第07讲 Scene Builder与JavaFX程序设计

主要内容

- [x] 使用 Scene Builder 来设计用户界面
- [x] 使用 模型 - 视图 - 控制器（MVC）模式 构造基础的应用
- [x] Model 和 TableView
- [x] 与用户的交互
- [x] CSS 样式
- [x] 将数据用 XML 格式存储
- [x] 统计图
- [x] 部署

重点内容

- 理解J av a网络编程，通过S oc k e t 、UR L类处理。
-

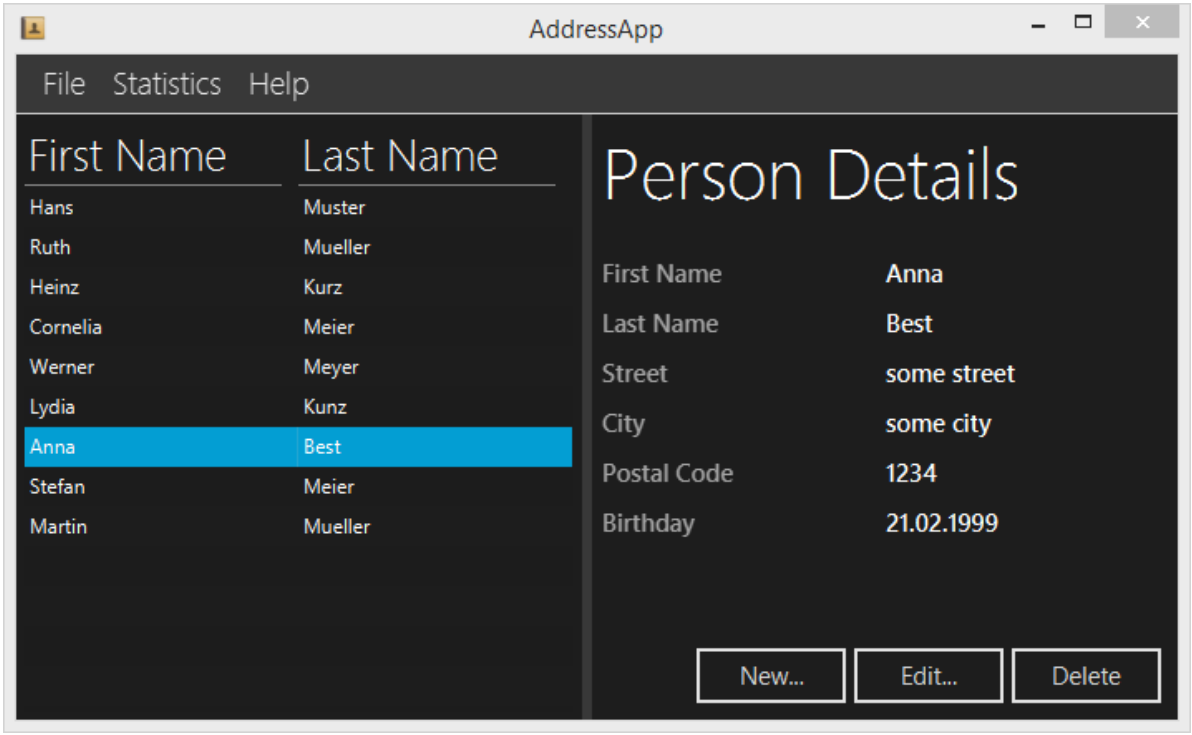
难点分析

- J av a的S oc k e t 类、UR L类以及Mai l 类的使用。
-

1.1 地址簿应用

2012一个非常详细的 JavaFX 2 系列教程。世界各地的人们都已阅读了这个教程并给了非常积极的反馈。所以我决定 **为 JavaFX 8 改写 JavaFX 2 的教程** (阅读关于 JavaFX 8 的变化 Update to JavaFX 8 - What's New).

这个教程将指导您设计，编写并部署一个联系人应用程序。应用程序最后将会是这个样子：

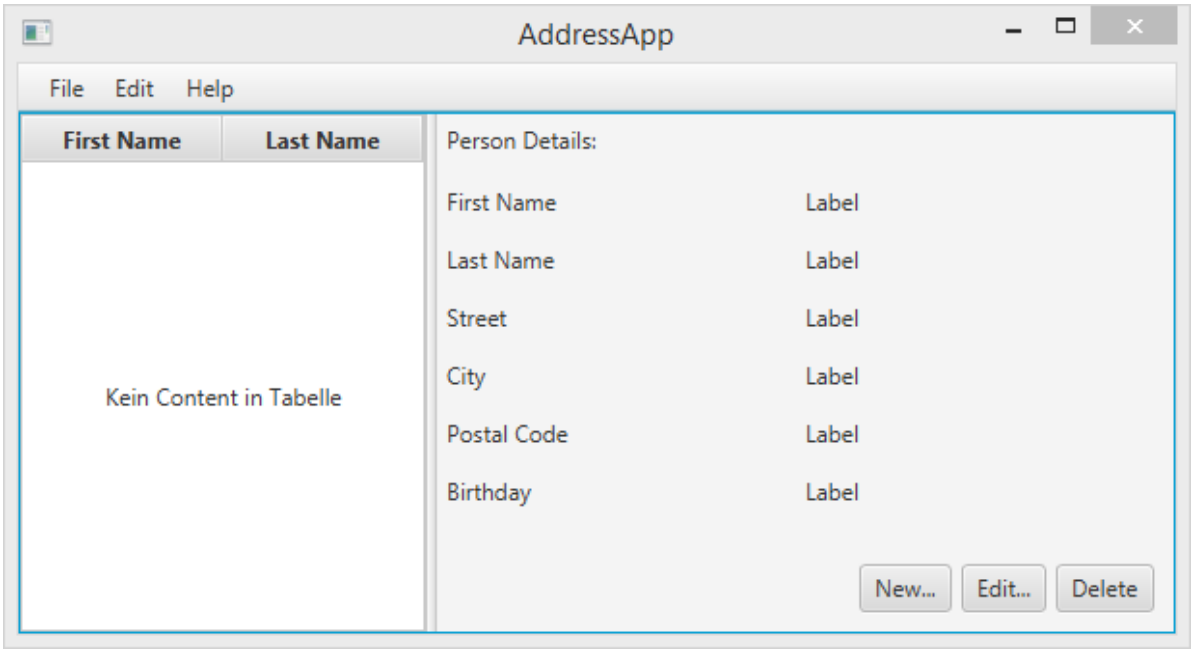


你将学到什么?

- 创建并启动一个 JavaFX 项目。
- 使用 Scene Builder 设计 UI 。
- 构造一个 模型 - 视图 - 控制器 (MVC) 模式的应用程序。
- 使用 `ObservableLists` 来自动更新用户界面。
- 使用 `TableView` 来响应列表中的选择。
- 创建一个 edit persons 的自定义弹出式对话框。
- 验证用户输入。
- 使用 CSS 样式化一个 JavaFX 应用程序。
- 使用 XML 保存数据。
- 在用户配置中保存最后一次打开文件的路径。
- 创建 JavaFX 的统计图表。
- 部署一个 JavaFX 到本机软件包。

这是相当多的! 所以，当你学习完这个教程后，你应该准备好使用 JavaFX 构建复杂的应用程序。

第一部分：Scene Builder



第一部分的主题

- 开始了解 JavaFX 。
- 创建并运行一个 JavaFX 项目。
- 使用 Scene Builder 来设计用户界面。
- 使用 模型 - 视图 - 控制器（MVC）模式 构造基础的应用。

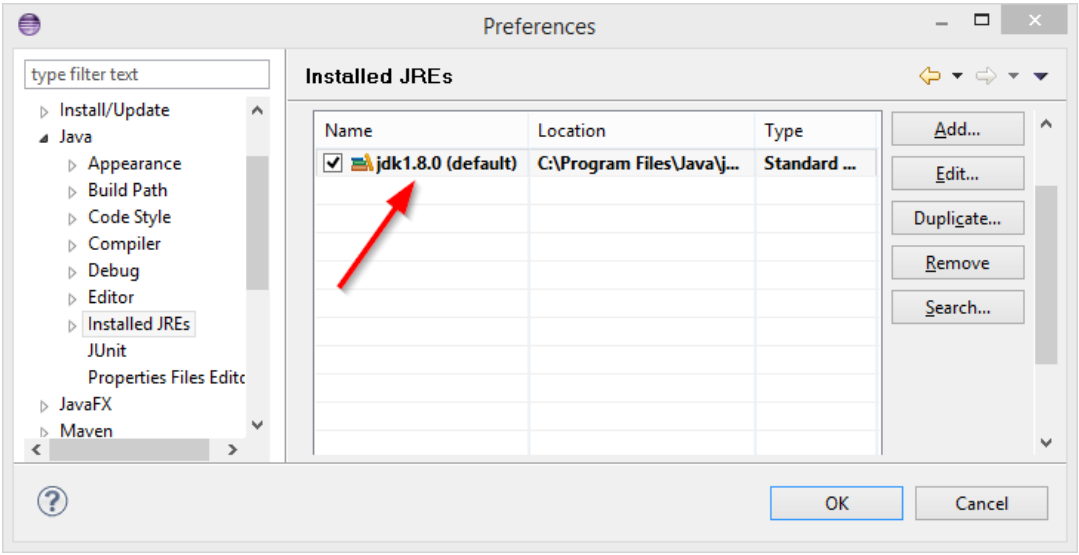
你需要准备

- 最新的 Java JDK 8 （包含 **JavaFX 8**）。
- Eclipse 4.3 或更高版本与 e(fx)clipse 插件。最简单的方法是从 e(fx)clipse 网站 下载预先配置的发行版本。作为一种备选你可以使用一个 update site 来给您的 Eclipse 安装。
- Scene Builder 2.0 或更高。

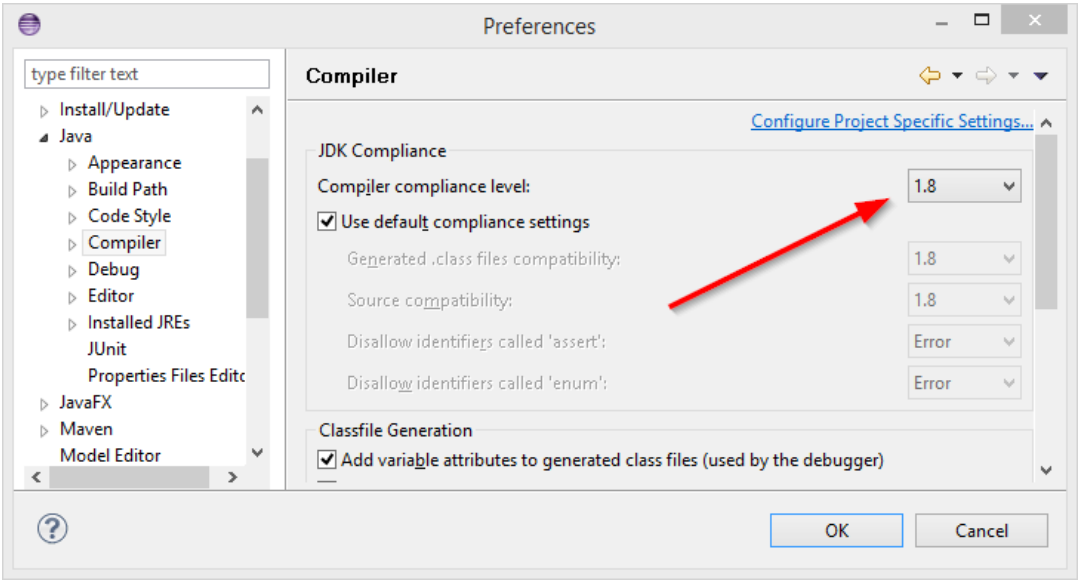
Eclipse 配置

配置Eclipse 所使用 JDK 和 Scene Builder：

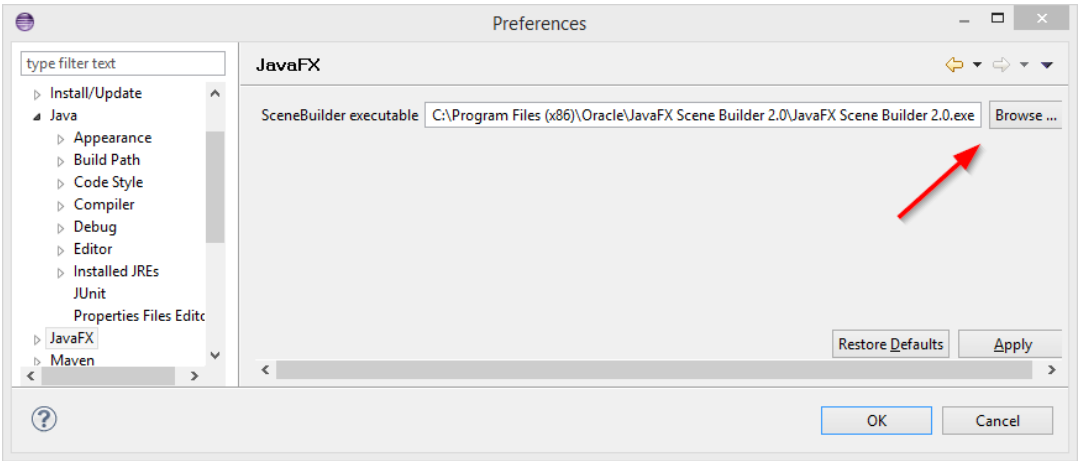
1. 打开 Eclipse 的设置并找到 *Java* | *Installed JREs* 。
2. 点击 *Add...*, 选择 *Standard VM* 并选择你安装 JDK 8 的 *Directory* 。
3. 移除其他的 JREs 或 JDKs 从而使 **JDK 8 成为默认。**



4. 在 *Java* | *Compiler* 中设置 **Compiler compliance level 到 1.8。**



5. 在 *JavaFX* 中指定你的 Scene Builder 可执行文件的路径。



帮助链接

你可能会想收藏下面的链接：

- Java 8 API - Java 标准类的文档。
- JavaFX 8 API - JavaFX 类的文档。
- ControlsFX API - ControlsFX project 额外 JavaFX 控件的文档。
- Oracle's JavaFX Tutorials - Oracle 的 JavaFX 官方教程。

一切就绪，让我们开始吧！

创建一个新的 JavaFX 项目

在 Eclipse（已安装 e(fx)clipse 的）中，点击 *File* | *New* | *Other...* 并选择 *JavaFX Project*。 指定这个项目的名字（e.g. *AddressApp*）并点击 *Finish*。

如果 *application* 包被自动创建，那么删除它和它的内容。

创建包

Model-View-Controller (MVC)是一个非常重要的软件设计原则。按照MVC模式可以将我们的应用程序划分成3个部分，然后为这每一部分建立自己的包 (在源代码文件夹上右键， 选择 新建 | 包):

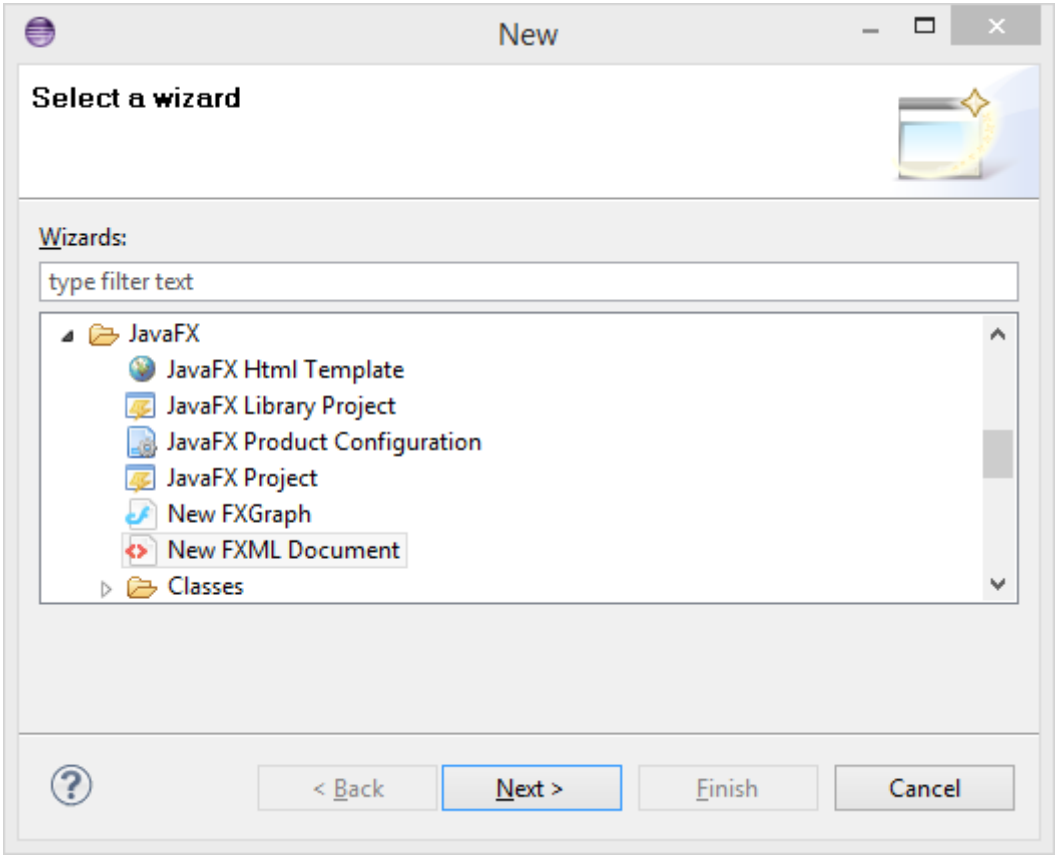
- `ch.makery.address` - 放置所有的控制器类(也就是应用程序的业务逻辑)
- `ch.makery.address.model` - 放置所有的模型类
- `ch.makery.address.view` - 放置所有界面和控件类

