxApplication.class, args);
    }
}
```

Класс JavaFX Application

Теперь наше приложение *@SpringBootApplication* вызывает класс *JavaFxApplication*, который еще не существует. Давайте создадим его.

```
public class JavaFxApplication extends Application {

    private ConfigurableApplicationContext applicationContext;

    @Override
    public void init() {
        String[] args = getParameters().getRaw().toArray(new String[0]);
```

```
        this.applicationContext = new SpringApplicationBuilder()
                .sources(SpringBootExampleApplication.class)
                .run(args);
    }

}
```

Когда JavaFX инициализируется, он создает новый контекст приложения на основе конфигурации в нашем *SpringBootExampleApplication* — главном классе приложения Spring Boot, которое мы изменили на предыдущем шаге.

Теперь у нас есть приложение Spring Boot, работающее с нашим новым контекстом приложения. Но мы должны убедиться, что контекст правильно закрыт, когда приложение JavaFX завершается. Например, когда вы закрываете окно. Давайте сделаем это сейчас.

```
@Override
public void stop() {
    this.applicationContext.close();
    Platform.exit();
}
```

Последняя часть приложения — это создание нового окна (Stage) и его показ.

```
@Override
public void start(Stage stage) {
    FxWeaver fxWeaver = applicationContext.getBean(FxWeaver.class);
    Parent root = fxWeaver.loadView(MyController.class);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
}
```

Это то место, где FxWeaver вступает в игру. Нам нужно получить его bean-компонент из контекста приложения, а затем использовать его для загрузки нашего FXML.

Контроллеры, управляемые Spring

Традиционно мы создавали бы нашу сцену, используя FXMLLoader, который загружал бы файл FXML и создавал экземпляр Controller, объявленный в нем для нас.

```
FXMLLoader loader = new FXMLLoader();
URL xmlUrl = getClass().getResource("/main-scene.fxml");
```

```
loader.setLocation(xmlUrl);
Parent root = loader.load();
Scene scene = new Scene(root);
stage.setScene(scene);
stage.show();
```

Так почему же мы используем FX Weaver? Проблема в том, что FXMLLoader создает для нас экземпляр контроллера. Это означает, что он не создан и не управляется Spring. Поэтому мы не можем использовать внедрение зависимостей и другие полезности Spring в наших контроллерах. И именно поэтому мы представили Spring в нашем JavaFX в первую очередь!

Но когда FX Weaver создает контроллер для нас, он создает его как управляемый Spring компонент, поэтому мы можем полностью использовать возможности Spring.

Включение Spring для контроллера

Первое, что нам нужно сделать — это аннотировать существующий контроллер JavaFX с помощью *Component*, чтобы Spring распознавал и управлял им. Затем нам нужно добавить аннотацию *@FxmlView*, чтобы она распознавалась FX Weaver.

```
import net.rgielen.fxweaver.core.FxmlView;
import org.springframework.stereotype.Component;

@Component
@FxmlView("main-stage.fxml")
public class MyController {
}
```

Обратите внимание на параметр *@FxmlView («main-stage.fxml»)*. Он указывает имя вашего *.fxml*файла, который должен соответствовать контроллеру. Это необязательно. Если вы не укажете его, Fx Weaver будет использовать имя класса контроллера в качестве имени файла с расширением *.fxml*. **Файл FXML должен находиться в том же пакете, что и контроллер, но в папке ресурсов.**

Убедимся, что все работает

Теперь давайте удостоверимся, что все работает и хорошо интегрируется. Давайте запустим наше приложение *@SpringBootApplication* с его *main* методом. Вы должны увидеть простое окно с надписью, ничего особенного.

Хорошо, это означает, что приложение запускается, но мы не сделали ничего специфического для Spring в нашем контроллере. Нет внедрения зависимости или чего-либо еще. Давайте попробуем сделать это теперь.

Добавление сервиса

Чтобы убедиться, что интеграция Spring работает правильно, давайте создадим новый сервис, управляемый Spring. Позже мы добавим его в наш контроллер и будем использовать там.

```
import org.springframework.stereotype.Service;

@Service
public class WeatherService {

    public String getWeatherForecast() {
        return "It's gonna snow a lot. Brace yourselves, the winter is coming.";
    }
}
```

Ничего особенного, это сервис для прогнозирования погоды, который сейчас не очень динамичен, но этого будет достаточно для нашего примера.

Внедрение сервиса

Теперь давайте добавим наш новый сервис в наш существующий контроллер. Это обычный Spring, ничего особенного здесь нет.

