Содержание

ОБ АВТОРЕ 11

ПРЕДИСЛОВИЕ 13

БЛАГОДАРНОСТИ 15

1 ЛЯМБДА-ВЫРАЖЕНИЯ 17
Назначение лямбда-выражений 18
Синтаксис лямбда-выражений 20
Функциональные интерфейсы 22
Ссылки на методы 24
Ссылки на конструкторы 25
Область действия переменных 26
Методы по умолчанию 29
Статические методы в интерфейсах 32
Упражнения 33

2 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ПОТОКОВ ВВОДА-ВЫВОДА 35
От итерации к операциям с потоками ввода-вывода 36
Создание потока ввода-вывода 38
Методы `filter()`, `map()` и `flatMap()` 39
Извлечение подпотоков и объединение потоков ввода-вывода 40
Преобразования с сохранением состояния 41
Простые операции сведения 42
Тип данных `Optional` 43
Обращение со значениями типа `Optional` 43
Формирование значений типа `Optional` 44
Составление функций дополнительных значений методом `flatMap()` 45
Операции сведения 46
Накопление результатов 47
Накопление данных в отображениях 49
Группирование и разделение 50
Потоки ввода-вывода примитивных типов 53
Параллельные потоки ввода-вывода 55
Функциональные интерфейсы 57
Упражнения 58

3 ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ ЛЯМБДА-ВЫРАЖЕНИЙ 61
Отложенное выполнение 62
Параметры лямбда-выражений 63
Выбор функционального интерфейса 64
Возврат функций 67
Составление операций 68
Отложенность операций 70
Распараллеливание операций 71
Обработка исключений 72

Лямбда-выражения и обобщения 74
Одноместные операции 76
Упражнения 77

4 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС JAVAFX 81
Краткая история программирования ГПИ средствами Java 82
Применение JavaFX 84
Обработка событий 85
Свойства JavaFX 86
Привязки 88
Компоновка 92
Язык разметки FXML 98
Таблицы стилей CSS 101
Анимация и спецэффекты 103
Декоративные элементы управления 105
Упражнения 109

5 НОВЫЙ ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ДЛЯ ДАТЫ И ВРЕМЕНИ 111
Временная шкала 112
Местные даты 115
Корректоры дат 117
Местное время 118
Поясное время 119
Форматирование и синтаксический анализ даты и времени 122
Взаимодействие с устаревшим кодом 125
Упражнения 126

6 УСОВЕРШЕНСТВОВАНИЯ ПАРАЛЛЕЛИЗМА 127
Атомарные значения 128
Усовершенствования в классе `ConcurrentHashMap` 131
Обновление значений 132
Групповые операции 134
Представления множеств 136
Параллельные операции с массивами 137
Завершаемые будущие действия 138
Будущие действия 138
Составление будущих действий 139
Конвейер составления 139
Составление асинхронных операций 140
Упражнения 143

7 ИНТЕРПРЕТАТОР NASHORN ЯЗЫКА JAVASCRIPT 145
Выполнение интерпретатора Nashorn из командной строки 146
Выполнение интерпретатора Nashorn из кода Java 148
Вызов методов 149
Построение объектов 150
Символьные строки 151
Числа 151
Обращение с массивами 152
Списки и отображения 153
Лямбда-выражения 154
Расширение классов и реализация интерфейсов Java 154
Исключения 156

Написание сценариев командного процессора 156
Выполнение команд из командного процессора 157
Интерполяция символьных строк 158
Ввод данных в сценарий 158
Nashorn и JavaFX 160
Упражнения 161

8 РАЗНЫЕ ПОЛЕЗНЫЕ СРЕДСТВА 163
Символьные строки 164
Числовые классы 165
Новые математические функции 166
Коллекции 167
Методы, введенные в классы коллекций 167
Компараторы 167
Класс `Collections` 169
Обращение с файлами 169
Потоки ввода-вывода строк 169
Потоки ввода-вывода содержимого каталогов 171
Кодировка Base64 172
Аннотации 173
Повторяющиеся аннотации 173
Аннотации к использованию типов 175
Рефлексия параметров метода 176
Различные незначительные изменения 177
Проверки пустых значений 177
Отложенные сообщения 177
Регулярные выражения 178
Региональные настройки 178
Технология JDBC 180
Упражнения 180

9 НЕДОСТАТОЧНО ОСВЕЩЕННЫЕ ЯЗЫКОВЫЕ СРЕДСТВА В JAVA 7 183
Изменения в обработке исключений 184
Оператор `try` с ресурсами 185
Подавляемые исключения 186
Перехват нескольких исключений 187
Упрощение обработки исключений для рефлексивных методов 187
Обращение с файлами 188
Пути 188
Чтение и запись данных в файлы 190
Создание файлов и каталогов 191
Копирование, перемещение и удаление файлов 192
Реализация методов `equals()`, `hashCode()` и `compareTo()` 193
Безопасная проверка на равенство пустым значениям 193
Вычисление хеш-кодов 193
Сравнение числовых типов 194
Требования к безопасности 195
Прочие изменения 198
Преобразование символьных строк в числа 198
Глобальный регистратор 198
Проверки на пустые значения 199
Класс `ProcessBuilder` 199
Класс `URLClassLoader` 200
Класс `BitSet` 200
Упражнения 201

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 203

Глава 4

Прикладной программный интерфейс JavaFX

В этой главе...

- Краткая история программирования ГПИ средствами Java
- Внедрение JavaFX
- Обработка событий
- Свойства JavaFX
- Привязки
- Компоновка
- Язык разметки FXML
- Таблицы стилей CSS
- Анимация и спецэффекты
- Декоративные элементы управления
- Упражнения

Прикладной программный интерфейс JavaFX является рекомендуемым инструментальным средством для разработки графического пользовательского интерфейса (ГПИ) приложений средствами Java. Теперь JavaFX входит в комплект версий платформы Java, поддерживаемых компанией Oracle. В этой главе поясняются основы разработки ГПИ средствами JavaFX. Если у вас имеется опыт разработки клиентских платформ с ГПИ, обладающим расширенными функциональными возможностями, то из этой главы вы узнаете, как лучше всего перейти от Swing к JavaFX. А если у вас нет такого опыта, то все равно прочитайте хотя бы бегло эту главу, чтобы понять из приведенных в ней примеров, как разрабатывать графические приложения средствами JavaFX.

В этой главе рассматриваются следующие основные вопросы.

- Граф сцены, состоящий из узлов, которые в свою очередь могут содержать узлы.
- Сцена, отображаемая на подмостках (в окне верхнего уровня, поверхности аплета или на всем экране).
- Элементы управления (например, кнопки), инициирующие события, хотя большинство событий в JavaFX наступают в результате изменения свойств.
- Свойства JavaFX, инициирующие события в результате изменений или недействительности данных.
- Обновление одного свойства, привязанного к другому свойству, когда последнее изменяется.
- Применение в JavaFX панелей компоновки, действующих аналогично диспетчерам компоновки в Swing.
- Определение компоновки средствами языка разметки FXML.
- Изменение внешнего вида приложения с помощью таблиц стилей CSS.
- Простые средства реализации анимации и спецэффектов.
- Усовершенствованные элементы управления, доступные в JavaFX, в том числе для построения диаграмм, проигрывания мультимедийных данных и просмотра встроенными средствами WebKit.

Краткая история программирования ГПИ средствами Java

Когда был разработан язык программирования Java, Интернет пребывал в зачаточном состоянии, а персональные компьютеры существовали лишь в настольном варианте. Бизнес-приложения были реализованы на платформе так называемых “толстых клиентов” — программ со многими кнопками, ползунками и текстовыми полями, предназначавшихся для взаимодействия с сервером. Такое решение считалось намного более изящным, чем “немые” (т.е. непрограммируемые) терминальные приложения из предшествовавшей эпохи. В состав Java 1.0 была включена библиотека AWT — набор инструментальных средств для разработки ГПИ независимо от конкретной платформы. Эта библиотека предназначалась для обслуживания клиентов зарождавшейся в то время Всемирной паутины (или просто веб), чтобы исключить затраты на сопровождение и обновление приложений на каждом настольном компьютере.

Разработка библиотеки AWT преследовала следующий благородный замысел: предоставить общий интерфейс для программирования машинно-зависимых кнопок, ползунков, текстовых полей и прочих элементов управления ГПИ в различных операционных системах. Но реализовать этот замысел на практике не удалось до конца. В каждой операционной системе выявились незначительные отличия в функциональных возможностях виджетов ГПИ, и поэтому первоначальный замысел "написано один раз, а выполняется везде" превратился в нечто другое: "написано много раз, а отлаживается везде".

Затем была разработана библиотека Swing, главный замысел которой состоял в том, чтобы не использовать виджеты, зависимые от отдельных платформ, а отображать свои. Благодаря этому обеспечивался общий стиль оформления ГПИ на каждой платформе. При желании пользователи могли выбрать машинно-зависимый стиль оформления, характерный для каждой платформы, а виджеты Swing должны были отображаться согласованно с машинно-зависимыми виджетами. Разумеется, все это происходило медленно, на что жаловались пользователи. Когда же появились более быстродействующие компьютеры, пользователи стали жаловаться на то, что библиотека Swing оказалась в конечном итоге хуже "родных" виджетов конкретных платформ, будучи перенасыщенной анимационными эффектами и спецэффектами. Более того, средства Flash стали все чаще применяться для создания ГПИ с еще более изощренными эффектами, которые вообще не применялись в "родных" элементах управления.