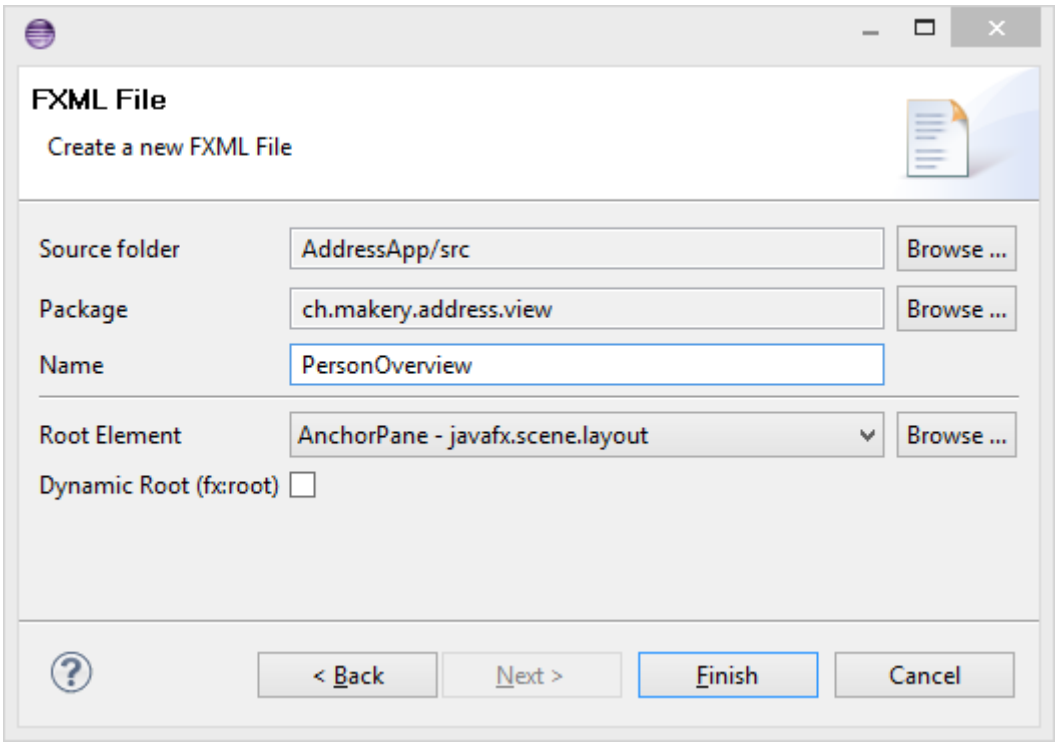
注意: view包里可能会包含一些控制器类，它可以直接被单个的view引用，我们叫它 **视图-控制器。**

创建FXML布局文件

有两种方式来创建用户界面，一种是通过XML文件来定义，另外一种则是直接通过java代码来创建. 这两种方式你都可以在网上搜到. 我们这里将使用XML的方式来创建大部分的界面。因为这种方式将会更好的将你的业务逻辑和你的界面开来，以保持代码的简洁。在接下来的内容里，我们将会介绍使用Scene Builder(所见即所得)来编辑我们的XML布局文件，它可以避免我们直接去修改XML文件。

在view包上右键创建一个新*FXML Document*，把它命名为`PersonOverview`。



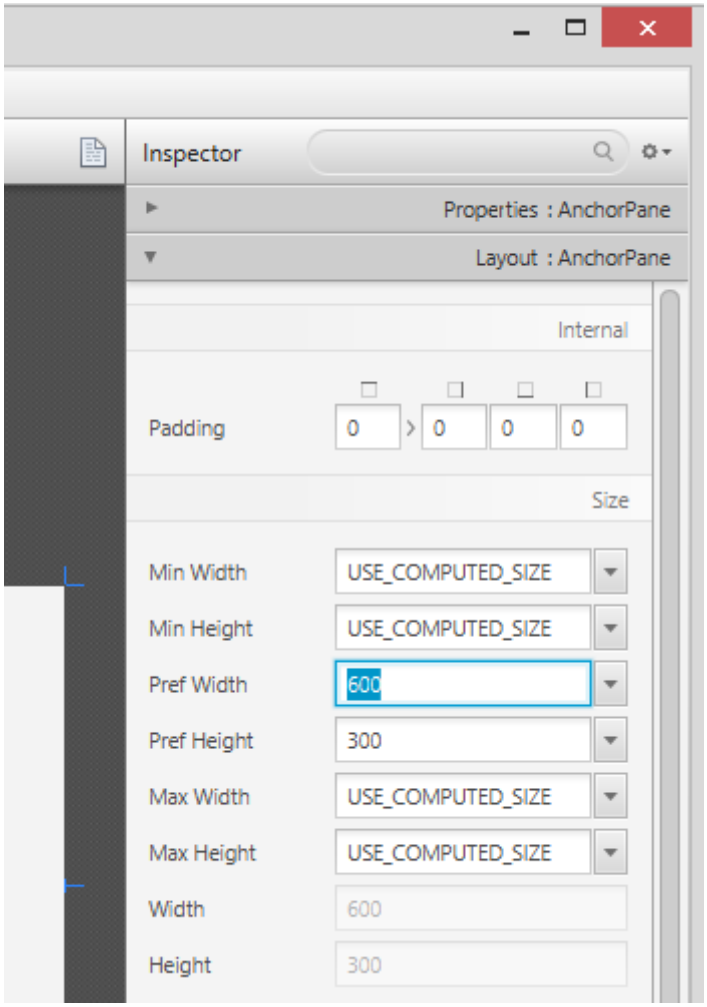


用Scene Builder来设计你的界面

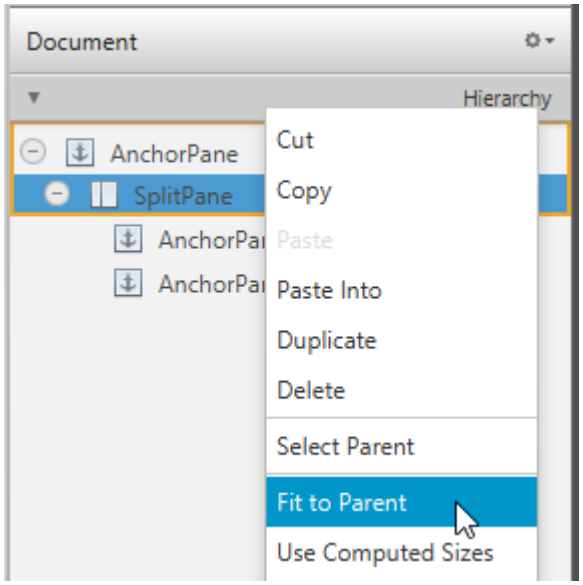
注意: 你可以下载这部分教程的源码，它里面已经包含了设计好的布局文件。

在`PersonOverview.fxml`右键选择 *Open with Scene Builder*，那么你将会在打开的Scene Builder里面看到一个固定的界面设计区域(在整个界面的左边)。

1. 选中这个界面设计区域，你就可以在右边的属性设置栏中对它的尺寸进行修改:

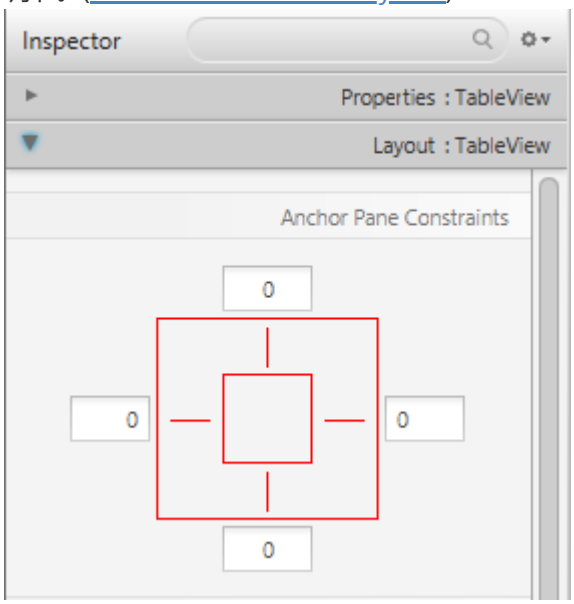


2. 从Scene Builder的左边控件栏中拖拽一个 *Splite Pane(Horizontal Flow)* 到界面设计区域，在Builder的右边视图结构中选择刚添加的Pane，在弹出的右键菜单中选择 *Fit to Parent* 。

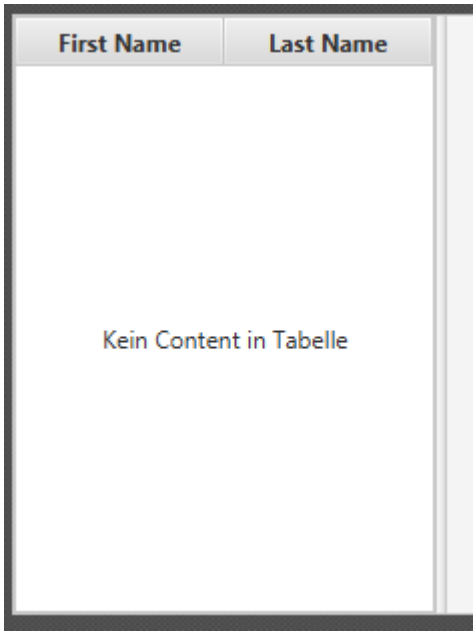


3. 同样从左边的控件栏中拖拽一个 *TableView* 到 *SplitPane* 的左边，选择这个TableView(而不是它的列)对它的布局进行设置，你可以在 *AnchorPane* 中对这个TableView四个边的外边距进行

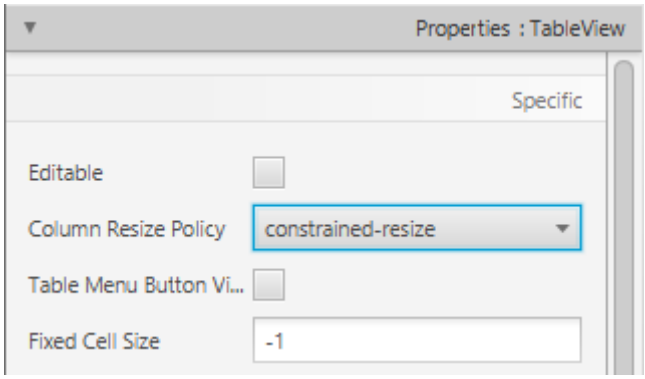
调节。(more information on Layouts).



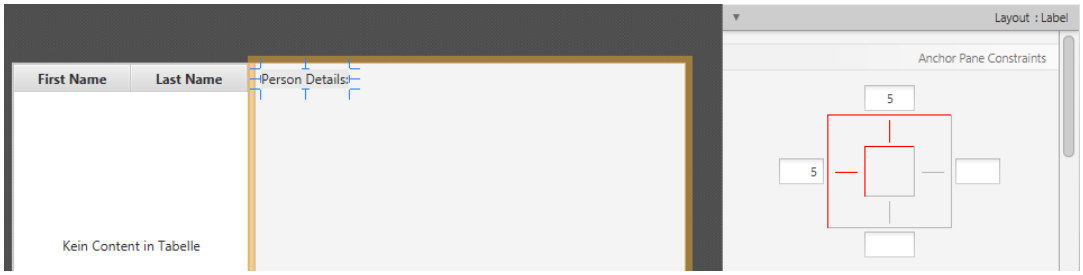
4. 点击菜单中的 *Preview | Show Preview in Window* 可以预览你设计好的界面，试着缩放预览的界面，你会发现TableView会随着窗口的缩放而变化。
5. 修改TableView中的列名字，“First Name” and “Last Name”，在右边面板中的属性设置项



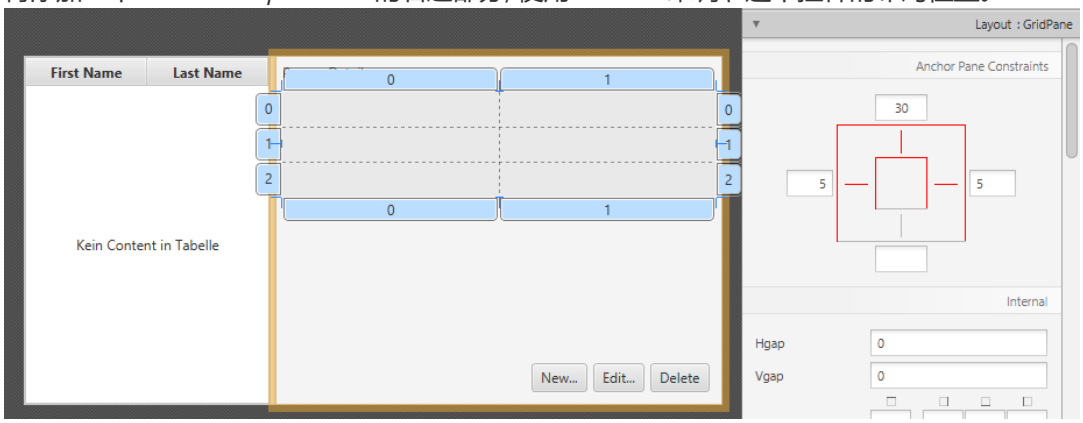
6. 选择这个 *TableView* ，在右边面板中将它的 *Column Resize Policy* 修改成 *constrained-resize* (同样是在属性设置项里面)。确保这个TableView的列能够铺满所有的可用空间。



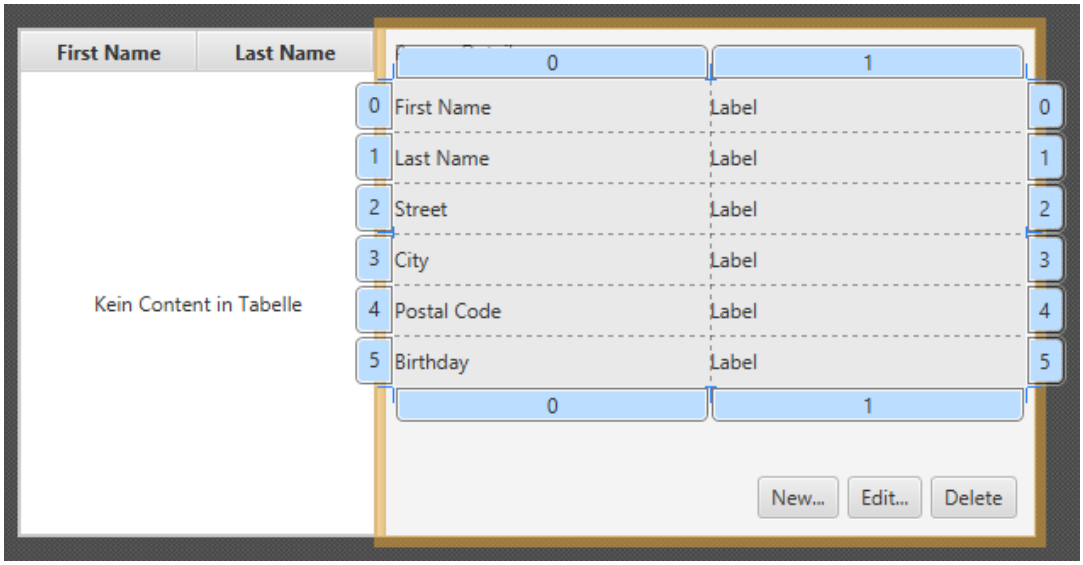
7. 添加一个 *Label* 到 *SplitePane*的右边部分，并设置它的显示文字为 “Person Details” (提示: 你可以通过搜索来找到 *Label* 这个控件)。 使用anchors来调节这个控件的布局位置。



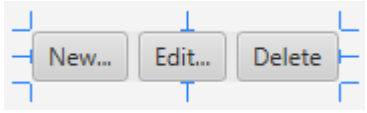
8. 再添加一个 *GridPane SplitePane*的右边部分, 使用anchors来调节这个控件的布局位置。



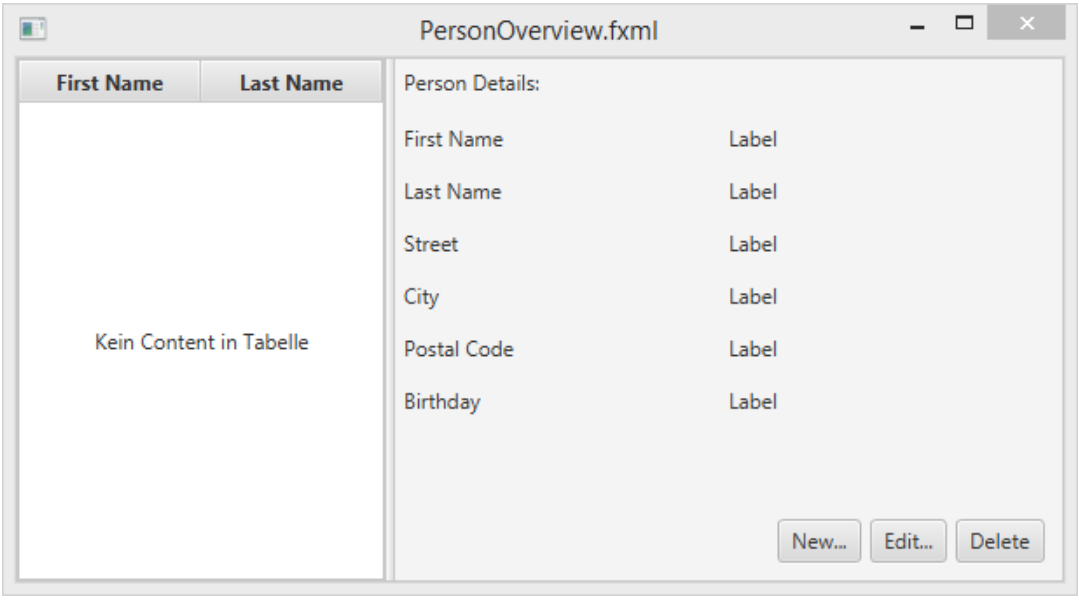
9. 按照下面的图添加多个 *Lables*到表格中去。
10. *注意: 添加一个控件到已经存在的行里面去，你可在这行的行号上右键选择"Add Row"。*



11. 添加3个按钮到这个 *GridPane* 的下面。 小提示: 选择这3个按钮，右键 *Wrap In | HBox*，那么它们会被放置到一个HBox里面。 你可能需要对这个HBox指定一个 *spacing*,同时也需要设置它们的右边和下边的anchors。



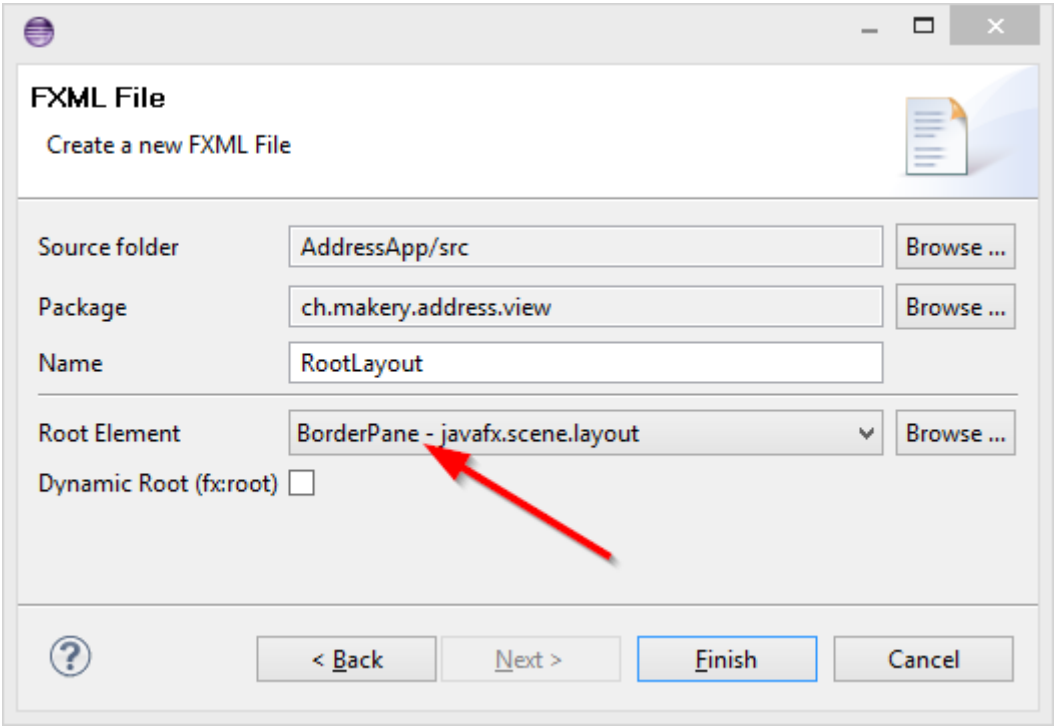
12. 那么基本已经完成了界面的设计，你可以通过 *Preview* 来预览一下你设计的界面，同时缩放一下窗口来检验一下各个控件的位置是否正确。



