ЛАБОРАТОРНАЯ РАБОТА №5.
ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»
ТЕМА: «РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА. РАБОТА С ФРЕЙМАМИ, ЭЛЕМЕНТАМИ УПРАВЛЕНИЯ, ГРАФИКОЙ (БИБЛИОТЕКИ SWING, JAVAFX)»

1-40 05 01 «Информационные системы и технологии (в бизнес-менеджменте и промышленной безопасности)»

Цель: научиться создавать приложения с графическим пользовательским интерфейсом.

ПАКЕТЫ БИБЛИОТЕКИ SWING

Библиотека *Swing* – крупная подсистема, в которой задействовано большое количество пакетов.

Ниже перечислены пакеты, определенные в библиотеке *Swing* на момент написания данного материала.

Самым главным среди них является пакет *javax.swing* (*https://docs.oracle.com/javase/8/docs/api/index.html?javax/swing/package-summary.html*). Его следует импортировать в любую прикладную программу, пользующуюся библиотекой *Swing*. В этом пакете содержатся классы, реализующие базовые компоненты *Swing*.

SWING-ПРИЛОЖЕНИЕ

Swing приложения предъявляют особые требования, связанные с многопоточной обработкой.

Имеются две разновидности программ на *Java*, в которых обычно применяется библиотека *Swing*: настольные приложения и аплеты. В этом разделе будет рассмотрен пример создания настольного *Swing*-приложения.

Несмотря на всю краткость рассматриваемого здесь примера программы, он демонстрирует один из способов разработки *Swing*-приложения, а также основные средства библиотеки *Swing*. В данном примере используются два компонента *Swing*: *JFrame* и *JLabel*. Класс *JFrame* представляет контейнер верхнего уровня, который обычно применяется в *Swing*-приложениях, а класс *JLabel* – компонент *Swing*, создающий метку для отображения информации.

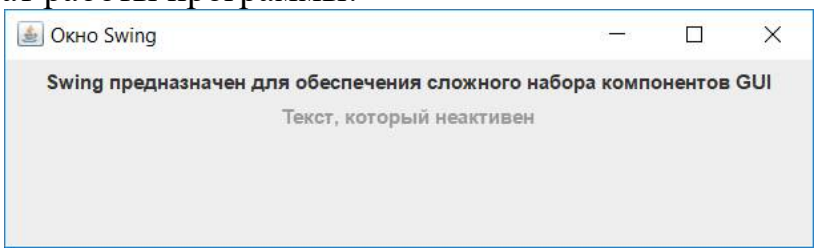
Метка является самым простым компонентом *Swing*, поскольку это пассивный компонент. Это означает, что **метка не реагирует на действия пользователя.** Она служит лишь для отображения выводимых данных. В данном примере контейнер типа *JFrame* служит для хранения метки в виде экземпляра класса *JLabel*. Метка отображает короткое текстовое сообщение.

```
//Пример №1. Простое Swing-приложение
import javax.swing.*;
import java.awt.*;
```

```
class SwingDemo {
    SwingDemo() {
        // создать новый контейнер типа JFrame
        JFrame jFrame = new JFrame("Окно Swing");
        jFrame.setLayout(new FlowLayout());
        // задать исходные размеры фрейма
        jFrame.setSize(500, 150);
        // завершить работу, если пользователь закрывает
приложение
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // создать метку с текстом сообщения
        JLabel jLabelFirst = new JLabel("Swing предназначен для
обеспечения сложного набора компонентов GUI");
        JLabel jLabelSecond = new JLabel("Текст, который
неактивен");
        jFrame.add(jLabelFirst);// ввести метку на панели
содержимого
        jFrame.add(jLabelSecond);// ввести метку на панели
содержимого
        jLabelSecond.setEnabled(false);
        jFrame.setVisible(true);// отобразить фрейм
    }
    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo();
            }
        });
        //альтернатива использования лямбда-выражения
        //SwingUtilities.invokeLater(() -> new SwingDemo());
        //альтернатива использования ссылки на конструктор
        //SwingUtilities.invokeLater(SwingDemo::new);
    }
}
```

Swing-приложения компилируются и выполняются таким же образом, как и остальные приложения *Java*.

Результат работы программы:



Это приложение демонстрирует ряд основных понятий библиотеки *Swing*. Данное *Swing* приложение начинается с импорт пакета *javax.swing*. Как упоминалось ранее, этот пакет содержит компоненты и модели, определяемые в библиотеке *Swing*. Так, в пакете *javax.swing* определяются классы, реализующие метки, кнопки, текстовые элементы управления и меню. Поэтому этот пакет обычно включается во все программы, пользующиеся

библиотекой *Swing*.

Затем объявляются класс *SwingDemo* и его конструктор, в котором сначала создается экземпляр класса *JFrame*:

```
JFrame jFrame = new JFrame("Окно Swing");
```

В методе *setSize()*, наследуемом классом *JFrame* от класса *Component* из библиотеки *AWT*, задаются размеры окна в пикселях. Ниже приведена общая форма этого метода. В данном примере задается ширина окна 500 пикселей, а высота – 100 пикселей.

```
void setSize(int width, int height);
```

Когда закрывается окно верхнего уровня (например, после того, как пользователь щелкнет на кнопке закрытия), по умолчанию окно удаляется с экрана, но работа приложения не прекращается. И хотя такое стандартное поведение иногда оказывается полезным, для большинства приложений оно не подходит. Чаще всего при закрытии окна верхнего уровня требуется завершить работу всего приложения. Это можно сделать двумя способами. Самый простой из них состоит в том, чтобы вызвать метод *setDefaultCloseOperation()*, что и делается в данном приложении следующим образом:

```
jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

В результате вызова этого метода приложение полностью завершает свою работу при закрытии окна. Общая форма метода *setDefaultCloseOperation()* выглядит следующим образом:

void	setDefaultCloseOperation(int operation) Sets the operation that will happen by default when the user initiates a "close" on this frame.

Значение константы, передаваемое в качестве параметра *operation* определяет, что именно происходит при закрытии окна. Помимо значения константы *JFrame.EXIT_ON_CLOSE*, имеются следующие значения:

Modifier and Type	Field and Description
static int	DISPOSE_ON_CLOSE The dispose-window default window close operation.
static int	DO_NOTHING_ON_CLOSE The do-nothing default window close operation.
static int	EXIT_ON_CLOSE The exit application default window close operation.
static int	HIDE_ON_CLOSE The hide-window default window close operation

Имена этих констант отражают выполняемые действия.

Эти константы объявляются в интерфейсе *WindowConstants*, который определяется в пакете *javax.swing* и реализуется в классе *JFrame*.

```
package javax.swing;

public interface WindowConstants {

    public static final int DO_NOTHING_ON_CLOSE = 0;
    public static final int HIDE_ON_CLOSE = 1;
    public static final int DISPOSE_ON_CLOSE = 2;
    public static final int EXIT_ON_CLOSE = 3;
}
```

Если открыто несколько *JFrames* и вы закрываете тот, для которого указана операция при закрытии *EXIT_ON_CLOSE,* он закроет все *JFrames* и выйдет из приложения. Если вы закроете тот, который имеет константу закрытия *DISPOSE_ON_CLOSE*, то будет закрыт только один *JFrame*. Если у вас есть только один *JFrame*, то между константами *EXIT_ON_CLOSE* и *DISPOSE_ON_CLOSE*.

В следующей строке кода создается компонент типа *JLabel* из библиотеки *Swing*:

```
JLabel jLabelFirst = new JLabel("Swing предназначен для обеспечения сложного набора компонентов GUI");
```

Класс *JLabel* определяет метку – самый простой в употреблении компонент, поскольку он не принимает вводимые пользователем данные, а только отображает информацию в виде текста, значка или того и другого. В данном примере создается метка, которая содержит только текст, передаваемый конструктору класса *JLabel*.

В следующей строке кода метка вводится на панели содержимого фрейма:

```
jFrame.add(jLabelFirst);
```

У всех контейнеров верхнего уровня имеется панель содержимого, на которой размещаются отдельные компоненты. Следовательно, чтобы ввести компонент во фрейм, его нужно ввести на панели содержимого фрейма. Для этого достаточно вызвать метод *add()* по ссылке на экземпляр класса *JFrame* (в данном случае экземпляр *jFrame*). Ниже приведена общая форма метода *add()*. Метод *add()* наследуется классом *JFrame* от класса *Container* из библиотеки *AWT*.

`Component`	`add(Component comp)` Appends the specified component to the end of this container.

По умолчанию на панели содержимого, связанной с компонентом типа *JFrame*, применяется граничная компоновка (*BorderLayout*). В отличие от диспетчера *FlowLayout*, который полностью контролирует позицию каждого компонента, диспетчер *BorderLayout* позволяет выбрать место для каждого компонента.

В приведенной выше форме метки вводится последовательно на панель

содержимого с использованием компоновки *FlowLayout*.

Прежде чем продолжить дальше, следует заметить, что до версии *JDK* 5 при вводе компонента на панели содержимого метод *add()* нельзя было вызывать непосредственно для экземпляра класса *JFrame*. Вместо этого метод *add()* приходилось вызывать для панели содержимого из объекта типа *JFrame*. Панель содержимого можно было получить в результате вызова метода *getContentPane()* для экземпляра класса *JFrame*. Ниже приведена общая форма метода *getContentPane()*.

`Container`	`getContentPane()` Returns the `contentPane` object for this frame.

Класс *Container* получает ссылку на окно содержимого. И по этой ссылке делается вызов метода *add()*, чтобы ввести компонент на панель содержимого. Следовательно, чтобы ввести метку *jLabel* во фрейм *jFrame*, раньше приходилось пользоваться следующим оператором:

```
jFrame.getContentPane().add(jLabel); // старый стиль
```

В этом операторе сначала вызывается метод *getContentPane()*, получающий ссылку на панель содержимого, а затем метод *add()*, вводит указанный компонент (в данном случае метку) в контейнер, связанный с этой панелью. Эта же процедура потребовалась бы и для вызова метода *remove()*, чтобы удалить компонент, или для вызова метода *setLayout()*, чтобы задать диспетчер компоновки для окна содержимого. Именно поэтому в коде, написанном на *Java* до версии 5.0, нередко встречаются вызовы метода *getContentPane()*.

Но теперь вызывать этот метод больше не нужно. Вместо этого достаточно вызвать методы *add()*, *remove()* и *setLayout()* непосредственно для объекта типа *JFrame*, поскольку они были специально изменены, чтобы автоматически оперировать панелью содержимого.

Последний оператор в конструкторе класса *SwingDemo* требуется для того, чтобы сделать окно видимым:

```
jFrame.setVisible(true);
```

Метод *setVisible()* наследуется от класса *Component* из библиотеки *AWT*. Если его аргумент принимает логическое значение *true*, то окно отобразится, а иначе оно будет скрыто. **По умолчанию контейнер типа *JFrame* невидим,** поэтому нужно вызвать метод *setVisible(true)*, чтобы показать его.

В методе *main()* создается объект типа *SwingDemo*, чтобы отобразить окно. Обратите внимание на то, что конструктор класса *SwingDemo* вызывается в следующих трех строка кода:

```
public static void main(String args[]) {
    // создать фрейм в потоке диспетчеризации событий
    SwingUtilities.invokeLater(new Runnable() {
```

```
        public void run() {
           new SwingDemo();
        }
});
```

Выполнение этой последовательности кода приводит к созданию объекта типа *SwingDemo* в потоке диспетчеризации событий, а не в главном потоке исполнения данного приложения, потому что *Swing*-приложения обычно выполняются под управлением событий.

Событие происходит, например, когда пользователь взаимодействует с компонентом. Событие передается приложению при вызове обработчика событий, определенного в этом приложении. Но обработчик выполняется в потоке диспетчеризации событий, предоставляемом библиотекой *Swing*, а не в главном потоке исполнения данного приложения. Таким образом, обработчики событий вызываются в потоке исполнения, который не был создан в приложении, хотя они и определены в нем.

Чтобы избежать осложнений, в том числе вероятной взаимной блокировки, все компоненты ГПИ из библиотеки *Swing* следует создавать и обновлять из потока диспетчеризации событий, а не из главного потока исполнения данного приложения. Но метод *main()* выполняется в главном потоке исполнения. Следовательно, метод *main()* не может напрямую создавать объект типа *SwingDemo*. Вместо этого нужно создать объект типа *Runnable*, который выполняется в потоке диспетчеризации событий, и с помощью этого объекта построить ГПИ.

Чтобы построить ГПИ в потоке диспетчеризации событий, следует вызвать один из следующих двух методов, определенных в классе *SwingUtilities*: *invokeLater()* и *invokeAndWait()*.

`static void`	`invokeAndWait(Runnable doRun)` Causes `doRun.run()` to be executed synchronously on the AWT event dispatching thread.
`static void`	`invokeLater(Runnable doRun)` Causes *doRun.run()* to be executed asynchronously on the AWT event dispatching thread.

Здесь параметр *doRun* обозначает объект типа *Runnable*, метод *run()* которого должен вызываться в потоке диспетчеризации событий. Единственное отличие этих двух методов заключается в том, что метод *invokeLater()* выполняет возврат немедленно, а метод *invokeAndWait()* ожидает возврат из метода *obj.run()*. Этими методами можно пользоваться для вызова метода, строящего ГПИ конкретного *Swing*-приложения, или вызывать их всякий раз, когда требуется изменить состояние ГПИ из кода, не выполняющегося в потоке диспетчеризации событий. Для этих целей обычно вызывается метод *invokeLater()*. А при построении исходного ГПИ для аплета потребуется метод *invokeAndWait()*.

ОБРАБОТКА СОБЫТИЙ

В предыдущем примере была продемонстрирована основная форма *Swing*- приложения, но ей недостает одной важной части: **обработки событий.** Компонент типа *JLabel* не принимает данные, вводимые пользователем, и не генерирует события. Но остальные компоненты библиотеки *Swing* реагируют

на вводимые пользователем данные, а, следовательно, требуется каким-то образом обработать события, наступающие в результате подобных взаимодействий. Событие происходит, когда, например, срабатывает таймер. Так или иначе, обработка событий занимает большую часть любого *Swing*-приложения.

Механизм обработки событий, применяемый в библиотеке *Swing*, ничем не отличается от аналогичного механизма из библиотеки *AWT*, называемого моделью делегирования событий.

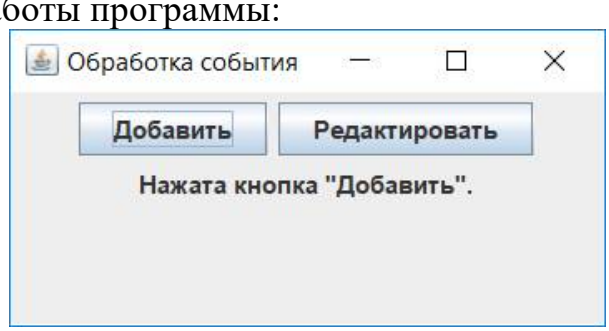
Как правило, в библиотеке *Swing* используются те же самые события, что и в библиотеке *AWT*, и эти события определены в пакете *java.awt.event.* А события, характерные только для библиотеки *Swing*, определены в пакете *javax.swing.event.*

В приведенной ниже программе обрабатывается событие, генерируемое после щелчка на экранной кнопке, определяемой соответствующим компонентом *Swing*.

```
//Пример №2. Обработка событий в Swing-приложении
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
    JLabel jLabel;
    EventDemo() {
        // создать новый контейнер типа JFrame
        JFrame jFrame = new JFrame("Обработка события");
        //   определить   диспетчер   поточной   компоновки   типа
FlowLayout
        jFrame.setLayout(new FlowLayout());
        // установить исходные размеры фрейма
        jFrame.setSize(300, 150);
        //завершить работу приложения, если пользователь закрывает
окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // создать две кнопки
        JButton jbtnAlpha = new JButton("Добавить");
        JButton jbtnBeta = new JButton("Редактировать");
        // ввести приемник действий от кнопки jbtnAlpha
        jbtnAlpha.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent){
                jLabel.setText("Нажата кнопка \"Добавить\".");
            }
        });
        // ввести приемник действий от кнопки jbtnBeta
        jbtnBeta.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent)
{
                jLabel.setText("Нажата                    кнопка
\"Редактировать\".");
```

```
            }
        });
        // создать текстовую метку
        jLabel = new JLabel("Нажмите на кнопку");
        // ввести кнопки на панель содержимого
        jFrame.add(jbtnAlpha);
        jFrame.add(jbtnBeta);
        // ввести метку на панель содержимого
        jFrame.add(jLabel);
        // отобразить фрейм
        jFrame.setVisible(true);
    }
    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(() -> new EventDemo());
    }}
```

Результаты работы программы:



Обратите внимание на то, что в данном примере программы импортируются пакеты *java.awt* и *java.awt.event*. Пакет *java.awt* требуется потому, что в нем содержится класс *FlowLayout*, поддерживающий стандартный диспетчер поточной компоновки, который применяется для размещения компонентов во фрейме. А пакет *java.awt.event* требуется потому, что в нем определяются интерфейс *ActionListener* и класс *ActionEvent*.

Сначала в конструкторе класса *EventDemo* создается контейнер *jFrame* типа *JFrame*, а затем устанавливается диспетчер компоновки типа *FlowLayout* для панели содержимого контейнера *jFrame*. Напомню, что по умолчанию для размещения компонентов на панели содержимого *JFrame* применяется диспетчер компоновки типа *BorderLayout*, но для данного примера больше подходит диспетчер компоновки типа *FlowLayout*. Обратите внимание на то, что диспетчер компоновки типа *FlowLayout* назначается с помощью следующего оператора:

```
jFrame.setLayout(new FlowLayout());
```

После определения размеров и стандартной операции, выполняемой при закрытии окна, в конструкторе класса *EventDemo* создаются две экранные кнопки:

```
JButton jbtnAlpha = new JButton("Добавить");
JButton jbtnBeta = new JButton("Редактировать");
```

Первая кнопка будет содержать надпись "*Добавить*", а вторая – надпись "*Редактировать*". Экранные кнопки из библиотеки *Swing* являются экземплярами класса *JButton*. В классе *JButton* предоставляется несколько конструкторов. Здесь используется приведенный ниже конструктор, где параметр *text* обозначает символьную строку, которая будет отображаться в виде надписи на экранной кнопке.

`JButton(String text)`	Creates a button with text.

После щелчка на экранной кнопке наступает событие типа *ActionEvent*. Поэтому в классе *JButton* предоставляется метод *addActionListener()*, который служит для ввода приемника действия. В классе *JButton* предоставляется также метод *removeActionListener()* для удаления приемника событий. В интерфейсе *ActionListener* определяется единственный метод *actionPerformed()*.

```
package java.awt.event;

import java.util.EventListener;

public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent ae);
}
```

Метод *actionPerformed()* автоматически вызывается после щелчка на экранной кнопке. Иными словами, это обработчик, который вызывается, когда наступает событие нажатия экранной кнопки.

Далее вводится приемник событий от двух экранных кнопок, как показано ниже. В данном случае используются анонимные внутренние классы, чтобы предоставить обработчики событий от двух экранных кнопок. Всякий раз, когда нажимается экранная кнопка, символьная строка, отображаемая на месте метки *jLabel*, изменяется в зависимости от того, какая кнопка была нажата.