В 2007 году компания Sun Microsystems внедрила новую технологию под названием JavaFX в качестве альтернативы технологии Flash. Эта технология выполнялась на виртуальной машине Java, но использовала свой язык программирования под названием JavaFX Script. Этот язык был оптимизирован для программирования анимационных эффектов и спецэффектов. Но программисты стали жаловаться на необходимость изучать этот новый язык и поэтому старались в своем большинстве держаться подальше от данной технологии. С появлением Java 7 в качестве обновления версии 6 технология JavaFX версии 2.2 вошла в комплект вместе с JDK и JRE, а в версии Java 8 она получила название JavaFX 8, отражающее результаты многисленных переработок.

Разумеется, технология Flash ныне постепенно отходит в небытие, а ГПИ приложений были значительно усовершенствованы и нашли широкое распространение на мобильных устройствах. Но до сих пор имеются случаи, когда "толстый" клиент на настольном компьютере повышает производительность труда пользователя. Кроме того, код Java теперь выполняется на процессорах с архитектурой ARM. Имеются также встроенные системы, которым требуется ГПИ, в том числе информационные киоски и автомобильные средства отображения.

Предлагая технологию JavaFX, компания Oracle рекомендует применять ее в подобных приложениях. Почему же компания Oracle не внедрила лучшие части JavaFX в Swing? Для того чтобы повысить эффективность работы Swing на современных графических аппаратных средствах, эту библиотеку пришлось бы полностью переделать. Поэтому в компании Oracle было решено, что делать этого не стоит, а лучше объявить о прекращении дальнейшей разработки Swing.

В этой главе рассматриваются основы создания ГПИ в JavaFX. При этом главное внимание уделяется бизнес-приложениям с кнопками, ползунками и текстовыми полями, а не броским эффектам, послужившим первоначально к разработке JavaFX.

Применение JavaFX

Начнем с самого простого примера программы, выводящей сообщение (рис. 4.1). Как и в Swing, для этой цели используется метка следующим образом:

```
Label message = new Label("Hello, JavaFX!");
```

НА ЗАМЕТКУ. Следует иметь в виду, что в названии компонентов JavaFX отсутствует префикс **J**, характерный для компонентов Swing. Этот префикс используется в Swing для того, чтобы отличать, например, компонент `JLabel` от компонента `Label` в AWT.

Рис. 4.1. Результат вывода из программы сообщения "`Hello, JavaFX!`" средствами JavaFX

Увеличим размер шрифта, как показано ниже. Для этой цели служит конструктор класса `Font`, создающий объект, который по умолчанию представляет шрифт размером **100** пунктов.

```
message.setFont(new Font(100));
```

Все, что требуется отобразить средствами JavaFX, размещается на *сцене*, которую можно оформить и оживить "актерами", в роли которых выступают элементы управления и формы. Рассматриваемой здесь программе никакого оформления или анимации не требуется, но все же нужна сцена. А сцена должна находиться на *подмостках*, которые представляют собой окно верхнего уровня, если программа выполняется на рабочем столе операционной системы, или прямоугольную область, если программа выполняется в виде аплета. Подмостки передаются в виде параметра методу `start()`, который необходимо переопределить в подклассе, производном от класса `Application`, как показано в приведенном ниже примере.

```
public class HelloWorld extends Application {
  public void start(Stage stage) {
    Label message = new Label("Hello, JavaFX!");
    message.setFont(new Font(100));
    stage.setScene(new Scene(message));
    stage.setTitle("Hello");
    stage.show();
  }
}
```

НА ЗАМЕТКУ. Как следует из приведенного выше примера, для запуска JavaFX-приложения на выполнение метод `main()` не требуется. Но в предыдущих версиях JavaFX метод `main()` требовалось включать в форму, как показано ниже.

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

Обработка событий

Графические пользовательские интерфейсы управляются событиями. Когда пользователи выбирают кнопки щелчком на них, регулируют ползунки и выполняют другие действия, ГПИ реагирует и обновляется. Как и в Swing, отдельный элемент управления (например, кнопка) снабжается обработчиком событий, чтобы получать уведомления о выборе кнопки щелчком на ней. Все это существенно упрощается благодаря лямбда-выражениям, как выделено полужирным в приведенном ниже примере кода.

```
Button red = new Button("Red");
red.setOnAction(event -> message.setTextFill(Color.RED));
```

При выборе кнопки щелчком на ней вызывается лямбда-выражение. В данном случае задается красный цвет текста.

Но обработка событий, возникающих в элементах управления JavaFX, происходит иначе. В качестве примера рассмотрим ползунок, показанный на рис. 4.2. Когда перемещается ползунок, регулируемое значение изменяется. Но принимать низкоуровневые события, инициируемые ползунком для уведомления об изменениях в этом элементе управления, не следует. Вместо этого у ползунка имеется *свойство* JavaFX, называемое `value`, а также инициирующее события об изменениях в нем. Более подробно свойства будут рассматриваться в следующем разделе, а ниже показано, как организовать прием событий от свойств и отрегулировать размер шрифта, которым набрано выводимое сообщение.

```
slider.valueProperty().addListener(property
    -> message.setFont(new Font(slider.getValue())));
```

Рис. 4.2. Обработка событий от ползунка

Прием событий от свойств весьма распространен в JavaFX. Так, если требуется изменить часть пользовательского интерфейса при вводе текста пользователем в текстовом поле, достаточно ввести приемник событий в свойство `text`.

НА ЗАМЕТКУ. Иное дело — кнопки. Щелчок на кнопке совсем не изменяет одно из ее свойств.

Свойства JavaFX

Свойство — это атрибут класса, значение которого можно читать или записывать. Как правило, свойство поддерживается полем, а метод получения и установки просто читает и записывает данные в свойство соответственно. Но метод получения и установки может также принимать другие действия, в том числе чтение значений из базы данных или отправку уведомлений об изменениях. Во многих языках программирования имеется удобный синтаксис для вызова методов получения и установки значений свойств. Как правило, если свойство указывается в правой части выражения присваивания, то вызывается метод получения. А если свойство указывается в левой части выражения присваивания, то вызывается метод установки. Ниже приведен характерный тому пример.

```
value = obj.property;
  // во многих языках программирования, кроме Java,
  // здесь вызывается метод получения
obj.property = value; // а здесь вызывается метод установки
```

К сожалению, подобный синтаксис отсутствует в Java. Тем не менее свойства поддерживаются в Java с версии 1.1. В спецификации JavaBeans поясняется, что свойство должно быть выведено из пары методов получения и установки. Например, считается, что в классе с методами `String getText()` и `void setText(String newValue)` имеется свойство `text`. А классы `Introspector` и `BeanInfo` из пакета `java.beans` позволяют перечислять все свойства класса.

В спецификации JavaBeans определяются также *привязанные свойства*, где объекты инициируют события об изменениях в свойствах при вызове методов установки. В JavaFX эта часть спецификации не используется. Вместо этого свойство в JavaFX, помимо методов получения и установки, снабжается третьим методом, возвращающим объект класса, реализующего интерфейс `Property`. Например, у свойства `text` в JavaFX имеется метод `Property<String> textProperty()`. К объекту свойства можно присоединить приемник событий. Этим JavaFX отличается от прежней технологии JavaBeans. В JavaFX объект свойства, а не компонент JavaBeans, посылает уведомления об изменениях. И для таких изменений имеются веские основания. Для реализации привязанных свойств в JavaBeans требовался шаблонный код, выполнявший ввод, удаление и запуск приемников событий. А в JavaFX все сделано намного проще, поскольку все бремя хлопот берут на себя библиотечные классы.

В качестве примера рассмотрим реализацию свойства `text` в классе `Greeting`. Ниже показан простейший способ сделать это.

```
public class Greeting {
  private StringProperty text = new SimpleStringProperty("");
  public final StringProperty textProperty() { return text; }
  public final void setText(String newValue) { text.set(newValue); }
  public final String getText() { return text.get(); }
}
```

Класс `StringProperty` заключает символьную строку в оболочку. В нем имеются методы для получения и установки заключенных в оболочку значений, а также для управления приемниками событий. Как видите, для реализации свойства в JavaFX требуется некоторый шаблонный код, и, к сожалению, в Java нет никакой

возможности сформировать такой код автоматически. Но, по крайней мере, об управлении приемниками событий можно не беспокоиться. И хотя объявлять как `final` методы получения и установки значений свойств совсем не обязательно, тем не менее разработчики JavaFX рекомендуют поступать именно так.

НА ЗАМЕТКУ. По такому шаблону объект свойства требуется для каждого свойства независимо от того, принимаются ли где-нибудь события об изменениях в нем. В упражнении 2, приведенном в конце этой главы, используется полезная оптимизация для шаблона, состоящая в создании объектов свойств по требованию.

В предыдущем примере был определен класс `StringProperty`. Для свойства примитивного типа в качестве оболочки используется один из следующих классов: `IntegerProperty`, `LongProperty`, `DoubleProperty`, `FloatProperty` или `BooleanProperty`. Для этой цели имеются также классы `ListProperty`, `MapProperty` и `SetProperty`, а для всего остального — обобщенный класс `ObjectProperty<T>`. Все эти классы являются абстрактными и имеют конкретные подклассы `SimpleIntegerProperty`, `SimpleObjectProperty<T>` и т.д.