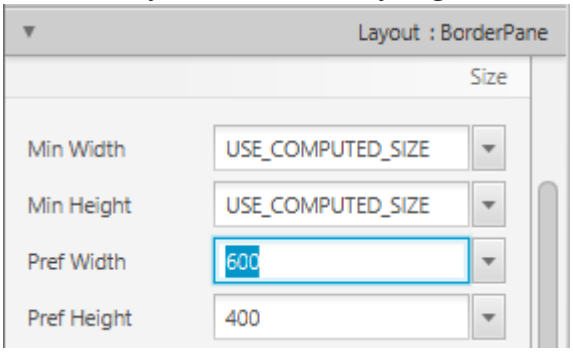
创建主应用程序

我们还需要新建一个*FXML*文件来做为主布局文件，它将包含菜单栏并存放我们之前创建的布局文件 `PersonOverview.fxml` 。

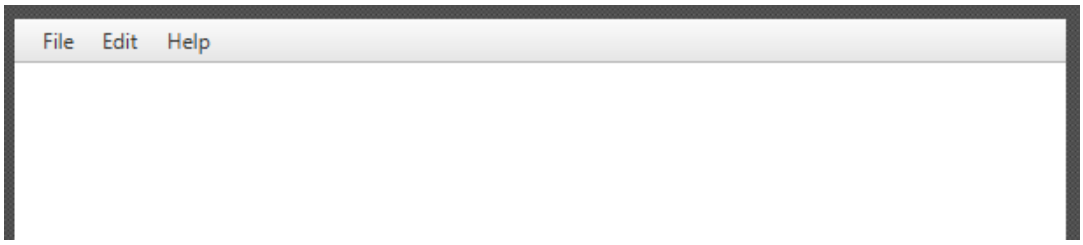
1. 在view包里面创建一个新的 *FXML Document* 叫做 `RootLayout.fxml`, 这一次，选择 *BorderPane* 做为它的根节点



2. 在Scene Builder中打开 `RootLayout.fxml` 。
3. 通过设置 *Pref Width* 为600和 *Pref Height* 为400来改变这个 *BorderPane*的尺寸。



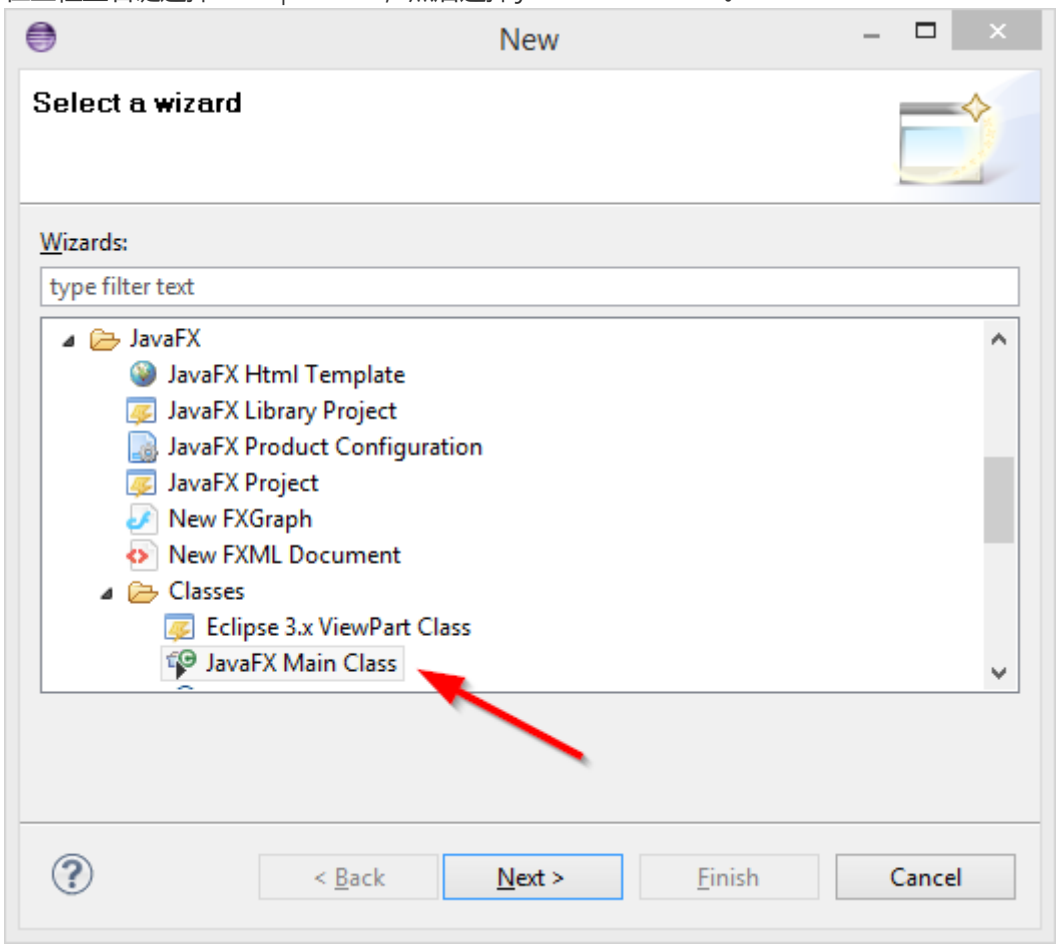
4. 在最顶上添加一个 *MenuBar*，先不去给这个菜单添加任何的功能。



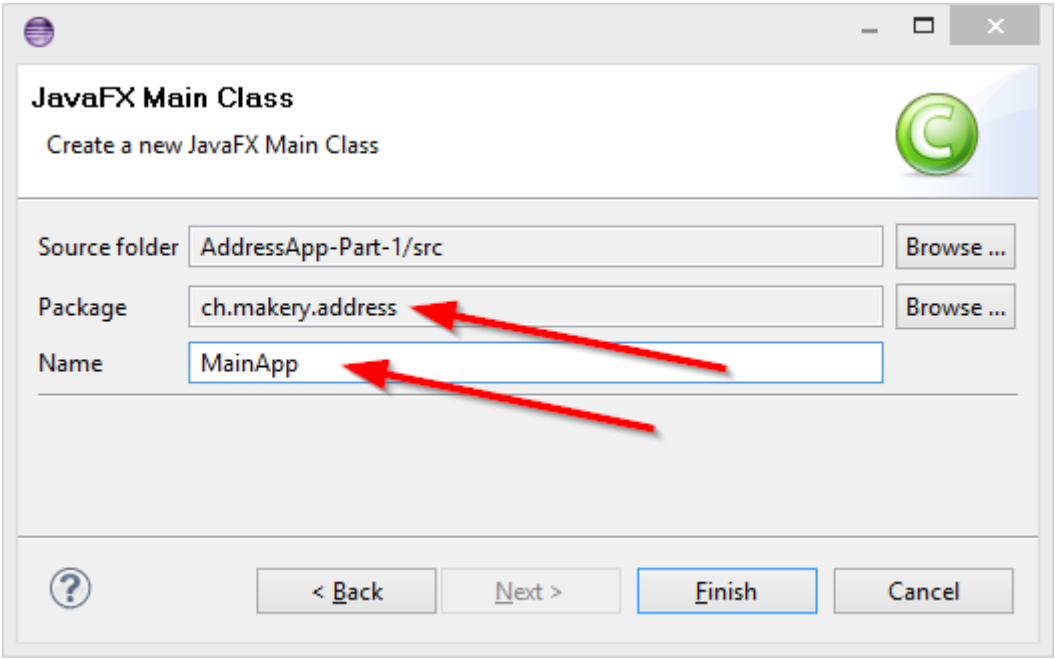
The JavaFX Main Class

现在，我们需要创建一个 **main java class** 用来加载 `RootLayout.fxml` ，同时添加 `PersonOverview.fxml` 到*RootLayout.fxml*中去，这个main class将做为我们这个应用程序的入口。

1. 在工程上右键选择 *New* | *Other...*，然后选择 *JavaFX Main Class*。



2. 将这个class命名为 `MainApp`，将它放置到controller包中，也就是上面建的 `ch.makery.address` (注意: 这个包下有两个子包，分别是`view` 和 `model`)。



你可能注意到了IDE生成的 `MainApp.java` 继承自 `Application` 同时包含了两个方法， 这是一个JavaFX应用程序的最基本的代码结构，这里最重要的方法是 `start(Stage primaryStage)` ，它将会在应用程序运行时通过内部的 `main` 方法自动调用。

正如你所看到的，这个 `start(...)` 方法会接收一个 `Stage` 类型的参数，下面的图向你展示了一个JavaFX应用程序的基本结构。

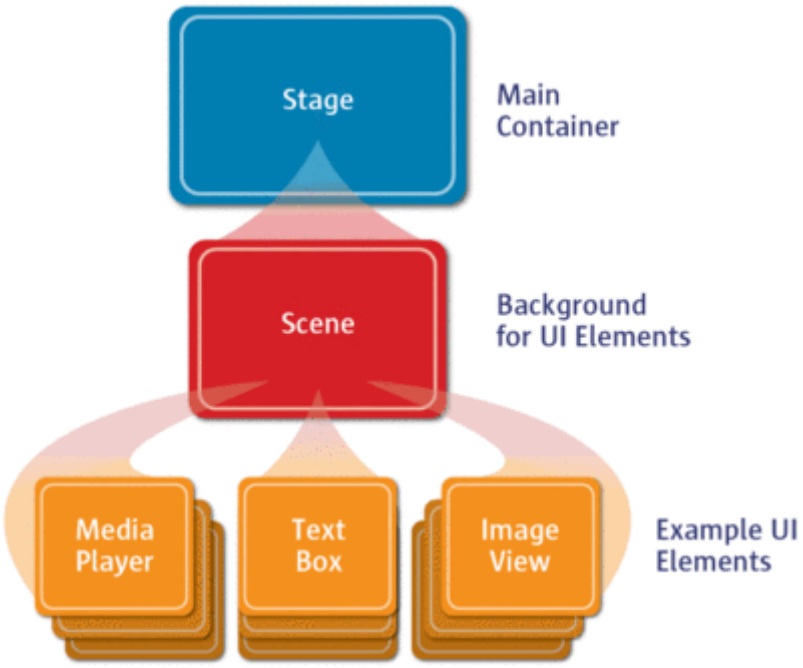


Image Source: http://www.oracle.com

一切看起来象是剧场里表演: 这里的 `Stage` 是一个主容器，它就是我们通常所认为的窗口（有边，高和宽，还有关闭按钮）。在这个 `Stage` 里面，你可以放置一个 `Scene`，当然你可以切换别的 `Scene`，而在这个 `Scene` 里面，我们就可以放置各种各样的控件。

更详细的信息，你可以参考 Working with the JavaFX Scene Graph.

打开 `MainApp.java`，将已有的代码替换成下面的代码:

```
package ch.makery.address;
```

```
import java.io.IOException;

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;

public class MainApp extends Application {

    private Stage primaryStage;
    private BorderPane rootLayout;

    @Override
    public void start(Stage primaryStage) {
        this.primaryStage = primaryStage;
        this.primaryStage.setTitle("AddressApp");

        initRootLayout();

        showPersonOverview();
    }

    /**
     * Initializes the root layout.
     */
    public void initRootLayout() {
        try {
            // Load root layout from fxml file.
            FXMLLoader loader = new FXMLLoader();

 loader.setLocation(MainApp.class.getResource("view/RootLayout.fxml"));
            rootLayout = (BorderPane) loader.load();

            // Show the scene containing the root layout.
            Scene scene = new Scene(rootLayout);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Shows the person overview inside the root layout.
     */
    public void showPersonOverview() {
        try {
            // Load person overview.
            FXMLLoader loader = new FXMLLoader();

 loader.setLocation(MainApp.class.getResource("view/PersonOverview.fxml"));
            AnchorPane personOverview = (AnchorPane) loader.load();

            // Set person overview into the center of root layout.
            rootLayout.setCenter(personOverview);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    /**
     * Returns the main stage.
     * @return
     */
    public Stage getPrimaryStage() {
        return primaryStage;
    }

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

代码中的注释会给你一些小提示，注明代码的含义。

如果你现在就运行这个程序，那么你将会看到和这篇文章开头所展示的图片那样的界面。

你有可能遇见的问题

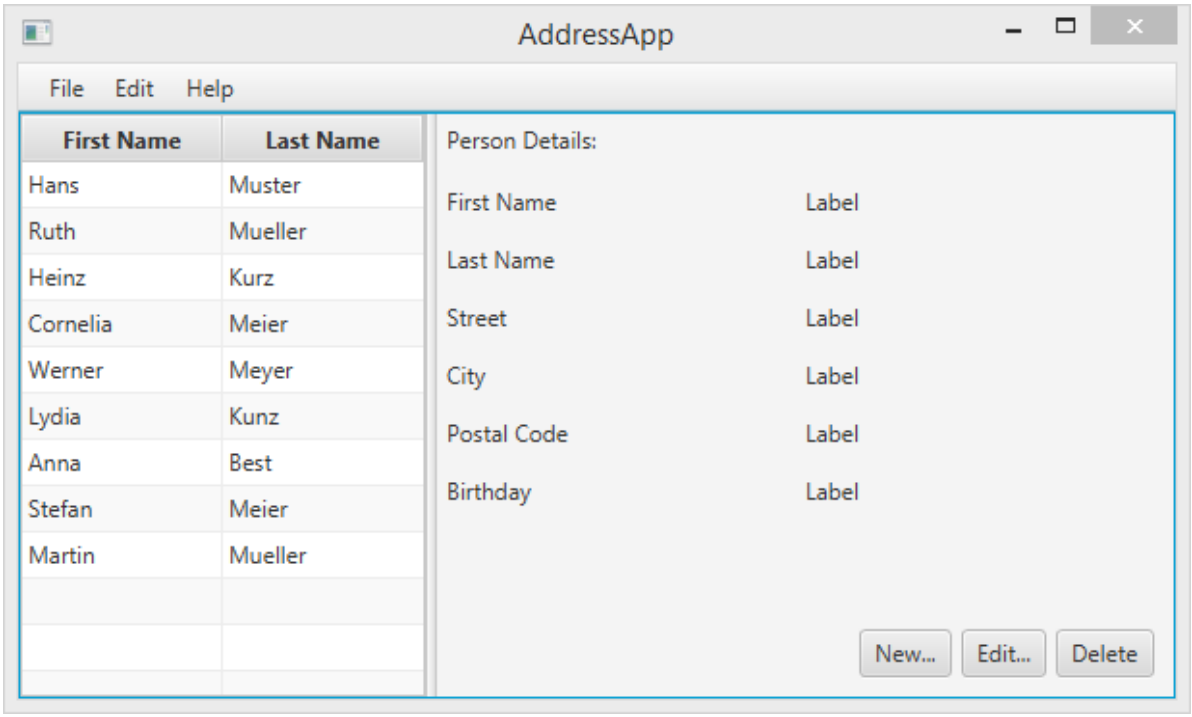
如果你的应用程序找不到你所指定的 `fxml` 布局文件，那么系统会提示以下的错误：

```
java.lang.IllegalStateException: Location is not set.
```

你可以检查一下你的 `fxml` 文件名是否拼写错误

如果还是不能工作，请下载这篇教程所对应的源代码，然后将源代码中的fxml文件替换掉你的

第二部分：Model 和 TableView



第二部分的主题

- 创建一个 **模型** 类。

- 在 **ObservableList** 使用模型类。
- 使用 **Controllers** 在 **TableView** 上显示数据。

创建 模型 类。

我们需要一个模型类来保存联系人信息到我们的通讯录中。在模型包中（`ch.makery.address.model`）添加一个叫 `Person` 的类。`Person` 类将会有一些变量，名字，地址和生日。将以下代码添加到类。在代码后，我将解释一些 JavaFX 的细节。

Person.java

```
package ch.makery.address.model;

import java.time.LocalDate;

import javafx.beans.property.IntegerProperty;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

/**
 * Model class for a Person.
 *
 * @author Marco Jakob
 */
public class Person {

    private final StringProperty firstName;
    private final StringProperty lastName;
    private final StringProperty street;
    private final IntegerProperty postalCode;
    private final StringProperty city;
    private final ObjectProperty<LocalDate> birthday;

    /**
     * Default constructor.
     */
    public Person() {
        this(null, null);
    }

    /**
     * Constructor with some initial data.
     *
     * @param firstName
     * @param lastName
     */
    public Person(String firstName, String lastName) {
        this.firstName = new SimpleStringProperty(firstName);
        this.lastName = new SimpleStringProperty(lastName);

        // Some initial dummy data, just for convenient testing.
        this.street = new SimpleStringProperty("some street");
        this.postalCode = new SimpleIntegerProperty(1234);
```

```
        this.city = new SimpleStringProperty("some city");
        this.birthday = new SimpleObjectProperty<LocalDate>(LocalDate.of(1999,
2, 21));
    }

    public String getFirstName() {
        return firstName.get();
    }

    public void setFirstName(String firstName) {
        this.firstName.set(firstName);
    }

    public StringProperty firstNameProperty() {
        return firstName;
    }

    public String getLastName() {
        return lastName.get();
    }

    public void setLastName(String lastName) {
        this.lastName.set(lastName);
    }

    public StringProperty lastNameProperty() {
        return lastName;
    }

    public String getStreet() {
        return street.get();
    }

    public void setStreet(String street) {
        this.street.set(street);
    }

    public StringProperty streetProperty() {
        return street;
    }

    public int getPostalCode() {
        return postalCode.get();
    }

    public void setPostalCode(int postalCode) {
        this.postalCode.set(postalCode);
    }

    public IntegerProperty postalCodeProperty() {
        return postalCode;
    }

    public String getCity() {
        return city.get();
    }

    public void setCity(String city) {
```

```
        this.city.set(city);
    }

    public StringProperty cityProperty() {
        return city;
    }

    public LocalDate getBirthday() {
        return birthday.get();
    }

    public void setBirthday(LocalDate birthday) {
        this.birthday.set(birthday);
    }

    public ObjectProperty<LocalDate> birthdayProperty() {
        return birthday;
    }
}
```

解释

- 在JavaFX中,对一个模型类的所有属性使用 `Properties` 是很常见的. 一个 `Property` 允许我们, 打个比方, 当 `lastName` 或其他属性被改变时自动收到通知, 这有助于我们保持视图与数据的同步， 阅读 Using JavaFX Properties and Binding 学习更多关于 `Properties` 的内容。
- `birthday`, 我们使用了 `LocalDate` 类型, 这在 Date and Time API for JDK 8 中是一个新的部分.

人员列表

我们的应用主要管理的数据是一群人的信息.让我们在 `MainApp` 类里面创建一个 `Person` 对象的列表。稍后其他所有的控制器类将存取 `MainApp` 的核心列表。

ObservableList

我们处理JavaFX的view classes需要在人员列表发生任何改变时都被通知. 这是很重要的,不然视图就会和数据不同步.为了达到这个目的,JavaFX引入了一些新的集合类.

在这些集合中, 我们需要的是`ObservableList`. 将以下代码增加到`MainApp`类的开头去创建一个新的`ObservableList`. 我们也会增加一个构造器去创建一些样本数据和一个公共的getter方法:

MainApp.java

```
    // ... AFTER THE OTHER VARIABLES ...