```
@Component
@FxmlView("main-stage.fxml")
public class MyController {

    private WeatherService weatherService;

    @Autowired
    public MyController(WeatherService weatherService) {
        this.weatherService = weatherService;
    }
}
```

Загрузка прогноза

Теперь нам нужно как-то загрузить данные из нашего сервиса. Давайте изменим наше представление FMXL, так:

1. Создадим кнопку, по нажатию на которую загружаются данные из *WeatherService*
2. Загруженные данные отображаются в метке

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="com.vojtechruzicka.javafxweaverexample.MyController"
      prefHeight="100.0" prefWidth="350.0" spacing="10" alignment="CENTER">

   <Label fx:id="weatherLabel"/>
   <Button onAction="#loadWeatherForecast">Get weather</Button>
</VBox>
```

Обратите внимание на идентификатор , мы будем использовать его для получения доступа к указанной метке в нашем контроллере, чтобы изменить ее текст.

onAction="# loadWeatherForecast" — это метод нашего контроллера, который следует вызывать при нажатии кнопки. Нам все еще нужно добавить его в контроллер. Давайте сделаем это.

Логика контроллера

Последний шаг — это изменить наш контроллер, чтобы он реагировал на нажатие кнопок в представлении, загружал данные прогноза погоды и устанавливал их для нашей метки.

Поэтому нам нужна ссылка на метку с нашей точки зрения, чтобы мы могли

изменить ее текст. Нам нужно выбрать его имя, соответствующее .

```
@FXML
private Label weatherLabel;
```

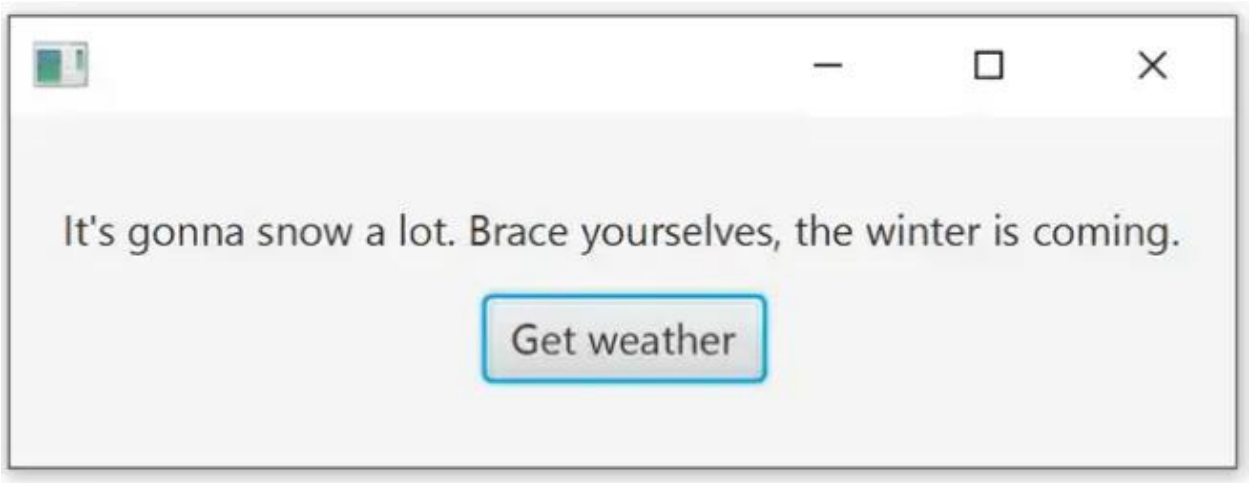
Теперь нам нужно добавить метод, который вызывается при нажатии кнопки — *onAction="# loadWeatherForecast"*.

```
public void loadWeatherForecast(ActionEvent actionEvent) {
   this.weatherLabel.setText(weatherService.getWeatherForecast());
}
```

В этом методе мы берем прогноз погоды из сервиса и присваиваем его метке,

которую мы определили ранее.

Если вы запустите приложение сейчас, после нажатия кнопки оно должно загрузить текущий прогноз погоды.



Доступ к компонентам из представления

Как и в обычном JavaFX, вы можете объявить компоненты из представления для внедрения в ваш контроллер, чтобы вы могли взаимодействовать с ними.

```
@FXML
private Label weatherLabel;
```

Мы уже видели, что все работает хорошо. Вам просто нужно быть осторожным с выбором времени. Наш контроллер аннотирован *Component*, так что это обычный bean-объект, управляемый Spring. Это означает, что он создается в Spring при запуске контекста приложения и внедрении всех зависимостей. Однако ткачество по FX Weaver происходит позже. И во время этого переплетения вставляются ссылки на компоненты.

Это имеет один смысл. В вашем конструкторе и *@PostConstruct* вы уже можете работать с внедренными зависимостями Spring как обычно. Однако следует помнить, что в течение этого времени ссылки на компоненты из представления еще не доступны и, следовательно, являются нулевыми.

Заключение

JavaFX Weaver предоставляет удобный и простой способ интеграции Spring с приложениями JavaFX. В остальном это не так просто, поскольку JavaFX управляет собственным жизненным циклом и жизненным циклом своих контроллеров. JavaFX Weaver делает интеграцию возможной и довольно простой, поэтому вы, наконец, можете использовать все интересные возможности Spring даже с JavaFX.