НА ЗАМЕТКУ. Если требуется лишь управлять приемниками событий, методы свойств могут возвращать объекты типа `ObjectProperty<T>` или даже интерфейс `Property<T>`. А более специализированные классы полезны для вычислений с помощью свойств, как поясняется в разделе "Привязки" далее в этой главе.

НА ЗАМЕТКУ. У классов свойств имеются методы `getValue()` и `setValue()`, помимо методов `get()` и `set()`. В классе `StringProperty` метод `get()` равнозначен методу `getValue()`, а метод `set()` — методу `setValue()`. Но для примитивных типов они разные. Например, в классе `IntegerProperty` метод `getValue()` возвращает объект типа `Integer`, а метод `get()` — значение типа `int`. Как правило, используются методы `get()` и `set()`, если только не написать обобщенный код специально для обращения со свойствами любого типа.

К свойству могут быть присоединены две разновидности приемников событий. В частности, приемник событий типа `ChangeListener` уведомляется, когда значение свойства было изменено, а приемник событий типа `InvalidationListener` вызывается, когда значение свойства *может* быть изменено. Главное отличие заключается в следующем: вычисляется ли свойство *по требованию*. Как будет показано в следующем разделе, одни свойства вычисляются из других, причем вычисление выполняется лишь по мере надобности. В результате обратного вызова интерфейса `ChangeListener` сообщается старое и новое значение, а это означает, что в нем придется вычислять новое значение. В интерфейсе `InvalidationListener` новое значение не вычисляется, но это означает, что обратный вызов может быть получен, когда значение фактически не изменялось.

В большинстве случаев это отличие оказывается несущественным. Ведь совершенно не важно, будет ли новое значение получено в виде параметра обратного вызова или же из свойства. И, как правило, не стоит беспокоиться о вычисленных свойствах, которые остались без изменения, несмотря на то, что изменилось одно из их входных значений. В примере кода из предыдущего раздела интерфейс `InvalidationListener` применялся ради упрощения кода.

ВНИМАНИЕ. Пользоваться интерфейсом `ChangeListener` для обращения со свойствами не так-то просто. Можно было бы сделать следующий вызов:

```
slider.valueProperty().addListener((property, oldValue, newValue)
    -> message.setFont(new Font(newValue)));
```

Но такой вариант не пройдет. В классе `DoubleProperty` реализуется интерфейс `Property<Number>`, а не `Property<Double>`. Следовательно, типом параметров `oldValue` и `newValue` является `Number`, а не `Double`. Это означает, что их придется распаковать следующим образом:

```
slider.valueProperty().addListener((property, oldValue, newValue)
    -> message.setFont(new Font(newValue.doubleValue())));
```

Привязки

Главное назначение свойств в JavaFX состоит в уведомлении о *привязке* — автоматическом обновлении одного свойства при изменении другого. В качестве примера рассмотрим приложение, приведенное на рис. 4.3. Когда пользователь редактирует верхний адрес, обновляется также нижний адрес.

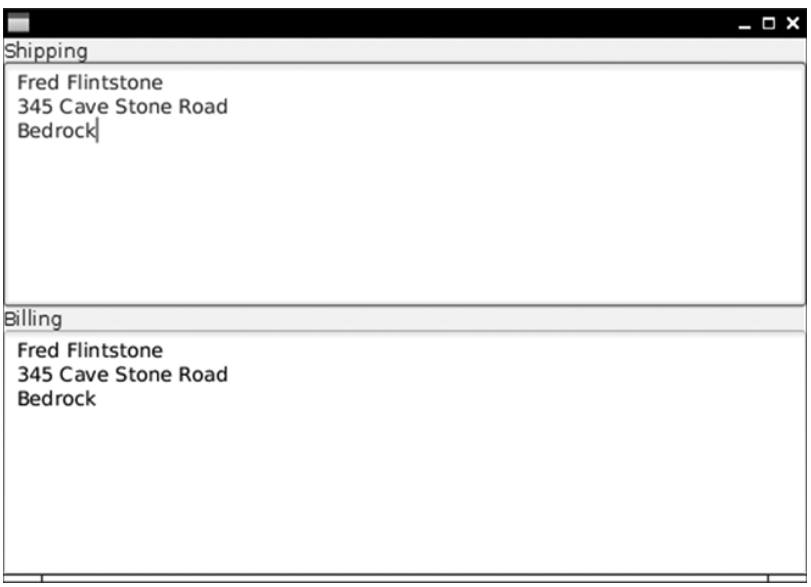

Рис. 4.3. Привязанное текстовое свойство обновляется автоматически

Такой результат достигается привязкой одного свойства к другому следующим образом:

```
billing.textProperty().bind(shipping.textProperty());
```

Приемник событий об изменениях подспудно присоединяется к текстовому свойству объекта `shipping`, устанавливающего текстовое свойство объекта `billing`. С другой стороны, можно сделать следующий вызов:

```
billing.textProperty().bindBidirectional(shipping.textProperty());
```

Если изменяется любое из этих свойств, то обновляется и другое. Для отмены привязки свойств следует вызвать метод `unbind()` или `unbindBidirectional()`.

Механизм привязки разрешает типичное затруднение, возникающее при программировании пользовательского интерфейса. В качестве примера рассмотрим поле даты и селектор даты из календаря. Когда пользователь выбирает дату из календаря, поле даты должно быть обновлено автоматически, а вместе с ним и свойство даты в модели.

Как правило, одно свойство зависит от другого, но эта взаимосвязь сложнее, чем кажется на первый взгляд. Например, круг обычно требуется расположить по центру сцены (рис. 4.4). Это означает, что значение свойства `centerX` должно составлять половину ширины сцены, определяемой свойством `width`.

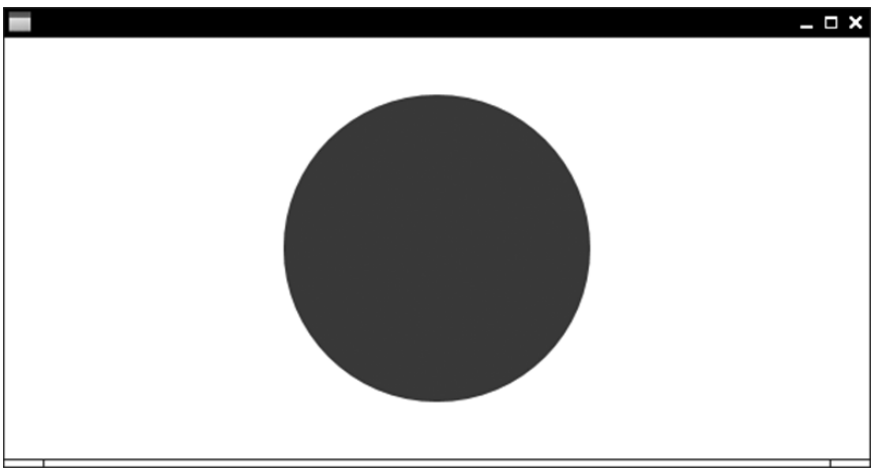

Рис. 4.4. Центр этого круга привязан к половине ширины и высоты сцены

Для достижения такого результата необходимо произвести вычисленное свойство. И для этой цели в классе `Bindings` имеются соответствующие статические методы. Например, при вызове метода `Bindings.divide(scene.widthProperty(), 2)` вычисляется свойство, значение которого составляет половину ширины сцены. Это свойство автоматически обновляется при изменении ширины сцены. Остается лишь привязать это вычисленное свойство к свойству `centerX` круга следующим образом:

```
circle.centerXProperty().bind(Bindings.divide(scene.widthProperty(), 2));
```

НА ЗАМЕТКУ. С другой стороны, можно вызвать метод `scene.widthProperty().divide(2)`. Применение статических методов из класса `Bindings` в более сложных выражениях повышает удобочитаемость исходного кода, особенно если сделать сначала следующее объявление:

```
import static javafx.beans.binding.Bindings.*;
```

а затем такой вызов:

```
divide(scene.widthProperty(), 2).
```

Рассмотрим более реалистичный пример. Допустим, требуется сделать недоступными кнопки Smaller (Мельче) и Larger (Крупнее), когда размер слишком мал или велик (рис. 4.5). Когда ширина меньше или равна нулю, недоступной становится кнопка Smaller. А когда ширина больше или равна **100**, недоступной становится кнопка Larger. Ниже показано, как все это реализуется непосредственно в коде.

```
smaller.disableProperty().bind(
    Bindings.lessThanOrEqual(gauge.widthProperty(), 0));
larger.disableProperty().bind(
    Bindings.greaterThanOrEqual(gauge.widthProperty(), 100));
```

Рис. 4.5. Когда размер достигает одного из крайних пределов, недоступной становится соответствующая кнопка его изменения

В табл. 4.1 перечислены все операции и соответствующие им методы, поддерживаемые в классе `Bindings`. В качестве одного или обоих аргументов указываются объекты, в классах которых реализуется интерфейс `Observable` или производные от него интерфейсы. В интерфейсе `Observable` предоставляются методы для ввода и удаления приемника событий типа `InvalidationListener`. В интерфейсе `ObservableValue` вводится приемник событий типа `ChangeListener` и метод `getValue()`, а в производных от него интерфейсах — методы для получения значения подходящего типа. Например, метод `get()` из интерфейса `ObservableStringValue` возвращает значение типа `int`. Типы, возвращаемые из методов класса `Bindings`, являются интерфейсами, производными от интерфейса `Binding`, который, в свою очередь, является производным от интерфейса `Observable`. Интерфейсу `Binding` известны все свойства, от которых он зависит. На практике разбираться во всех этих интерфейсах совсем не обязательно. Для привязки одного свойства к другому с целью добиться желаемого результата достаточно употребить эти свойства в нужном сочетании.