    /**
     * The data as an observable list of Persons.
     */
    private ObservableList<Person> personData =
FXCollections.observableArrayList();

    /**
     * Constructor
     */
    public MainApp() {
```

```
        // Add some sample data
        personData.add(new Person("Hans", "Muster"));
        personData.add(new Person("Ruth", "Mueller"));
        personData.add(new Person("Heinz", "Kurz"));
        personData.add(new Person("Cornelia", "Meier"));
        personData.add(new Person("Werner", "Meyer"));
        personData.add(new Person("Lydia", "Kunz"));
        personData.add(new Person("Anna", "Best"));
        personData.add(new Person("Stefan", "Meier"));
        personData.add(new Person("Martin", "Mueller"));
    }

    /**
     * Returns the data as an observable list of Persons.
     * @return
     */
    public ObservableList<Person> getPersonData() {
        return personData;
    }

    // ... THE REST OF THE CLASS ...
```

The PersonOverviewController

现在我们终于要将数据加入到表格中了,我们需要一个控制器为了`PersonOverview.fxml`,.

1. 在**view**包下创建一个名为`PersonOverviewController.java`的普通java类(我们需要将这个类放在和`PersonOverview.fxml`相同的包下, 不然SceneBuilder会找不到它 - 至少在当前的版本).
2. 我们需要增加一些实例变量来访问表格和在视图中的标签.这些属性和一些方法有一个特殊的`@FXML`注解. 这对于fxml文件访问私有属性和私有方法来说是必需的. 当将一切都在fxml文件中设置好之后, 应用程序会在fxml文件被载入时自动地填充这些变量. 让我们添加以下的代码:

Note: 记住要使用 **javafx imports**, 而不是awt和swing!

PersonOverviewController.java

```
package ch.makery.address.view;

import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import ch.makery.address.MainApp;
import ch.makery.address.model.Person;

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;

    // Reference to the main application.
    private MainApp mainApp;

    /**
     * The constructor.
     * The constructor is called before the initialize() method.
     */
    public PersonOverviewController() {
    }

    /**
     * Initializes the controller class. This method is automatically called
     * after the fxml file has been loaded.
     */
    @FXML
    private void initialize() {
        // Initialize the person table with the two columns.
        firstNameColumn.setCellValueFactory(cellData ->
cellData.getValue().firstNameProperty());
        lastNameColumn.setCellValueFactory(cellData ->
cellData.getValue().lastNameProperty());
    }

    /**
     * Is called by the main application to give a reference back to itself.
     *
     * @param mainApp
     */
    public void setMainApp(MainApp mainApp) {
        this.mainApp = mainApp;

        // Add observable list data to the table
        personTable.setItems(mainApp.getPersonData());
    }
}
```

可能需要解释一下这段代码:

- 所有fxml文件需要访问的属性和方法必须加上 `@FXML` 注解.实际上,只有在私有的情况下才需要, 但是让它们保持私有并且用注解标记的方式更好!
- `initialize()` 方法在fxml文件完成载入时被自动调用. 那时, 所有的FXML属性都应已被初始化.
- 我们在表格列上使用`setCellValueFactory(...)` 来确定为特定列使用 `Person` 对象的某个属性. 箭头 `->` 表示我们在使用Java 8的 *Lambdas* 特性. (另一个选择是使用 PropertyValueFactory, 但它不是类型安全的).

连接 MainApp 和 PersonOverviewController

`setMainApp(...)` 必须被 `MainApp` 类调用. 这让我们可以访问 `MainApp` 对象并得到 `Persons` 的列表和其他东西. 用以下代码替换 `showPersonOverview()` 方法. 它包含了新增的两行:

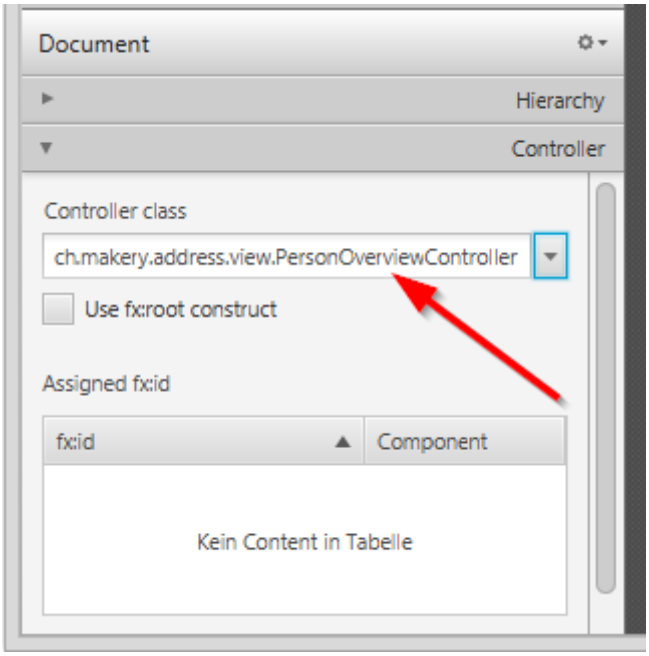
MainApp.java - new showPersonOverview() method

```
/**
 * Shows the person overview inside the root layout.
 */
public void showPersonOverview() {
    try {
        // Load person overview.
        FXMLLoader loader = new FXMLLoader();

 loader.setLocation(MainApp.class.getResource("view/PersonOverview.fxml"));
        AnchorPane personOverview = (AnchorPane) loader.load();

        // Set person overview into the center of root layout.
        rootLayout.setCenter(personOverview);

        // Give the controller access to the main app.
        PersonOverviewController controller = loader.getController();
        controller.setMainApp(this);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

将View与Controller挂钩

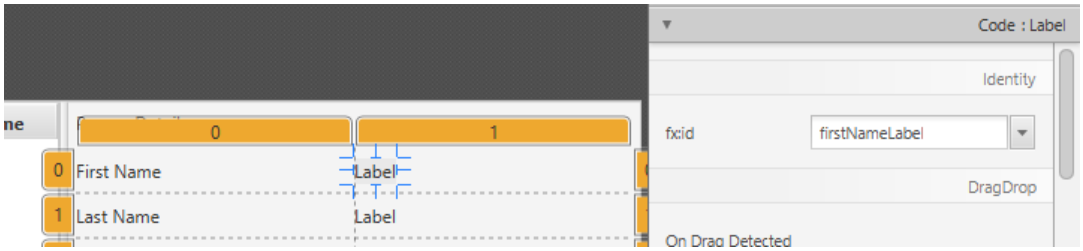
我们快要完成了! 但是有件小事被遗漏了: 至今没有告诉 `PersonOverview.fxml` 使用的是哪个控制器以及元素与控制器中的属性的对应关系.

1. 使用*SceneBuilder* 打开 `PersonOverview.fxml`.
2. 打开左边的 *Controller* 组选择 `PersonOverviewController` 作为 **controller class**.



3. 在 *Hierarchy* 组选择 `TableView` 并选择 *Code* 组将 `personTable` 作为 **fx:id**.

4. 对列做相同的事并且将 `firstNameColumn` and `lastNameColumn` 分别作为 **fx:id** .
5. 对在第二列的 **each label** , 选择对应的 **fx:id**.

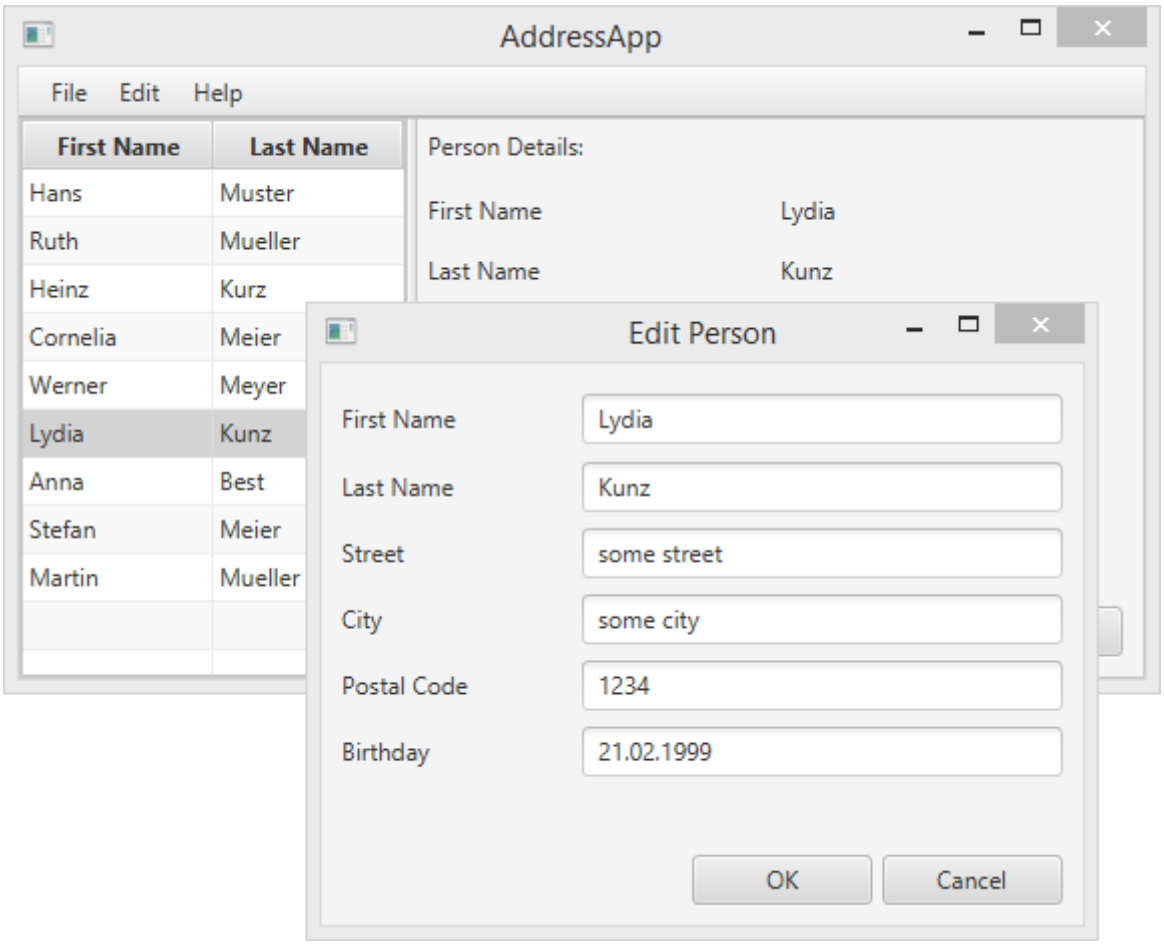


6. 重要事项: 回到eclipse并且 **refresh the entire AddressApp project** (F5). 这是必要的因为有时候eclipse并不知道在Scene Builder中作出的改变.

启动应用程序

当你现在启动了你的应用,你应该看到了类似这篇博客开头的截图的程序界面. 恭喜!

第三部分：与用户的交互



第3部分的主题:

1. 在表中**反应选择的改变**（TableView中）。
2. 增加**增加**，**编辑**和**删除**按钮的功能。
3. 创建自定义**弹出对话框**编辑人员。
4. **验证用户输入**。

响应表的选择

显然，我们还没有使用应用程序的右边。想法是当用户选择表中的人员时，在右边显示人员的详情。

首先，让我们在`PersonOverviewController`添加一个新的方法，帮助我们使用单个人员的数据填写标签。

创建方法`showPersonDetails(Person person)`。遍历所有标签，并且使用`setText(…)`方法设置标签的文本为个人的详情。如果null作为参数传递，所有的标签应该被清空。

PersonOverviewController.java

```
/**
 * Fills all text fields to show details about the person.
 * If the specified person is null, all text fields are cleared.
 *
 * @param person the person or null
 */
private void showPersonDetails(Person person) {
    if (person != null) {
        // Fill the labels with info from the person object.
        firstNameLabel.setText(person.getFirstName());
        lastNameLabel.setText(person.getLastName());
        streetLabel.setText(person.getStreet());
        postalCodeLabel.setText(Integer.toString(person.getPostalCode()));
        cityLabel.setText(person.getCity());

        // TODO: We need a way to convert the birthday into a String!
        // birthdayLabel.setText(...);
    } else {
        // Person is null, remove all the text.
        firstNameLabel.setText("");
        lastNameLabel.setText("");
        streetLabel.setText("");
        postalCodeLabel.setText("");
        cityLabel.setText("");
        birthdayLabel.setText("");
    }
}
```

转换生日日期为字符串

你注意到我们没有设置`birthday`到标签中，因为它是`LocalDate`类型，不是`String`。我们首先需要格式化日期。

在几个地方上我们使用`LocalDate`和`String`之间的转换。好的实践是创建一个带有`static`方法的帮助类。我们称它为`DateUtil`，并且把它放到单独的包中，称为`ch.makery.address.util`。

DateUtil.java

```
package ch.makery.address.util;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;

/**
 * Helper functions for handling dates.
 *
 * @author Marco Jakob
 */
public class DateUtil {

    /** The date pattern that is used for conversion. Change as you wish. */
    private static final String DATE_PATTERN = "dd.MM.yyyy";

    /** The date formatter. */
    private static final DateTimeFormatter DATE_FORMATTER =
            DateTimeFormatter.ofPattern(DATE_PATTERN);

    /**
     * Returns the given date as a well formatted String. The above defined
     * {@link DateUtil#DATE_PATTERN} is used.
     *
     * @param date the date to be returned as a string
     * @return formatted string
     */
    public static String format(LocalDate date) {
        if (date == null) {
            return null;
        }
        return DATE_FORMATTER.format(date);
    }

    /**
     * Converts a String in the format of the defined {@link
DateUtil#DATE_PATTERN}
     * to a {@link LocalDate} object.
     *
     * Returns null if the String could not be converted.
     *
     * @param dateString the date as String
     * @return the date object or null if it could not be converted
     */
    public static LocalDate parse(String dateString) {
        try {
            return DATE_FORMATTER.parse(dateString, LocalDate::from);
        } catch (DateTimeParseException e) {
            return null;
        }
    }

    /**
     * Checks the String whether it is a valid date.
     *
```

```
     * @param dateString
     * @return true if the String is a valid date
     */
    public static boolean validDate(String dateString) {
        // Try to parse the String.
        return DateUtil.parse(dateString) != null;
    }
}
```

提示：你能通过改变`DATE_PATTERN`修改日期的格式。所有可能的格式参考 DateTimeFormatter.

使用DateUtil

现在，我们需要在`PersonOverviewController`的`showPersonDetails`方法中使用我们新建的`DateUtil`。使用下面这样替代我们添加的`TODO`。

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

监听表选择的改变

为了当用户在人员表中选择一个人时获得通知，我们需要**监听改变**。

在JavaFX中有一个接口称为`ChangeListener`，带有一个方法`changed()`。该方法有三个参数：`observable`，`oldValue`和`newValue`。

我们使用*Java 8 lambda*表达式创建这样一个`ChangeListener`。让我们添加一些行到`PersonOverviewController`的`initialize()`方法中。现在看起来是这样的。

PersonOverviewController.java

```
@FXML
private void initialize() {
    // Initialize the person table with the two columns.
    firstNameColumn.setCellValueFactory(
            cellData -> cellData.getValue().firstNameProperty());
    lastNameColumn.setCellValueFactory(
            cellData -> cellData.getValue().lastNameProperty());

    // Clear person details.
    showPersonDetails(null);

    // Listen for selection changes and show the person details when changed.
    personTable.getSelectionModel().selectedItemProperty().addListener(
            (observable, oldValue, newValue) -> showPersonDetails(newValue));
}
```

使用`showPersonDetails(null)`，我们重设个人详情。

使用`personTable.getSelectionModel...`，我们获得人员表的`selectedItemProperty`，并且添加监听。不管什么时候用户选择表中的人员，都会执行我们的`lambda`表达式。我们获取新选择的人员，并且把它传递给`showPersonDetails(...)`方法。

现在试着**运行你的应用程序**，验证当你选择表中的人员时，关于该人员的详情是否正确的显示。

如果有些事情不能工作，你可以对比下PersonOverviewController.java中的`PersonOverviewController`类

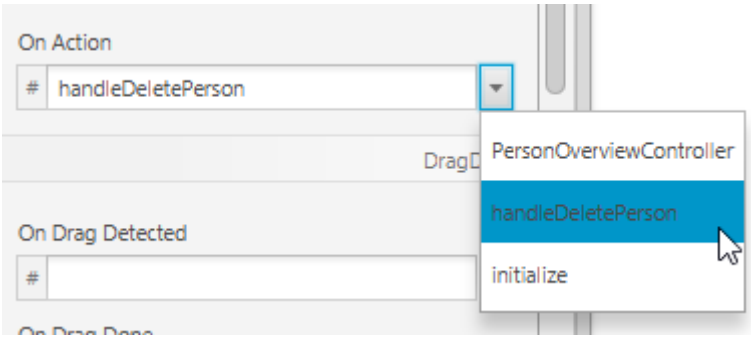
删除按钮

我们的用户接口已经包含一个删除按钮，但是没有任何功能。我们能在*SceneBuilder*中的按钮上选择动作。在我们控制器中的任何使用`@FXML`（或者它是公用的）注释的方法都可以被*Scene Builder*访问。因此，让我们在`PersonOverviewController`类的最后添加一个删除方法。

PersonOverviewController.java

```
/**
 * Called when the user clicks on the delete button.
 */
@FXML
private void handleDeletePerson() {
    int selectedIndex = personTable.getSelectionModel().getSelectedIndex();
    personTable.getItems().remove(selectedIndex);
}
```

现在，使用*SceneBuilder*打开`PersonOverview.fxml`文件，选择*Delete*按钮，打开*Code*组，在**On Actin**的下拉菜单中选择`handleDeletePerson`。