```
// ввести приемник действий от кнопки jbtnAlpha
jbtnAlpha.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent)
{
                jLabel.setText("Нажата кнопка \"Добавить\".");
            }
        });
// ввести приемник действий от кнопки jbtnBeta
jbtnBeta.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent)
{
                jLabel.setText("Нажата                    кнопка
\"Редактировать\".");
```

```
        }
    });
```

Начиная с версии *JDK* 8 для реализации обработчиков событий можно также воспользоваться лямбда-выражениями. Например, обработчик событий от кнопки *jbtnAlpha* можно было бы написать следующим образом:

```
jbtnAlpha.addActionListener(actionEvent -> jLabel.setText("Нажата кнопка \"Добавить\"."));
```

Этот код более краткий. В последующих примерах лямбда-выражения не применяются для лучшего запоминания используемых конструкций. Но при написании нового кода следует непременно пользоваться лямбда-выражениями.

Затем кнопки вводятся на панели содержимого следующим образом:

```
jFrame.add(jbtnAlpha);
jFrame.add(jbtnBeta);
```

Так же на панели содержимого вводится метка *jLabel*, и окно приложения делается видимым. Если после запуска данного *Swing*-приложения щелкнуть на любой из двух экранных кнопок, то на месте метки отобразится сообщение, извещающее, какая именно кнопка была нажата.

Не следует забывать, что все обработчики событий вроде метода *actionPerformed()* вызываются в потоке диспетчеризации событий. Следовательно, возврат из обработчика событий должен быть произведен быстро, чтобы не замедлить выполнение *Swing*-приложения. **Если же при наступлении некоторого события в приложении требуется выполнить операции, отнимающие много времени, то для этой цели следует организовать отдельный поток исполнения.**

Ниже перечислены классы компонентов *Swing*, рассматриваемые в этой теме. Все эти компоненты являются легковесными. Это означает, что все они происходят от класса *JComponent*.

JButton	*JCheckBox*	*JComboBox*	*JLabel*
JList	*JRadioButton*	*JScrollPane*	*JTabbedPane*
JTable	*JtextField*	*JToggleButton*	*JTree*

В этой теме рассматривается также класс *ButtonGroup*, инкапсулирующий взаимоисключающий ряд кнопок *Swing*, а также класс *ImageIcon*, инкапсулирующий графическое изображение. Каждый из этих классов определяется в библиотеке *Swing* и входит в пакет *javax.swing*.

КЛАССЫ JLABEL, IMAGEICON

Класс *JLabel* представляет метку – самый простой в употреблении компонент *Swing*.

С помощью компонента типа *JLabel* можно отображать текст и/или

значок. Этот компонент является пассивным в том отношении, что он не реагирует на данные, вводимые пользователем. В классе *JLabel* определяется несколько конструкторов.

Constructor and Description
`JLabel()` Creates a `JLabel` instance with no image and with an empty string for the title.
`JLabel(Icon image)` Creates a `JLabel` instance with the specified image.
`JLabel(Icon image, int horizontalAlignment)` Creates a `JLabel` instance with the specified image and horizontal alignment.
`JLabel(String text)` Creates a `JLabel` instance with the specified text.
`JLabel(String text, Icon icon, int horizontalAlignment)` Creates a `JLabel` instance with the specified text, image, and horizontal alignment.
`JLabel(String text, int horizontalAlignment)` Creates a `JLabel` instance with the specified text and horizontal alignment.

Здесь параметры *text* и *image* обозначают соответственно текст и значок, которые будут использоваться в качестве метки, а параметр *horizontalAlignment* – вид выравнивания текста и/или значка по горизонтали в пределах метки. Этот параметр должен принимать одно из значений следующих констант: *LEFT*, *RIGHT*, *CENTER*, *LEADING* или *TRAILING*. Наряду с рядом других констант, используемых в классах из библиотеки *Swing*, эти константы определяются в интерфейсе *SwingConstants*.

LEFT – выравнивание по левому краю, *CENTER* – по центру, *RIGHT* – по правому краю. Для языков, у которых написание текста идет слева направо *LEADING* — это левый край, а *TRAILING* — правый. Для языков, у которых написание происходит справа налево, *LEADING* — это правый край, а *TRAILING* — левый.

Обратите внимание на то, что значки определяются с помощью объектов типа *Icon*, относящегося к интерфейсу, определяемому в библиотеке *Swing*. Получить значок проще всего средствами класса *ImageIcon*, реализующего интерфейс *Icon* и инкапсулирующего изображение. Следовательно, объект типа *ImageIcon* можно передать в качестве параметра типа *Icon* конструктору класса *JLabel*.

javax.swing

Class ImageIcon

java.lang.Object
  javax.swing.ImageIcon

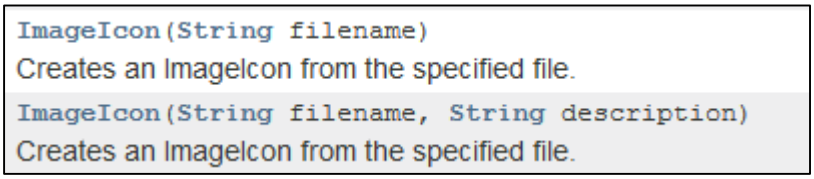
All Implemented Interfaces:

Serializable, Accessible, Icon

```
public class ImageIcon
extends Object
implements Icon, Serializable, Accessible
```

An implementation of the Icon interface that paints Icons from Images. Images that are created from a URL, filename or byte array are preloaded using MediaTracker to monitor the loaded state of the image.

Предоставить изображение можно несколькими способами, включая чтение изображения из файла или его загрузку по указанному *URL*. Ниже приведен конструктор класса *ImageIcon*, который получает изображение из файла, обозначаемого параметром *filename*.



Значок и текст, связанные с меткой, можно получить с помощью следующих методов:

```
Icon getIcon()
String getText()
```

Значок и текст, связанные с меткой, можно установить, вызывая приведенные ниже методы.

```
void setIcon (Icon icon)
void setText (String string)
```

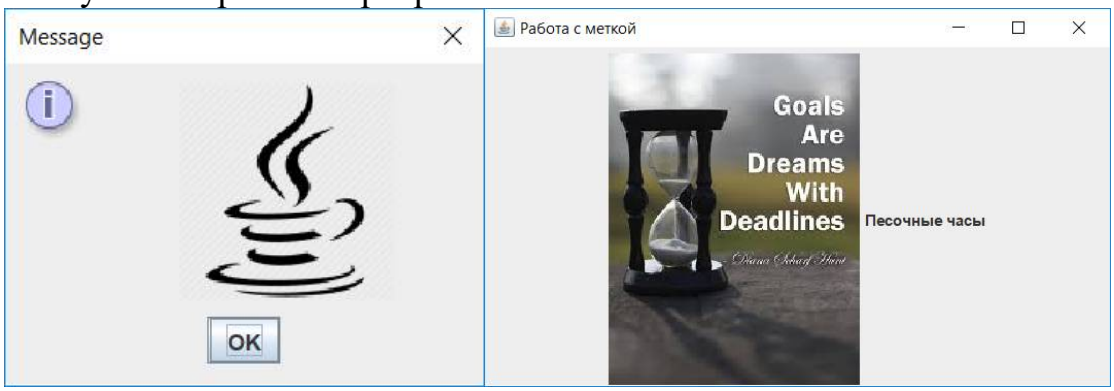
Здесь параметры *icon* и *string* обозначают указываемый значок и текст соответственно. Таким образом, с помощью метода *setText()* можно изменить текст метки во время выполнения программы.

В приведенном ниже примере демонстрируется порядок создания и отображения метки, состоящей из значка и символьной строки. Сначала создается объекта типа *ImageIcon* для отображения песочных часов из файла изображения *hourglass.png*. Этот объект указывается в качестве второго параметра при вызове конструктора класса *JLabel*. А первый и последний параметры этого конструктора представляют текст метки и его выравнивание. И, наконец, созданная метка вводится на панели содержимого.

```
//Пример №3. Применение компонентов типа JLabel и ImageIcon
import java.awt.*;
import java.net.MalformedURLException;
import java.net.URL;
import javax.swing.*;
class JLabelDemo {
    ImageIcon imageIcon;
    JLabel jLabel;
    JFrame jFrame;
    public JLabelDemo() throws MalformedURLException {
        URL url = new
URL("https://cdn0.iconfinder.com/data/icons/huge-basic-icons-
part-3/512/Java.png");
        javax.swing.ImageIcon imageIcon = new
javax.swing.ImageIcon(url); // load the image to a imageIcon
        Image image = imageIcon.getImage(); // transform it
        Image newimg = image.getScaledInstance(120, 120,
java.awt.Image.SCALE_SMOOTH); // scale it the smooth way
        imageIcon = new ImageIcon(newimg);  // transform it back
        javax.swing.JOptionPane.showMessageDialog(null,
imageIcon);
        jFrame = new JFrame("Работа с меткой");
```

```
        // определить диспетчер поточной компоновки типа FlowLayout
        jFrame.setLayout(new FlowLayout());
        // установить исходные размеры фрейма
        jFrame.setSize(500, 300);
        //завершить работу приложения, если пользователь закрывает окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // создать значок
       //              imageIcon              =              new ImageIcon("c:\\Work\\IBA_programms\\Pictures\\Hourglass.jpg");
        imageIcon = new ImageIcon("images/HourglassPhrase.jpg");
        // создать метку
        jLabel   =   new   JLabel("Песочные   часы",   imageIcon, JLabel.CENTER);
        // ввести метку на панель содержимого
        jFrame.add(jLabel);
        // отобразить фрейм
        jFrame.setVisible(true);
    }
    public static void main(String args[]) {
        try {
            // создать фрейм в потоке диспетчеризации событий
            SwingUtilities.invokeLater(() -> {
                try {
                    new JLabelDemo();
                } catch (MalformedURLException e) {
                    e.printStackTrace();
                }
            });
        } catch (Exception e) {
            System.out.println("Невозможно отобразить фрейм " + e);
        }}}
```

Результаты работы программы:



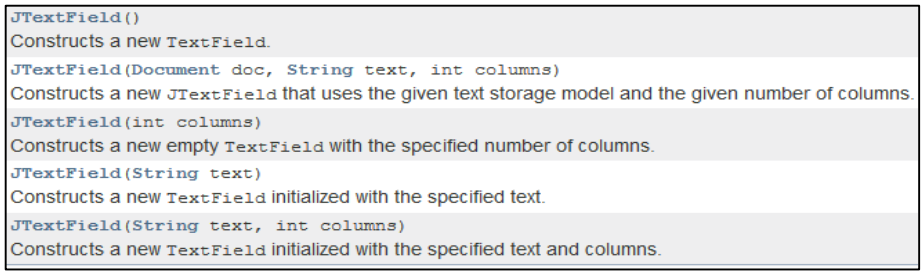
КЛАСС JTEXTFIELD

Класс *JTextField* представляет простейший текстовый компонент из библиотеки *Swing*.

Это наиболее употребительный текстовый компонент, который позволяет отредактировать одну строку текста. Класс *JTextField* является

производным от класса *JTextComponent*, наделяющим функциональными возможностями текстовые компоненты *Swing*.

В качестве своей модели класс *JTextField* использует интерфейс *Document*.


`JTextField()`
Constructs a new `TextField`.
`JTextField(Document doc, String text, int columns)`
Constructs a new `JTextField` that uses the given text storage model and the given number of columns.
`JTextField(int columns)`
Constructs a new empty `TextField` with the specified number of columns.
`JTextField(String text)`
Constructs a new `TextField` initialized with the specified text.
`JTextField(String text, int columns)`
Constructs a new `TextField` initialized with the specified text and columns.

Здесь параметр *text* обозначает первоначально предоставляемую символьную строку, а параметр *columns* – количество столбцов в текстовом поле. Если параметр *text* не задан, то текстовое поле оказывается исходно пустым. А если не задан параметр *columns*, то размеры текстового поля выбираются таким образом, чтобы оно могло уместиться в указанной символьной строке.

Компонент типа *JTextField* генерирует события в ответ на действия пользователя. Например, событие типа *ActionEvent* наступает при нажатии пользователем клавиши <*Enter*>, а событие типа *CaretEvent* – при каждом изменении позиции каретки (т.е. курсора).


javax.swing.event
Class CaretEvent
java.lang.Object
java.util.EventObject
javax.swing.event.CaretEvent
All Implemented Interfaces:
Serializable
`public abstract class CaretEvent extends EventObject`
CaretEvent is used to notify interested parties that the text caret has changed in the event source.

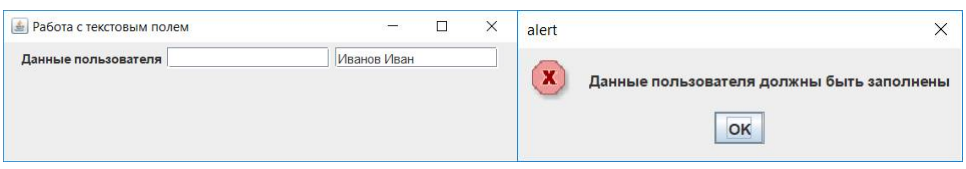
Событие типа *CaretEvent* определяется в пакете *javax.swing.event*. Возможны и другие события. Как правило, эти события не нужно обрабатывать в прикладной программе. Вместо этого достаточно получить символьную строку, находящуюся в данный момент в текстовом поле. Для ее получения следует вызвать метод *getText()*.

В приведенном ниже примере демонстрируется применение компонент типа *JTextField*. В данном примере сначала создается компонент типа *JTextField*, который затем вводится на панель содержимого. Когда пользователь нажимает клавишу <*Enter*>, наступает событие действия. В результате обработки этого события текст текущего *JTextField* отображается текст в другом поле типа *JTextField*.

```
//Пример №4. Применение компонента JTextField
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class JTextFieldDemo {
```

```
    JTextField jTextFieldFirst;
    JTextField jTextFieldSecond;
    JFrame jFrame;
    public JTextFieldDemo() {
        JLabel jLabel = new JLabel("Данные пользователя");
        jFrame = new JFrame("Работа с текстовым полем");
        // применить поточную компоновку
        jFrame.setLayout(new FlowLayout());
        // ввести текстовое поле на панель содержимого
        jTextFieldFirst = new JTextField(15);
        jTextFieldFirst.setActionCommand("jTextFieldFirst");
        jTextFieldSecond = new JTextField(15);
        jTextFieldSecond.setActionCommand("jTextFieldSecond");
        // установить исходные размеры фрейма
        jFrame.setSize(500, 150);
        ActionListener actionListener = new ActionListener(){
            public void actionPerformed(ActionEvent ae) {//
отобразить текст, когда пользователь нажимает клавишу <Enter>
                if (jTextFieldFirst.getText().equals("") &&
jTextFieldSecond.getText().equals("")) {
                    JOptionPane.showMessageDialog(null, "Данные
пользователя должны быть заполнены", "alert",
JOptionPane.ERROR_MESSAGE);
                }

jTextFieldSecond.setText(jTextFieldFirst.getText());
                jTextFieldFirst.setText("");
            }
        };
        //завершить работу приложения, если пользователь закрывает
окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jTextFieldFirst.addActionListener(actionListener);
        jTextFieldSecond.addActionListener(actionListener);
        jFrame.add(jLabel);
        jFrame.add(jTextFieldFirst);
        jFrame.add(jTextFieldSecond);
        // отобразить фрейм
        jFrame.setVisible(true);
    }
    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new JTextFieldDemo();
            }
        });
    }}
```



КНОПКИ В БИБЛИОТЕКЕ SWING

В библиотеке *Swing* определены четыре класса кнопок: *JButton, JToggleButton, JCheckBox, JRadioButton*. Все они являются производными от класса *AbstractButton*, расширяющего класс *JComponent*.

Таким образом, у кнопок имеются общие черты. Класс *AbstractButton* содержит методы, позволяющие управлять поведением кнопок. С их помощью можно, например, определить различные значки, которые будут отображаться на месте кнопки, когда она отключена, нажата или выбрана. Другой значок можно использовать для динамической подстановки, чтобы он отображался при наведении указателя мыши на кнопку. Ниже приведены общие формы методов, с помощью которых можно задавать эти значки.

void	**setDisabledIcon**(Icon disabledIcon) Sets the disabled icon for the button.
void	**setPressedIcon**(Icon pressedIcon) Sets the pressed icon for the button.
void	**setSelectedIcon**(Icon selectedIcon) Sets the selected icon for the button.
void	**setRolloverIcon**(Icon rolloverIcon) Sets the rollover icon for the button.

Параметры *disabledIcon, pressedIcon, selectedIcon* и *rolloverIcon* определяют значки, используемые для обозначения различных состояний. Текст, связанный с кнопкой, можно прочитать и записать с помощью приведенных ниже методов, где параметр *text* обозначает текст надписи на кнопке.

String	**getText**() Returns the button's text.
void	**setText**(String text) Sets the button's text.

Модель, применяемая во всех кнопках, определяется в интерфейсе *ButtonModel*. Кнопка генерирует событие действия, когда ее нажимает пользователь. Возможны и другие события. В последующих разделах будут подробнее рассмотрены классы отдельных кнопок.

КЛАСС JBUTTON

В классе *JButton* определяются функциональные возможности экранной кнопки.

javax.swing

Class JButton

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        javax.swing.AbstractButton
          javax.swing.JButton

All Implemented Interfaces:

ImageObserver, ItemSelectable, MenuContainer, Serializable, Accessible, SwingConstants

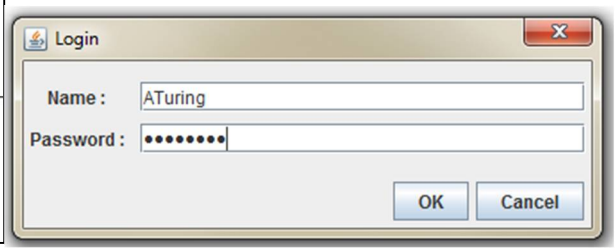
Direct Known Subclasses:

BasicArrowButton, MetalComboBoxButton

```
public class JButton
extends AbstractButton
implements Accessible
```

An implementation of a "push" button.

Buttons can be configured, and to some degree controlled, by Actions.



Компонент типа *JButton* позволяет связать с экранной кнопкой значок, символьную строку или же и то, и другое. Ниже приведены три конструктора класса *JButton*, где параметры *icon* и *text* обозначат соответственно значок и строку, используемые для кнопки.

JButton()
Creates a button with no set text or icon.

JButton(Action a)
Creates a button where properties are taken from the Action supplied.

JButton(Icon icon)
Creates a button with an icon.

JButton(String text)
Creates a button with text.

JButton(String text, Icon icon)
Creates a button with initial text and an icon.

Когда пользователь щелкает на экранной кнопке, наступает событие типа *ActionEvent*. Используя объект типа *ActionEvent*, передаваемый методу *actionPerformed()* зарегистрированного приемник действий типа *ActionListener*, можно получить символьную строку с **командой действия**, связанной с данной кнопкой. По умолчанию эта символьная строка отображается в пределах кнопки. Но команду действия можно также задать, вызвав метод *setActionCommand()* для кнопки. А получить команду действия можно, вызвав метод *getActionCommand()* для объекта события. Этот метод объявляется следующим образом:

String	**getActionCommand()** Returns the action command for this button.

Команда действия обозначает кнопку. Так, если в одном приложении используются две кнопки или больше, команда действия позволяет определить, какая именно кнопка была нажата.

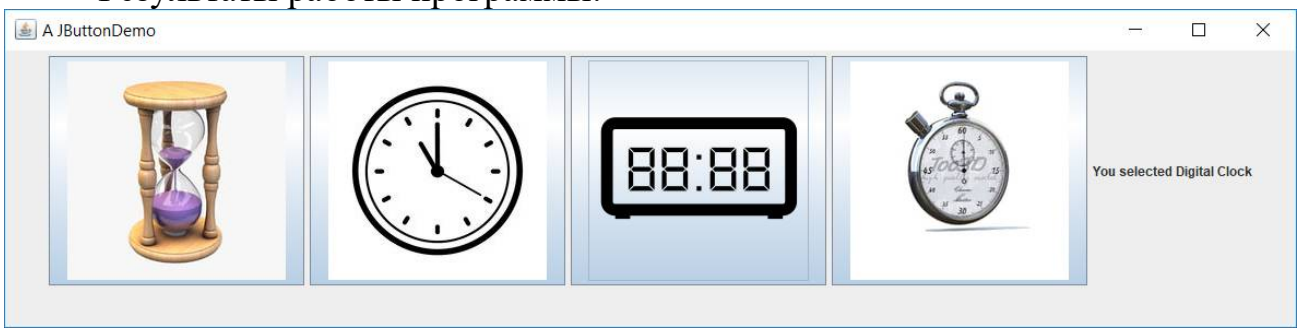
В приведенном ниже примере демонстрируется кнопка в виде значка. В данном примере отображаются четыре экранные кнопки и одна метка. Каждая кнопка отображает значок, представляющий разновидность часов. Когда пользователь щелкает на кнопке, в метке появляется название часов.