Таблица 4.1. Операции и соответствующие им методы, поддерживаемые в классе `Bindings`

Имя метода	Аргументы
add, **subtract**, **multiply**, **divide**, **max**, **min**	Оба типа **ObservableNumberValue**, **int**, **long**, **float**, **double**
negate	Типа **ObservableNumberValue**

Окончание табл. 4.1

Имя метода	Аргументы
greaterThan, **greaterThanOrEqual**, **lessThan**, **lessThanOrEqual**	Оба типа **ObservableNumberValue**, **int**, **long**, **float**, **double** или же типа **ObservableStringValue**, **String**
equal, **notEqual**	Оба типа **ObservableObjectValue**, **ObservableNumberValue**, **int**, **long**, **float**, **double**, **Object**
equalIgnoreCase, **notEqualIgnoreCase**	Оба типа **ObservableStringValue**, **String**
isEmpty, **isNotEmpty**	Типа **Observable(List|Map|Set|StringValue)**
isNull, **isNotNull**	Типа **ObservableObjectValue**
length	Типа **ObservableStringValue**
size	Типа **Observable(List|Map|Set)**
and, **or**	Оба типа **ObservableBooleanValue**
not	Типа **ObservableBooleanValue**
convert	Типа **ObservableValue**, преобразуемого в символьную строку для привязки
concat	Последовательность объектов, значения которых сцепляются в символьные строки, возвращаемые методом **toString()**. Если любой из объектов относится к изменяющемуся типу **ObservableValue**, то изменяется и результат сцепления строк
format	Дополнительные региональные настройки, символьная строка, отформатированная средствами класс **MessageFormat**, последовательность форматируемых объектов. Если любой из объектов относится к изменяющемуся типу **ObservableValue**, то изменяется и отформатированная строка
valueAt (double|float|integer|long) ValueAt stringValueAt	Типа **ObservableList** и индекс или типа **ObservableMap** и ключ
create(Boolean|Double|Float|Integer|Long|Object|String)Binding	Типа **Callable** и список зависимостей
select select(Boolean|Double|Float|Integer|Long|String)	Типа **Object** или **ObservableValue** и последовательность имен открытых свойств, дающих свойство *obj*.p_1.p_2. . . . .p_n
when	Дает построитель условного оператора. Привязка **when(***b***).then(***v_1***).otherwise(***v_2***)** дает v_1 или v_2 в зависимости от того, имеет ли аргумент *b* типа **ObservableBooleanValue** логическое значение **true**. Здесь v_1 или v_2 может принимать регулярное или наблюдаемое значение. Условное значение вычисляется повторно всякий раз, когда изменяется наблюдаемое значение

Привязка вычисленного значения с помощью методов из класса `Bindings` может получиться довольно вычурной. Впрочем, имеется другой, более простой способ получения вычисленных привязок. Для этого достаточно разместить вычисляемое выражение в лямбда-выражении и предоставить список зависимых свойств. При

изменении любых свойств лямбда-выражение вычисляется повторно, как показано в приведенном ниже примере. В упражнении 5, приведенном в конце этой главы, предлагается еще более изящный способ вычисления привязок по требованию с помощью лямбда-выражений.

```
larger.disableProperty().bind(
  createBooleanBinding(
    () -> gauge.getWidth() >= 100, // это выражение вычисляется . . .
    gauge.widthProperty())); // . . . когда изменяется данное свойство
```

НА ЗАМЕТКУ. Компилятор языка JavaFX Script анализирует выражения привязки и автоматически определяет зависимые свойства. В рассмотренном выше примере кода было объявлено следующее: **`disable bind gauge.width >= 100`**, и поэтому компилятор присоединил приемник событий к свойству **`gauge.width property`**. А программирующему на Java подобные сведения приходится предоставлять вручную.

Компоновка

Если ГПИ содержит несколько элементов управления, они должны быть расположены на экране как с точки зрения выполняемых ими функций, так и с точки зрения удобства применения. Компоновку элементов управления ГПИ можно осуществить с помощью специального инструментального средства, предназначенного для разработки пользовательского интерфейса. Пользователь такого инструментального средства (зачастую графический дизайнер) перетаскивает графические изображения элементов управления в представлении конструирования, располагая их, изменяя размеры и настраивая. Но такой способ построения ГПИ может вызвать определенные трудности, например, при изменении размеров элементов, поскольку метки могут иметь разную длину при интернационализации программы на разных языках.

С другой стороны, компоновка может быть достигнута программным способом, т.е. написанием кода в методе установки, вводящим элементы управления в пользовательский интерфейс на отдельных позициях. Именно так и было сделано в библиотеке `Swing` с помощью объектов диспетчеров компоновки.

Еще один способ состоит в том, чтобы определить компоновку на декларативном, непроцедурном языке. Например, веб-страницы компонуются средствами HTML и CSS. Аналогично на платформе Android имеется отдельный язык XML для определения компоновок.

В JavaFX поддерживаются все три упомянутых выше способа. В частности, SceneBuilder выполняет в JavaFX роль визуального конструктора ГПИ. Это инструментальное средство можно загрузить по адресу `www.oracle.com/technetwork/java/javafx/overview`. На рис. 4.6 показан моментальный снимок экрана SceneBuilder. Это инструментальное средство здесь не рассматривается, но, уяснив основные принципы компоновки ГПИ, вы сможете без особого труда пользоваться SceneBuilder на практике.

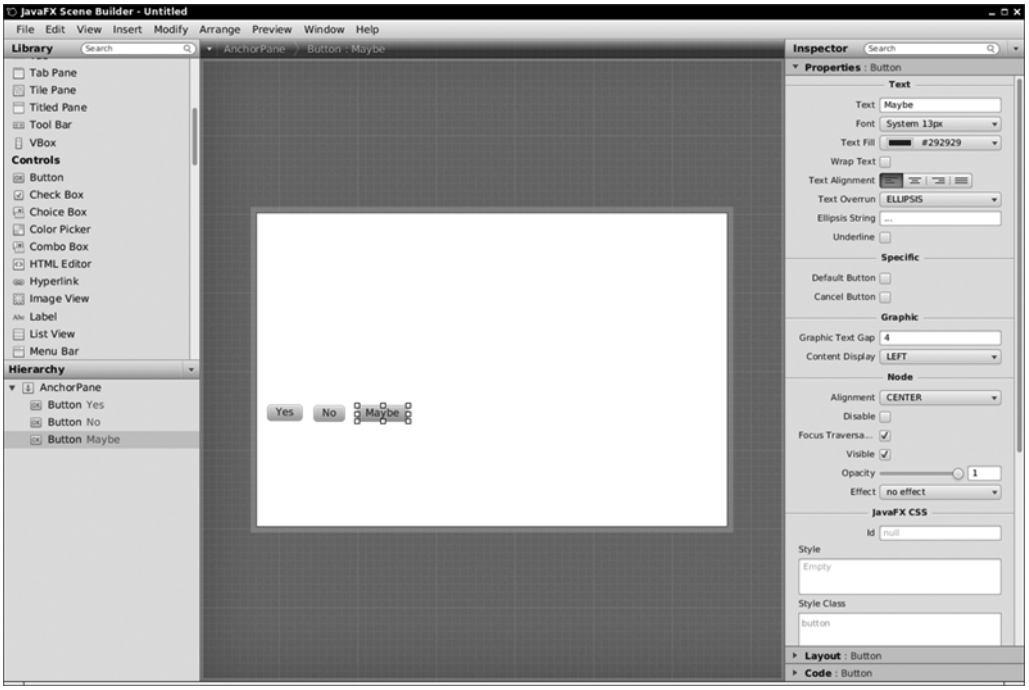

Рис. 4.6. Инструментальное средство SceneBuilder в JavaFX

Программная компоновка осуществляется во многом так же, как и в Swing. Но вместо диспетчеров компоновки, вводимых на произвольных панелях, в данном случае используются *панели* — контейнеры с правилами компоновки. Например, на панели типа `BorderPane` имеются пять областей: северная, западная, южная, восточная и центральная. В приведенном ниже примере показано, каким образом кнопки располагаются в этих областях панели, а результат приведен на рис. 4.7.

```
BorderPane pane = new BorderPane();
pane.setTop(new Button("Top"));
pane.setLeft(new Button("Left"));
pane.setCenter(new Button("Center"));
pane.setRight(new Button("Right"));
pane.setBottom(new Button("Bottom"));
stage.setScene(new Scene(pane));
```

Рис. 4.7. Компоновка кнопок на панели типа `BorderPane`

НА ЗАМЕТКУ. С помощью компонента `BorderLayout` из библиотеки Swing кнопки компоновались таким образом, чтобы заполнить собой всю компонуемую область. А в JavaFX кнопка не выходит на свои естественные размеры.