错误处理

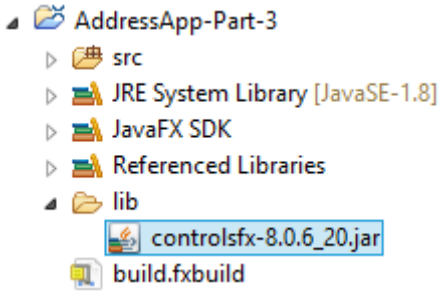
如果你现在运行应用程序，你应该能够从表中删除选择的人员。但是，**当你没有在表中选择人员时点击删除按钮时会发生什么呢。**

这里有一个`ArrayIndexOutOfBoundsException`，因为它不能删除掉索引为-1人员项目。索引-1由`getSelectedIndex()`返回，它意味着你没有选择项目。

当然，忽略这种错误不是非常好。我们应该让用户知道在删除时必须选择一个人员。（更好的是我们应该禁用删除按钮，以便用户没有机会做错误的事情）。

我们添加一个弹出对话框通知用户，你将需要**添加一个库*Dialogs*：

1. 下载controlsfx-8.0.6_20.jar （你也能从ControlsFX Website中获取）。 **重要：ControlsFX必须是8.0.6_20以上版本才能在`JDK8U20`以上版本工作。**
2. 在项目中创建一个**lib**子目录，添加controlsf jar文件到该目录下。
3. 添加库到你的项目**classpath**中。在Eclipse中*右击jar文件*| 选择*Build Path*| *Add to Build Path*。现在Eclipse知道这个库了。



对`handleDeletePerson()`方法做一些修改后，不管什么时候用户没有选择表中的人员时按下删除按钮，我们能显示一个简单的对话框。

PersonOverviewController.java

```
/**
 * Called when the user clicks on the delete button.
 */
@FXML
private void handleDeletePerson() {
    int selectedIndex = personTable.getSelectionModel().getSelectedIndex();
    if (selectedIndex >= 0) {
        personTable.getItems().remove(selectedIndex);
    } else {
        // Nothing selected.
        Dialogs.create()
            .title("No Selection")
            .masthead("No Person Selected")
            .message("Please select a person in the table.")
            .showWarning();
    }
}
```

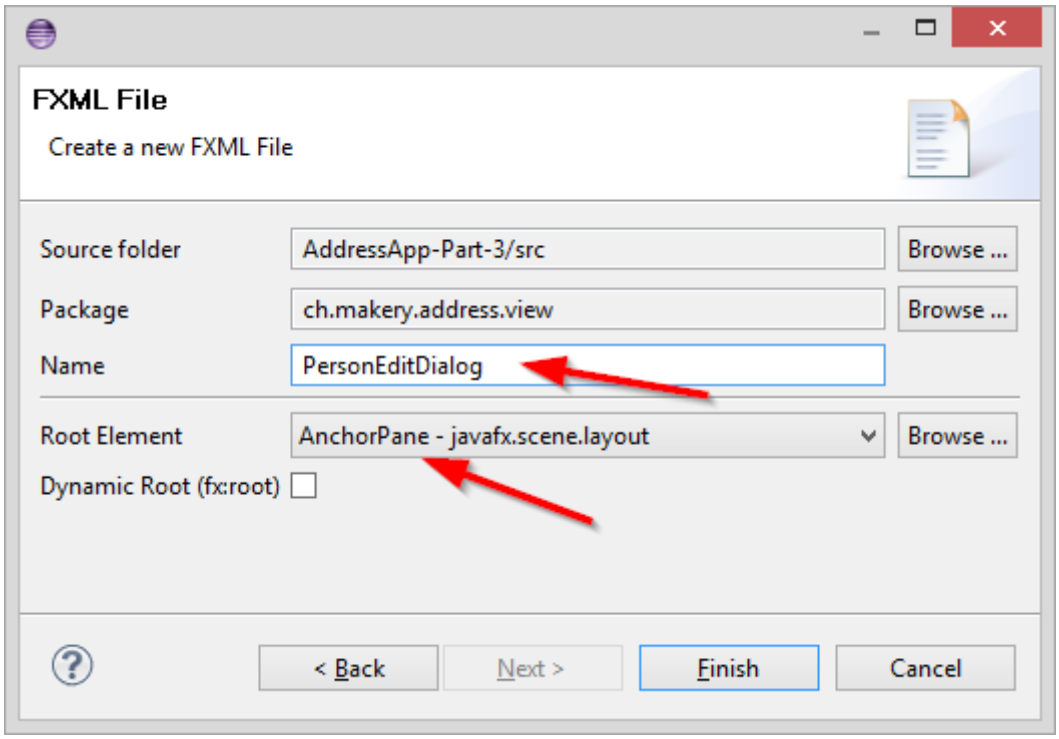
更多如何使用Dialog的示例，请阅读JavaFX 8 Dialogs.

新建和编辑对话框

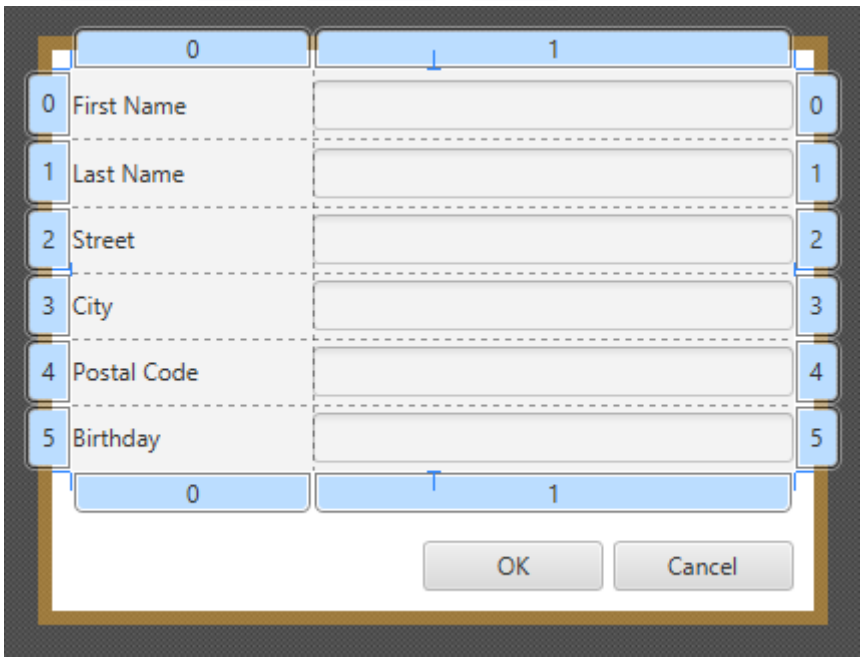
新建和编辑的动作有点工作：我们需要一个自定义带表单的对话框（例如：新的Stage），询问用户关于人员的详情。

设计对话框

1. 在*view*包中创建新的fxml文件，称为`PersonEditDialog.fxml`



2. 使用`GridPan`，`Label`，`TextField`和`Button`创建一个对话框，如下所示:



如果你不能完成工作，你能下载这个PersonEditDialog.fxml.

创建控制器

为对话框创建控制器`PersonEditDialogController.java`:

PersonEditDialogController.java

```
package ch.makery.address.view;

import javafx.fxml.FXML;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

import org.controlsfx.dialog.Dialogs;
```

```
import ch.makery.address.model.Person;
import ch.makery.address.util.DateUtil;

/**
 * Dialog to edit details of a person.
 *
 * @author Marco Jakob
 */
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;

    /**
     * Initializes the controller class. This method is automatically called
     * after the fxml file has been loaded.
     */
    @FXML
    private void initialize() {
    }

    /**
     * Sets the stage of this dialog.
     *
     * @param dialogStage
     */
    public void setDialogStage(Stage dialogStage) {
        this.dialogStage = dialogStage;
    }

    /**
     * Sets the person to be edited in the dialog.
     *
     * @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 if the user clicked OK, false otherwise.
     *
     * @return
     */
    public boolean isOkClicked() {
        return okClicked;
    }

    /**
     * Called when the user clicks 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.setCity(cityField.getText());
            person.setBirthday(DateUtil.parse(birthdayField.getText()));

            okClicked = true;
            dialogStage.close();
        }
    }

    /**
     * Called when the user clicks cancel.
     */
    @FXML
    private void handleCancel() {
        dialogStage.close();
    }

    /**
     * Validates the user input in the text fields.
     *
     * @return true if the input is valid
     */
    private boolean isInputValid() {
        String errorMessage = "";

        if (firstNameField.getText() == null ||
firstNameField.getText().length() == 0) {
            errorMessage += "No valid first name!\n";
        }
        if (lastNameField.getText() == null || lastNameField.getText().length()
== 0) {
            errorMessage += "No valid last name!\n";
        }
        if (streetField.getText() == null || streetField.getText().length() ==
0) {
            errorMessage += "No valid street!\n";
```

```
        }

        if (postalCodeField.getText() == null ||
postalCodeField.getText().length() == 0) {
            errorMessage += "No valid postal code!\n";
        } else {
            // try to parse the postal code into an int.
            try {
                Integer.parseInt(postalCodeField.getText());
            } catch (NumberFormatException e) {
                errorMessage += "No valid postal code (must be an integer)!\n";
            }
        }

        if (cityField.getText() == null || cityField.getText().length() == 0) {
            errorMessage += "No valid city!\n";
        }

        if (birthdayField.getText() == null || birthdayField.getText().length()
== 0) {
            errorMessage += "No valid birthday!\n";
        } else {
            if (!DateUtil.validDate(birthdayField.getText())) {
                errorMessage += "No valid birthday. Use the format
dd.mm.yyyy!\n";
            }
        }

        if (errorMessage.length() == 0) {
            return true;
        } else {
            // Show the error message.
            Dialogs.create()
                .title("Invalid Fields")
                .masthead("Please correct invalid fields")
                .message(errorMessage)
                .showError();
            return false;
        }
    }
}
```

关于该控制器的一些事情应该注意：

1. `setPerson(…)` 方法可以从其它类中调用，用来设置编辑的人员。
2. 当用户点击OK按钮时，调用 `handleOK()` 方法。首先，通过调用 `isInputValid()` 方法做一些验证。只有验证成功，Person对象使用输入的数据填充。这些修改将直接应用到Person对象上，传递给 `setPerson(…)`。
3. 布尔值 `okClicked` 被使用，以便调用者决定用户是否点击OK或者Cancel按钮。

连接视图和控制器

使用已经创建的视图（FXML）和控制器，需要连接到一起。

1. 使用SceneBuilder打开 `PersonEditDialog.fxml` 文件
2. 在左边的*Controller*组中选择 `PersonEditDialogController` 作为控制器类
3. 设置所有**TextField**的 `fx:id` 到相应的控制器字段上。

4. 设置两个按钮的**onAction**到相应的处理方法上。

打开对话框

在`MainApp`中添加一个方法加载和显示编辑人员的对话框。

MainApp.java

```
/**
 * Opens a dialog to edit details for the specified person. If the user
 * clicks OK, the changes are saved into the provided person object and true
 * is returned.
 *
 * @param person the person object to be edited
 * @return true if the user clicked OK, false otherwise.
 */
public boolean showPersonEditDialog(Person person) {
    try {
        // Load the fxml file and create a new stage for the popup dialog.
        FXMLLoader loader = new FXMLLoader();

 loader.setLocation(MainApp.class.getResource("view/PersonEditDialog.fxml"));
        AnchorPane page = (AnchorPane) loader.load();

        // Create the dialog Stage.
        Stage dialogStage = new Stage();
        dialogStage.setTitle("Edit Person");
        dialogStage.initModality(Modality.WINDOW_MODAL);
        dialogStage.initOwner(primaryStage);
        Scene scene = new Scene(page);
        dialogStage.setScene(scene);

        // Set the person into the controller.
        PersonEditDialogController controller = loader.getController();
        controller.setDialogStage(dialogStage);
        controller.setPerson(person);

        // Show the dialog and wait until the user closes it
        dialogStage.showAndWait();

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

添加下面的方法到`PersonOverviewController`中。当用户按下*New*或*Edit*按钮时，这些方法将从`MainApp`中调用`showPersonEditDialog(...)`。

PersonOverviewController.java

```
/**
 * Called when the user clicks the new button. Opens a dialog to edit
 * details for a new person.
 */
@FXML
```

```
private void handleNewPerson() {
    Person tempPerson = new Person();
    boolean okClicked = mainApp.showPersonEditDialog(tempPerson);
    if (okClicked) {
        mainApp.getPersonData().add(tempPerson);
    }
}

/**
 * Called when the user clicks the edit button. Opens a dialog to edit
 * details for the selected person.
 */
@FXML
private void handleEditPerson() {
    Person selectedPerson = personTable.getSelectionModel().getSelectedItem();
    if (selectedPerson != null) {
        boolean okClicked = mainApp.showPersonEditDialog(selectedPerson);
        if (okClicked) {
            showPersonDetails(selectedPerson);
        }

    } else {
        // Nothing selected.
        Dialogs.create()
            .title("No Selection")
            .masthead("No Person Selected")
            .message("Please select a person in the table.")
            .showWarning();
    }
}
```

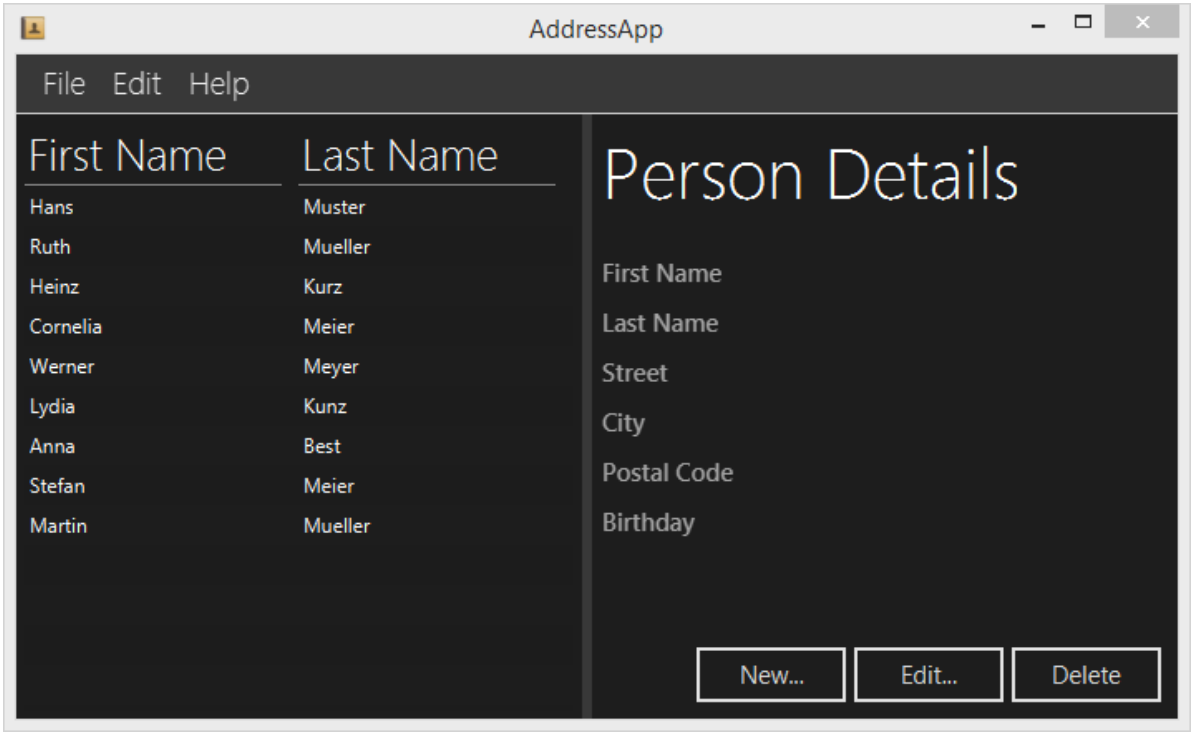
在Scene Builder中打开`PersonOverview.fxml`文件，为New和Edit按钮的*On Action*中选择对应的方法。

完成!

现在你应该有一个可以工作的*Address应用*。应用能够添加、编辑和删除人员。这里甚至有一些文本字段的验证避免坏的用户输入。

我希望本应用的概念和结构让开始编写自己的JavaFX应用！玩的开心。

第四部分：CSS 样式



第4部分主题

- **CSS样式表**
- 添加**应用程序图标**

CSS样式表

在JavaFX中，你能使用层叠样式表修饰你的用户接口。这非常好！自定义Java应用界面从来不是件简单的事情。

在本教程中，我们将创建一个*DarkTheme*主题，灵感来自于Windows 8 Metro设计。按钮的CSS来至于Pedro Duque Vieia的博客Java中JMetro-Windows 8 Metro控件。

熟悉CSS

如果你希望修饰你的JavaFX应用，通常你应该对CSS有一个基本的了解。一个好的起点是CSS教程.

关于CSS更多JavaFX指定信息:

- 使用CSS换肤JavaFX应用 - Oracle教程
- JavaFX CSS参考 - 官方

缺省的JavaFX CSS

在JavaFX 8中缺省的CSS风格源码是一个称为`modena.css`文件。该CSS文件可以在JavaFX jar文件`jfxrt.jar`中找到，它位于Java目录`/jdk1.8.x/jre/lib/ext/jfxrt.jar`。

解压`jfxrt.jar`，你应该能在`com/sun/javafx/scene/control/skin/modena/`目录下找到`modena.css`。

缺省的样式表总是应用到JavaFX应用上。通过添加自定义样式表，你能覆盖`modena.css`中缺省的样式。

提示：查看缺省的CSS文件能够让你模板你需要覆盖掉那些样式。

添加CSS样式表

添加下面的CSS文件`DarkTheme.css`到`view`包中。

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 {
```

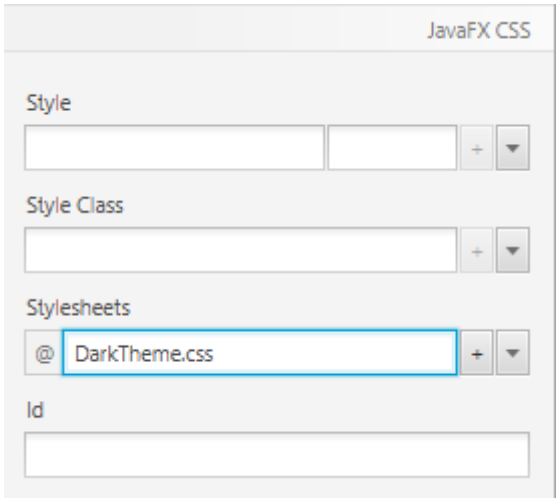
```
    -fx-font-size: 20pt;
    -fx-font-family: "Segoe UI Light";
    -fx-text-fill: white;
    -fx-alignment: center-left;
    -fx-opacity: 1;
}

.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";
}

/*
 * Metro style Push Button
 * Author: 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%);
}
```