```
//Пример №5. Применение компонента типа JButton в виде значка
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class JButtonDemo implements ActionListener {
    JLabel jLabel;
    JFrame jFrame;
    public JButtonDemo() {
        jFrame = new JFrame("Работа с кнопками");
        // изменить поточную компоновку
        jFrame.setLayout(new FlowLayout());
        // ввести кнопки на панели содержимого
        ImageIcon hourglassImageIcon = new
ImageIcon("c:\\Work\\IBA_programms\\Pictures\\Hourglass.jpg");
        JButton jButton = new JButton(hourglassImageIcon);
        jButton.setActionCommand("Hourglass"); // Песочные часы
        jButton.addActionListener(this);
        jFrame.add(jButton);
        ImageIcon analogImageIcon = new
ImageIcon("c:\\Work\\IBA_programms\\Pictures\\AnalogClock.jpg");
        jButton = new JButton(analogImageIcon);
        jButton.setActionCommand("Analog Clock");//Аналоговые
часы
        jButton.addActionListener(this);
        jFrame.add(jButton);
        ImageIcon digitalImageIcon = new
ImageIcon("c:\\Work\\IBA_programms\\Pictures\\DigitalClock.png")
;
        jButton = new JButton(digitalImageIcon);
        jButton.setActionCommand("Digital Clock");//Цифровые часы
        jButton.addActionListener(this);
        jFrame.add(jButton);
        ImageIcon stopwatchImageIcon = new
ImageIcon("c:\\Work\\IBA_programms\\Pictures\\Stopwatch.jpg");
        jButton = new JButton(stopwatchImageIcon);
        jButton.setActionCommand("Stopwatch");//Секундомер
        jButton.addActionListener(this);
        jFrame.add(jButton);
        // создать метку и ввести ее на панели содержимого
        jLabel = new JLabel("Выберите часы");//выбрать часы
        jFrame.add(jLabel);
        // установить исходные размеры фрейма
        jFrame.setSize(1200, 300);
        //завершить работу приложения, если пользователь закрывает
окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // отобразить фрейм
        jFrame.setVisible(true);
    }
    // обработать события от кнопок
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        // выбраны указанные часы
        jLabel.setText("Вы выбрали: " +
```

```
actionEvent.getActionCommand());
    }
    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new JButtonDemo();
            }
        });
    }}
```

Результаты работы программы:



ФЛАЖКИ (JCHECKBOX)

Класс *JCheckBox* определяет функции флажка. Его суперклассом служит класс *JToggleButton*, поддерживающий кнопки с двумя состояниями. В классе *JCheckBox* определяется ряд конструкторов.

`JCheckBox()` Creates an initially unselected check box button with no text, no icon.
`JCheckBox(Action a)` Creates a check box where properties are taken from the Action supplied.
`JCheckBox(Icon icon)` Creates an initially unselected check box with an icon.
`JCheckBox(Icon icon, boolean selected)` Creates a check box with an icon and specifies whether or not it is initially selected.
`JCheckBox(String text)` Creates an initially unselected check box with text.
`JCheckBox(String text, boolean selected)` Creates a check box with text and specifies whether or not it is initially selected.
`JCheckBox(String text, Icon icon)` Creates an initially unselected check box with the specified text and icon.
`JCheckBox(String text, Icon icon, boolean selected)` Creates a check box with text and icon, and specifies whether or not it is initially selected.

Конструктор создает флажок с текстом метки, определяемым в качестве параметра *text*. Остальные конструкторы позволяют определить исходное состояние выбора флажка и указать значок.

Если пользователь устанавливает или сбрасывает флажок, генерируется событие типа *ItemEvent*. Чтобы получить ссылку на компонент типа *JCheckBox*, сгенерировавший событие, следует вызвать метод *getItem()* для объекта типа *ItemEvent*, который передается в качестве события от элемента методу *itemStateChanged(),* определяемому в интерфейсе *ItemListener*. Определить состояние флажка можно вызвав метод *isSelected()* для экземпляра класса *JCheckBox*.

В приведенном ниже примере демонстрируется применение флажков.

```
//Пример №7. Применение компонента типа JCheckBox
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
 class SwingJCheckBoxDemo extends JFrame {
    // create three check boxes that allow selecting three numbers
    private JCheckBox checkboxOne = new JCheckBox("$100");
    private JCheckBox checkboxTwo = new JCheckBox("$200");
    private JCheckBox checkboxThree = new JCheckBox("$300");
    // a label and a text field to display sum
    private JLabel labelSum = new JLabel("Sum: ");
    private JTextField textFieldSum = new JTextField(10);
    private int sum = 0;// sum of 3 numbers
    public SwingJCheckBoxDemo() {
        super("Работа с флажками");
        setLayout(new FlowLayout());
        // add the check boxes to this frame
        add(checkboxOne);
        add(checkboxTwo);
        add(checkboxThree);
        add(labelSum);
        textFieldSum.setEditable(false);
        add(textFieldSum);
        // add action listener for the check boxes
        ActionListener actionListener = new ActionHandler();
        checkboxOne.addActionListener(actionListener);
        checkboxTwo.addActionListener(actionListener);
        checkboxThree.addActionListener(actionListener);
        pack();
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    class ActionHandler implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent event) {
            JCheckBox checkbox = (JCheckBox) event.getSource();
            if (checkbox.isSelected()) {
                if (checkbox == checkboxOne) {
                    sum += 100;
                } else if (checkbox == checkboxTwo) {
                    sum += 200;
                } else if (checkbox == checkboxThree) {
                    sum += 300;
                }
            } else {
                if (checkbox == checkboxOne) {
                    sum -= 100;
                } else if (checkbox == checkboxTwo) {
```

```
                    sum -= 200;
                } else if (checkbox == checkboxThree) {
                    sum -= 300;
                }
            }
            textFieldSum.setText(String.valueOf(sum));
        }
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new SwingJCheckBoxDemo().setVisible(true);
            }
        });
    }}
```

Результаты работы программы:

КНОПКИ-ПЕРЕКЛЮЧАТЕЛИ

Кнопки-переключатели образуют группу взаимоисключающих кнопок, из которых можно выбрать только одну. Они поддерживаются в классе *JRadioButton*, расширяющем класс *JToggleButton*.

В классе *JRadioButton* предоставляется несколько конструкторов.

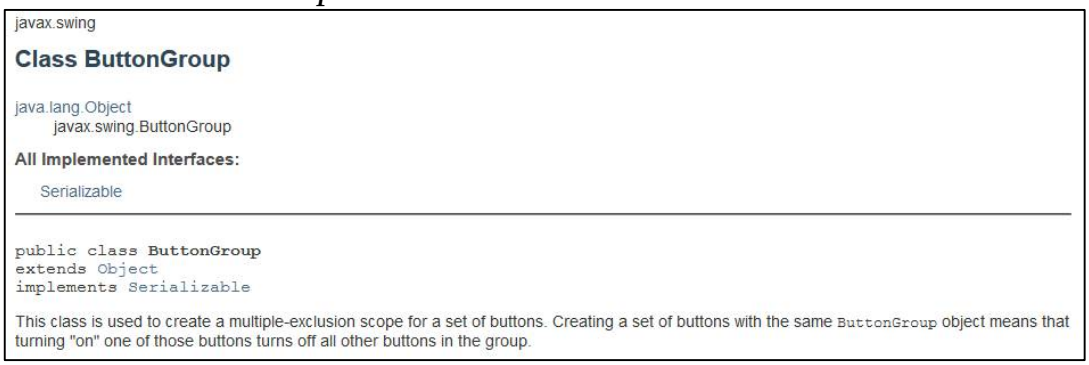
`JRadioButton()` Creates an initially unselected radio button with no set text.
`JRadioButton(Action a)` Creates a radiobutton where properties are taken from the Action supplied.
`JRadioButton(Icon icon)` Creates an initially unselected radio button with the specified image but no text.
`JRadioButton(Icon icon, boolean selected)` Creates a radio button with the specified image and selection state, but no text.
`JRadioButton(String text)` Creates an unselected radio button with the specified text.
`JRadioButton(String text, boolean selected)` Creates a radio button with the specified text and selection state.
`JRadioButton(String text, Icon icon)` Creates a radio button that has the specified text and image, and that is initially unselected.
`JRadioButton(String text, Icon icon, boolean selected)` Creates a radio button that has the specified text, image, and selection state.

Здесь параметр *text* обозначает метку кнопки-переключателя. Остальные конструкторы позволяют определить исходное состояние кнопки-переключателя и указать для нее значок.

Кнопки-переключатели следует объединить в группу, где можно выбрать только одну из них.

Так, если пользователь выбирает какую-нибудь кнопку-переключатель из группы, то кнопка-переключатель, выбранная ранее в этой группе,

автоматически выключается. Для создания группы кнопок-переключателей служит класс *ButtonGroup*.



javax.swing

Class ButtonGroup

java.lang.Object
  javax.swing.ButtonGroup

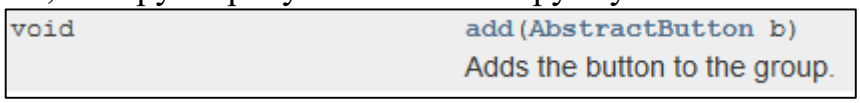
All Implemented Interfaces:

Serializable

```
public class ButtonGroup
extends Object
implements Serializable
```

This class is used to create a multiple-exclusion scope for a set of buttons. Creating a set of buttons with the same `ButtonGroup` object means that turning "on" one of those buttons turns off all other buttons in the group.

С этой целью вызывается его конструктор по умолчанию. После этого в группу можно ввести отдельные кнопки-переключатели с помощью приведенного ниже метода, где параметр *b* обозначает ссылку на кнопку-переключатель, которую требуется ввести в группу.



`void`	`add(AbstractButton b)`
	Adds the button to the group.

Компонент типа *JRadioButton* генерирует события действия, события от элементов и события изменения всякий раз, когда выбирается другая кнопка-переключатель в группе. Зачастую обрабатывается событие действия, а это, как правило, означает необходимость реализовать интерфейс *ActionListener*, в котором определяется единственный метод *actionPerformed()*. В этом методе можно несколькими способами выяснить, какая именно кнопка-переключатель была выбрана.

Во-первых, можно проверить ее команду действия, вызвав метод *getActionCommand()*. По умолчанию команда действия аналогична метке кнопки, но, вызвав метод *setActionCommand()* для кнопки-переключателя, можно задать какую-нибудь другую команду действия.

Во-вторых, можно вызвать метод *getSource()* для объекта типа *ActionEvent* и проверить ссылку по отношению к кнопкам-переключателям.

И наконец, для каждой кнопки можно вызвать свой обработчик событий действия, реализуемый в виде анонимного класса или лямбда-выражения. Не следует забывать, что всякий раз, когда наступает событие действия, оно означает, что выбранная кнопка-переключатель была изменена и что была выбрана одна и только одна кнопка-переключатель.

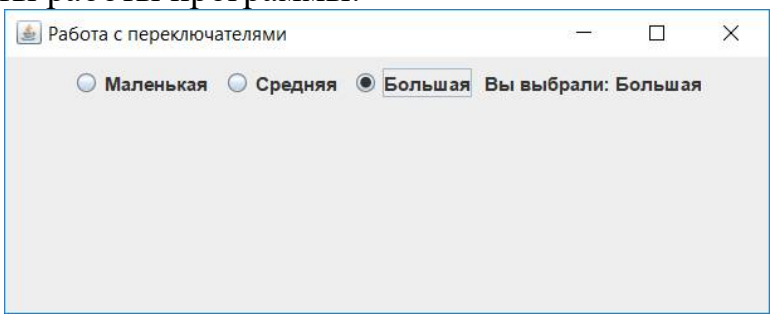
В приведенном ниже примере демонстрируется применение кнопок-переключателей. В этом примере создаются и объединяются в группу три кнопки-переключателя. Это требуется для того, чтобы они действовали, взаимно исключая друг друга. При выборе кнопки-переключателя наступает событие действия, которое обрабатывается медом *actionPerformed()*. В этом обработчике событий метод *getActionCommand()* получает текст, связанный с кнопкой-переключателем, чтобы отобразить его текст на месте метки.

```
//Пример №8. Применение компонента JRadioButton
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
class JRadioButtonDemo implements ActionListener {
    JLabel jLabel;
    JFrame jFrame;
    public JRadioButtonDemo() {
        jFrame = new JFrame("Работа с переключателями");
        // изменить поточную компоновку
        jFrame.setLayout(new FlowLayout());
        // установить исходные размеры фрейма
        jFrame.setSize(500, 200);
        //завершить работу приложения, если пользователь закрывает
окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // отобразить фрейм
        jFrame.setVisible(true);
        // изменить поточную компоновку
        jFrame.setLayout(new FlowLayout());
        //создать кнопки-переключатели и ввести их на панель
содержимого
        JRadioButton jRadioButtonSmall = new
JRadioButton("Маленькая");
        jRadioButtonSmall.addActionListener(this);
        jFrame.add(jRadioButtonSmall);
        JRadioButton jRadioButtonMedium = new
JRadioButton("Средняя",true);
        jRadioButtonMedium.setActionCommand("Средняя");
        jRadioButtonMedium.addActionListener(this);
        jFrame.add(jRadioButtonMedium);
        JRadioButton jRadioButtonBig = new
JRadioButton("Большая");
        jRadioButtonBig.addActionListener(this);
        jFrame.add(jRadioButtonBig);
        // определить группу кнопок
        ButtonGroup buttonGroup = new ButtonGroup();
        buttonGroup.add(jRadioButtonSmall);
        buttonGroup.add(jRadioButtonMedium);
        buttonGroup.add(jRadioButtonBig);
        // создать метку и ввести ее на панель содержимого
        jLabel = new JLabel("Выберите размер пиццы");
        jFrame.add(jLabel);
        jLabel.setText("Вы выбрали: " +
buttonGroup.getSelection().getActionCommand());
    }
    // обработать событие выбора кнопки-переключателя
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        jLabel.setText("Вы выбрали: " +
actionEvent.getActionCommand());
    }
    public static void main(String args[]) {
            // создать фрейм в потоке диспетчеризации событий
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
```

```
                new JRadioButtonDemo();
            }
        });
    }}
```

Результаты работы программы:



КЛАСС JLIST

Базовым для составления списков в *Swing* служит класс *JList*. В этом классе поддерживается выбор одного или нескольких элементов из списка. Зачастую список состоит из символьных строк, но ничто не мешает составить список из любых объектов, которые только можно отобразить.

Раньше элементы списка типа *JList* были представлены ссылками на класс *Object*. Но в версии *JDK* 7 класс *JList* стал обобщенным и теперь объявляется приведенным ниже образом, где параметр *E* обозначает тип элементов в списке.

```
class JList<E>
```

В классе *JList* предоставляется несколько конструкторов.

`JList()` Constructs a `JList` with an empty, read-only, model.
`JList(E[] listData)` Constructs a `JList` that displays the elements in the specified array.
`JList(ListModel<E> dataModel)` Constructs a `JList` that displays elements from the specified, `non-null`, model.
`JList(Vector<? extends E> listData)` Constructs a `JList` that displays the elements in the specified `Vector`.

Конструктор создает список типа *JList*, содержащий элементы в массиве, обозначаемом в качестве параметра *listData* или *dataModel*.

Класс *JList* основывается на двух моделях. Первая модель определяется в интерфейсе *ListModel* и устанавливает порядок доступа к данным в списке. Вторая модель определяется в интерфейсе *ListSelectionModel*, где объявляются методы, позволяющие выявить выбранный из списка элемент (или элементы).

Несмотря на то, что компонент типа *JList* вполне способен действовать самостоятельно, он обычно размещается на панели типа *JScrollPane*. Благодаря этом длинные списки становятся автоматически прокручиваемыми. Этим упрощается не только построение ГПИ, но и изменение количества записей в списке, не требуя изменять размеры компонента типа *JList*.

Компонент типа *JList* генерирует событие типа *ListSelectionEvent*, когда пользователь выбирает элемент или изменяет выбор элемента в списке.

Событие *ListSelectionEvent* наступает и в том случае, если пользователь отменяет выбор элемента. Оно обрабатывается приемником событий, реализующим интерфейс *ListSelectionListener*, в котором определяется единственный метод *valueChanged()*:

void	valueChanged(ListSelectionEvent e) Called whenever the value of the selection changes.

где *e* обозначает ссылку на событие. И хотя в классе *ListSelectionEvent* предоставляются свои методы для выяснения событий, наступающих при выборе элементов из списка, как правило, для этой цели достаточно обратиться непосредственно к объекту типа *JList*. Класс *ListSelectionEvent* и интерфейс *ListSelectionListener* определены в пакете *javax.swing.event.*

По умолчанию компонент типа *JList* позволяет выбирать несколько элементов из списка, но это поведение можно изменить, вызвав метод *setSelectionModel(),* определяемый в классе *JList*. Ниже приведена его общая форма.

void	setSelectionMode(int selectionMode) Sets the selection mode for the list.

Здесь параметр *selectionMode* обозначает заданный режим выбора. Этот параметр должен принимать значение одной из следующих констант, определенных в интерфейсе *ListSelectionModel*:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

По умолчанию выбирается значение последней константы, позволяющее выбирать несколько интервалов элементов из списка. Если же задан режим выбора в одном интервале (константа *SINGLE_INTERVAL_SELECTION*), то выбрать можно только один ряд элементов из списка. А если задан режим выбора одного элемента (константа *SINGLE_SELECTION*), то выбрать можно только один элемент из списка. Разумеется, один элемент можно выбрать и в двух других режимах, но эти режимы позволяют также выбирать несколько элементов из списка.

Вызвав метод *getSelectedIndex(),* можно получить индекс первого

выбранного элемент, который оказывается также индексом единственного выбранного элемента в режиме *SINGLE_SELECTION*.

int	getSelectedIndex() Returns the smallest selected cell index; *the selection* when only a single item is selected in the list.

Индексация элементов списка начинается с нуля. Так, если выбран первый элемент, метод *getSelectedIndex()* возвращает нулевое значение. А если не выбрано ни одного элемента, то возвращается значение -1.

Вместо того, чтобы получать индекс выбранного элемент, можно получить значение, связанное с выбранным элементом, вызвав метод *getSelectedValue()*.

E	getSelectedValue() Returns the value for the smallest selected cell index; *the selected value* when only a single item is selected in the list.

Этот метод возвращает ссылку на первое выбранное значение. Если не выбрано ни одного значения, то возвращается пустое значение *null*.

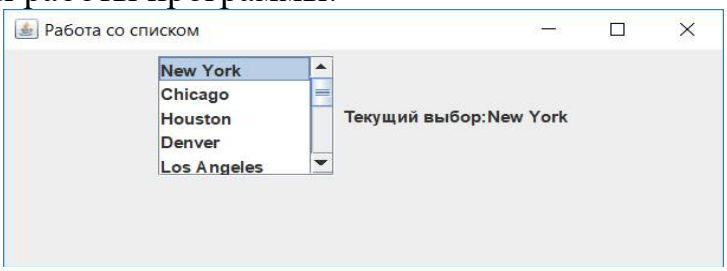
В приведенном ниже примере демонстрируется применение простого компонента типа *JList*, содержащего список городов. Всякий раз, когда из этого списка выбирается город, наступает событие типа *ListSelectionEvent*, обрабатываемое методом *valueChanged()*, определяемым в интерфейсе *ListSelectionListener*. Этот метод получает индекс выбранного элемента и отображает имя выбранного города на месте метки.

```
//Пример №9. Применение компонента JList
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
class JListDemo extends JFrame {
    JList<String> stringJList;
    JLabel jLabel;
    JScrollPane jScrollPane;
    JFrame jFrame;
    // создать массив из названий городов
    String cities[] = {"New York", "Chicago ", "Houston",
"Denver", "Los Angeles", "Seattle", "London", "Paris", "New
Delhi", "Hong Kong", "Tokyo", "Sydney"};
    public JListDemo() {
        jFrame = new JFrame("Работа со списком");
        // изменить поточную компоновку
        jFrame.setLayout(new FlowLayout());
        // установить исходные размеры фрейма
        jFrame.setSize(500, 200);
        //завершить работу приложения,если пользователь закрывает
окно
        jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // отобразить фрейм
        jFrame.setVisible(true);
        // создать список на основе компонента типа JList
        stringJList = new JList<String>(cities);
        // задать режим выбора единственного элемента из списка

stringJList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION
```

```
);
        // ввести список на панели с полосам прокрутки
        jScrollPane = new JScrollPane(stringJList);
        //  задать  предпочтительные  размеры  панели  с  полосам
прокрутки
        jScrollPane.setPreferredSize(new Dimension(120, 90));
        // создать метку для отображения выбранного города
        jLabel = new JLabel("Выберите город");
        // ввести приемник событий выбора из списка
        stringJList.addListSelectionListener(new
ListSelectionListener() {
            @Override
            public void valueChanged(ListSelectionEvent le) {
                // получить индекс измененного элемента
                int idx = stringJList.getSelectedIndex();
                //отобразить сделанный выбор, если элемент выбран
из списка
                if (idx != -1) {
                    jLabel.setText("Текущий       выбор:"       +
cities[idx]);
                } // Текущий выбор: указанный город
                else // выбрать город из списка
                    jLabel.setText("Выберите город");
            }
        });
        // ввести список и метку на панель содержимого
        jFrame.add(jScrollPane);
        jFrame.add(jLabel);
    }
    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new JListDemo();
            }
        });
    }}
```

Результаты работы программы:



КЛАСС JCOMBOBOX

С помощью класса *JComboBox* из библиотеки *Swing* определяется компонент, называемый **комбинированным списком** и сочетающий текстовое поле с раскрывающимся списком. Как правило, комбинированный список отображает одну запись, но он может отображать и раскрывающийся

список, позволяющий выбирать другие элементы. Имеется также возможность создать комбинированный список, позволяющий вводить выбираемый элемент в текстовом поле.

Раньше элементы комбинированного списка на основе компонент типа *JComboBox* были представлены ссылками на класс *Object*.

Но в версии *JDK* 7 класс *JComboBox* был сделан обобщенным и теперь объявляется приведенным ниже образом, где параметр *E* обозначает тип элементов комбинированного списка.