А теперь допустим, что в южной области компоновки требуется расположить больше одной кнопки. Для этой цели служит панель типа `HBox`, как показано ниже и на рис. 4.8.

```
HBox box = new HBox(10); // промежуток 10 пикселей между элементами управления
box.getChildren().addAll(yesButton, noButton, maybeButton);
```

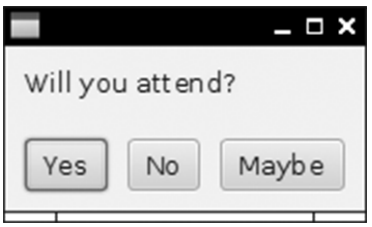

Рис. 4.8. Компоновка кнопок с помощью панели `HBox`

Безусловно, для компоновки элементов управления по вертикали имеется панель типа `VBox`. А компоновка кнопок, приведенная на рис. 4.8, была достигнута с помощью следующего фрагмента кода:

```
VBox pane = new VBox(10);
pane.getChildren().addAll(question, buttons);
pane.setPadding(new Insets(10));
```

Обратите внимание на свойство `padding`. Без него метка и кнопки будут сливаться с границей окна, не отделяясь от нее заданным промежутком.

ВНИМАНИЕ. Размеры в JavaFX указываются в пикселях. В данном примере для заполнения и расстановки кнопок с пробелами выбрана величина промежутка, равная **10** пикселям. Но в настоящее время указывать размеры в пикселях уже стало неуместно, поскольку плотность расположения пикселей может заметно отличаться. Это затруднение можно, в частности, преодолеть, вычисляя размеры в *корневых круглых шпациях* (rem), как это делается по стандарту CSS3. (Корневая круглая шпация обозначает высоту исходного шрифта в корне документа.) В приведенном ниже примере кода показано, как это делается.

```
final double rem = new Text("").getLayoutBounds().getHeight();
pane.setPadding(new Insets(0.8 * rem));
```

Но это все, чего можно добиться с помощью горизонтальных и вертикальных прямоугольных областей компоновки. Если в Swing компонент `GridBagLayout` составлял основу всех диспетчеров компоновки, то в JavaFX аналогичную роль выполняет панель типа `GridPane`. Ее можно сравнить с HTML-таблицей. На этой панели можно задать выравнивание всех ячеек по горизонтали и по вертикали, а при желании — сделать так, чтобы ячейки заполняли несколько рядов и столбцов. В качестве примера на рис. 4.9 показана компоновка диалогового окна регистрации.

Рис. 4.9. Панель типа `GridPane` позволяет расположить элементы управления в этом диалоговом окне регистрации

Имейте в виду следующее:

- метки `"User name"` (Имя пользователя) и `"Password"` (Пароль) выравниваются по правому краю;
- кнопки располагаются на панели типа `HBox`, охватывающем два ряда.

Когда на панели типа `GridPane` вводится дочерний элемент, следует указать индексы ряда и столбца его расположения (именно в таком порядке, т.е. аналогично координатам *x*, *y*). В приведенном ниже примере кода показано, как это делается.

```
pane.add(usernameLabel, 0, 0);
pane.add(username, 1, 0);
pane.add(passwordLabel, 0, 1);
pane.add(password, 1, 1);
```

Если же дочерний элемент охватывает несколько столбцов или рядов, следует также указать охватываемую область после места его расположения. Например, в следующей строке кода указывается, что панель кнопок охватывает два столбца и один ряд:

```
pane.add(buttons, 0, 2, 2, 1);
```

А если требуется, чтобы дочерний элемент охватывал все оставшиеся ряды или столбцы, следует воспользоваться свойством `GridPane.REMAINING`. Для выравнивания дочернего элемента по горизонтали достаточно вызвать метод `setHalignment()`, передав ссылку на дочерний элемент, а также константу `LEFT`, `CENTER` или `RIGHT` из перечисления типа `HPos`, как показано ниже. Аналогично для вертикального выравнивания достаточно вызвать метод `setValignment()`, передав ему константу `TOP`, `CENTER` или `BOTTOM` из перечисления типа `VPos`.

```
GridPane.setHalignment(usernameLabel, HPos.RIGHT);
```

НА ЗАМЕТКУ. Вызовы этих статических методов выглядят не совсем изящно в коде Java. Тем не менее они имеют смысл в языке разметки FXML, как поясняется в следующем разделе.

ВНИМАНИЕ. *Не* выравнивайте по центру панель типа `HBox` с кнопками, расположенными сеткой. Эта прямоугольная область компоновки распространяется по всей ширине, поэтому центровка не изменяет расположение данной области. Поэтому лучше отцентровать не саму панель типа `HBox`, а его содержимое, как показано ниже.

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

Ряды и столбцы, а также саму таблицу требуется также отделить пробелами следующим образом:

```
pane.setHgap(0.8 * em);
pane.setVgap(0.8 * em);
pane.setPadding(new Insets(0.8 * em));
```

СОВЕТ. Для целей отладки было бы также полезно отобразить границы ячеек (рис. 4.10), сделав следующий вызов:

```
pane.setGridLinesVisible(true);
```

Если же требуется отобразить границы отдельного дочернего элемента (например, для того, чтобы было видно, заполняет ли он всю ячейку), их следует задать. И это проще всего сделать с помощью таблицы стилей CSS следующим образом:

```
buttons.setStyle("-fx-border-color: red;");
```

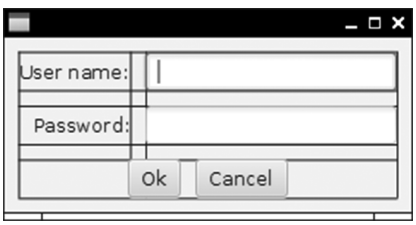

Рис. 4.10. При отладке компоновки на панели типа `GridPane` используются видимые линии сетки

Упомянутых выше панелей должно быть достаточно для компоновки ГПИ большинства приложений. В табл. 4.2 приведены все компоновки, создаваемые средствами JavaFX.

Таблица 4.2. Компоновки в JavaFX

Класс панели	Описание
`HBox`, **`VBox`**	Выравнивает дочерние элементы по горизонтали или по вертикали
`GridPane`	Располагает дочерние элементы в виде табличной сетки аналогично компоненту **`GridBagLayout`** в Swing
`TilePane`	Располагает дочерние элементы в виде сетки и одинакового размера аналогично компоненту **`GridLayout`** в Swing
`BorderPane`	Предоставляет для компоновки северную, восточную, южную, западную и центральную области аналогично компоненту **`BorderLayout`** в Swing
`FlowPane`	Располагает дочерние элементы рядами, создавая новые ряды, если недостаточно свободного места, аналогично компоненту **`FlowLayout`** в Swing
`AnchorPane`	Дочерние элементы могут быть расположены на абсолютных позициях или относительно границ панели. Такой режим выбирается по умолчанию в инструменте компоновки SceneBuilder
`StackPane`	Располагает дочерние элементы друг над другом. Такая компоновка может оказаться полезной для оформления компонентов, например, для размещения кнопки на цветном прямоугольнике

НА ЗАМЕТКУ. В этом разделе рассматривается построение пользовательских интерфейсов ручным вложением панелей и элементов управления. В языке JavaFX Script имелся синтаксис "построителя" для описания таких вложенных структур, называемых "графом сцены". В версии JavaFX 2 применялись классы построителей для имитации данного синтаксиса. Ниже показано, как построить подобными средствами диалоговое окно регистрации.

```
GridPane pane = GridPaneBuilder.create()
    .hgap(10)
    .vgap(10)
    .padding(new Insets(10))
    .children(
       usernameLabel = LabelBuilder.create()
       .text("User name:")
       .build(),
        passwordLabel = LabelBuilder.create()
       .text("Password:")
        .build(),
        username = TextFieldBuilder.create().build(),
        password = PasswordFieldBuilder.create().build(),
        buttons = HBoxBuilder.create()
       .spacing(10)
       .alignment(Pos.CENTER)
       .children(
          okButton = ButtonBuilder.create()
          .text("Ok")
          .build(),
          cancelButton = ButtonBuilder.create()
          .text("Cancel")
          .build())
       .build())
    .build();
```

Такой код выглядит довольно громоздким, и тем не менее он не избавляет от необходимости указывать ограничения на сетку. Построители элементов ГПИ стали нерекомендованными к применению в версии JavaFX 8 не из-за громоздкости кода, а из-за трудностей реализации. Ради экономии кода в построителях применяется дерево наследования, распараллеливающее наследование соответствующих узлов. Например, класс `GridPaneBuilder` расширяет класс `PaneBuilder`, поскольку класс `GridPane` расширяет класс `Pane`. Но при этом возникает вопрос: что именно должен возвращать метод `PaneBuilder.children()`? Если он возвращает только объект типа `PaneBuilder`, то пользователь должен проявить особую аккуратность, настроив сначала свойства подкласса, а затем и свойства суперкласса.

Разработчики постарались разрешить данное затруднение с помощью обобщений. Методы из обобщенного класса `PaneBuilder<B>` возвращают объект типа `B`, а следовательно, класс `GridPaneBuilder` может расширять класс `PaneBuilder<GridPaneBuilder>`. Но дело в том, что сам класс `GridPaneBuilder` является обобщенным, и поэтому вполне допустимо его расширение `GridPaneBuilder<GridPaneBuilder>` или даже `GridPaneBuilder<GridPaneBuilder<нечто>>`. Подобное зацикливание было преодолено специальными приемами, но они не совсем надежны и вряд ли окажутся работоспособными в последующих версиях Java. Поэтому от построителей пришлось отказаться. Если же построители элементов ГПИ вас вполне устраивают, воспользуйтесь языком Scala или Groovy и его привязками к JavaFX (`https://code.google.com/p/scalafx`; `http://groovyfx.org`).