现在我们需要把CSS添加到我们的场景中。我们能在Java代码中编程完成，但是我们将使用SceneBuilder来添加它到fxml文件中。

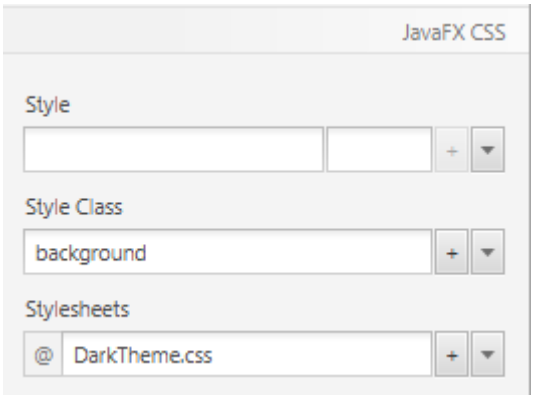
添加CSS到RootLayout.fxml

1. 在Scene Builder中打开`RootLayout.fxml`
2. 在*Hierarchy*视图中选择根节点`BorderPane`。在*Properties*组中添加`DarkTheme.css`作为样式表。



添加CSS到PersonEditDialog.fxml

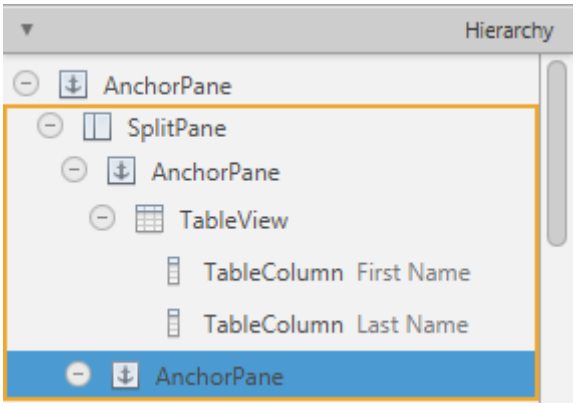
1. 在Scene Builder中打开`PersonEditDialog.fxml`。选择根节点`AnchorPane`，并且在*Properties*组中选择`DarkTheme.css`作为样式表。
2. 背景仍然是白色的，因此添加样式类`background`到根节点`AnchorPane`。



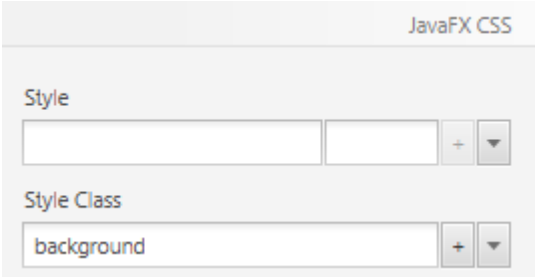
1. 选择OK按钮，在*Properties*视图中选择*Default Button*单选框。这将修改它的颜色，当用户*输入*关键词时，使用它作为缺省的按钮。

添加CSS到PersonOverview.fxml

1. 在Scene Builder中打开文件`PersonOverview.fxml`。在*Hierarchy*组中选择根节点`AnchorPane`。在*Properties*下面添加`DarkTheme.css`文件作为样式表。
2. 你现在应该已经看到一些修改，表和按钮是黑色的。来自`modena.css`中所有类样式`.table-view`和`.button`应用到表和按钮。因为我们已经在自定义CSS中重定义（因此覆盖掉）一些样式。新的样式自动应用。
3. 你可能需要调整按钮的大小，以便显示所有的文本。
4. 选择`SplitPane`中右边的`AnchorPane`。



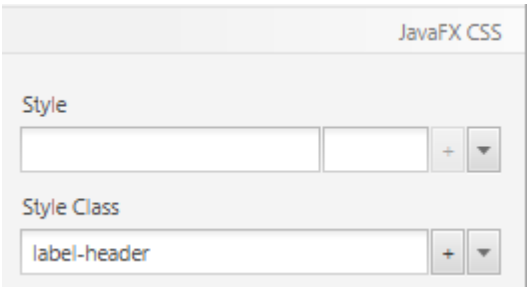
5. 进入到*Properties*组，并且选择`background`作为样式表。背景现在应该变为黑色。



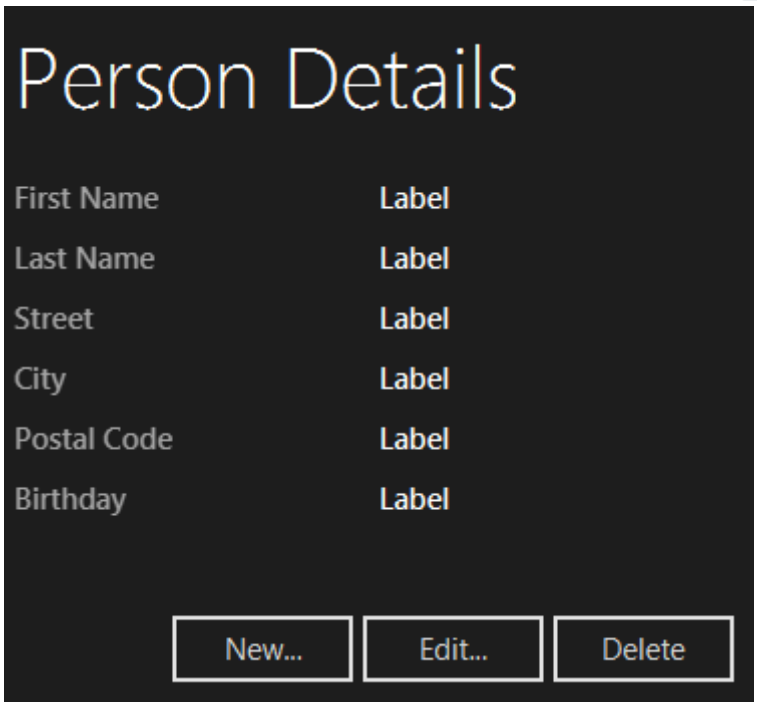
使用不同样式的标签

现在，在左边的所有的标签都有相同的大小。这里已经有一些样式定义在CSS文件中，称为`.label-header`和`.label-bright`。我们将使用更多样式的标签Label。

1. 选择*Person Detail*标签，添加`label-header`作为样式类。

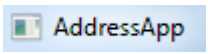


2. 给右边一列的每个Label（显示人员详情的那列）添加CSS样式类`label-bright`。

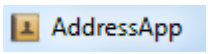


添加应用图标

现在，在标题栏和任务栏中，我们的应用只有一个缺省图标:



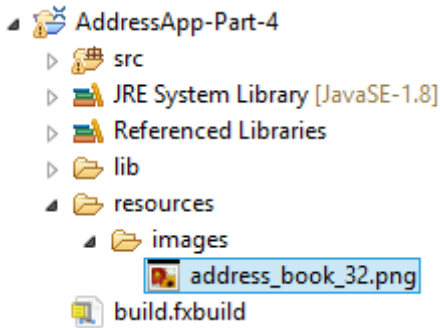
使用自定义图标看起来更好了。



图标文件

获取图标的一个可能地方是Icon Finder。我下载了一个地址本的图标.

通常在你的AddressApp项目中创建一个目录称为**resources**，在它中子目录称为**images**。把你选择的图标放入到images目录中。现在，你的目录结构应该看上去如下所示:



设置图标到场景

为了给你场景设置图标，添加下面一行到`MainApp.jar`的`start(…)`方法中。

MainApp.java

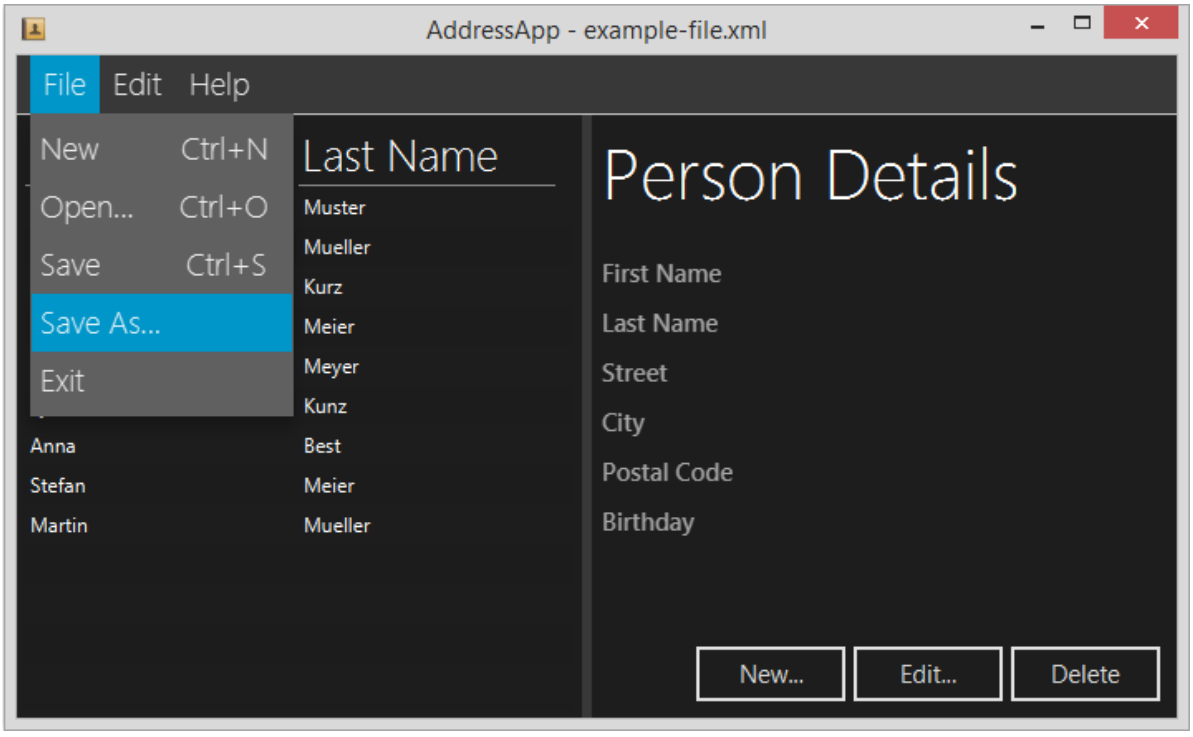
```
this.primaryStage.getIcons().add(new
Image("file:resources/images/address_book_32.png"));
```

现在，整个`start(…)`方法看上去应该是这样的。：

```
public void start(Stage primaryStage) {
    this.primaryStage = primaryStage;
    this.primaryStage.setTitle("AddressApp");

    // Set the application icon.
    this.primaryStage.getIcons().add(new
Image("file:resources/images/address_book_32.png"));

    initRootLayout();

    showPersonOverview();
}
```

当然，你也应该添加图标到人员编辑对话框的Stage中。

第五部分：用 XML 格式存储数据



第5部分的主题

- **持久化数据为XML**
- 使用JavaFX的**FileChooser**
- 使用JavaFX的**菜单**
- 在**用户设置**中保存最后打开的文件路径。

现在我们的地址应用程序的数据只保存在内存中。每次我们关闭应用程序，数据将丢失，因此是时候开始考虑持久化存储数据了。

保存用户设置

Java允许我们使用`Preferences`类保存一些应用状态。依赖于操作系统，`Perferences`保存在不同的地方（例如：Windows中的注册文件）。

我们不能使用`Preferences`来保存全部地址簿。但是它允许我们**保存一些简单的应用状态。**一件这样事情是**最后打开文件的路径。**使用这个信息，我们能加载最后应用的状态，不管用户什么时候重启应用程序。

下面两个方法用于保存和检索Preference。添加它们到你的`MainApp`类的最后：

MainApp.java

```
/**
 * Returns the person file preference, i.e. the file that was last opened.
 * The preference is read from the OS specific registry. If no such
 * preference can be found, null is returned.
 *
 * @return
 */
public File getPersonFilePath() {
    Preferences prefs = Preferences.userNodeForPackage(MainApp.class);
    String filePath = prefs.get("filePath", null);
    if (filePath != null) {
        return new File(filePath);
    } else {
        return null;
    }
}

/**
 * Sets the file path of the currently loaded file. The path is persisted in
 * the OS specific registry.
 *
 * @param file the file or null to remove the path
 */
public void setPersonFilePath(File file) {
    Preferences prefs = Preferences.userNodeForPackage(MainApp.class);
    if (file != null) {
        prefs.put("filePath", file.getPath());

        // Update the stage title.
        primaryStage.setTitle("AddressApp - " + file.getName());
    } else {
        prefs.remove("filePath");

        // Update the stage title.
        primaryStage.setTitle("AddressApp");
    }
}
```

持久性数据到XML

为什么是XML?

持久性数据的一种最常用的方法是使用数据库。数据库通常包含一些类型的关系数据（例如：表），当我们需要保存的数据是对象时。这称object-relational impedance mismatch。匹配对象到关系型数据库表有很多工作要做。这里有一些框架帮助我们匹配（例如：Hibernate，最流行的一个）。但是它仍然需要相当多的设置工作。

对于简单的数据模型，非常容易使用XML。我们使用称为JAXB（**J**ava **A**rchitecture for **X**ML **B**inding）的库。只需要几行代码，JAXB将允许我们生成XML输出，如下所示：

示例XML输出

```
<persons>
    <person>
        <birthday>1999-02-21</birthday>
        <city>some city</city>
        <firstName>Hans</firstName>
        <lastName>Muster</lastName>
        <postalCode>1234</postalCode>
        <street>some street</street>
    </person>
    <person>
        <birthday>1999-02-21</birthday>
        <city>some city</city>
        <firstName>Anna</firstName>
        <lastName>Best</lastName>
        <postalCode>1234</postalCode>
        <street>some street</street>
    </person>
</persons>
```

使用JAXB

JAXB已经包含在JDK中。这意味着我们不需要包含任何其它的库。

JAXB提供两个主要特征：**编列(marshal)**Java对象到XML的能力，**反编列(unmarshal)**XML到Java对象。

为了让JAXB能够做转换，我们需要准备我们的模型。

准备JAXB的模型类

我们希望保持的数据位于`MainApp`类的`personData`变量中。JAXB要求使用`@XmlRootElement`注释作为最顶层的类。`personData`是`ObservableList`类，我们不能把任何注释放到`ObservableList`上。因此，我们需要创建另外一个类，它只用于保存`Person`列表，用于存储成XML文件。

创建的新类名为`PersonListWrapper`，把它放入到`ch.makery.address.model`包中。

PersonListWrapper.java

```
package ch.makery.address.model;

import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

```java
/**
 * Helper class to wrap a list of persons. This is used for saving the
 * list of persons to XML.
 *
 * @author Marco Jakob
 */
@XmlRootElement(name = "persons")
public class PersonListWrapper {

    private List<Person> persons;

    @XmlElement(name = "person")
    public List<Person> getPersons() {
        return persons;
    }

    public void setPersons(List<Person> persons) {
        this.persons = persons;
    }
}
```

注意两个注释：

- `@XmlRootElement` 定义根元素的名称。
- `@XmlElement` 一个可选的名称，用来指定元素。

使用JAXB读写数据

我们让 `MainApp` 类负责读写人员数据。添加下面两个方法到 `MainApp.java` 的最后：

```java
/**
 * Loads person data from the specified file. The current person data will
 * be replaced.
 *
 * @param file
 */
public void loadPersonDataFromFile(File file) {
    try {
        JAXBContext context = JAXBContext
                .newInstance(PersonListWrapper.class);
        Unmarshaller um = context.createUnmarshaller();

        // Reading XML from the file and unmarshalling.
        PersonListWrapper wrapper = (PersonListWrapper) um.unmarshal(file);

        personData.clear();
        personData.addAll(wrapper.getPersons());

        // Save the file path to the registry.
        setPersonFilePath(file);

    } catch (Exception e) { // catches ANY exception
        Dialogs.create()
                .title("Error")
                .masthead("Could not load data from file:\n" + file.getPath())
                .showException(e);
    }
```

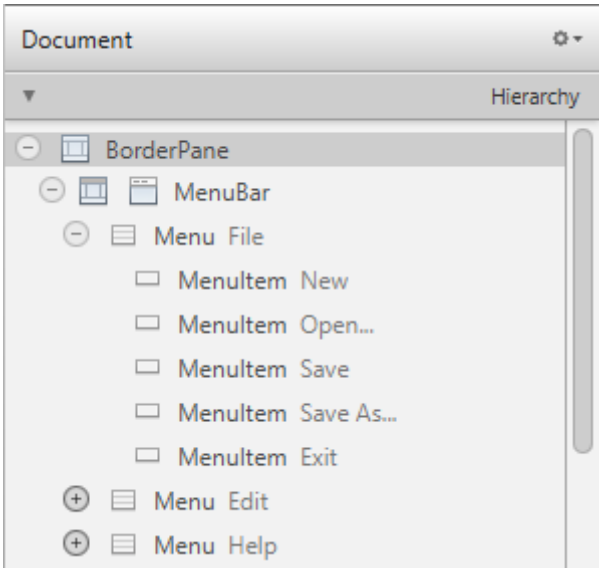
```
}

/**
 * Saves the current person data to the specified file.
 *
 * @param file
 */
public void savePersonDataToFile(File file) {
    try {
        JAXBContext context = JAXBContext
                .newInstance(PersonListWrapper.class);
        Marshaller m = context.createMarshaller();
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

        // Wrapping our person data.
        PersonListWrapper wrapper = new PersonListWrapper();
        wrapper.setPersons(personData);

        // Marshalling and saving XML to the file.
        m.marshal(wrapper, file);

        // Save the file path to the registry.
        setPersonFilePath(file);
    } catch (Exception e) { // catches ANY exception
        Dialogs.create().title("Error")
                .masthead("Could not save data to file:\n" + file.getPath())
                .showException(e);
    }
}
```

*编组和解组*已经准备好，让我们创建保存和加载的菜单实际的使用它。

处理菜单响应

在我们`RootLayout.fxml`中，这里已经有一个菜单，但是我们没有使用它。在我们添加响应到菜单中之前，我们首先创建所有的菜单项。

在Scene Builder中打开`RootLayout.fxml`，从*library*组中拖曳必要的菜单到*Hierarchy*组的`MemuBar`中。创建**New**，**Open...**，**Save**，**Save As...*和*Exit**菜单项。



提示：使用*Properties*组下的*Accelerator*设置，你能设置菜单项的快捷键。

RootLayoutController

为了处理菜单动作，我们需要创建一个新的控制器类。在控制器包`ch.makery.address.view`中创建一个类`RootLayoutController`。

添加下面的内容到控制器中：

RootLayoutController.java

```
package ch.makery.address.view;

import java.io.File;

import javafx.fxml.FXML;
import javafx.stage.FileChooser;

import org.controlsfx.dialog.Dialogs;

import ch.makery.address.MainApp;

/**
 * The controller for the root layout. The root layout provides the basic
 * application layout containing a menu bar and space where other JavaFX
 * elements can be placed.
 *
 * @author Marco Jakob
 */
public class RootLayoutController {

    // Reference to the main application
    private MainApp mainApp;