```
class JComboBox<E>
```

Ниже приведены конструкторы класса *JComboBox*.

`JComboBox()` Creates a `JComboBox` with a default data model.
`JComboBox(ComboBoxModel<E> aModel)` Creates a `JComboBox` that takes its items from an existing `ComboBoxModel`.
`JComboBox(E[] items)` Creates a `JComboBox` that contains the elements in the specified array.
`JComboBox(Vector<E> items)` Creates a `JComboBox` that contains the elements in the specified Vector.

Здесь параметры *items* и *aModel* обозначает массив, инициализирующий комбинированный список.

В классе *JComboBox* применяется модель, определяемая в интерфейсе *ComboBoxModel*. Для создания изменяемых комбинированных списков (т.е. таких списков, элементы которых могут изменяться) применяется модель, определяемая в интерфейсе *MutableComboBoxModel*.

Кроме передачи массива элементов, которые должны быть отображены в раскрывающемся списке, элементы можно динамически вводить в список с помощью метода *addItem()*. Ниже приведена общая форма этого метода.

`void`	`addItem(E item)` Adds an item to the item list.

Здесь параметр *item* обозначает объект, вводимый в комбинированный список. **Метод *addItem()* должен применяться только к изменяемым комбинированным спискам.**

Компонент типа *JComboBox* генерирует событие действия, когда пользователь выбирает элемент из комбинированного списка. Этот компонент генерирует также событие от элемента, когда изменяется состояние выбора, что происходит при выборе или отмене выбора элемента. **Таким образом, при изменении выбора происходят два события: одно – от того элемента, выбор которого был отменен, другое – от выбранного элемента.** Нередко оказывается достаточно принимать события действия, но обрабатывать можно оба типа событий.

Вызвав метод *getSelectedItem()*, можно получить элемент, выбранный из комбинированного списка. Ниже приведена общая форма этого метода.

`Object`	`getSelectedItem()` Returns the current selected item.

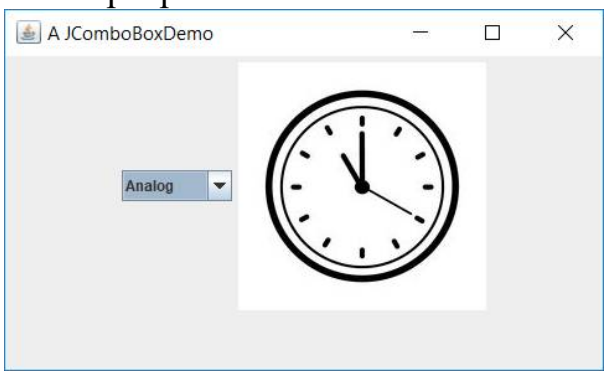
Значение, возвращаемое этим методом, следует привести к типу

объекта, хранящегося в списке. В приведенном ниже примере демонстрируется применение комбинированного списка. Этот список содержит элементы "*Hourglass*" (Песочные часы), "*Analog*" (Аналоговые часы), "*Digital*" (Цифровые часы) и "*Stopwatch*" (Секундомер). Если пользователь выберет часы, метка обновится, отображая значок данной разновидности часов. Обратите внимание, как мало кода требуется, чтобы воспользоваться этим сложным компонентом.

```
//Пример №10. Применение компонента типа JComboBox
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JComboBoxDemo {
    JLabel jlab;
    ImageIcon hourglass, analog, digital, stopwatch;
    JComboBox<String> jcb;
    String tirepieces[] = {"Hourglass", "Analog", "Digital",
"Stopwatch"};
    JFrame jfrm;
    JComboBoxDemo() {
        jfrm = new JFrame("A JComboBoxDemo");
        // изменить поточную компоновку
        jfrm.setLayout(new FlowLayout());
        // установить исходные размеры фрейма
        jfrm.setSize(500, 300);
     //завершить работу приложения, если пользователь закрывает
окно
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // отобразить фрейм
        jfrm.setVisible(true);
        //получить экземпляр объекта комбинированного списка и
ввести его на панели содержимого
        jcb = new JComboBox<String>(tirepieces);
        jfrm.add(jcb);
        // обработать события выбора элементов из списка
        jcb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                String s = (String) jcb.getSelectedItem();
                jlab.setIcon(new ImageIcon(s + ".png"));
            }}
        );
        // создать метку и ввести ее на панель содержимого
        jlab = new JLabel(new ImageIcon("hourglass.png"));
        jfrm.add(jlab);
    }
    public static void main(String args[]) {
            // создать фрейм в потоке диспетчеризации событий
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new JComboBoxDemo();
                }
            });
```

```
}}
```

Результаты работы программы:



КЛАСС JTABLE

Класс *JTable* представляет компонент, отображающий данные в виде строк и столбцов таблицы. Чтобы изменить размеры столбцов, достаточно перетащить их границы мышью. Кроме того, весь столбец можно перетащить в другое место. В зависимости от конфигурации можно выбрать символьную строку, столбец или ячейку в таблице, а также изменить данные в ячейке. Компонент типа *JTable* довольно сложный, он предоставляет немало вариантов выбора и средств, рассмотреть которые полностью просто невозможно в одном разделе. Это едва ли не самый сложный компонент *Swing*. Но в своей стандартной конфигурации компонент типа *JTable* предоставляет простые в употреблении средства, которых должно быть достаточно для представления данных в табличном виде.

Как и с компонентом типа *JTree*, с компонентом типа *JTable* связаны многие классы и интерфейсы. Все они входят в состав пакет *javax.swing.table*.

В своей основе компонент типа *JTable* очень прост. Он состоит из одного или нескольких столбцов с данными. Вверху каждого столбца находится заголовок. Кроме описания данных в столбце, заголовок предоставляет механизм, с помощью которого пользователь может изменять размеры столбца или его местоположение в таблице. Компонент типа *JTable* не предоставляет никаких возможностей для прокрутки, поэтому он, как правило, размещается в контейнере типа *JScrollPane*.

В классе *JTable* предоставляется несколько конструкторов. Ниже приведен один из них, где параметр **data** обозначает двумерный массив данных, представляемых в табличном виде, а параметр **cloumnsName** – одномерный массив, содержащий заголовки столбцов.

```
JTable (Object data [] [], Object cloumnsName[])
```

Компонент типа *JTable* основывается на трех моделях. Первой из них является модель таблицы, определяемая в интерфейсе *TableModel*. Эта модель определяет все, что связано с отображением данных в двухмерном формате. А второй является модель столбца таблицы, определяемая в интерфейсе *TableColumModel*. Компонент типа *JTable* обозначает столбцы, а модель типа *TableColumnModel* – характеристики столбцов. Обе эти модели входят в состав

пакета *javax.swing.table*. И наконец, третья модель обозначает порядок выбора элементов и определяется в интерфейсе *ListSelectionModel* (она упоминалась выше при рассмотрении класса *JList*).

Компонент типа *JTable* может генерировать целый ряд разных событий. К самым основным относятся события типа *ListSelectionEvent* и *TableModelEvent*.

Событие *ListSelectionEvent* наступает, когда пользователь выбирает что-нибудь в таблице. По умолчанию компонент типа *JTable* позволяет выбрать полностью одну строк таблицы или больше, но это поведение можно изменить, чтобы пользователь мог выбрать один или несколько столбцов или одну или несколько отдельных ячеек таблицы. Событие типа *TableModelEvent* наступает, когда данные каким-нибудь образом изменяются в таблице. Обработка таких событий требует больших затрат труда, чем обработка событий в описанных ранее компонентах. Но если компонент типа *JTable* требуется лишь для отображения данных в табличном виде, как в приведенном ниже примере, то никаких событий обрабатывать не придется.

Чтобы создать простой компонент типа *JTable* для отображения данных в табличном виде, достаточно выполнить следующие действия.

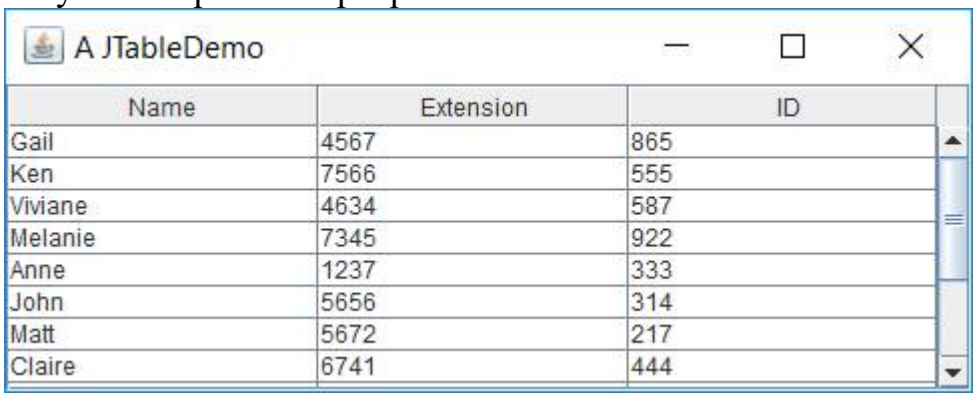
1. Создать экземпляр класса *JTable*.
2. Создать экземпляр класса *JScrollPane*, определив таблицу в качестве прокручиваемого объекта.
3. Ввести таблицу на панели с полосами прокрутки.
4. Ввести панель с полосами прокрутки на панели содержимого.

В приведенном ниже примере демонстрируется создание и применение простой таблицы. В данном примере сначала создается одномерный массив символьных строк *colHeads* для заголовков столбцов, а также двухмерный массив символьных строк *data* для данных в ячейках таблицы. Каждый элемент массива *data* является массивом из трех символьных строк. Эти массивы передаются конструктору класса *JTable*. Таблица вводится на панели с полосами прокрутки, а та – на панели содержимого. В таблице отображаются данные из массива *data*. Стандартна конфигурация таблицы позволяет также редактировать содержимое ячеек. Все вносимые в них изменения отражаются на содержимом базового массива *data*.

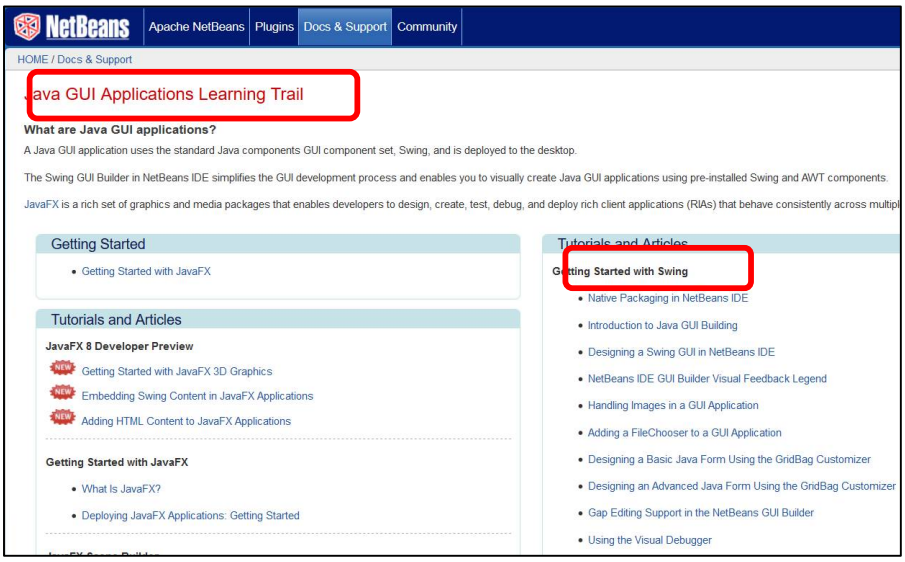
```
//Пример №11. Применение компонента JTable
import javax.swing.*;
public class JTableDemo {
    JFrame jfrm;
    public JTableDemo() {
        jfrm = new JFrame("A JTableDemo");
        // установить исходные размеры фрейма
        jfrm.setSize(500, 200);
     //завершить работу приложения, если пользователь закрывает
окно
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // отобразить фрейм
        jfrm.setVisible(true);
```