Язык разметки FXML

Для описания компоновок в JavaFX применяется язык разметки FXML под названием FXML. Это язык разметки рассматривается здесь в некоторых подробностях потому, что он основывается на ряде принципов, интерес к которым выходит за рамки JavaFX, а реализуется он обыкновенным способом. Ниже приведен пример разметки в коде FXML диалогового окна регистрации из предыдущего примера.

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

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

<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  <children>
    <Label text="User name:" GridPane.columnIndex="0" GridPane.rowIndex="0"
        GridPane.halignment="RIGHT" />
    <Label text="Password: " GridPane.columnIndex="0" GridPane.rowIndex="1"
        GridPane.halignment="RIGHT" />
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
    <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1" />
    <HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
         GridPane.columnSpan="2" alignment="CENTER" spacing="10">
      <children>
        <Button text="Ok" />
        <Button text="Cancel" />
      </children>
    </HBox>
  </children>
</GridPane>
```

Если проанализировать приведенный выше код FXML-разметки, то в нем можно обнаружить команды обработки `<?import ....*?>` для импорта пакетов Java. (Как правило, команды обработки в XML служат спасительным выходом для тех приложений, где требуется обрабатывать XML-документы.)

А теперь рассмотрим структуру данного FXML-документа. Прежде всего, вложение панели типа `GridPane`, меток и текстовых полей, панели типа `HBox` и ее дочерних элементов в виде кнопок отражает вложение, организованное в коде Java из предыдущего раздела. Большинство атрибутов соответствуют методам установки свойств. Например, приведенная ниже строка кода означает следующее: "построить панель типа `GridPane` и установить свойства `hgap` и `vgap`".

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

Если атрибут начинается с имени класса и статического метода, этот метод вызывается. Например, следующая строка кода означает, что будут вызваны статические методы `GridPane.setColumnIndex(thisTextField, 1)` и `GridPane.setRowIndex(thisTextField, 0)`:

```
<TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

НА ЗАМЕТКУ. Как правило, элемент разметки FXML сначала конструируется своим конструктором по умолчанию, а затем специально настраивается с помощью вызываемых методов установки свойств или статических методов в духе спецификации JavaBeans. Некоторые исключения из данного правила будут рассмотрены далее в этой главе.

Если свойство слишком трудно выразить в виде символьной строки, то вместо атрибутов применяются вложенные элементы. Рассмотрим следующий пример:

```
<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  ...
```

Свойство `padding` содержит объект типа `Insets`, построенный с помощью дочернего элемента разметки `<Insets ...>`, в котором определяется порядок установки его свойств. И наконец, для списочных свойств имеется особое правило. Например, `children` — это списочное свойство, и в результате обращения к нему в следующем коде разметки вводятся кнопки в список, возвращаемый методом `getChildren()`.

```
<HBox ...>
  <children>
    <Button text="Ok" />
    <Button text="Cancel" />
  </children>
</HBox>
```

Файлы FXML-документов можно составить вручную или же воспользоваться для этой цели инструментальным средством SceneBuilder, упоминавшимся в предыдущем разделе. Получив такой файл, его можно загрузить следующим образом:

```
public void start(Stage stage) {
  try {
    Parent root = FXMLLoader.load(getClass().getResource("dialog.fxml"));
    stage.setScene(new Scene(root));
    stage.show();
  } catch (IOException ex) {
    ex.printStackTrace();
    System.exit(0);
  }
}
```

Разумеется, такой код пока еще не является полезным сам по себе. Пользовательский интерфейс отображается, но программа не имеет доступа к значениям, предоставляемым пользователем. Установить связь между элементами управления пользовательского интерфейса и программой можно, в частности, воспользовавшись атрибутами `id`, как это обычно делается в JavaScript. Атрибуты `id` предоставляются в FXML-файле следующим образом:

```
<TextField id="username" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

А в программе элемент управления находится следующим образом:

```
TextField username = (TextField) root.lookup("#username");
```

Но имеется и лучший способ. В частности, аннотацию `@FXML` можно использовать для "внедрения" управляющих объектов в класс *контроллера*. А в классе контроллера должен быть реализован интерфейс `Initializable`. В методе `initialize()` из класса контроллера средства привязки связываются с обработчиками событий. В качестве контроллера может служить любой класс — даже само JavaFX-приложение. Например, в приведенном ниже коде реализуется контроллер для управления диалоговым окном регистрации.

```
public class LoginDialogController implements Initializable {
    @FXML private TextField username;
    @FXML private PasswordField password;
    @FXML private Button okButton;

    public void initialize(URL url, ResourceBundle rb) {
      okButton.disableProperty().bind(
        Bindings.createBooleanBinding(
            () -> username.getText().length() == 0
                || password.getText().length() == 0,
            username.textProperty(),
            password.textProperty()));
        okButton.setOnAction(event ->
            System.out.println("Verifying " + username.getText()
                + ":" + password.getText()));
}
```

В файле FXML-документа следует предоставить имена переменных экземпляра контроллера для соответствующих элементов управления в этом файле. Для этой цели служит атрибут `fx:id`, а не атрибут `id`, как выделено ниже полужирным.

```
<TextField fx:id="username" GridPane.columnIndex="1"
    GridPane.rowIndex="0"/>
<PasswordField fx:id="password" GridPane.columnIndex="1"
    GridPane.rowIndex="1" />
<Button fx:id="okButton" text="Ok" />
```

В корневом элементе необходимо также объявить класс контроллера. А для этой цели служит атрибут `fx:controller`, как показано ниже. Обратите внимание на атрибут для внедрения пространства имен FXML.

```
<GridPane xmlns:fx="http://javafx.com/fxml" hgap="10" vgap="10"
    fx:controller="LoginDialogController">
```

НА ЗАМЕТКУ. Если у контроллера отсутствует конструктор по умолчанию (возможно, потому что он инициализирован по ссылке на бизнес-сервис), то его можно установить программным способом, как показано ниже.

```
FXMLLoader loader = new FXMLLoader(getClass().getResource(...));
loader.setController(new Controller(service));
Parent root = (Parent) loader.load();
```

ВНИМАНИЕ. Если контроллер устанавливается программным способом, то для этой цели вполне подходит исходный код из предыдущей врезки. Приведенный ниже код будет скомпилирован, но в нем будет вызван статический метод `FXMLLoader.load()` в обход построенного загрузчика.

```
FXMLLoader loader = new FXMLLoader();
loader.setController(...);
Parent root = (Parent) loader.load(getClass().getResource(...));
  // Ошибка: вызывается статический метод
```

При загрузке файла FXML-документа конструируется граф сцены, а ссылки на именованные объекты управления внедряются в аннотированные поля объекта контроллера. Затем вызывается его метод `initialize()`.

Большую часть инициализации вполне возможно сделать в файле FXML-документа. В частности, можно определить простые привязки и задать аннотированные методы контроллера в качестве приемников событий. Не вдаваясь в подробности, заметим, что документация на соответствующий синтаксис доступна по адресу `http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html`. По-видимому, визуальный дизайн лучше отделить от поведения программы, чтобы разработчик пользовательского интерфейса мог заниматься графическим оформлением программы, а программист — реализацией ее поведения.

НА ЗАМЕТКУ. В файл FXML-документа можно также ввести сценарии на JavaScript или другом языке написания сценариев. Этот вопрос вкратце обсуждается в главе 7.

Таблицы стилей CSS

В JavaFX допускается изменение внешнего вида пользовательского интерфейса с помощью таблиц стилей CSS, что, как правило, удобнее, чем предоставление атрибутов FXML-разметки или вызов методов в Java. Таблицу стилей CSS можно загрузить программно и применить ее к графу сцены следующим образом:

```
Scene scene = new Scene(pane);
scene.getStylesheets().add("scene.css");
```

В таблице стилей можно обращаться к любым элементам управления, имеющим идентификатор. В качестве примера рассмотрим, как организовать управление внешним видом панели типа `GridPane`. Идентификатор этой панели устанавливается в коде следующим образом:

```
GridPane pane = new GridPane();
pane.setId("pane");
```

А любое заполнение промежутков или расстановка устанавливается не в коде, а в таблице CSS, как показано ниже.

```
#pane {
  -fx-padding: 0.5em;
  -fx-hgap: 0.5em;
  -fx-vgap: 0.5em;
  -fx-background-image: url("metal.jpg")
}
```

К сожалению, вместо известных атрибутов CSS придется освоить и применять специальные атрибуты JavaFX, имена которых начинаются с префикса **`-fx-`** и образуются из имен свойств в нижнем регистре и дефисов вместо смешанного написания.

Например, свойство `textAlignment`, определяющее выравнивание текста, превращается в соответствующий атрибут `-fx-text-alignment`. Все атрибуты CSS, поддерживаемые в JavaFX, можно найти по адресу `http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html`.

Пользоваться таблицами CSS удобнее, чем загромождать код деталями компоновки ГПИ. Кроме того, можно без особого труда пользоваться круглыми шпациями как единицами измерения, независящими от разрешения. Разумеется, пользование таблицами CSS способно принести не только пользу, но и нанести вред. В частности, следует удерживаться от искушения применять неуместные фоновые текстуры в диалоговых окнах регистрации (рис. 4.11).

Рис. 4.11. Неуместное применение фоновой текстуры для стилизации пользовательского интерфейса средствами CSS

Вместо стилизации по отдельным идентификаторам можно воспользоваться классами стилей. С этой целью класс стиля вводится в узловой объект следующим образом:

```
HBox buttons = new HBox();
buttons.getStyleClass().add("buttonrow");
```

Затем он стилизуется с помощью следующего обозначения класса CSS:

```
.buttonrow {
  -fx-spacing: 0.5em;
}
```

Каждый класс элемента управления или формы в JavaFX относится к классу CSS, имя которого образуется из имени класса Java, приведенного в нижний регистр. Например, все узлы типа `Label` относятся к классу `label`. В качестве примера ниже показано, как заменить на Comic Sans шрифт всех меток, хотя на самом деле делать этого не нужно.