    /**
     * Is called by the main application to give a reference back to itself.
     *
     * @param mainApp
     */
    public void setMainApp(MainApp mainApp) {
        this.mainApp = mainApp;
    }

    /**
     * Creates an empty address book.
     */
    @FXML
    private void handleNew() {
        mainApp.getPersonData().clear();
        mainApp.setPersonFilePath(null);
    }

    /**
     * Opens a FileChooser to let the user select an address book to load.
     */
    @FXML
    private void handleOpen() {
        FileChooser fileChooser = new FileChooser();
```

```
        // Set extension filter
        FileChooser.ExtensionFilter extFilter = new FileChooser.ExtensionFilter(
                "XML files (*.xml)", "*.xml");
        fileChooser.getExtensionFilters().add(extFilter);

        // Show save file dialog
        File file = fileChooser.showOpenDialog(mainApp.getPrimaryStage());

        if (file != null) {
            mainApp.loadPersonDataFromFile(file);
        }
    }

    /**
     * Saves the file to the person file that is currently open. If there is no
     * open file, the "save as" dialog is shown.
     */
    @FXML
    private void handleSave() {
        File personFile = mainApp.getPersonFilePath();
        if (personFile != null) {
            mainApp.savePersonDataToFile(personFile);
        } else {
            handleSaveAs();
        }
    }

    /**
     * Opens a FileChooser to let the user select a file to save to.
     */
    @FXML
    private void handleSaveAs() {
        FileChooser fileChooser = new FileChooser();

        // Set extension filter
        FileChooser.ExtensionFilter extFilter = new FileChooser.ExtensionFilter(
                "XML files (*.xml)", "*.xml");
        fileChooser.getExtensionFilters().add(extFilter);

        // Show save file dialog
        File file = fileChooser.showSaveDialog(mainApp.getPrimaryStage());

        if (file != null) {
            // Make sure it has the correct extension
            if (!file.getPath().endsWith(".xml")) {
                file = new File(file.getPath() + ".xml");
            }
            mainApp.savePersonDataToFile(file);
        }
    }

    /**
     * Opens an about dialog.
     */
    @FXML
    private void handleAbout() {
        Dialogs.create()
            .title("AddressApp")
```

```
                .masthead("About")
                .message("Author: Marco Jakob\nWebsite: http://code.makery.ch")
                .showInformation();
    }

    /**
     * Closes the application.
     */
    @FXML
    private void handleExit() {
        System.exit(0);
    }
}
```

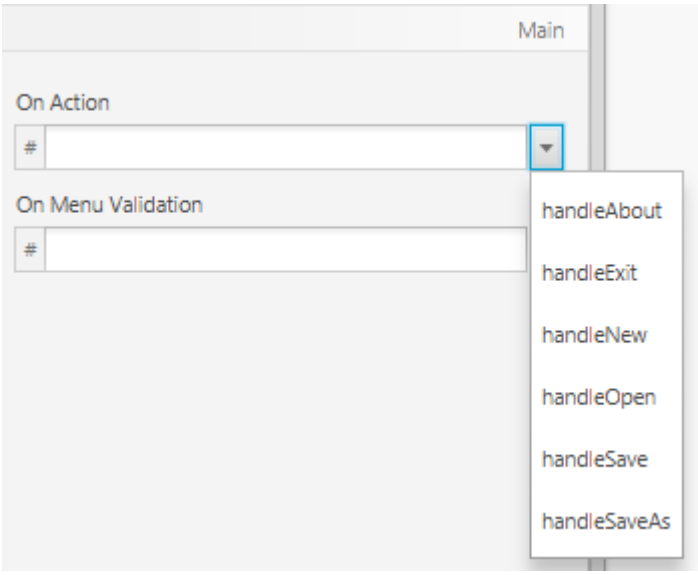
FileChooser

注意在上面的`RootLayoutController`中使用`FileCooser`的方法。首先，创建新的`FileChooser`类对象的，然后，添加扩展名过滤器，以至于只显示以`.xml`结尾的文件。最后，文件选择器显示在主Stage的上面。

如果用户没有选择一个文件关闭对话框，返回`null`。否则，我们获得选择的文件，我们能传递它到`MainApp`的`loadPersonDataFromFile(…)`或`savePersonDataToFile()`方法中。

连接fxml视图到控制器

1. 在Scene Builder中打开`RootLayout.fxml`。在*Controller*组中选择`RootLayoutController`作为控制器类。
2. 回到*Hierarchy*组中，选择一个菜单项。在*Code*组中**On Action**下，应该看到所有可用控制器方法的选择。为每个菜单项选择响应的方法。



3. 为每个菜单项重复第2步。
4. 关闭Scene Builder，并且在项目的根目录上按下**刷新F5**。这让Eclipse知道在Scene Builder中所做的修改。

连接MainApp和RootLayoutController

在几个地方，`RootLayoutController`需要引用`MainApp`类。我们也没有传递一个`MainApp`的引用到`RootLayoutController`。

打开`MainApp`类，使用下面的替代`initRootLayout()`方法：

```
/**
 * Initializes the root layout and tries to load the last opened
 * person file.
 */
public void initRootLayout() {
    try {
        // Load root layout from fxml file.
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MainApp.class
                .getResource("view/RootLayout.fxml"));
        rootLayout = (BorderPane) loader.load();

        // Show the scene containing the root layout.
        Scene scene = new Scene(rootLayout);
        primaryStage.setScene(scene);

        // Give the controller access to the main app.
        RootLayoutController controller = loader.getController();
        controller.setMainApp(this);

        primaryStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }

    // Try to load last opened person file.
    File file = getPersonFilePath();
    if (file != null) {
        loadPersonDataFromFile(file);
    }
}
```

注意两个修改：一行*给控制器访问MainApp*和最后三行*加载最新打开的人员文件*。

测试

做应用程序的测试驱动，你应该能够使用菜单保存人员数据到文件中。

当你在编辑器中打开一个`xml`文件，你将注意到生日没有正确保存，这是一个空的`<birthday/>`标签。原因是JAXB不只奥如何转换`LocalDate`到XML。我们必须提供一个自定义的`LocalDateAdapter`定义这个转换。

在`ch.makery.address.util`中创建新的类，称为`LocalDateAdapter`，内容如下：

LocalDateAdapter.java

```
package ch.makery.address.util;

import java.time.LocalDate;

import javax.xml.bind.annotation.adapters.XmlAdapter;

/**
 * Adapter (for JAXB) to convert between the LocalDate and the ISO 8601
 * String representation of the date such as '2012-12-03'.
 *
 * @author Marco Jakob
```

```
 */
public class LocalDateAdapter extends XmlAdapter<String, LocalDate> {

    @Override
    public LocalDate unmarshal(String v) throws Exception {
        return LocalDate.parse(v);
    }

    @Override
    public String marshal(LocalDate v) throws Exception {
        return v.toString();
    }
}
```

然后打开`Person.jar`，添加下面的注释到`getBirthday()`方法上：

```
@XmlJavaTypeAdapter(LocalDateAdapter.class)
public LocalDate getBirthday() {
    return birthday.get();
}
```

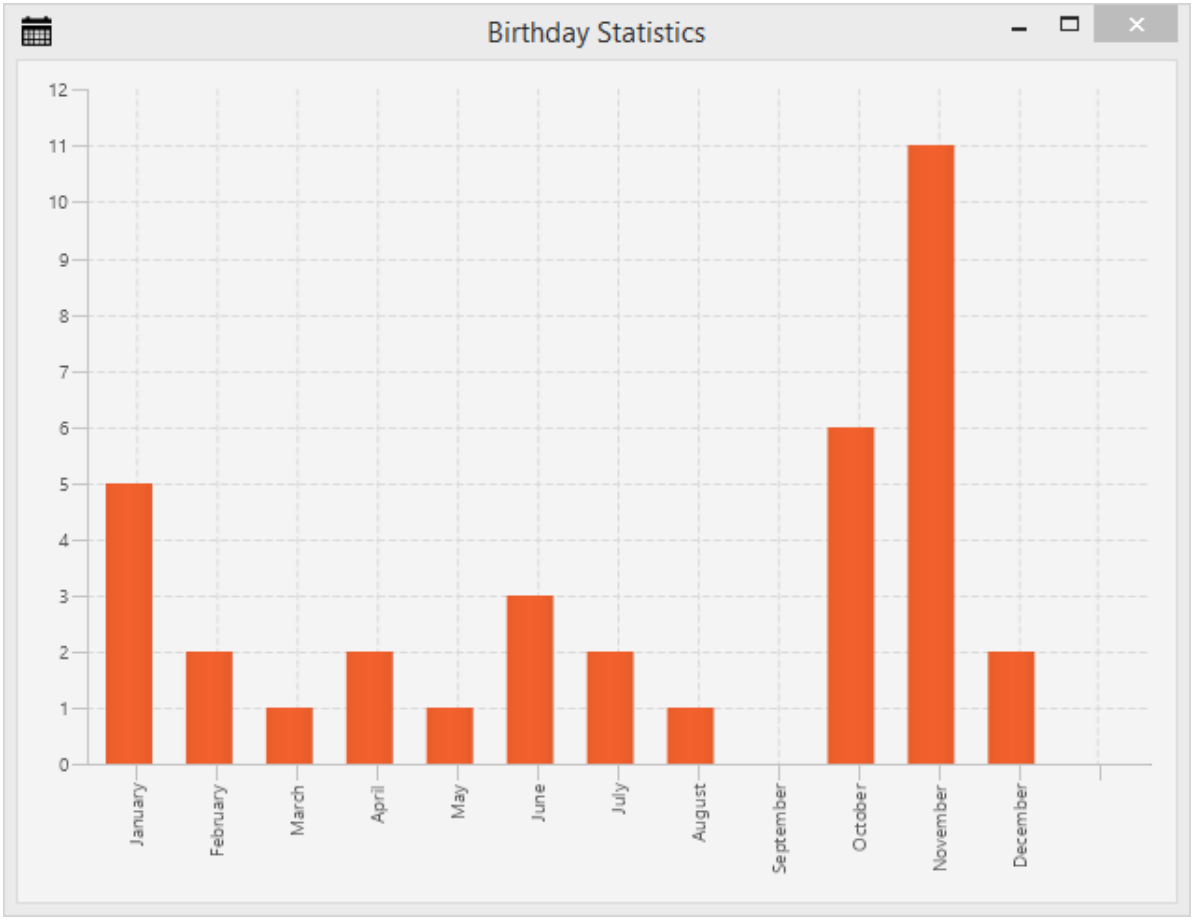
现在，再次测试。试着保存和加载XML文件。在重启之后，它应该自动加载最后使用的文件。

它如何工作

让我们看下它是如何一起工作的：

1. 应用程序使用`MainApp`中的`main(…)`方法启动。
2. 调用`public MainApp()`构造函数添加一些样例数据。
3. 调用`MainApp`的`start(…)`方法，调用`initRootLayout()`从`RootLayout.fxml`中初始化根布局。fxml文件有关于使用控制器的信息，连接视图到`RootLayoutController`。
4. `MainApp`从fxml加载器中获取`RootLayoutController`，传递自己的引用到控制器中。使用这些引用，控制器随后可以访问`MainApp`的公开方法。
5. 在`initRootLayout`方法结束，我们试着从`Perferences`中获取*最后打开的人员文件*。如果`Perferences`知道有这样一个XML文件，我们将从这个XML文件中加载数据。这显然会覆盖掉构造函数中的样例数据。

第六部分：统计图



第6部分的主题

- 创建一个**统计图**显示生日的分布。

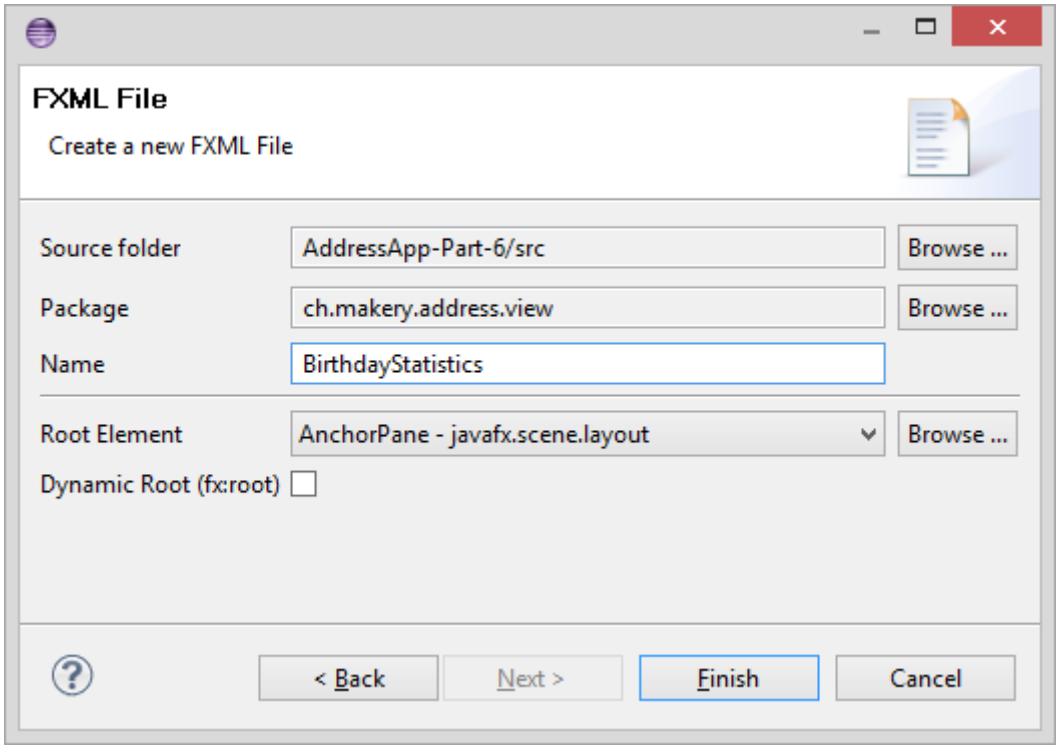
生日统计

在AddressApp中所有人员都有生日。当我们人员庆祝他们生日的时候，如果有一些生日的统计不是会更好。

我们使用**柱状图**，包含每个月的一个条形。每个条形显示在指定月份中有多少人需要过生日。

统计FXML视图

1. 在`ch.makery.address.view`包中我们开始创建一个`BirthdayStatistics.fxml`（*右击包|New|other..|New FXML Document*）



2. 在Scene Builder中打开`BirthdayStatistics.fxml`文件。
3. 选择根节点`AnchorPane`。在*Layout*组中设置*Pref Width*为620，*Pref Height*为450。
4. 添加`BarChart`到`AnchorPane`中。
5. 右击`BarChart`并且选择*Fit to Parent*。
6. 保存fxml文件，进入到Eclipse中，F5刷新项目。

在我们返回到Scene Builder之前，我们首先创建控制器，并且在我们的`MainApp`中准备好一切。

统计控制器

在view包`ch.makery.address.view`中创建一个Java类，称为`BirthdayStatisticsController.java`。

在开始解释之前，让我们看下整个控制器类。

BirthdayStatisticsController.java

```
package ch.makery.address.view;

import java.text.DateFormatSymbols;
import java.util.Arrays;
import java.util.List;
import java.util.Locale;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.chart.BarChart;
import javafx.scene.chart.CategoryAxis;
import javafx.scene.chart.XYChart;
import ch.makery.address.model.Person;

/**
 * The controller for the birthday statistics view.
 *
 * @author Marco Jakob
```

```
 */
public class BirthdayStatisticsController {

    @FXML
    private BarChart<String, Integer> barChart;

    @FXML
    private CategoryAxis xAxis;

    private ObservableList<String> monthNames =
FXCollections.observableArrayList();

    /**
     * Initializes the controller class. This method is automatically called
     * after the fxml file has been loaded.
     */
    @FXML
    private void initialize() {
        // Get an array with the English month names.
        String[] months =
DateFormatSymbols.getInstance(Locale.ENGLISH).getMonths();
        // Convert it to a list and add it to our ObservableList of months.
        monthNames.addAll(Arrays.asList(months));

        // Assign the month names as categories for the horizontal axis.
        xAxis.setCategories(monthNames);
    }

    /**
     * Sets the persons to show the statistics for.
     *
     * @param persons
     */
    public void setPersonData(List<Person> persons) {
        // Count the number of people having their birthday in a specific month.
        int[] monthCounter = new int[12];
        for (Person p : persons) {
            int month = p.getBirthday().getMonthValue() - 1;
            monthCounter[month]++;
        }

        XYChart.Series<String, Integer> series = new XYChart.Series<>();

        // Create a XYChart.Data object for each month. Add it to the series.
        for (int i = 0; i < monthCounter.length; i++) {
            series.getData().add(new XYChart.Data<>(monthNames.get(i),
monthCounter[i]));
        }

        barChart.getData().add(series);
    }
}
```

控制器如何工作

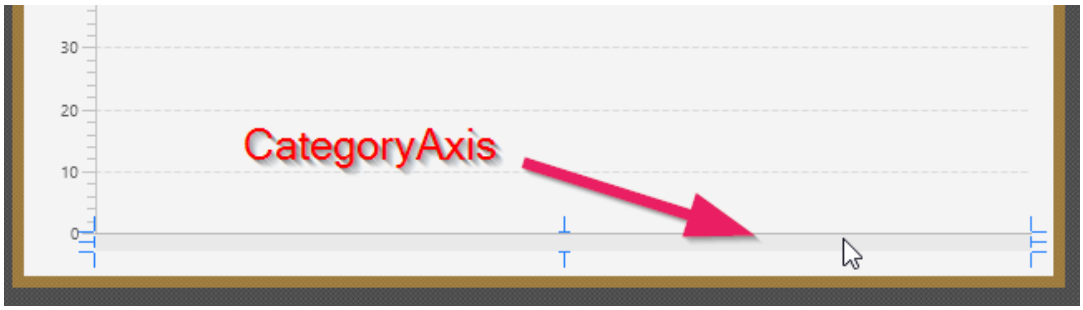
1. 控制器需要从FXML文件中访问两个元素:

- `barChar`：它有`String`和`Integer`类型。`String`用于x轴上的月份，`Integer`用于指定月份中人员的数量。
- `xAxis`：我们使用它添加月字符串

1. `initialize()`方法使用所有月的列表填充`x-axis`。
2. `setPersonData(…)`方法将由`MainApp`访问，设置人员数据。它遍历所有人员，统计出每个月生日的人数。然后它为每个月添加`XYChart.Data`到数据序列中。每个`XYChart.Data`对象在图表中表示一个条形。

连接视图和控制器

1. 在Scene Builder中打开`BirthdayStatistics.fxml`。
2. 在**Controller**组中设置`BirthdayStatisticsController`为控制器。
3. 选择`BarChart`，并且选择`barChar`作为fx:id属性（在*Code*组中）
4. 选择`CategoryAxis`，并且选择`xAxis`作为fx:id属性。



5. 你可以添加一个标题给`BarChar`（在*Properties*组中）进一步修饰。

连接View/Controller和MainApp

我们为*生日统计*使用与*编辑人员对话框*相同的机制，一个简单的弹出对话框。

添加下面的方法到`MainApp`类中