```
        // инициализировать заголовки столбцов
        String[] colHeads = {"Name ", "Extension", " ID"};
        // Имя, добавочный номер телефона, идентификационный номер
        // инициализировать данные
        Object[][] data = {{"Gail", "4567", "865"},
        {"Ken", "7566", "555"},
        {"Viviane", "4634", "587"},
        {"Melanie", "7345", "922"},
        {"Anne", "1237", "333"},
        {"John", "5656", "314"},
        {"Matt", "5672", "217"},
        {"Claire", "6741", "444"},
        {"Erwin", "9023", "519"},
        {"Ellen", "1134", "532"},
        {"Jennifer", "5689 ", "772"},
        {"Ed", "9030 ", "133"},
        {"Helen", "6751", "145"}};
        // создать таблицу
        JTable table = new JTable(data, colHeads);
        // ввести таблицу на панели с полосам прокрутки
        JScrollPane jsp = new JScrollPane(table);
        // ввести панель с полоса прокрутки на панели содержимого
        jfrm.add(jsp);
    }
    public static void main(String args[]) {
        try {
            // создать фрейм в потоке диспетчеризации событий
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    new JTableDemo();
                }
            });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }}}
```

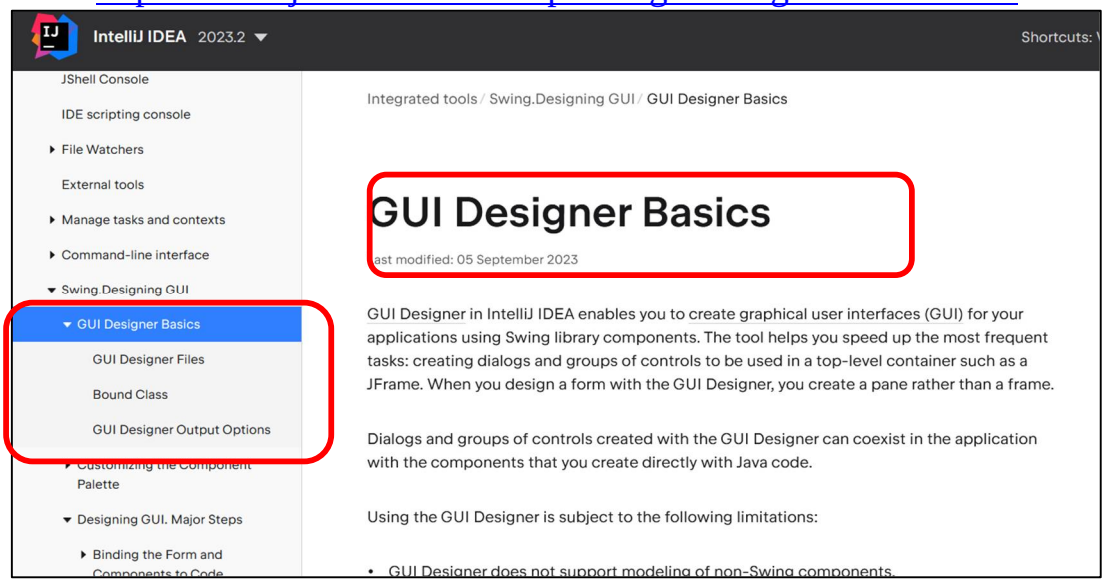
Результаты работы программы:



Дополнительные данные по созданию пользовательского интерфейса можно получить по следующим ссылкам: https://netbeans.org/kb/trails/matisse.html

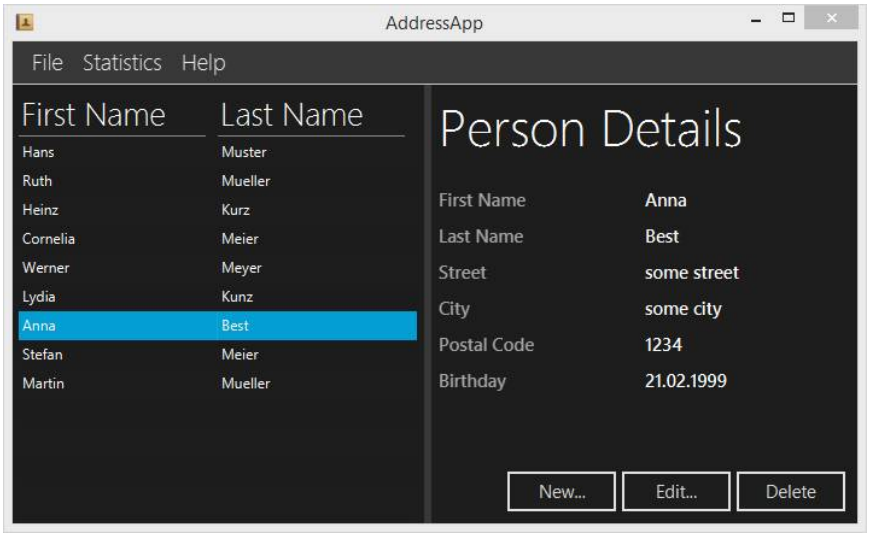


https://www.jetbrains.com/help/idea/gui-designer-basics.html



БИБЛИОТЕКА JAVAFX

В этой части рассмотрим создание приложения с функциональностью адресной книги.



СОЗДАНИЕ СТРУКТУРЫ ПАКЕТОВ

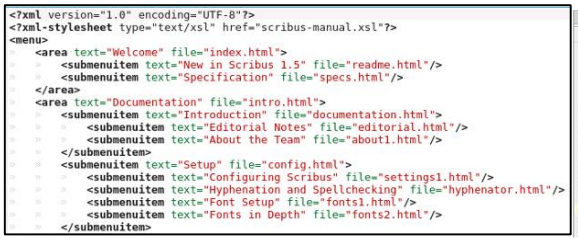
С самого начала будем следовать принципам проектирования программного обеспечения. Один из них – это шаблон проектирования «Модель-Представление-Контроллер (*MVC*)». Опираясь на этот шаблон, разобьём код приложения на три части и создадим для каждой из них свой пакет (контекстное меню папки *src->New->Package*):

- `iba.address` – будет содержать *большинство* классов-контроллеров (*Controller*, класс бизнес-логики);
- `iba.address.model` – содержит классы Моделей (*Model*, классы-данные);
- `iba.address.view` – содержит Представления (*View*, классы-отображения).

Внутри пакета `view` также будут лежать некоторые классы-контроллеры, которые непосредственно связаны с конкретными представлениями. Назовем их контроллеры-представлений (***view-controllers***).

СОЗДАНИЕ ФАЙЛА РАЗМЕТКИ FXML

Есть два пути создания пользовательского интерфейса: либо использовать файл разметки *FXML*, либо писать код на языке *Java*. Более рациональным и масштабируемым является использование файла разметки в формате *XML* (файл с расширением *.`fxml`).

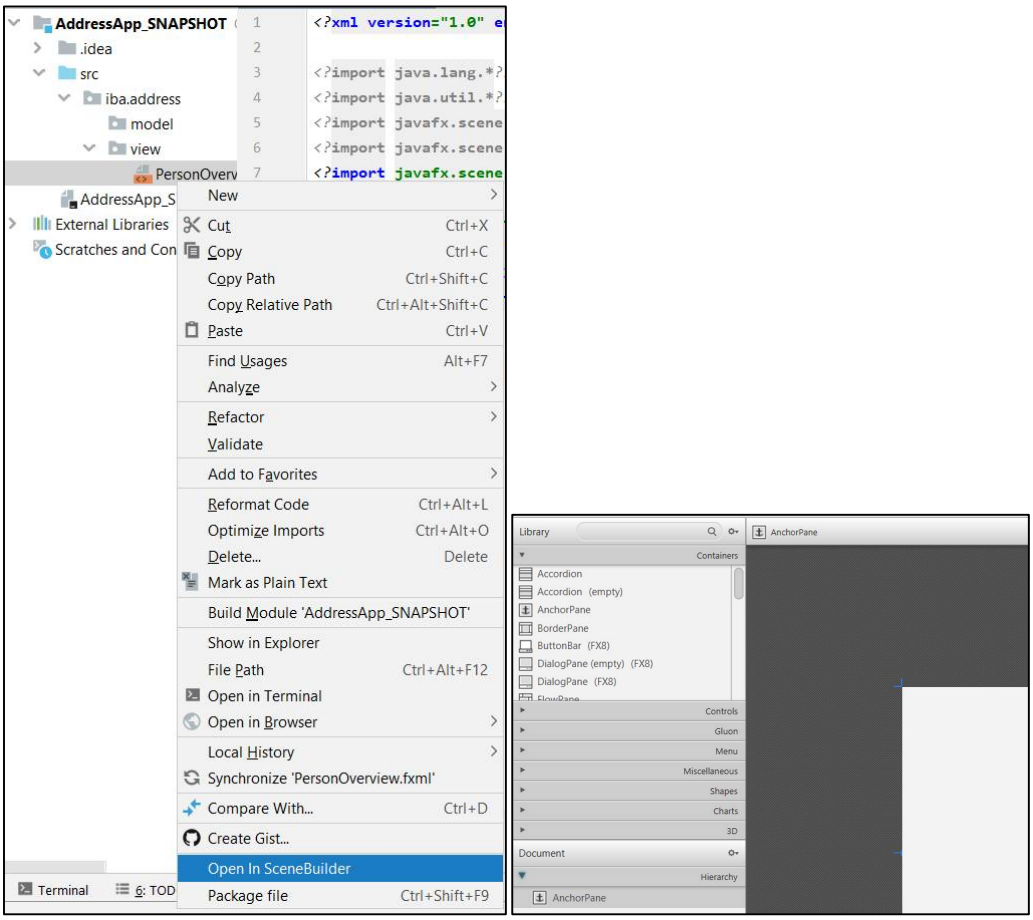


Этот подход соответствует паттерну *MVC* и используется **с целью отделения контроллеров от представлений.** Для визуального редактирования содержимого *XML*-файлов будем использовать `Scene Builder` (`графический редактор для пользовательского интерфейса`). А это значит, что разработчик может работать с содержимым *XML* не напрямую, а через графический редактор, что значительно оптимизирует процесс создания пользовательского интерфейса.

Кликните на пакете `view` правой кнопкой мышки и создайте новый документ *FXML* с названием `PersonOverview`.

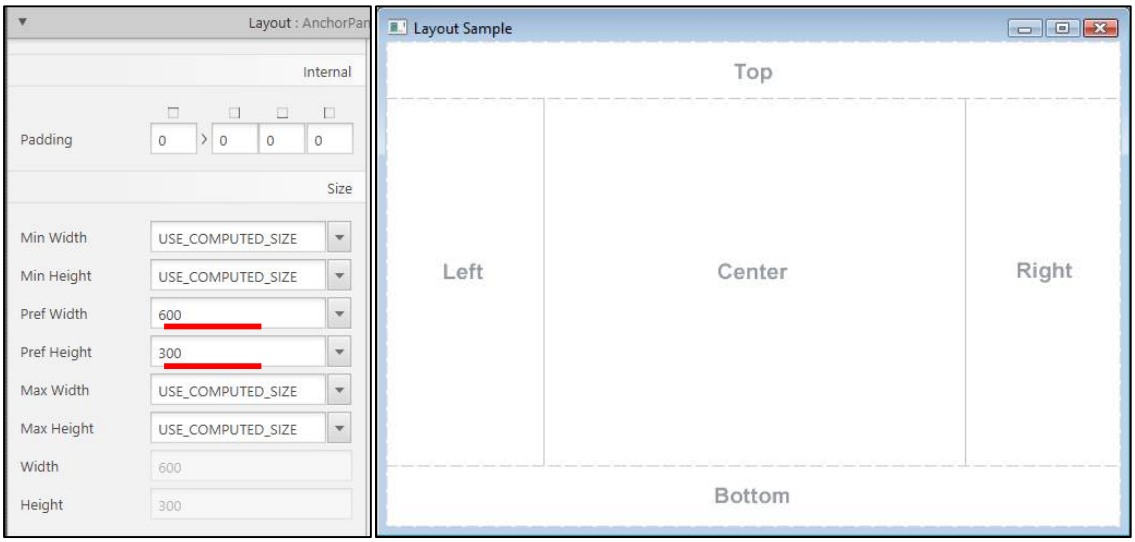
ВИЗУАЛЬНЫЙ ИНТЕРФЕЙС В SCENE BUILDER

Откройте только что созданный *fxml*-документ в приложении `Scene Builder` – клик правой кнопкой мышки по файлу `PersonOverview.fxml` и выберите пункт контекстного меню «*Open in SceneBuilder*».



В приложении *SceneBuilder* на вкладке *Hierarchy* должен находиться единственный компонент *AnchorPane*. Если `Scene Builder` не запустился, то необходимо настроить путь к **исполняемому файлу (.exe) установленного приложения** `Scene Builder`.

На вкладке *Hierarchy* выберите компонент *AnchorPane*, и справа, на вкладке *Layout (расположение)* установите значение характеристикам `Pref Width` (предпочитаемая ширина) и `Pref Height` (предпочитаемая ширина) – 600 и 300 соответственно.



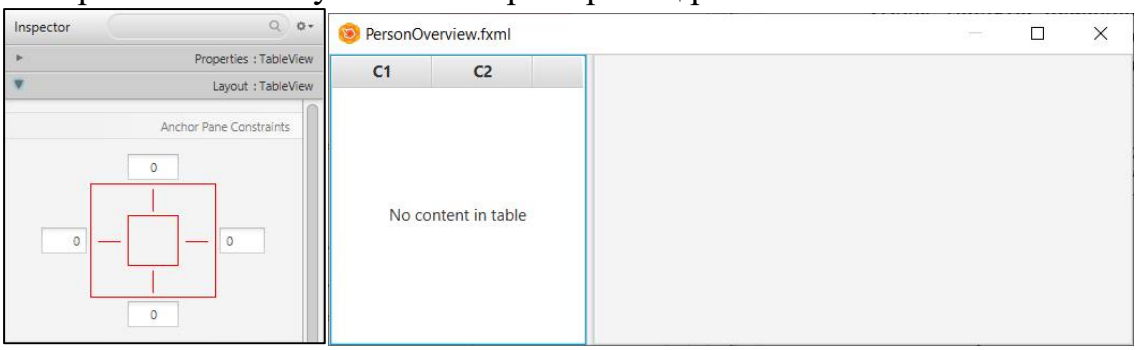
AnchorPane это контейнер (*container*), который разделяет поверхность окна на 5 логических областей, чтобы поставить анкер (*anchor*, якорь) подкомпонента. Компонент находящийся в контейнере типа *AnchorPane*

может поставить анкер в один или более логических областей *AnchorPane*. Изображение ниже иллюстрирует компонент в *AnchorPane*, закрепленный на левой, верхней и правой стороне *AnchorPane*. И когда *AnchorPane* меняет длину, длина компонента так же меняется. Компонент может быть закреплен анкером с 4-х сторон *AnchorPane*.

На вкладке *Hierarchy* в компонент *AnchorPane* добавьте новый компонент *SplitPane (horizontal)*. Кликните по нему правой кнопкой мыши и выберите *Fit to Parent*.

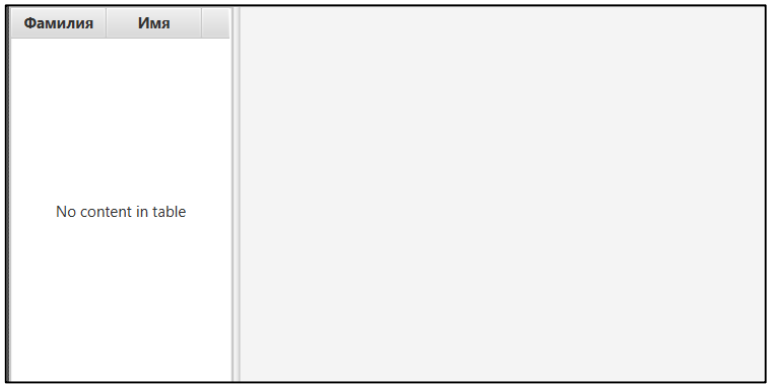
Компонент *SplitPane* представляет панель с горизонтальным или вертикальным набором разделенных областей. В левую часть компонента *SplitPane* со вкладки *Controls* перетащите компонент *TableView*. Компонент *TableView* используется вместе с компонентами *TableColumn* и *TableCell* и позволяет отобразить данные в виде таблицы (*Tabular form*).

Выделите его целиком (а не отдельный столбец) и проставьте отступы от краёв так, как показано на рисунке. Внутри компонента *AnchorPane* всегда можно проставить отступы от четырёх границ рамки.

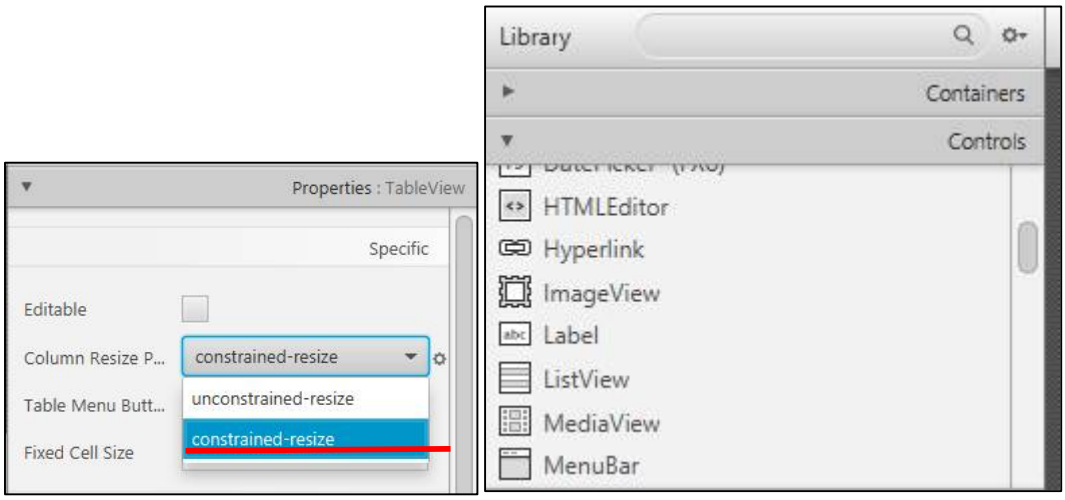


Чтобы увидеть, правильно ли отображается созданное окно, выполните пункт меню *Preview->Show Preview in Window*. Попробуйте поменять размер окна. Добавленная таблица должна изменятся вместе с окном, так как она прикреплена к границам окна.

В таблице измените заголовки колонок (вкладка *Properties* компонента *TableColumn*) на “Фамилия” и “Имя”.

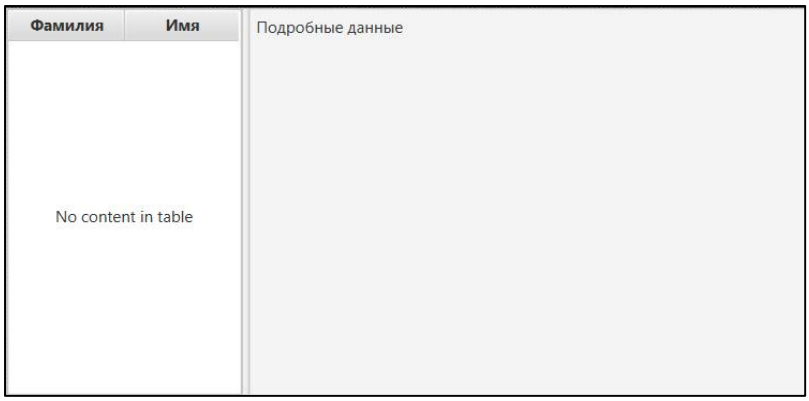


Выберите компонент *TableView* и во вкладке *Properties* измените значение *Column Resize Policy* на *constrained-resize*. Выбор этой характеристики гарантирует, что колонки таблицы всегда будут занимать всё доступное пространство контейнера.



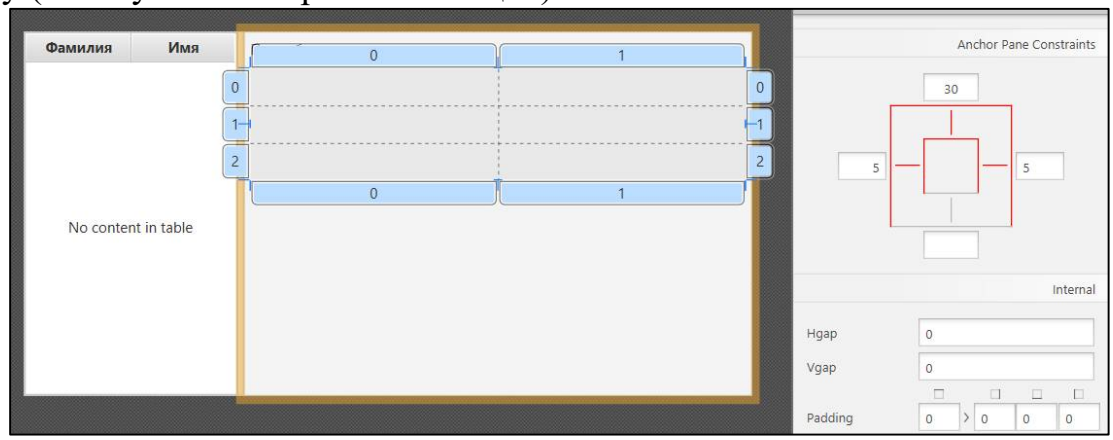
В правую часть компонента *SplitPane* перетащите компонент *Label* (со вкладки «*Controls*») и измените его текст на “`Подробные данные`”. Вместо навигации по вкладкам библиотеки компонентов можно использовать поиск для скорейшего их нахождения.

Используя привязки к границам (вкладка *Layout*) скорректируйте его положение.



На правую панель *SplitPane* добавьте компонент *GridPane* и так же настройте привязки к границам, как показано на рисунке.

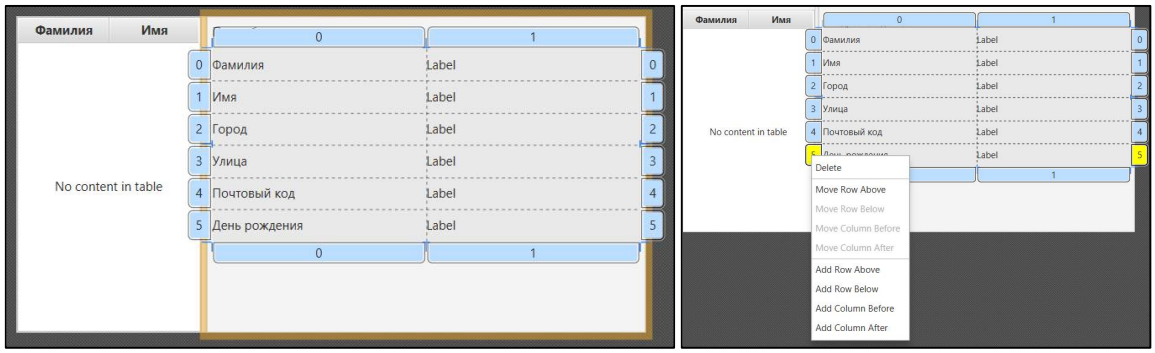
GridPane представляет собой контейнер, который делит поверхность на сетку (совокупность строк и столбцов).



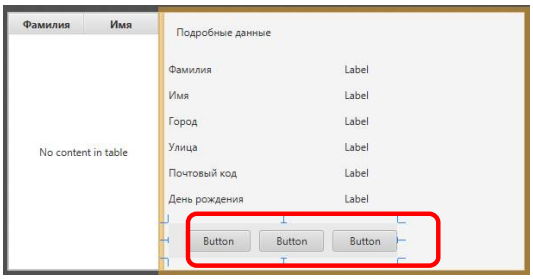
Приведите своё окно в соответствие с тем, что показано на рисунке, добавляя компоненты *Label* внутрь ячеек компонента *GridPane*.

Для того, чтобы добавить новый ряд в компонент *GridPane*, выберите существующий номер ряда (он окрасится жёлтым), кликните правой кнопкой

мышки на номере ряда и выберите пункт “*Add Row Above*” или “*Add Row Below*”.

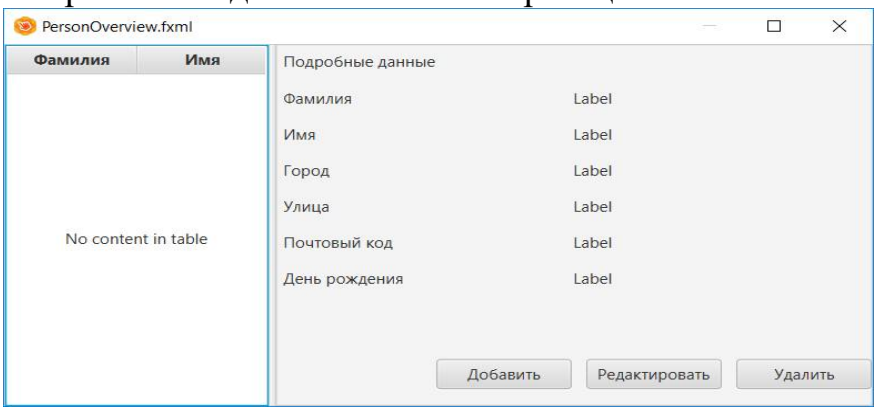


Внизу добавьте `ButtonBar`, а в него три кнопки *Button*. Установите привязку `ButtonBar` к границам (правой и нижней), чтобы `ButtonBar` всегда находилась справа.



Так как панель ButtonBar доступна только с JavaFX 8, и её поддержка в Scene Builder может несколько хромать, то имеется альтернативный способ. Добавьте три компонента *Button* в правую часть так, как показано на предыдущем рисунке. Выделите их всех вместе (`Shift` + клик), кликните по ним правой кнопкой мышки и выберите пункт *Wrap In | HBox*. Это действие их сгруппирует. Вы можете задать расстояние (*Spacing*) между компонентами во вкладке *Properties* компонента *HBox*.

Если всё сделано правильно, то должно получится окно, показанное на рисунке ниже. Используйте пункт меню *Preview| Show preview in window*, чтобы протестировать созданное окно и его реакцию на изменение размеров.



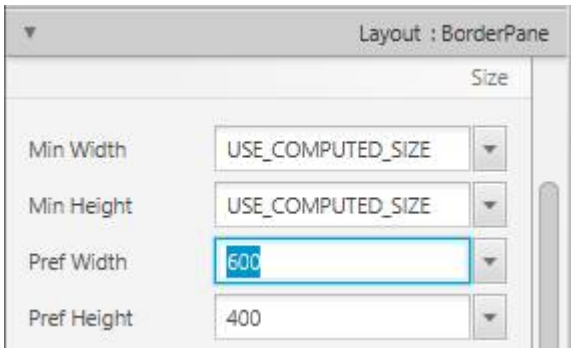
СОЗДАНИЕ ОСНОВНОГО ПРИЛОЖЕНИЯ

Необходимо создать ещё один файл *fxml*-разметки, в котором будет компонент полосы меню. Этот файл будет служить обёрткой для только что созданного `PersonOverview.fxml`.

В пакете `view` создайте другой `fxml`-документ, и назовите его `RootLayout.fxml`.

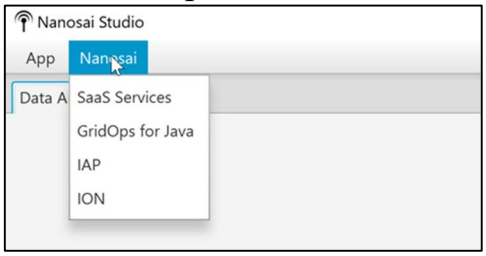
Откройте файл RootLayout.fxml в приложении Scene Builder. Добавьте в качестве корневого элемента BorderPane. BorderPane представляет собой контейнер (container), который разделен на 5 отдельных областей, каждая область может содержать компонент.

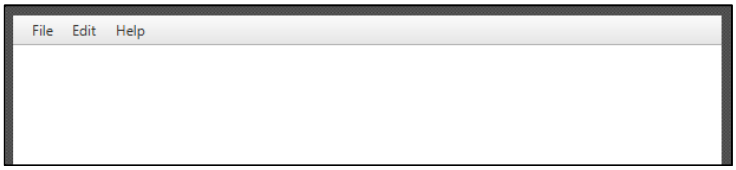
Установите предпочитаемое значение ширины и высоты компонента: 600 и 400 соответственно.



В верхний слот компонента *BorderPane* добавьте компонент *MenuBar*. Функциональность меню мы будем реализовывать позже.

`MenuBar` предоставляет приложениям визуальное выпадающее меню, подобное тому, что большинство настольных приложений имеют в верхней части окна приложения.





ОСНОВНОЙ КЛАСС ПРИЛОЖЕНИЯ JAVAFX

Теперь нам надо создать **основной класс** приложения `JavaFX`, который будет запускать приложение с `RootLayout.fxml` и добавлять в его центральную область содержимое файла `PersonOverview.fxml`.

Кликните правой кнопкой мыши по пакету *address* и выберите «*New-Java Class*». Назовите класс `MainApp`.

Созданный класс `MainApp.java` должен расширять класс `Application` и обязательно содержать два метода. Это базовая структура, которая необходима для запуска любого приложения `JavaFX`. Должен быть реализован метод `start(Stage primaryStage)`. Он автоматически вызывается при вызове метода `launch(...)` из метода `main()`.

Как можно заметить, метод `start(...)` в качестве параметра принимает экземпляр класса `Stage`. На следующем рисунке представлена структура любого приложения `JavaFX`.

Это как театральное представление `Stage` (театральные подмостки) является основным контейнером, который, как правило, представляет собой обрамлённое окно со стандартными кнопками: закрыть, свернуть, развернуть. Внутрь `Stage` добавляется сцена `Scene`, которая может быть заменена другой `Scene`. Внутрь `Scene` добавляются стандартные компоненты типа

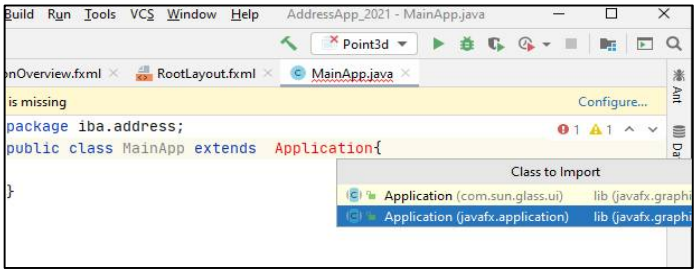
AnchorPane, *TextBox* и другие для размещения на них элементов управления.

Откройте класс *MainApp.java* и замените его содержимое на это:

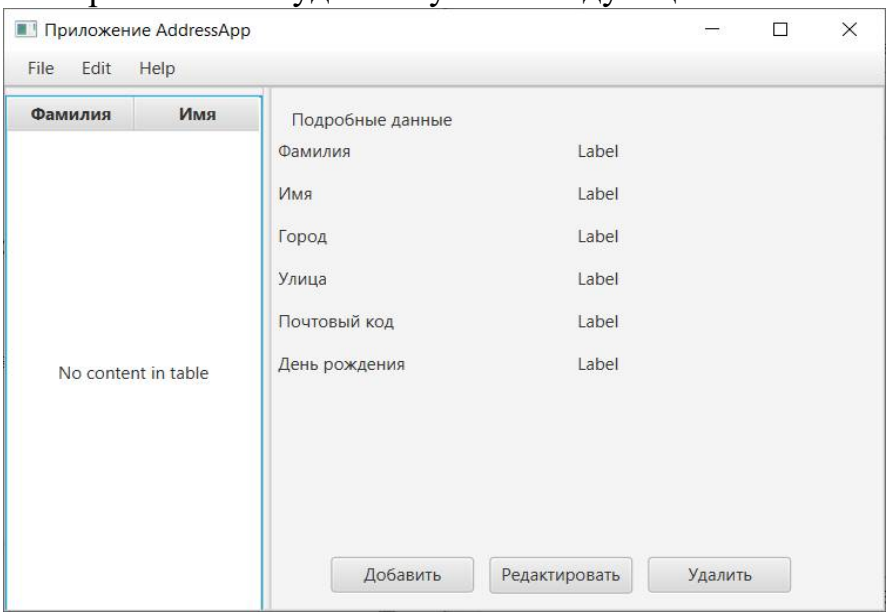
```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
package iba.address;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
import java.io.IOException;
public class MainApp extends Application {
    private Stage primaryStage;
    private BorderPane rootLayout;
    @Override
    public void start(Stage stage) {
        this.primaryStage = stage;
        this.primaryStage.setTitle("Приложение AddressApp");
        initRootLayout();
        showPersonOverview();
    }
    /**
     * Инициализирует корневой макет
     */
    private void initRootLayout() {
        try {
            //создаем стандартный загрузчик fxml файла
            FXMLLoader loader = new FXMLLoader();
            // Загружаем корневой макет из fxml файла
loader.setLocation(MainApp.class.getResource("view/RootLayout.fx
ml"));
            rootLayout = (BorderPane) loader.load();
            // Отображаем сцену, содержащую корневой макет
            Scene scene = new Scene(rootLayout);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Показывает в корневом макете сведения об адресатах
     */
    public void showPersonOverview() {
        try {
            //создаем стандартный загрузчик fxml файла
            FXMLLoader loader = new FXMLLoader();
            // Загружаем сведения об адресатах
```