```
.label {
  -fx-font-family: "Comic Sans MS";
}
```

Кроме того, таблицы стилей CSS можно применять в FXML-компоновках. Для этого достаточно присоединить таблицу стилей к корневой панели следующим образом:

```
<GridPane id="pane" stylesheets="scene.css">
```

Код FXML-разметки снабжается атрибутами `id` или `styleClass`, как показано в приведенном ниже примере.

```
<HBox styleClass="buttonrow">
```

Затем бо́льшую часть стилизации можно указать в таблице стилей CSS, а FXML-разметку использовать только для компоновки. К сожалению, стилизацию нельзя полностью удалить из FXML-разметки. Так, в настоящее время отсутствует способ, позволяющий указывать выравнивание ячейки сеточной компоновки в таблице стилей CSS.

НА ЗАМЕТКУ. Стили CSS можно применять и программным способом, например, так, как показано ниже.

```
buttons.setStyle("-fx-border-color: red;");
```

Такой способ может оказаться удобным для отладки, но, как правило, лучше пользоваться внешними таблицами стилей.

Анимация и спецэффекты

На момент появления JavaFX спецэффекты были весьма распространены, и поэтому в JavaFX имеются средства, позволяющие без особого труда отбрасывать тени, размывать изображение или перемещать объекты. В Интернете можно найти немало примеров, демонстрирующих беспорядочное движение всплывающих пузырьков, нервно скачущий текст и прочие анимационные эффекты. В этой связи было бы полезно дать некоторые рекомендации по поводу применения подобных анимационных эффектов в прикладных программах. В качестве примера на рис. 4.12 показано приложение, в котором размеры кнопки Yes (Да) постепенно увеличиваются, кнопка No (Нет) сливается с фоном, а кнопка Maybe (Может быть) вращается.

Рис. 4.12. Кнопки, которые постепенно увеличиваются, обесцвечиваются и вращаются

В JavaFX определен целый ряд *монтажных переходов*, изменяющих свойства узлов во времени. В качестве примера ниже показано, каким образом в коде реализуется увеличение размеров узла наполовину в направлении обеих осей координат x и y в течение трех секунд.

```
ScaleTransition st = new ScaleTransition(Duration.millis(3000));
st.setByX(1.5);
st.setByY(1.5);
st.setNode(yesButton);
st.play();
```

В качестве узла для монтажного перехода может служить любой узел графа сцены, например, кружок в анимации мыльных пузырьков или более привлекательная кнопка Yes в диалоговом окне, приведенном на рис. 4.12. Установленный однажды монтажный переход завершается по достижении своей цели. Его циклическое повторение можно организовать следующим образом:

```
st.setCycleCount(Animation.INDEFINITE);
st.setAutoReverse(true);
```

Теперь узел будет то увеличиваться, то уменьшаться до бесконечности. При монтажном переходе типа `FadeTransition` постепенно изменяется непрозрачность узла. Ниже показано, каким образом в коде реализуется постепенное слияние кнопки No с фоном.

```
FadeTransition ft = new FadeTransition(Duration.millis(3000));
ft.setFromValue(1.0);
ft.setToValue(0);
ft.setNode(noButton);
ft.play();
```

Все узлы JavaFX могут вращаться вокруг своего центра. Так, при монтажном переходе типа `RotateTransition` изменяется свойство `rotate` вращаемого узла. В приведенном ниже фрагменте кода осуществляется анимация кнопки Maybe.

```
RotateTransition rt = new RotateTransition(Duration.millis(3000));
rt.setByAngle(180);
rt.setCycleCount(Animation.INDEFINITE);
rt.setAutoReverse(true);
rt.setNode(maybeButton);
rt.play();
```

Монтажные переходы можно составлять, сочетая объекты типа `ParallelTransition` и `SequentialTransition`, чтобы выполнять эти переходы параллельно или последовательно. Если же требуется анимация нескольких узлов, их достаточно разместить в групповом узле типа `Group`, а затем привести в движение. И для того чтобы добиться такого поведения, очень удобно пользоваться классами JavaFX.

Спецэффекты создаются так же просто. Если, например, требуется отбрасывать тени от нарядной надписи, такой эффект можно создать с помощью класса `DropShadow`, задав его в качестве свойства `effect` узла. На рис. 4.13 приведен результат применения эффекта отбрасывания тени к узлу типа `Text`, а ниже показано, каким образом данный эффект реализуется в коде.

```
DropShadow dropShadow = new DropShadow();
dropShadow.setRadius(5.0);
dropShadow.setOffsetX(3.0);
dropShadow.setOffsetY(3.0);
dropShadow.setColor(Color.GRAY);

Text text = new Text();
text.setFill(Color.RED);
text.setText("Drop shadow");
text.setFont(Font.font("sans", FontWeight.BOLD, 40));
text.setEffect(dropShadow);
```

Рис. 4.13. Спецэффекты, получаемые средствами JavaFX

Эффекты свечения или размытия, приведенные на рис. 4.13, реализуются так же просто. Ниже показано, как это делается.

```
text2.setEffect(new Glow(0.8));
text3.setEffect(new GaussianBlur());
```

Правда, эффект свечения выглядит не очень привлекательно, а эффект размытия вряд ли найдет широкое применение в большинстве прикладных программ. Тем не менее нельзя не отметить, насколько просто получить такие спецэффекты.

Декоративные элементы управления

Разумеется, в JavaFX, как и в Swing, имеются комбинированные списки, панели вкладок, деревья, таблицы, а также элементы управления пользовательского интерфейса, отсутствующие в Swing, в том числе селектор дат и меню-гармошка. Для подробного их описания потребовалась бы отдельная книга. Поэтому, для того чтобы развеять ностальгию по Swing, в этом разделе будут рассмотрены три декоративных элемента управления, функциональные возможности который выходят далеко за рамки того, что может предложить Swing.

На рис. 4.14 показана одна из многих диаграмм, которые можно построить средствами JavaFX. И для этого не потребуется специально устанавливать сторонние библиотеки.

А в приведенном ниже фрагменте кода показано, насколько просто построить круговую диаграмму средствами JavaFX.

```
ObservableList<PieChart.Data> pieChartData =
  FXCollections.observableArrayList(
    new PieChart.Data("Asia", 4298723000.0),
    new PieChart.Data("North America", 355361000.0),
    new PieChart.Data("South America", 616644000.0),
    new PieChart.Data("Europe", 742452000.0),
    new PieChart.Data("Africa", 1110635000.0),
```