```
/**
 * Opens a dialog to show birthday statistics.
 */
public void showBirthdayStatistics() {
    try {
        // Load the fxml file and create a new stage for the popup.
        FXMLLoader loader = new FXMLLoader();

 loader.setLocation(MainApp.class.getResource("view/BirthdayStatistics.fxml"));
        AnchorPane page = (AnchorPane) loader.load();
        Stage dialogStage = new Stage();
        dialogStage.setTitle("Birthday Statistics");
        dialogStage.initModality(Modality.WINDOW_MODAL);
        dialogStage.initOwner(primaryStage);
        Scene scene = new Scene(page);
        dialogStage.setScene(scene);

        // Set the persons into the controller.
        BirthdayStatisticsController controller = loader.getController();
```

```
            controller.setPersonData(personData);

            dialogStage.show();

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

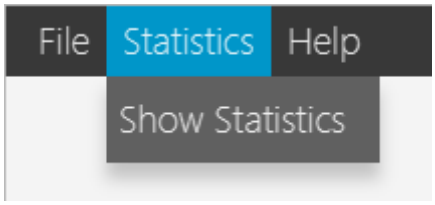
一切设置完毕，但是我们没有任何东西实际上调用新的`showBirthdayStatistics()`方法。幸运的是我们已经在`RootLayout.fxml`中有一个菜单，它可以用于这个目的。

显示生日统计菜单

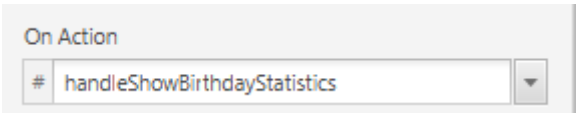
在`RootLayoutController`中添加下面的方法，它将处理*显示生日统计*菜单项的用户点击。

```
/**
 * Opens the birthday statistics.
 */
@FXML
private void handleShowBirthdayStatistics() {
  mainApp.showBirthdayStatistics();
}
```

现在，使用Scene Builder打开`RootLayout.fxml`文件。创建*Staticstic* `菜单`，带有一个*Show Statistcs* `MenuItem`：

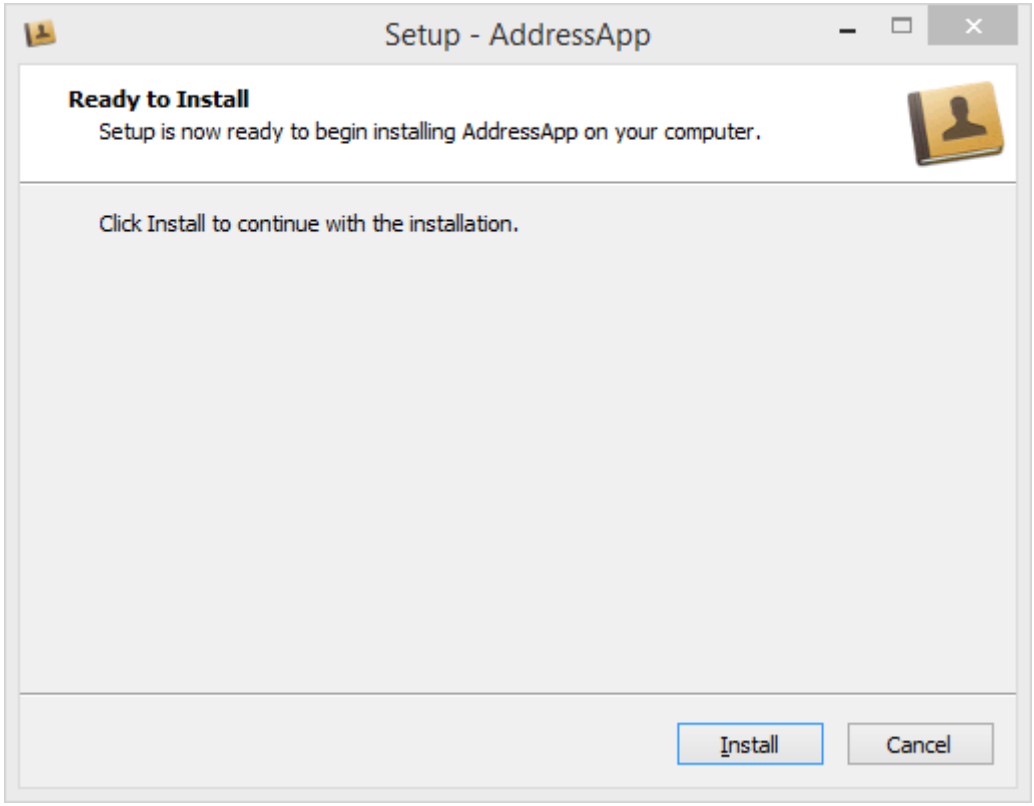


选择*Show Statistics* `MenuItem`，并且选择`handleShowBirthdayStatistics`作为`On Action`（在*Code*组中）。



进入到Eclipse，刷新项目，**测试它。**

第七部分：软件的部署



我想已经写到本教程系列的最后一部分了，应该教你如何部署（例如：打包和发布）AddressApp

第7部分的主题

- 使用e(fx)clipse**本地包（Native Package）**部署我们的JavaFX应用程序。

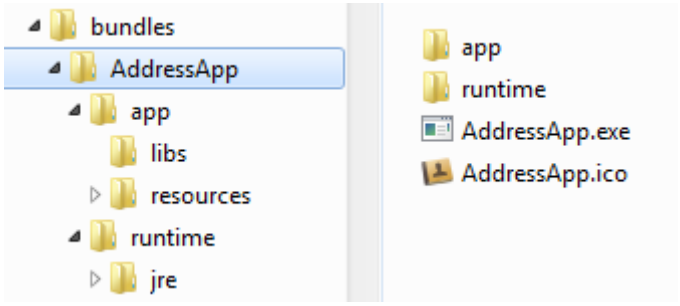
什么是部署

部署是打包和发布软件给用户的过程。这是软件开发的关键部分，因为它是第一次与使用我们软件的用户交流。

Java的广告口号是**编写一次，到处运行**，这说明Java语言的**跨平台**好处。理想情况下，这意味着我们Java应用可以运行在任何装备有JVM的设备上。 在过去，安装Java应用程序的用户经验不总是平滑的。如果用户在系统中没有要求的Java版本，它必须首先直接安装它。这导致有些困难，例如，需要管理员权限，Java版本之间的兼容问题等等。 幸运的是，JavaFX提供新的部署选项称为**本地打包**（也称为自包含应用程序包）。一个本地包是一个包含你的应用代码和平台指定的Java运行时的打包程序。Oracle提供的官方JavaFx文档包含一个所有JavaFX部署选项的扩展指南。 在本章节中，我们教你如何使用Eclipse和**e(fx)clipse插件**创建**本地包**。

创建本地包

目标是在用户的计算机上单个目录中创建一个自包含的应用程序。下面是AddressApp应用看起来的样子（在Windows上）：



`app`目录包含我们的应用数据和`runtime`目录（包含平台相关的Java运行时）。

为了让用户更加舒适，我们也提供一个安装器：

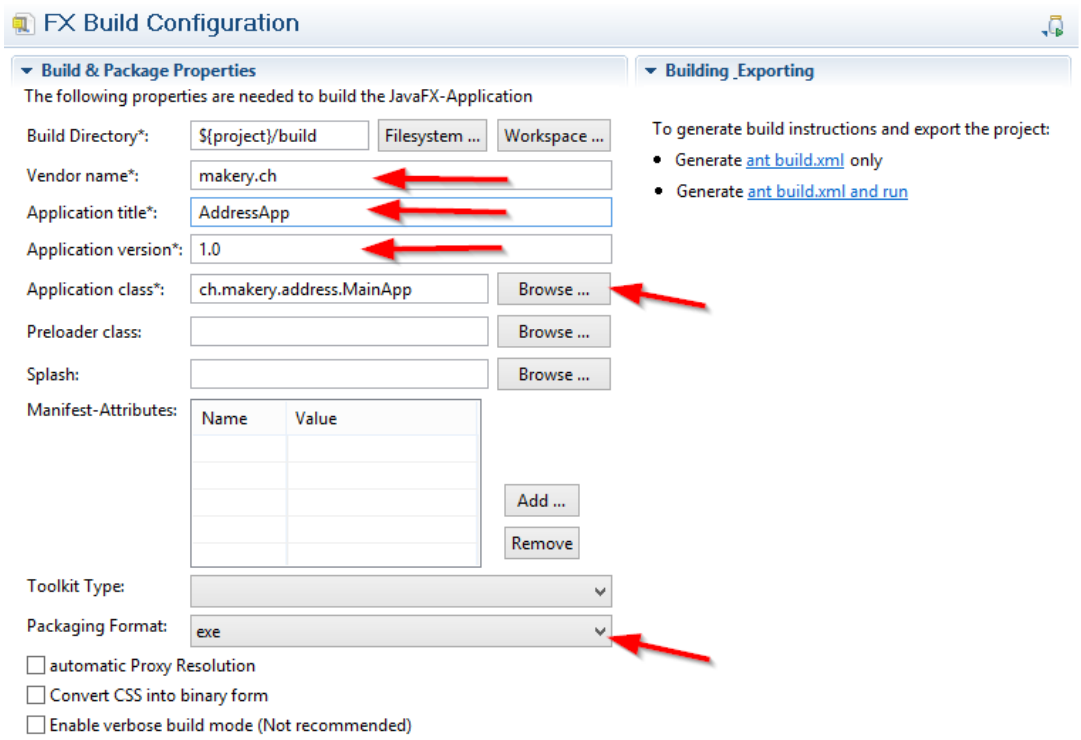
- Windows下的`exe`文件安装器
- MacOS下的`dmg`（拖放）安装器。

E(fx)clipse插件会帮助我们生成本地包和安装器。

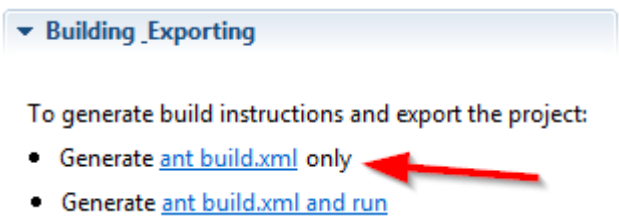
第1步 编辑build.fxbuild

E(fx)clipse使用`build.fxbuild`文件生成一个被Ant编译工具使用的文件。（如果你没有一个`build.fxbuid`文件，在Eclipse中创建一个新的**Java FX项目**，并且拷贝生成的文件过来。

1. 从项目的根目录下打开`build.fxbuild`。
2. 填写包含一个星号的字段。*对于MacOS：在应用程序标题中不能使用空格，因为好像会产生问题。*



3. 在Windows下**Packaging Format**选择`exe`，MacOS下选择`dmg`，Linux下选择`rpm`
4. 点击`Generate ant build.xml only`的连接（在右边可以找到）。



5. 验证是否创建一个新的`build`目录和文件`build.xml`

第2步 添加安装程序的图标

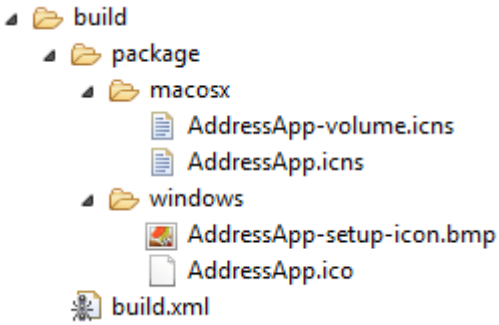
我们希望安装程序有一些好看的图标:

- AddressApp.ico 安装文件图标
- AddressApp-setup-icon.bmp 安装启动画面图标
- AddressApp.icns Mac安装程序图标

1. 在`build`目录下创建下面的子目录:

- `build/package/windows` (只用于Windows)
- `build/package/macos` (只用于macos)

1. 拷贝上面的相关图标到这些目录中，现在它应该看起来如下所示:



2. **重要**：图标的名称必须精确匹配`build.fxbuild`中指定的**Application的标题名**:

- `YourAppTitle.ico`
- `YourAppTitle-setup-icon.bmp`
- `YourAppTitle.icns`

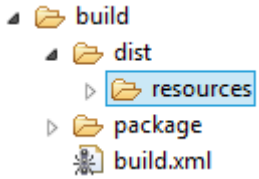
第3步 添加资源

我们的`resources`目录不能自动拷贝。我们必须手动添加它到build目录下:

1. 在`build`目录下创建下面的子目录:

- `build/dist`

1. 拷贝`resources`目录（包含我们应用的图标）到`build/dist`.



第4步 编辑build.xml包含图标

E(fx)clipse生成的`build/build.xml`文件（准备使用**Ant**执行）。我们的安装器图标和资源图像不能正常工作。

当e(fx)clipse没有告诉它包含其它资源，例如`resources`目录和上面添加的安装文件图标时，我们必须手动编辑`build.xml`文件。

打开`build.xml`文件，找到路径`fxant`。添加一行到`${basedir}`（将让我们安装器图标可用）。

build.xml - 添加"basedir"

```
<path id="fxant">
  <filelist>
    <file name="${java.home}\..\lib\ant-javafx.jar"/>
    <file name="${java.home}\lib\jfxrt.jar"/>
    <file name="${basedir}"/>
  </filelist>
</path>
```

找到块`fx:resources id="appRes"`，文件的更下面位置。为`resources`添加一行：

build.xml - 添加"resources"

```
<fx:resources id="appRes">
    <fx:fileset dir="dist" includes="AddressApp.jar"/>
    <fx:fileset dir="dist" includes="libs/*"/>
    <fx:fileset dir="dist" includes="resources/**"/>
</fx:resources>
```

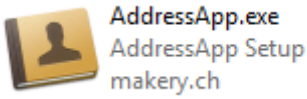
有时候，版本数不能添加到`fx:application`中，使得安装器总是缺省的版本`1.0`（在注释中很多人指出这个问题）。为了修复它，手动添加版本号（感谢Marc找到解决办法）。 解决):

build.xml - 添加 “version”

```
<fx:application id="fxApplication"
    name="AddressApp"
    mainClass="ch.makery.address.MainApp"
    version="1.0"
/>
```

现在，我们已经能够使用ant编译运行`build.xml`了。这将会生成一个可运行的项目jar文件。但是我们希望更进一步，创建一个很好的安装器。

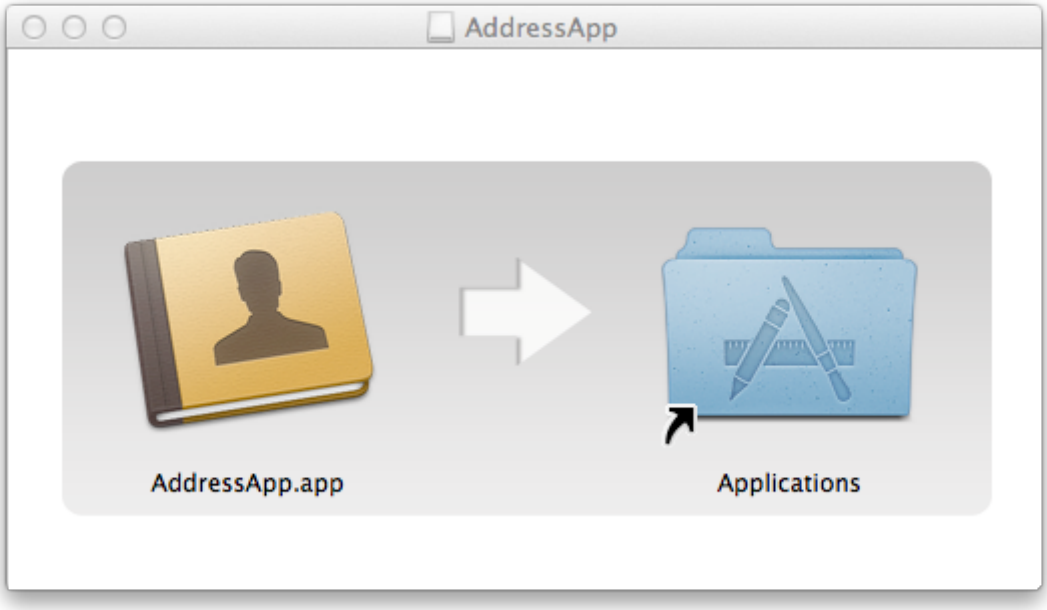
第5步（Windows） - Windows exe安装器



使用**Inno Setup**，我们能为我们的应用程序创建一个单独`.exe`文件的Windows安装器。生成的`.exe`执行用户级别的安装（无需管理员权限）。也创建一个快捷方式（菜单和桌面）。

1. 下载Inno Setup 5以后版本，安装Inno程序到你的计算机上。我们的Ant脚本将使用它自动生成安装器。
2. 告诉Windows Inno程序的安装路径（例如：`C:\Program Files (x86)\Inno Setup 5`）。添加Inno安装路径到Path环境变量中。如果你不知道哪里可以找到它，阅读Windows中如何设置路径和环境变量.
3. 重启Eclipse，并且继续第6步。

第5步（MAC） - MacOS dmg安装器



为了创建Mac OS `dmg` 拖放安装器，不需要任何的要求。

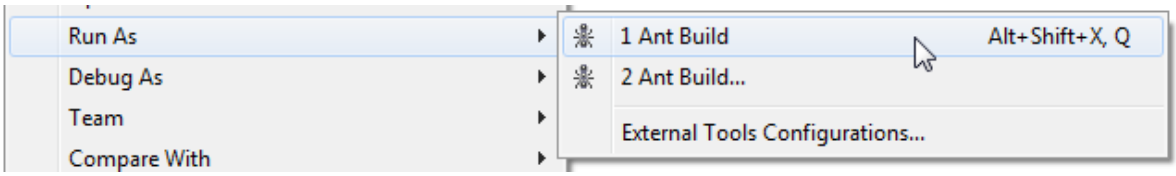
注意：为了让安装器映像能工作，它的名称必须与应用名称相同。

第5步（Linux等） Linux rpm安装器

其它打包选项（Windows的`msi`，Linux的`rpm`）参考本地打包博客 或者本oracle 文档.

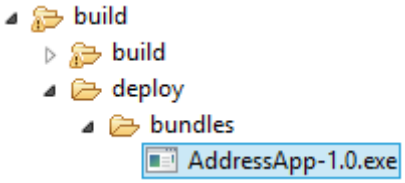
第6步 运行build.xml

最后一步，我们使用ant运行`build.xml`，*右击* `build.xml` 文件| *Run As* | *Ant Build*。



编译将**运行一会**（在我的计算机上大概1分钟）。

如果一切都成功，你应该在`build/deploy/bundles`目录下找到本地打包。Windows版本看起来如下所示：



文件`AddressApp-1.0.exe`可以用作为单个文件安装应用。该安装程序将拷贝打包到`C:/Users/[yourname]/AppData/Local/AddressApp`目录下。