```
loader.setLocation(MainApp.class.getResource("view/PersonOverview.fxml"));
            AnchorPane personOverview = (AnchorPane) loader.load();
            // Помещаем сведения об адресатах в центр корневого макета
            rootLayout.setCenter(personOverview);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Возвращает главную сцену
     */
    public Stage getPrimaryStage() {
        return primaryStage;
    }
    public static void main(String[] args) {
        launch(args);
    }}
```

При выборе подключаемого класса обращайте внимание на используемый пакет:



Запустив приложение будет получено следующее окно:



Если *JavaFX* не может найти указанный *fxml*-файл, то вы получите следующее сообщение об ошибке:

```
Caused by: java.lang.IllegalStateException: Location is not set.
    at javafx.fxml/javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:2459)
    at javafx.fxml/javafx.fxml.FXMLLoader.load(FXMLLoader.java:2435)
    at ch.makery.address.MainApp.showPersonOverview(MainApp.java:54)
    at ch.makery.address.MainApp.start(MainApp.java:24)
```

Для решения этой проблемы внимательно проверьте правильность указания пути ко всем необходимым файлам и правильность написания их имен.

В ходе запуска приложения возникнет такого рода ошибка:

```
javafx.fxml.LoadException:
/C:/Work/IBA/2_Java/AddressApp_SNAPSHOT/out/production/AddressApp_SNAPSHOT/iba/address/view/PersonOverview.fxml:14

    at javafx.fxml/javafx.fxml.FXMLLoader.constructLoadException(FXMLLoader.java:2625)
    at javafx.fxml/javafx.fxml.FXMLLoader.access$700(FXMLLoader.java:105)
    at javafx.fxml/javafx.fxml.FXMLLoader$ValueElement.processAttribute(FXMLLoader.java:930)
    at javafx.fxml/javafx.fxml.FXMLLoader$InstanceDeclarationElement.processAttribute(FXMLLoader.java:980)
    at javafx.fxml/javafx.fxml.FXMLLoader$Element.processStartElement(FXMLLoader.java:227)
    at javafx.fxml/javafx.fxml.FXMLLoader$ValueElement.processStartElement(FXMLLoader.java:752)
    at javafx.fxml/javafx.fxml.FXMLLoader.processStartElement(FXMLLoader.java:2722)
    at javafx.fxml/javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:2552)
    at javafx.fxml/javafx.fxml.FXMLLoader.loadImpl(FXMLLoader.java:2466)
    at javafx.fxml/javafx.fxml.FXMLLoader.load(FXMLLoader.java:2435)
    at iba.address.MainApp.showPersonOverview(MainApp.java:47)
    at iba.address.MainApp.start(MainApp.java:18)
    at javafx.graphics/com.sun.javafx.application.LauncherImpl.lambda$launchApplication1$9(LauncherImpl.java:846)
    at javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runAndWait$12(PlatformImpl.java:455)
    at javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLater$10(PlatformImpl.java:428)
    at java.base/java.security.AccessController.doPrivileged(AccessController.java:391)
    at javafx.graphics/com.sun.javafx.application.PlatformImpl.lambda$runLater$11(PlatformImpl.java:427)
    at javafx.graphics/com.sun.glass.ui.InvokeLaterDispatcher$Future.run(InvokeLaterDispatcher.java:96)
    at javafx.graphics/com.sun.glass.ui.win.WinApplication._runLoop(Native Method)
    at javafx.graphics/com.sun.glass.ui.win.WinApplication.lambda$runLoop$3(WinApplication.java:174)
    at java.base/java.lang.Thread.run(Thread.java:830)
Caused by: java.lang.ClassNotFoundException: iba.address.view.PersonOverview <2 internal calls>
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
    at javafx.fxml/javafx.fxml.FXMLLoader$ValueElement.processAttribute(FXMLLoader.java:928)
    ... 18 more
```

Её в следующей части приложения необходимо будет исправить.

СОЗДАНИЕ КЛАССА-МОДЕЛИ

Класс-модель необходим для хранения в адресной книге информации об адресатах. Добавьте класс ***Person.java*** в пакет ***iba.address.model.*** В нём будет несколько переменных для хранения информации об имени, адресе и дне рождения. Добавьте в этот класс следующий код.

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//Person.java
package iba.address.model;
import javafx.beans.property.*;
import java.time.LocalDate;

/**
 * Класс-модель для адресата (Person)
 */
public class Person {
    private final StringProperty firstName;
    private final StringProperty lastName;
    private final StringProperty city;
    private final StringProperty street;
    private final IntegerProperty postalCode;
    private final ObjectProperty<LocalDate> birthday;
    /**
     * Конструктор по умолчанию
     */
    public Person() {
```

```
    this(null,null);
}
public String getFirstName() {
    return firstName.get();
}
public StringProperty firstNameProperty() {
    return firstName;
}
public void setFirstName(String firstName) {
    this.firstName.set(firstName);
}
public String getLastName() {
    return lastName.get();
}
public StringProperty lastNameProperty() {
    return lastName;
}
public void setLastName(String lastName) {
    this.lastName.set(lastName);
}
public String getCity() {
    return city.get();
}
public StringProperty cityProperty() {
    return city;
}
public void setCity(String city) {
    this.city.set(city);
}
public String getStreet() {
    return street.get();
}
public StringProperty streetProperty() {
    return street;
}
public void setStreet(String street) {
    this.street.set(street);
}
public int getPostalCode() {
    return postalCode.get();
}
public IntegerProperty postalCodeProperty() {
    return postalCode;
}
public void setPostalCode(int postalCode) {
    this.postalCode.set(postalCode);
}
public LocalDate getBirthday() {
    return birthday.get();
}
public ObjectProperty<LocalDate> birthdayProperty() {
    return birthday;
}
```

```
    public void setBirthday(LocalDate birthday) {
        this.birthday.set(birthday);
    }
    /**
     * Конструктор с некоторыми начальными данными.
     *
     * @param firstName
     * @param lastName
     */
    public Person(String firstName, String lastName) {
        this.firstName = new SimpleStringProperty(firstName);
        this.lastName = new SimpleStringProperty(lastName);
        this.city=new SimpleStringProperty("г. Минск");
        this.street=new                SimpleStringProperty("пр.
Независимости");
        this.postalCode=new SimpleIntegerProperty(1234);
        this.birthday=new
SimpleObjectProperty<LocalDate>(LocalDate.of(2000,01,01));
    }
}
```

В *JavaFX* для всех полей класса-модели предпочтительно использовать *Properties*. Типы этих классов стоит использовать тогда, когда эта переменные будут наблюдаться другими блоками или модулями программы. *StringProperty* – это абстрактный базовый класс для наблюдаемых свойств строки, *SimpleStringProperty* – это конкретная реализация. 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.

Свойство модели *JavaFX* является наблюдаемым контейнером, который облегчает привязку данных: представление может быть связано со свойствами в модели представления, так что оно автоматически обновляется при изменении модели. Поэтому **стоит использовать поля свойств, если нужна привязка данных, и использовать обычные строки, когда не нужна привязка данных.**

В любом случае свойство должно быть привязано к конкретному объекту модели. Свойство следует рассматривать как особый вид переменной, а не как обычный объект. Поэтому передача свойства в сеттере или в конструкторе очень необычна. Вместо этого геттеры, сеттеры и конструкторы обычно работают с данными, хранящимися в свойстве. Таким образом, классы на основе свойств также могут использоваться в качестве компонентов *Java*.

Класс *LocalDate*, тип которого выбран для переменной *birthday*, это часть нового *Date and Time API* для *JDK* 8.

Основные данные, которыми оперирует разрабатываемое приложение – это группа экземпляров класса *Person*. Создадим в классе *MainApp.java* список объектов класса *Person*. Все остальные классы-контроллеры позже получат доступ к этому центральному списку внутри этого класса.

СПИСОК OBSERVABLELIST

Мы работаем с классами-представлениями *JavaFX*, которые необходимо информировать при любых изменениях в списке адресатов. Это важно, потому что, не будь этого, мы бы не смогли синхронизировать представление данных с самими данными. Для этой цели в *JavaFX* были введены некоторые новые классы коллекций.

Из этих классов в разрабатываемом приложении понадобится класс *ObservableList*. Для создания экземпляра данного класса добавьте приведённый код в начало класса *MainApp.java*. Мы так же добавим в код конструктор, который будет создавать некоторые демонстрационный данные и метод-геттер с публичным модификатором доступа:

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
package iba.address;
import iba.address.model.Person;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
import java.io.IOException;
public class MainApp extends Application {
    private Stage primaryStage;
    private BorderPane rootLayout;
    private ObservableList<Person> personData = FXCollections.observableArrayList();
    @Override
    public void start(Stage stage) {
        this.primaryStage = stage;
        this.primaryStage.setTitle("Приложение AddressApp");
        initRootLayout();
        showPersonOverview();
    }
    public MainApp() {
        personData.add(new Person("Александра", "Мишкина"));
        personData.add(new Person("Иван", "Иванов"));
        personData.add(new Person("Петр", "Захаров"));
        personData.add(new Person("Даниил", "Петров"));
        personData.add(new Person("Светлана", "Захарова"));
        personData.add(new Person("Лидия", "Адамова"));
        personData.add(new Person("Анна", "Сергеева"));
        personData.add(new Person("Стефан", "Маховский"));
        personData.add(new Person("Мартин", "Петровский"));
    }
    /**
     * Возвращает данные в виде наблюдаемого списка адресатов.
     * @return
```

```
     */
    public     ObservableList<Person>     getPersonData(){return
personData;}
    /**
     * Инициализирует корневой макет.
     */
    private void initRootLayout() {
        try {
            // Загружаем корневой макет из fxml файла
            FXMLLoader loader = new FXMLLoader();

loader.setLocation(MainApp.class.getResource("view/RootLayout.fx
ml"));
            rootLayout = (BorderPane) loader.load();
            // Отображаем сцену, содержащую корневой макет
            Scene scene = new Scene(rootLayout);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Показывает в корневом макете сведения об адресатах
     */
    public void showPersonOverview() {
        try {
            // Загружаем сведения об адресатах
            FXMLLoader loader = new FXMLLoader();

loader.setLocation(MainApp.class.getResource("view/PersonOvervie
w.fxml"));
            AnchorPane     personOverview     =     (AnchorPane)
loader.load();
            // Помещаем сведения об адресатах в центр корневого
макета
            rootLayout.setCenter(personOverview);
            // Даём контроллеру доступ к главному приложению
            PersonOverviewController
controller=loader.getController();
            controller.setMainApp(this);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     * Возвращает главную сцену.
     */
    public Stage getPrimaryStage() {
        return primaryStage;
    }
    public static void main(String[] args) {
        launch(args);
```

```
}}
```

КЛАСС PERSONOVERVIEWCONTROLLER

Теперь мы отобразим в таблице некоторые данные. Для этого необходимо создать класс-контроллер для представления *PersonOverview.fxml*. Для этого:

– создайте новый класс внутри пакета *view* и назовите его *PersonOverviewController.java*. Мы должны разместить этот класс-контроллер в том же пакете, где находится файл разметки *PersonOverview.fxml*, иначе *Scene Builder* не сможет его найти.

– для того, чтобы получить доступ к таблице и меткам представления, мы определим некоторые переменные. Эти переменные и некоторые методы имеют специальную аннотацию *@FXML*. Она необходима для того, чтобы *fxml*-файл имел доступ к приватным полям и методам класса. После этого мы настроим *fxml*-файл так, что при его загрузке приложение автоматически заполняло эти переменные данными. Добавим следующий код в класс *PersonOverviewController.java.*

При импорте пакетов всегда используйте пакет *javafx*, а НЕ пакеты *awt* или *swing*!

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
package iba.address.view;
import iba.address.MainApp;
import iba.address.model.Person;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
public class PersonOverviewController {
    @FXML
    private TableView<Person> personTable;
    @FXML
    private TableColumn<Person, String> firstNameColumn;
    @FXML
    private TableColumn<Person, String> lastNameColumn;
    @FXML
    private Label firstNameLabel;
    @FXML
    private Label lastNameLabel;
    @FXML
    private Label streetLabel;
    @FXML
    private Label postalCodeLabel;
    @FXML
    private Label cityLabel;
    @FXML
    private Label birthdayLabel;
    // Ссылка на главное приложение
    private MainApp mainApp;
```

```
    /**
     * Конструктор.
     * Конструктор вызывается раньше метода initialize().
     */
    public PersonOverviewController() {
    }
    /*Инициализация класса-контроллера. Этот метод вызывается автоматически после того, как fxml-файл будет загружен*/
    @FXML
    private void initialize() {
        //метод setCellValueFactory определяет, как необходимо заполнить данные каждой ячейки
        // Инициализация столбца с данными об именах
        firstNameColumn.setCellValueFactory(cellData -> cellData.getValue().firstNameProperty());
        // Инициализация столбца с данными об фамилиях
        lastNameColumn.setCellValueFactory(cellData -> cellData.getValue().lastNameProperty());
    }
    /**
     * Вызывается главным приложением, которое даёт на себя ссылку
     *
     * @param mainApp
     */
    public void setMainApp(MainApp mainApp) {
        this.mainApp = mainApp;
        personTable.setItems(mainApp.getPersonData());
    }
}
```

Все поля и методы, к которым *fxml*-файлу потребуется доступ, должны быть отмечены аннотацией *@FXML*. Несмотря на то, что это требование предъявляется только для полей и методов с модификатором *private*, лучше оставить их закрытыми и помечать аннотацией, чем делать публичными!

```
@Retention(RUNTIME)
@Target({FIELD,METHOD})
public @interface FXML
```

Annotation that tags a field or method as accessible to markup.

При запуске javaFX приложения сначала вызывается конструктор, затем заполняются все аннотированные @FXML поля, затем вызывается метод initialize(). Таким образом, конструктор не имеет доступа к полям @FXML, ссылающимся на компоненты, определенные в файле fxml, в то время как initialize() имеет к ним доступ. После загрузки *fxml*-файла автоматически вызывается метод *initialize()*. На этот момент все *FXML*-поля должны быть инициализированы.

Метод *setCellValueFactory(...)* определяет, какое поле внутри класса *Person* будут использоваться для заполнения конкретного столбца в таблице.

В примере для столбцов таблицы использовались только значения *StringProperty*. Если понадобится использовать *IntegerProperty*, *DoubleProperty*, то *setCellValueFactory(...)* должен иметь дополнительный метод *asObject():*

```
myIntegerColumn.setCellValueFactory(cellData ->
cellData.getValue().myIntegerProperty().asObject());
```

Это добавление необходимо сделать из-за неудачного решения при проектировании *JavaFX*.

СОЕДИНЕНИЕ КЛАССА MAINAPP С КЛАССОМ PERSONOVERVIEWCONTROLLER

Метод *setMainApp(...)* должен быть вызван из класса *MainApp*. Это даст нашему контроллеру доступ к экземпляру *MainApp*, к коллекции записей *personList* внутри него и к другим элементам класса. Для этого в метод *showPersonOverview()* необходимо добавить две дополнительные строки:

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//MainApp.java - метод showPersonOverview()
  /**
   * Показывает в корневом макете сведения об адресатах
   */
  public void showPersonOverview() {
      try {
        // Загружаем сведения об адресатах
        FXMLLoader loader = new FXMLLoader();