```
    new PieChart.Data("Oceania", 38304000.0));
final PieChart chart = new PieChart(pieChartData);
chart.setTitle("Population of the Continents");
```

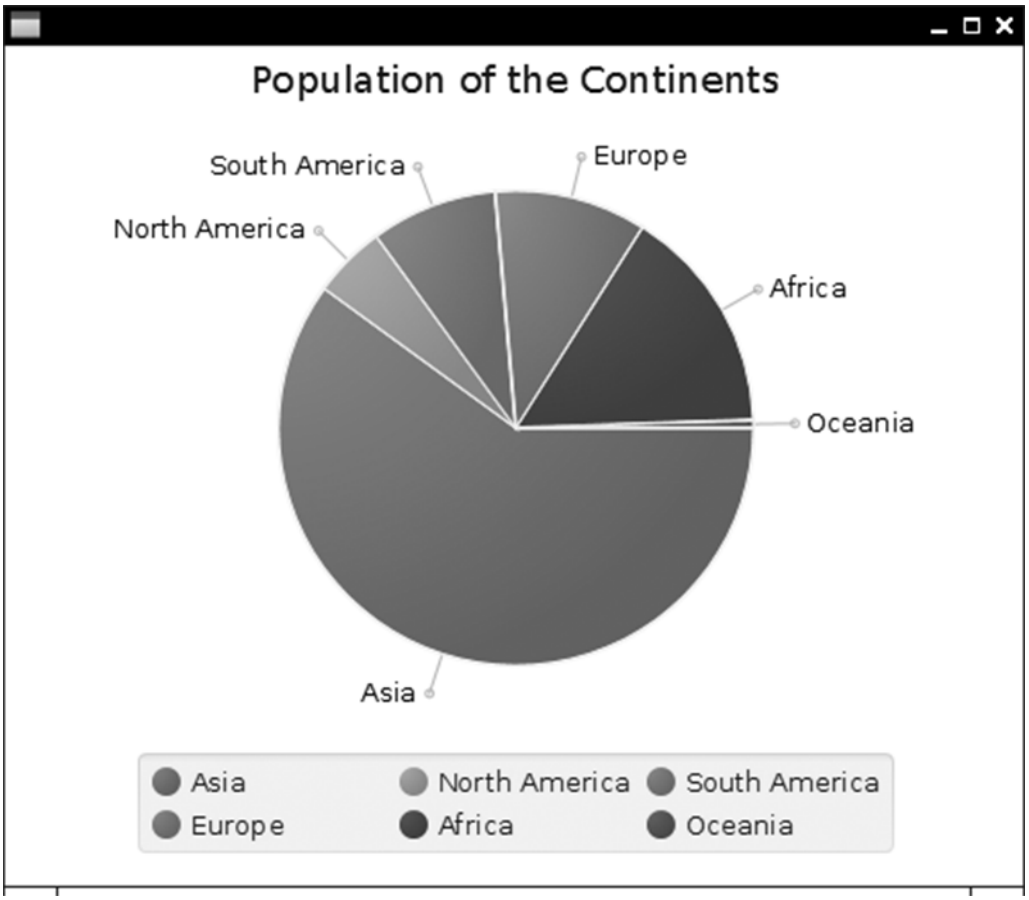

Рис. 4.14. Круговая диаграмма, построенная средствами JavaFX

Вообще в JavaFX имеется с полдесятка разновидностей диаграмм, которые можно применять в готовом виде или же специально настраивать. Подробнее об этом можно узнать из документации, доступной по адресу http://docs.oracle.com/javafx/2/charts/jfxpub-charts.htm.

В Swing можно было отображать HTML-разметку в компоненте `JEditorPane`, но воспроизведение настоящих HTML-документов, как правило, поддерживалось слабо. И это понятно, ведь реализовать подобные функциональные возможности на уровне большинства браузеров не так-то просто, поскольку они служат надстройкой над открытым кодом механизма WebKit. То же самое происходит и в JavaFX. В частности, класс `WebView` отображает встроенное машинно-зависимое окно WebKit, как показано на рис. 4.15.

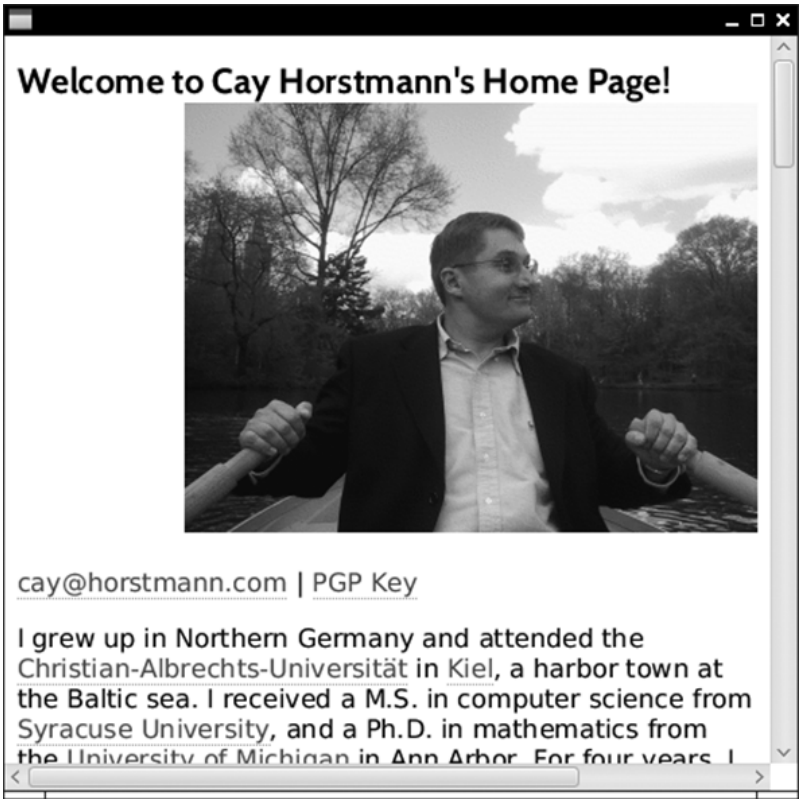

Рис. 4.15. Просмотр веб-страницы

Ниже приведен фрагмент кода для отображения веб-страницы, приведенной на рис. 4.15, средствами JavaFX.

```
String location = "http://horstmann.com";
WebView browser = new WebView();
WebEngine engine = browser.getEngine();
engine.load(location);
```

Браузер действует, реагируя обычным образом на выбор ссылок щелчком кнопкой мыши. Действует также и код сценариев JavaScript. Но если требуется отобразить строку состояния или всплывающие сообщения из сценария JavaScript, то придется установить обработчики уведомлений и реализовать отображение строки состояния или всплывающих сообщений самостоятельно.

НА ЗАМЕТКУ. В классе `WebView` не поддерживается ни один из подключаемых модулей, и поэтому его нельзя использовать для отображения Flash-анимации, документов формата PDF и даже аплетов.

До появления JavaFX воспроизведение мультимедийного содержимого было достойно сожаления. В частности, дополнительная загрузка такого содержимого была доступна в прикладном программном интерфейсе Java Media Framework, но это

мало привлекало разработчиков. Разумеется, реализовать воспроизведение мультимедийного содержимого сложнее, чем написать браузер. Поэтому в JavaFX используется оптимальное сочетание уже существующего набора инструментальных средств и библиотеки GStreamer с открытым кодом.

Для воспроизведения видеозаписей следует построить сначала объект типа `Media` из символьной строки с URL, затем мультимедийный проигрыватель типа `MediaPlayer` и, наконец, представление типа `MediaView` для отображения мультимедийного проигрывателя, как показано ниже.

```
Path path = Paths.get("moonlanding.mp4");
String location = path.toUri().toString();
Media media = new Media(location);
MediaPlayer player = new MediaPlayer(media);
player.setAutoPlay(true);
MediaView view = new MediaView(player);
view.setOnError(e -> System.out.println(e));
```

Как показано на рис. 4.16, видеозапись воспроизводится, но, к сожалению, отсутствуют элементы управления ее воспроизведением. И хотя их можно ввести самостоятельно (подробнее об этом см. в документации по адресу `http://docs.oracle.com/javafx/2/media/playercontrol.htm`), тем не менее, для этой цели было бы неплохо представить стандартный набор элементов управления.

Рис 4.16. Воспроизведение видеозаписи средствами JavaFX

НА ЗАМЕТКУ. Несмотря на все сказанное выше, в библиотеке GStreamer отсутствуют средства для обработки отдельных файлов видеозаписи. Обработчик ошибок в коде отображает сообщения GStreamer для целей диагностики ошибок, возникающих при воспроизведении.

На этом краткий обзор функциональных возможностей JavaFX завершается. С JavaFX связывается будущее разработки настольных приложений на Java. Эта технология еще не отлажена полностью в силу той поспешности, с которой пришлось переходить на нее из исходного языка написания сценариев. Но, безусловно, пользоваться средствами JavaFX не труднее, чем Swing, а имеющиеся среди них элементы управления намного более полезны и привлекательны, чем те, что когда-либо имелись в Swing.

Упражнения

1. Напишите программу с текстовым полем и соответствующей меткой. Как и в программе, выводящей сообщение "Hello, JavaFX!" средствами JavaFX, метка должна содержать это сообщение, выделенное шрифтом размером 100 пунктов. Инициализируйте текстовое поле тем же самым сообщением. А метку обновляйте по мере редактирования содержимого текстового поля.
2. Разработайте класс со многими свойствами JavaFX, например, для составления таблицы или диаграммы. Скорее всего, в конкретном приложении к большинству этих свойств не будут присоединены приемники событий, поэтому не стоит снабжать каждое свойство отдельным объектом. Покажите, каким образом можно устанавливать свойство по требованию, сначала с помощью обычного поля, в котором должно храниться значение свойства, а затем с помощью объекта свойства, но только в тот момент, когда метод `xxxProperty()` вызывается в первый раз.
3. Разработайте класс со многими свойствами JavaFX, большинство исходных значений которых вообще не изменяются. Покажите, каким образом свойство может быть установлено по умолчанию, когда задается неисходное значение или же когда метод `xxxProperty()` вызывается в первый раз.
4. Усовершенствуйте программу из раздела "Привязки" этой главы таким образом, чтобы круг остался отцентрованным и всегда касался, по крайней мере, двух сторон сцены.
5. Напишите следующие методы:

```
public static <T, R> ObservableValue<R> observe(
    Function<T, R> f, ObservableValue<T> t)
public static <T, U, R> ObservableValue<R> observe(
    BiFunction<T, U, R> f, ObservableValue<T> t, ObservableValue<U> u)
```

Они должны возвращать наблюдаемые значения, для которых метод `getValue()` возвращает значение лямбда-выражения, а приемники событий о недостоверности и изменениях данных запускаются в тот момент, когда любые вводимые данные становятся недостоверными или изменяются, как показано в приведенном ниже примере.

```
larger.disableProperty().bind(observe(
    t -> t >= 100, gauge.widthProperty()));
```

6. Отцентруйте верхнюю и нижнюю кнопки, приведенные на рис. 4.7.
7. Придумайте, как установить границу элемента управления, не прибегая к таблице стилей CSS.
8. Для синтаксического анализа файлов FXML-документов в JavaFX отсутствует соответствующая логика, поэтому придумайте пример загрузки объекта с рядом вложенных объектов, но не имеющего никакого отношения к JavaFX, а также установите свойства, используя синтаксис FXML. Было бы лучше, если бы вы воспользовались для этой цели внедрением кода.
9. Организуйте анимацию окружности, представляющей планету, чтобы она перемещалась по эллиптической орбите. Воспользуйтесь для этого классом `PathTransition`.
10. Используя средство просмотра веб-содержимого, реализуйте браузер со строкой для ввода URL и кнопкой возврата к прежней веб-странице. *Подсказка:* воспользуйтесь для этой цели методом `WebEngine.getHistory()`.