loader.setLocation(MainApp.class.getResource("view/PersonOverview.fxml"));
       AnchorPane personOverview = (AnchorPane) loader.load();
       // Помещаем сведения об адресатах в центр корневого макета
       rootLayout.setCenter(personOverview);
       // создание экземпляра класса контроллера
       PersonOverviewController controller=loader.getController();
       // передача контроллеру ссылки на главное приложение
         controller.setMainApp(this);
     } catch (IOException e) {
        e.printStackTrace();
    }
  }
```

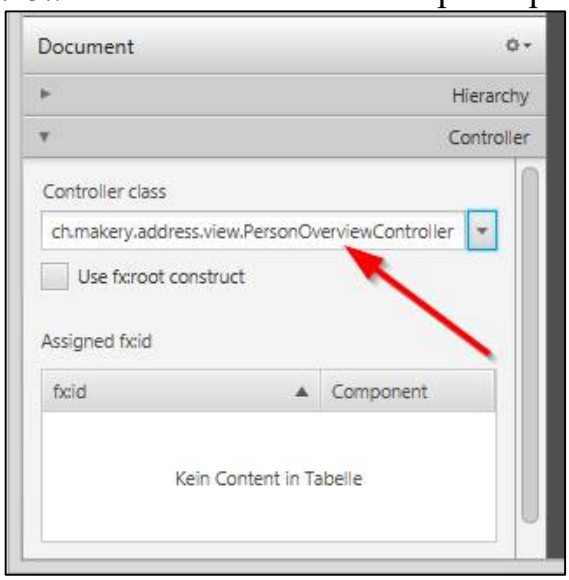
ПРИВЯЗКА КЛАССА-КОНТРОЛЛЕРА К FXML-ФАЙЛУ

Необходимо сообщить файлу *PersonOverview.fxml*, какой контроллер он должен использовать, а также необходимо указать соответствие между элементами представления и полями внутри класса-контроллера.

Откройте файл *PersonOverview.fxml* в приложении *Scene Builder*.

Откройте вкладку *Controller* слева на панели *Document* и выберите класс

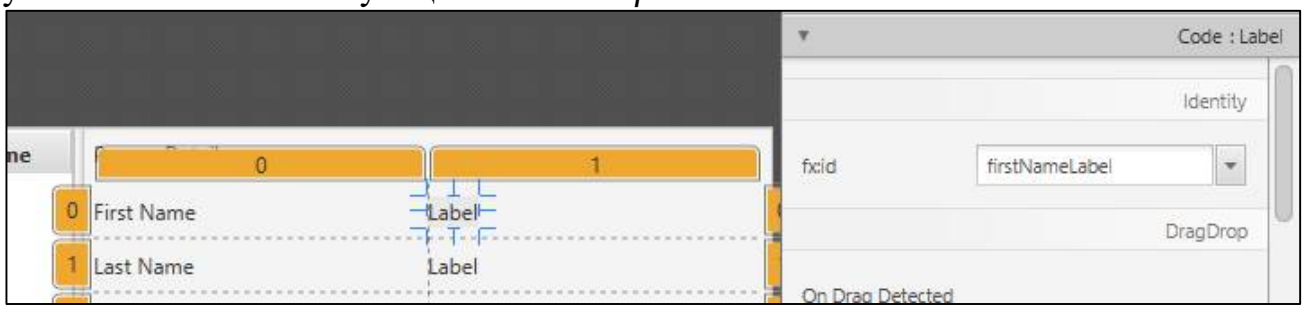
PersonControllerOverview в качестве класса-контроллера.



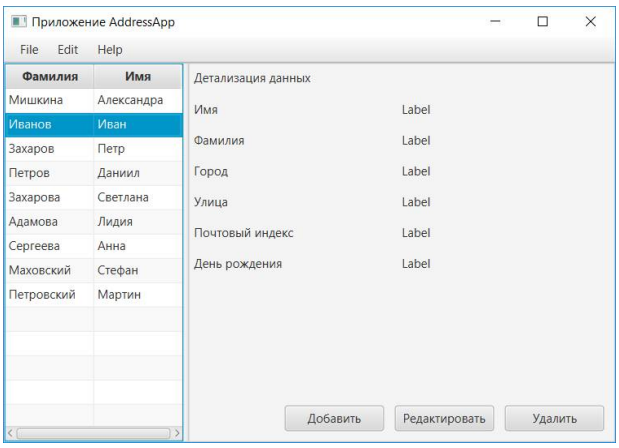
Выберите компонент *TableView* на вкладке *Hierarchy*, перейдите на вкладку *Code* и в поле *fx:id* установите значение *personTable*.

Сделайте то же самое для колонок таблицы и установите значения свойства *fx:id firstNameColumn* и *lastNameColumn* соответственно.

Для каждой метки во второй колонке компонента *GridPane* также установите соответствующие значения *fx:id.*



Важно: сохраните файл *PersonOverview.fxml*, вернитесь в среду разработки и обновите весь проект *AdressApp*. Это необходимо для того, чтобы приложение “увидело” те изменения, которые мы сделали в приложении *Scene Builder*. После запуска приложения мы должны увидеть следующее окно:



РЕАКЦИЯ НА ВЫБОР АДРЕСАТОВ В ТАБЛИЦЕ

Пока мы ещё не использовали правую часть приложения. Идея заключается в том, чтобы при выборе адресата в таблице, в правой части приложения отображать дополнительную информацию об этом адресате.

Сначала добавим новый метод в класс *PersonOverviewController*. Он будет заполнять текстовые метки данными указанного адресата (*Person*).

Создайте метод *showPersonDetails(Person person)*. Его код проходится по всем меткам, и, используя метод *setText(...)*, присваивает им соответствующие значения, взятые из переданного в параметре объекта *Person*. Если в качестве параметра передаётся *null*, то весь текст в метках будет очищен.

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//Файл PersonOverviewController.java
/*Метод заполняет все текстовые поля, отображая подробности об адресате. Если указанный адресат = null, то все текстовые поля очищаются.
     * @param person – адресат типа Person или null
     */
    private void showPersonDetails(Person person) {
        if (person != null) {
            // Заполняем метки информацией из объекта person.
            firstNameLabel.setText(person.getFirstName());
            lastNameLabel.setText(person.getLastName());
            streetLabel.setText(person.getStreet());

postalCodeLabel.setText(Integer.toString(person.getPostalCode())
);
            cityLabel.setText(person.getCity());
            // TODO: Нам нужен способ для перевода дня рождения в тип String!
            // birthdayLabel.setText(...);
        } else {
            // Если Person = null, то убираем весь текст.
            firstNameLabel.setText("");
            lastNameLabel.setText("");
            streetLabel.setText("");
            postalCodeLabel.setText("");
```

```
            cityLabel.setText("");
            birthdayLabel.setText("");
        }
    }
```

ПРЕОБРАЗОВАНИЕ ДНЯ РОЖДЕНИЯ В СТРОКУ

Обратите внимание, что мы просто так не можем присвоить текстовой метке значение поля *birthday*, так как тип его значения *LocalDate,* а не *String*. Для того чтобы это сделать, надо сначала отформатировать дату.

Мы будем преобразовывать тип *LocalDate* в тип *String* и обратно в нескольких местах программы, поэтому, хорошей практикой считается создание для этой цели вспомогательного класса, содержащего статические методы. Этот вспомогательный класс мы назовем *DateUtil* и разместим его в новом пакете *ch.makery.address.util:*

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
public class DateUtil {
    /**
     * Шаблон даты, используемый для преобразования
     */
    private static final String DATE_PATTERN = "dd.MM.yyyy";
    /**
     * Форматировщик даты.
     */
    private static final DateTimeFormatter DATE_FORMATTER = DateTimeFormatter.ofPattern(DATE_PATTERN);
    /**
     * Возвращает полученную дату в виде отформатированной строки.
     *      Используется определённый выше {@link DateUtil#DATE_PATTERN}.
     *
     * @param date - дата, которая будет возвращена в виде строки
     * @return отформатированную строку
     */
    public static String format(LocalDate date) {
        if (date == null) return null;
        return DATE_FORMATTER.format(date);
    }
    /**
     * Преобразует строку, которая отформатирована по правилам
     *  шаблона {@link DateUtil#DATE_PATTERN} в объект {@link LocalDate}.
     * <p>
     * Возвращает null, если строка не может быть преобразована.
     *
     * @param dateString - дата в виде String
     * @return объект даты или null, если строка не может быть преобразована
```

```
     */
    public static LocalDate parse(String dateString) {
        try {
            return                DATE_FORMATTER.parse(dateString,
LocalDate::from);
        } catch (DateTimeParseException e) {
            return null;
        }
    }
    /**
     * Проверяет, является ли строка корректной датой
     *
     * @param dateString
     * @return true, если строка является корректной датой
     */
    public static boolean validDate(String dateString) {
        // Пытаемся разобрать строку
        return DateUtil.parse(dateString) != null;
    }
}
```

Формат даты можно поменять, просто изменив константу *DATE_PATTERN*. Все возможные форматы описаны в документации к классу *DateTimeFormatter*.

ИСПОЛЬЗОВАНИЕ КЛАССА DATEUTIL

Теперь можно использовать новый класс *DateUtil* в методе *showPersonDetails()* класса *PersonOverviewController*. Замените комментарий *TODO* следующей строкой:

```
birthdayLabel.setText(DateUtil.format(person.getBirthday()));
```

НАБЛЮДЕНИЕ ЗА ВЫБОРОМ АДРЕСАТОВ В ТАБЛИЦЕ

Чтобы знать о том, что пользователь выбрал определённую запись в таблице, необходимо прослушивать изменения в таблице.

Для этого в *JavaFX* существует интерфейс *ChangeListener* с единственным методом *changed(...)*. Этот метод имеет три параметра: *observable*, *oldValue* и *newValue*.

Modifier and Type	Method and Description
void	changed(ObservableValue<? extends T> observable, T oldValue, T newValue) This method needs to be provided by an implementation of ChangeListener.

Мы будем реализовывать интерфейс *ChangeListener* с помощью лямбда-выражений. Добавим несколько строчек кода к методу *initialize()* класса *PersonOverviewController*. Теперь этот метод выглядит так:

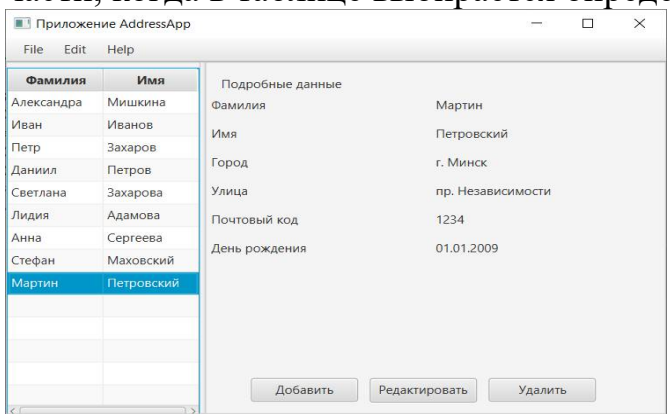
```
//Пример №11. Поэтапное создание приложения с использованием
библиотеки JavaFX
//Файл PersonOverviewController.java
/*Инициализация класса-контроллера. Этот метод вызывается
автоматически после того, как fxml-файл будет загружен.*/
    @FXML
    private void initialize() {
```

```
        // Инициализация таблицы адресатов с двумя столбцами
        firstNameColumn.setCellValueFactory(cellData ->
cellData.getValue().firstNameProperty());
        lastNameColumn.setCellValueFactory(cellData ->
cellData.getValue().lastNameProperty());
        // Очистка дополнительной информации об адресате
        showPersonDetails(null);
        // Слушаем изменения выбора, и при изменении отображаем
дополнительную информацию об адресате
personTable.getSelectionModel().selectedItemProperty().addListen
er((observable, oldValue, newValue) ->
showPersonDetails(newValue));
    }
```

Если мы передаём в параметр метода *showPersonDetails(...)* значение *null*, то все значения меток будут очищены.

В строке *personTable.getSelectionModel()...* получаем *selectedItemProperty* таблицы и добавляем к нему слушателя. Когда пользователь выбирает запись в таблице, выполняется указанное лямбда-выражение. Выбранная запись передаётся её в метод *showPersonDetails(...)*.

Запустите свое приложение и проверьте, отображаются ли детали адресата в правой части, когда в таблице выбирается определённый адресат.

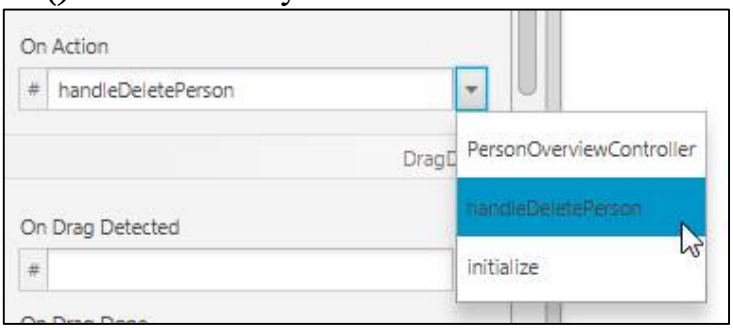


В приложении *Scene Builder* можно указать действие, которое будет выполняться при нажатии на кнопку. Любой метод внутри класса-контроллера, помеченный аннотацией *@FXML* (или публичный), доступен *Scene Builder*. Добавим в конец класса *PersonOverviewController* метод удаления адресата, а потом назначим его обработчиком кнопки *Delete*.

```
//Пример №11. Поэтапное создание приложения с использованием
библиотеки JavaFX
PersonOverviewController.java
/**
     * Вызывается, когда пользователь кликает по кнопке удаления
     */
    @FXML
    private void handleDeletePerson() {
        int selectedIndex =
personTable.getSelectionModel().getSelectedIndex();
        personTable.getItems().remove(selectedIndex);
```

```
    }
```

Теперь в приложении *Scene Builder* откройте файл *PersonOverview.fxml*. Выберите кнопку *Delete*, откройте вкладку *Code* и укажите метод *handleDeletePerson()* в значение пункта *On Action*.



После изменения *FXML*-файла в *SceneBuilder* для того, чтобы изменения вступили в силу, может понадобится обновить проект в *IDE*.

ОБРАБОТКА ОШИБОК

Если вы сейчас запустите приложение, то сможете удалять выбранных адресатов из таблицы. Но что произойдёт, если не будет выбран ни один адресат, а вы нажмёте кнопку *Delete*?

Будет создано исключение *ArrayIndexOutOfBoundsException*, потому что не получится удалить адресата с индексом -1. Значение -1 возвращается методом *getSelectedIndex()*, когда в таблице ничего не выделено.

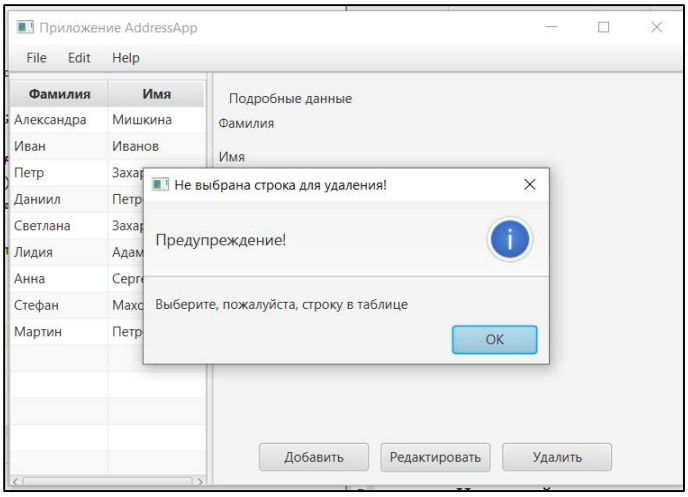
```
Caused by: java.lang.IndexOutOfBoundsException: Index -1 out of bounds for length 9
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:248)
    at java.base/java.util.Objects.checkIndex(Objects.java:372)
```

Игнорировать такого рода ошибки нельзя. Необходимо сообщить пользователю о том, что он, перед тем как нажимать кнопку *Delete*, должен выбрать запись в таблице. Ещё лучше совсем деактивировать кнопку, чтобы у пользователя не было возможности выбрать кнопку пока не выбрана строка в таблице.

Добавим в метод *handleDeletePerson()* некоторые изменения. Теперь, когда пользователь нажимает на кнопку *Delete*, а в таблице ничего не выбрано, он увидит простое диалоговое окно:

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//Файл PersonOverviewController.java
/* Вызывается, когда пользователь кликает по кнопке удаления */
@FXML
    private void handleDeletePerson() {
        int                    selectedIndex                    =
personTable.getSelectionModel().getSelectedIndex();
        if (selectedIndex >= 0)
            personTable.getItems().remove(selectedIndex);
        else{
            Alert alert=new Alert(Alert.AlertType.INFORMATION);
```

```
            alert.initOwner(mainApp.getPrimaryStage());
            alert.setTitle("Не выбрана строка для удаления!");
            alert.setHeaderText("Предупреждение!");
            alert.setContentText("Выберите, пожалуйста, строку в
таблице");
            alert.showAndWait();
        }    }
```

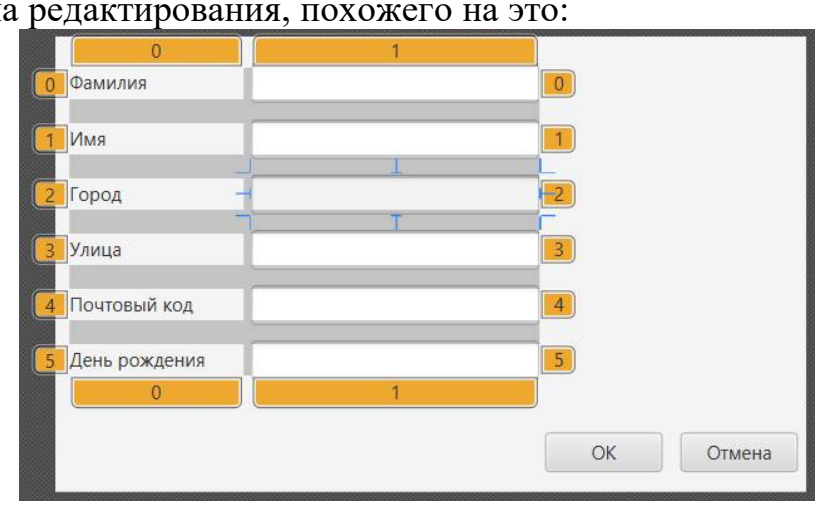


Для реализации методов-обработчиков создания и изменения адресатов необходимо создать новое пользовательское окно (то есть сцену) с формой, содержащей поля для заполнения всей необходимой информации об адресате.

ДИЗАЙН ОКНА РЕДАКТИРОВАНИЯ

Внутри пакета *view* создайте новый *fxml*-файл *PersonEditDialog*.

Используйте компоненты *GridPane*, *Label*, *TextField* и *Button* для создания окна редактирования, похожего на это:



СОЗДАНИЕ КОНТРОЛЛЕРА

Создайте класс-контроллер для нового окна *PersonEditDialogController.java*:

```
//Пример №11. Поэтапное создание приложения с использованием
библиотеки JavaFX
package iba.address.view;
//PersonEditDialogController.java
import iba.address.model.Person;
```

```
import iba.address.util.DateUtil;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.TextField;
import javafx.stage.Stage;
/*Окно для изменения информации об адресате*/
public class PersonEditDialogController {
    @FXML
    private TextField firstNameField;
    @FXML
    private TextField lastNameField;
    @FXML
    private TextField streetField;
    @FXML
    private TextField postalCodeField;
    @FXML
    private TextField cityField;
    @FXML
    private TextField birthdayField;
    private Stage dialogStage;
    private Person person;
    private boolean okClicked = false;
    /**
     * Инициализирует класс-контроллер. Этот метод вызывается
автоматически после того, как fxml-файл будет загружен.
     */
    @FXML
    private void initialize() {
    }
    /**
     * Устанавливает сцену для этого окна.
     *
     * @param dialogStage
     */
    public void setDialogStage(Stage dialogStage) {
        this.dialogStage = dialogStage;
    }
    /**
     * Задаёт данные об адресате, информацию о котором будем
менять.
     *
     * @param person
     */
    public void setPerson(Person person) {
        this.person = person;
        firstNameField.setText(person.getFirstName());
        lastNameField.setText(person.getLastName());
        streetField.setText(person.getStreet());

postalCodeField.setText(Integer.toString(person.getPostalCode())
);
        cityField.setText(person.getCity());
```

```
birthdayField.setText(DateUtil.format(person.getBirthday()));
        birthdayField.setPromptText("dd.mm.yyyy");
    }
    /**
     * Returns true, если пользователь кликнул OK, в другом случае
false.
     *
     * @return
     */
    public boolean isOkClicked() {
        return okClicked;
    }
    /**
     * Вызывается, когда пользователь кликнул по кнопке OK.
     */
    @FXML
    private void handleOk() {
        if (isInputValid()) {
            person.setFirstName(firstNameField.getText());
            person.setLastName(lastNameField.getText());
            person.setStreet(streetField.getText());

person.setPostalCode(Integer.parseInt(postalCodeField.getText())
);

person.setBirthday(DateUtil.parse(birthdayField.getText()));
            okClicked = true;
            dialogStage.close();
        }
    }
    /**
     * Вызывается, когда пользователь кликнул по кнопке Cancel.
     */
    @FXML
    private void handleCancel() {
        dialogStage.close();
    }
    /**
     * Метод проверяет пользовательский ввод в текстовых полях.
     *
     * @return true, если пользовательский ввод корректен
     */
    private boolean isInputValid() {
        String errorMessage = "";
        if     (firstNameField.getText()     ==     null     ||
firstNameField.getText().length() == 0) {
            errorMessage += "Некорректно введено имя!\n";
        }
        if     (lastNameField.getText()     ==     null     ||
lastNameField.getText().length() == 0) {
            errorMessage += "Некорректно введена фамилия!\n";
        }
        if     (streetField.getText()     ==     null     ||
```

```
streetField.getText().length() == 0) {
            errorMessage += "Некорректно введена улица!\n";
        }
        if     (postalCodeField.getText()     ==     null     ||
postalCodeField.getText().length() == 0) {
            errorMessage += "Некорректно введен почтовый код!\n";
        } else {
            // try to parse the postal code into an int.
            try {
                Integer.parseInt(postalCodeField.getText());
            } catch (NumberFormatException e) {
                errorMessage  +=  "Некорректно  введен  почтовый
код(должен быть целочисленным)!\n";
            }
        }
        if      (cityField.getText()      ==      null      ||
cityField.getText().length() == 0) {
            errorMessage += "Некорректно введен город!\n";
        }
        if     (birthdayField.getText()     ==     null     ||
birthdayField.getText().length() == 0) {
            errorMessage += "Некорректно введен день рождения!\n";
        } else {
            if (!DateUtil.validDate(birthdayField.getText())) {
                errorMessage += "Неверный формат даты. Используйте
формат дд.мм.гггг!\n";
            }
        }
        if (errorMessage.length() == 0) {
            return true;
        } else {
            // Show the error message.
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.initOwner(dialogStage);
            alert.setTitle("Неверно заполнены поля");
            alert.setHeaderText("Введите   корректные   значения
полей");
            alert.setContentText(errorMessage);
            alert.showAndWait();
            return false;
        }
    }}
```

В этом контроллере для указания адресата, данные которого должны быть изменены, используется метод *setPerson(...)*. Когда пользователь нажимает на кнопку *OK*, то вызывается метод *handleOK()*. Первым делом данные, введённые пользователем, проверяются в методе *isInputValid()*. Если проверка прошла успешно, то объект адресата заполняется данными, которые ввёл пользователь. Эти изменения будут напрямую применяться к объекту адресата, который был передан в качестве аргумента метода *setPerson(...)*. Логическая переменная *okClicked* служит для определения того, какую из двух

кнопок, *OK* или *Cancel* нажал пользователь.

ПРИВЯЗКА КЛАССА-КОНТРОЛЛЕРА К FXML-ФАЙЛУ

Теперь мы должны связать созданный класс-контроллер с *fxml*-файлом. Для этого выполните следующие действия.

Откройте файл *PersonEditDialog.fxml* в *Scene Builder*. С левой стороны во вкладке *Controller* установите класс *PersonEditDialogController* в качестве значения параметра *Controller Class*. Установите соответствующие значения *fx:id* для всех компонентов *TextField*. В значениях параметров *onAction* для кнопок укажите соответствующие методы-обработчики.

ВЫЗОВ ДИАЛОГА РЕДАКТИРОВАНИЯ

Добавьте в класс *MainApp* метод для загрузки и отображения диалога редактирования записей:

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//Файл MainApp.java
/**
     * Открывает диалоговое окно для изменения деталей указанного адресата.
     * Если пользователь кликнул OK, то изменения сохраняются в предоставленном
     * объекте адресата и возвращается значение true.
     *
     * @param person - объект адресата, который надо изменить
     * @return true, если пользователь кликнул OK, в противном случае false.
     */
    public boolean showPersonEditDialog(Person person) {
        // Загружаем fxml-файл и создаём новую сцену
        // для всплывающего диалогового окна.
        try {
            FXMLLoader loader = new FXMLLoader();

loader.setLocation(MainApp.class.getResource("view/PersonEditDialog.fxml"));
            AnchorPane page = loader.load();
            // Создаём диалоговое окно Stage.
            Stage dialogStage = new Stage();
            dialogStage.setTitle("Редактирование");
            dialogStage.initModality(Modality.WINDOW_MODAL);
            dialogStage.initOwner(primaryStage);
            Scene scene = new Scene(page);
            dialogStage.setScene(scene);
            // Передаём адресата в контроллер.
            PersonEditDialogController controller = loader.getController();
            controller.setDialogStage(dialogStage);
            controller.setPerson(person);
            // Отображаем диалоговое окно и ждём, пока пользователь его не закроет
            dialogStage.showAndWait();
```

```
            return controller.isOkClicked();
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
```

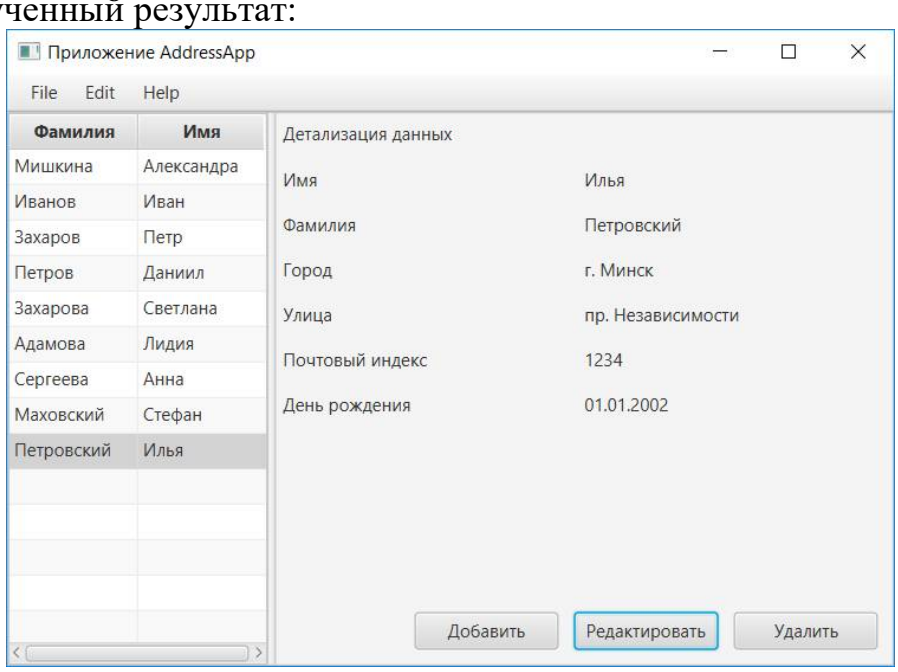
Добавьте следующие методы в класс *PersonOverviewController*. Когда пользователь будет нажимать на кнопки *«Добавить»* или «*Редактировать*», эти методы будут обращаться к методу *showPersonEditDialog(...)* в классе *MainApp*.

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//Файл PersonOverviewController.java
/**
     * Вызывается, когда пользователь кликает по кнопке "Добавить"
     * Открывает диалоговое окно с дополнительной информацией нового адресата
     */
    @FXML
    private void handleNewPerson() {
        Person person = new Person();
        boolean okClicked = mainApp.showPersonEditDialog(person);
        if (okClicked) mainApp.getPersonData().add(person);
    }
    /**
     * Вызывается, когда пользователь кликает по кнопка "Редактировать"
     * Открывает диалоговое окно для изменения выбранного адресата.
     */
    @FXML
    private void handleEditPersont() {
        Person selectedPerson = personTable.getSelectionModel().getSelectedItem();
        if (selectedPerson != null) {
            boolean okClicked = mainApp.showPersonEditDialog(selectedPerson);
            if (okClicked) showPersonDetails(selectedPerson);
        } else {
            // Ничего не выбрано.
            Alert alert = new Alert(Alert.AlertType.WARNING);
            alert.initOwner(mainApp.getPrimaryStage());
            alert.setTitle("Нет выбранной записи");
            alert.setHeaderText("Не выбрана запись");
            alert.setContentText("Выберите запись в таблице для редактирования");
            alert.showAndWait();
        }
    }
```

В приложении *Scene Builder* откройте представление

PersonOverview.fxml и для кнопок *«Добавить»* и «*Редактировать*» задайте соответствующие методы-обработчики в параметре *On Action*.

Полученный результат:



Сейчас у вас должно быть приложение адресной книги, которое позволяет добавлять, изменять и удалять адресатов. Это приложение так же осуществляет проверку данных, вводимых пользователем.

СТИЛИЗАЦИЯ С ПОМОЩЬЮ CSS

В *JavaFX* с помощью каскадных таблиц стилей (*CSS*) можно стилизовать интерфейс пользователя. В этом примере создадим тему *DarkTheme*, на базе Метро-дизайна из *Windows* 8. Принятые стили для кнопок заимствованы из статьи *JMetro - Windows* 8 *Metro controls on Java*, написанной *Pedro Duque Vieira*.

В *JavaFX* есть стиль, применяемый по умолчанию для всех приложений. Добавляя пользовательские стили, происходит переопределение базового стиля.

Добавьте файл *DarkTheme.css* в пакет *view*.

```
//Пример №11. Поэтапное создание приложения с использованием библиотеки JavaFX
//DarkTheme.css
.background {
    -fx-background-color: #1d1d1d;
}

.label {
    -fx-font-size: 11pt;
    -fx-font-family: "Segoe UI Semibold";
    -fx-text-fill: white;
    -fx-opacity: 0.6;
}
.label-bright {
    -fx-font-size: 11pt;
```

```
    -fx-font-family: "Segoe UI Semibold";
    -fx-text-fill: white;
    -fx-opacity: 1;
}
.label-header {
    -fx-font-size: 32pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-opacity: 1;
}
.table-view {
    -fx-base: #1d1d1d;
    -fx-control-inner-background: #1d1d1d;
    -fx-background-color: #1d1d1d;
    -fx-table-cell-border-color: transparent;
    -fx-table-header-border-color: transparent;
    -fx-padding: 5;
}
.table-view .column-header-background {
    -fx-background-color: transparent;
}
.table-view .column-header, .table-view .filler {
    -fx-size: 35;
    -fx-border-width: 0 0 1 0;
    -fx-background-color: transparent;
    -fx-border-color:
        transparent
        transparent
        derive(-fx-base, 80%)
        transparent;
    -fx-border-insets: 0 10 1 0;
}
.table-view .column-header .label {
    -fx-font-size: 20pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-alignment: center-left;
    -fx-opacity: 1;
}
.table-view:focused .table-row-cell:filled:focused:selected {
    -fx-background-color: -fx-focus-color;
}
.split-pane:horizontal > .split-pane-divider {
    -fx-border-color: transparent #1d1d1d transparent #1d1d1d;
    -fx-background-color: transparent, derive(#1d1d1d,20%);
}
.split-pane {
    -fx-padding: 1 0 0 0;
}
.menu-bar {
    -fx-background-color: derive(#1d1d1d,20%);
}
.context-menu {
```

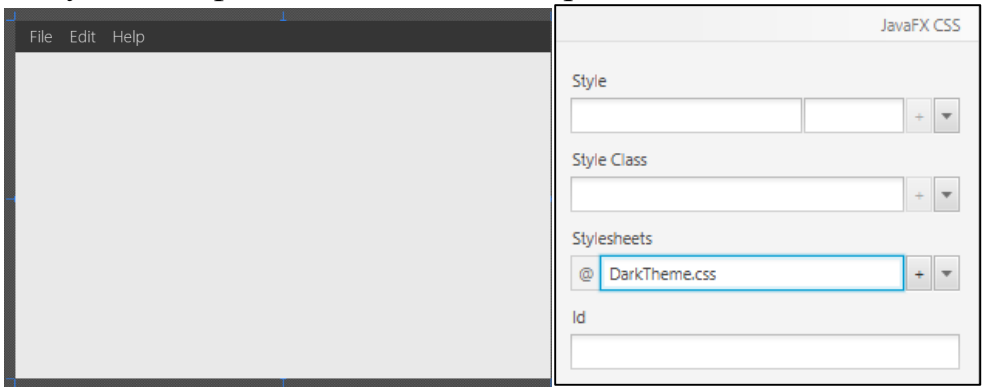
```
    -fx-background-color: derive(#1d1d1d,50%);
}

.menu-bar .label {
    -fx-font-size: 14pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-opacity: 0.9;
}
.menu .left-container {
    -fx-background-color: black;
}
.text-field {
    -fx-font-size: 12pt;
    -fx-font-family: "Segoe UI Semibold";
}
/*
 * Push Button в стиле Metro
 * Автор: Pedro Duque Vieira
 *    http://pixelduke.wordpress.com/2012/10/23/jmetro-windows-8-
controls-on-java/
 */
.button {
    -fx-padding: 5 22 5 22;
    -fx-border-color: #e2e2e2;
    -fx-border-width: 2;
    -fx-background-radius: 0;
    -fx-background-color: #1d1d1d;
    -fx-font-family: "Segoe UI", Helvetica, Arial, sans-serif;
    -fx-font-size: 11pt;
    -fx-text-fill: #d8d8d8;
    -fx-background-insets: 0 0 0 0, 0, 1, 2;
}
.button:hover {
    -fx-background-color: #3a3a3a;
}
.button:pressed, .button:default:hover:pressed {
  -fx-background-color: white;
  -fx-text-fill: #1d1d1d;
}
.button:focused {
    -fx-border-color: white, white;
    -fx-border-width: 1, 1;
    -fx-border-style: solid, segments(1, 1);
    -fx-border-radius: 0, 0;
    -fx-border-insets: 1 1 1 1, 0;
}
.button:disabled, .button:default:disabled {
    -fx-opacity: 0.4;
    -fx-background-color: #1d1d1d;
    -fx-text-fill: white;
}
.button:default {
```

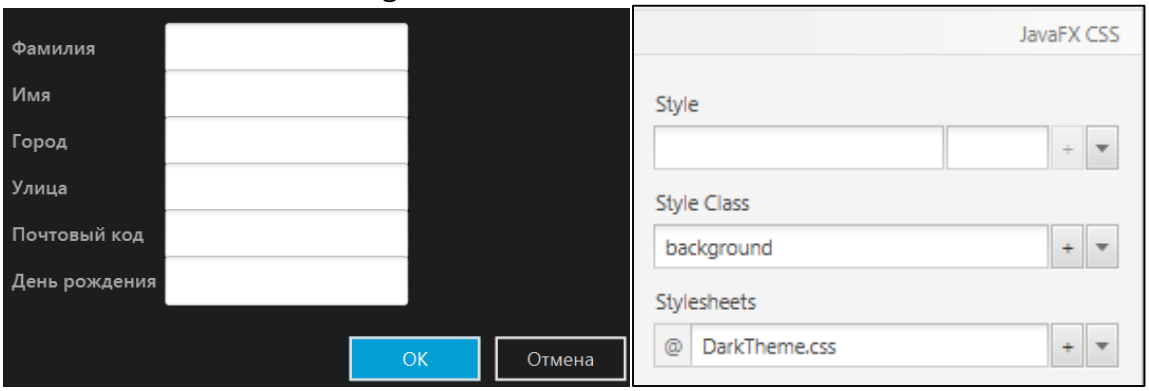
```
    -fx-background-color: -fx-focus-color;
    -fx-text-fill: #ffffff;
}
.button:default:hover {
    -fx-background-color: derive(-fx-focus-color,30%);
}
```

Теперь надо подключить эти стили к сцене. Мы можем сделать это программно в коде *Java*, а так же можно добавить стили в *fxml*-файлы, используя *Scene Builder*.

В приложении *Scene Builder* откройте файл *RootLayout.fxml*. Во вкладке *Hierarchy* выберите корневой контейнер *BorderPane*, перейдите на вкладку *Properties* и укажите файл *DarkTheme.css* в роли таблиц стилей.



В приложении *Scene Builder* откройте файл *PersonEditDialog.fxml*. Во вкладке *Hierarchy* выберите корневой контейнер *AnchorPane*, перейдите на вкладку *Properties* и укажите файл *DarkTheme.css* в роли таблиц стилей. Фон всё ещё белый, поэтому укажите для корневого компонента *AnchorPane* в классе стиля значение *background*.

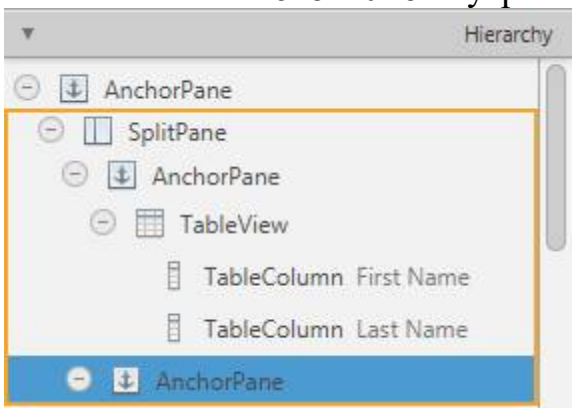


Выберите кнопку *OK* и отметьте свойство *Default Button* в вкладке *Properties*. В результате изменится цвет кнопки, и она будет использоваться по умолчанию, когда пользователь, находясь в окне, будет нажимать клавишу *Enter*.

В приложении *Scene Builder* откройте файл *PersonOverview.fxml*. Во вкладке *Hierarchy* выберите корневой контейнер *AnchorPane*, перейдите на вкладку *Properties* и укажите файл *DarkTheme.css* в роли таблиц стилей.

Вы сразу должны увидеть некоторые изменения: цвет таблицы и кнопок поменялся на чёрный. С того момента, как мы переопределили некоторые из стилей в *css*-файле, новые стили применяются автоматически. Возможно, потребуется изменить размер кнопок для того, чтобы отображался весь текст.

Выберите правый компонент *AnchorPane* внутри компонента *SplitPane*.



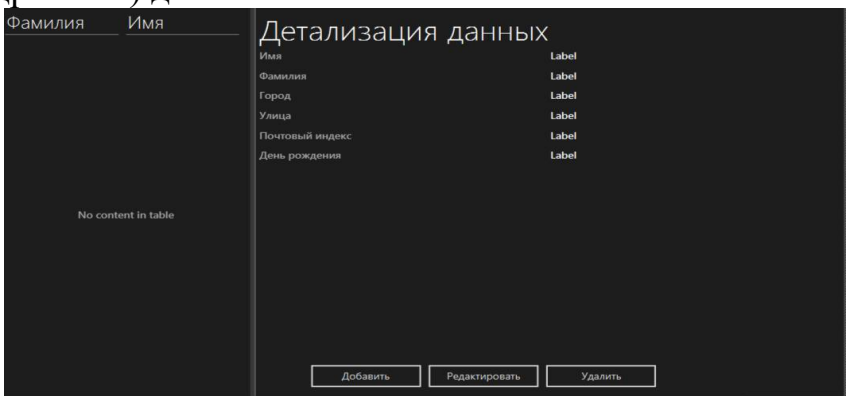
Перейдите на вкладку *Properties* и укажите в классе стиля значение *background*. Теперь фон станет чёрного цвета.

ТЕКСТОВЫЕ МЕТКИ С ДРУГИМИ СТИЛЯМИ

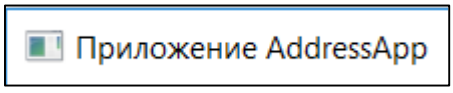
Сейчас все текстовые метки с правой стороны имеют одинаковый размер. Для дальнейшей стилизации текстовых меток мы будем использовать уже определённые стили *.label-header* и *label-bright.*

Выберите метку *«Подробные данные»* и добавьте в качестве класса стиля значение *label-header*.

Для каждой метки в правой колонке (где отображаются фактические данные об адресатах) добавьте в качестве класса стиля значение *label-bright.*



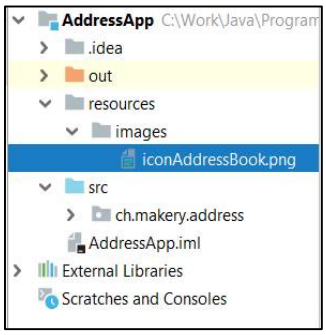
На данный момент в приложении в панели названия и панели задач используется иконка по умолчанию:



Можно указать пользовательскую иконку для приложения.

Одно из возможных мест, где можно свободно скачать иконки — это сайт *Icon Finder*.

Создайте внутри проекта *AddressApp* обычную папку *resources*, а в ней папку *images*. Поместите выбранную иконку в папку изображений. Структура папок должна иметь такой вид:



Для того, чтобы для сцены установить новую иконку, в классе *MainApp.java* добавьте следующий код в метод *start()*

```
//Файл MainApp.java
this.primaryStage.getIcons().add(new
Image("file:resources/images/user-256.png"));
```

Весь метод *start()* теперь будет выглядеть так:

```
@Override
    public void start(Stage stage) {
        this.primaryStage = stage;
        this.primaryStage.setTitle("Приложение AddressApp");
        // Устанавливаем иконку приложения
        this.primaryStage.getIcons().add(new
Image("file:resources/images/user-256.png"));
        initRootLayout();
        showPersonOverview();
    }
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Для чего предназначены библиотеки *Swing* и *JavaFX*? Какова между ними разница?
2. Что собой представляют легковесные и тяжеловесные компоненты пользовательского интерфейса? Приведите пример компонента пользовательского интерфейс.
3. Что такое событие пользовательского интерфейса? Что такое обработчик события пользовательского интерфейса? Приведите пример события и его обработчика.
4. Что такое менеджер компоновки (layout manager)? Каковы его задачи? Приведите примеры менеджеров компоновки.
5. Что такое контейнер и компонент пользовательского интерфейса? В чем между ними разница? Приведите примеры контейнера и компонента.
6. Что такое предпочтительный размер для компонента пользовательского интерфейса?
7. Какой метод используется для указания менеджера компоновки?
8. Что такое «поток диспетчеризации событий»?
9. Опишите жизненный цикли приложения JavaFX.
10. Какой класс находится в вершине иерархии событий библиотеки Swing?

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.
2. Ответить на контрольные вопросы лабораторной работы.
3. Разработать алгоритм программы по индивидуальному заданию.
4. Написать, отладить и проверить корректность работы созданной программы.
5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. диаграмму созданных классов в нотации UML
8. выводы по лабораторной работе

В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО РАЗРАБОТАТЬ ДВА GUI-ПРИЛОЖЕНИЯ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕК SWING И JAVAFX. АВТОМАТИЗИРОВАТЬ GRUD ОПЕРАЦИИ ПО ПРЕДМЕТНОЙ ОБЛАСТИ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ. ДАННЫЕ ПРИЛОЖЕНИЯ ДОЛЖНЫ СОХРАНЯТЬСЯ В ФАЙЛ.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:

№ варианта	Предметная область
1.	студенты ВУЗа
2.	компьютерные игры
3.	каталог книг
4.	тестирование знаний студентов
5.	транспортная техника
6.	web-приложения
7.	медицинские работники
8.	каталог книг
9.	банковские операции
10.	строительная техника
11.	мобильные приложения
12.	банковские сотрудники
13.	печатная продукция
14.	бронирование авиабилетов
15.	продажа и покупка недвижимости

16.	компьютерные сети
17.	аналитика данных
18.	ассортимент услуг
19.	бронирование авиабилетов
20.	бытовая техника
21.	участники спортивных мероприятий
22.	тестирование приложений
23.	издательство печатной продукции
24.	документооборот
25.	сотрудники ИТ-организации
26.	розничная продажа товаров и услуг
27.	тестирование по английскому языку
28.	работники строительной организации
29.	сотрудники библиотеки
30.	сельскохозяйственная продукция