

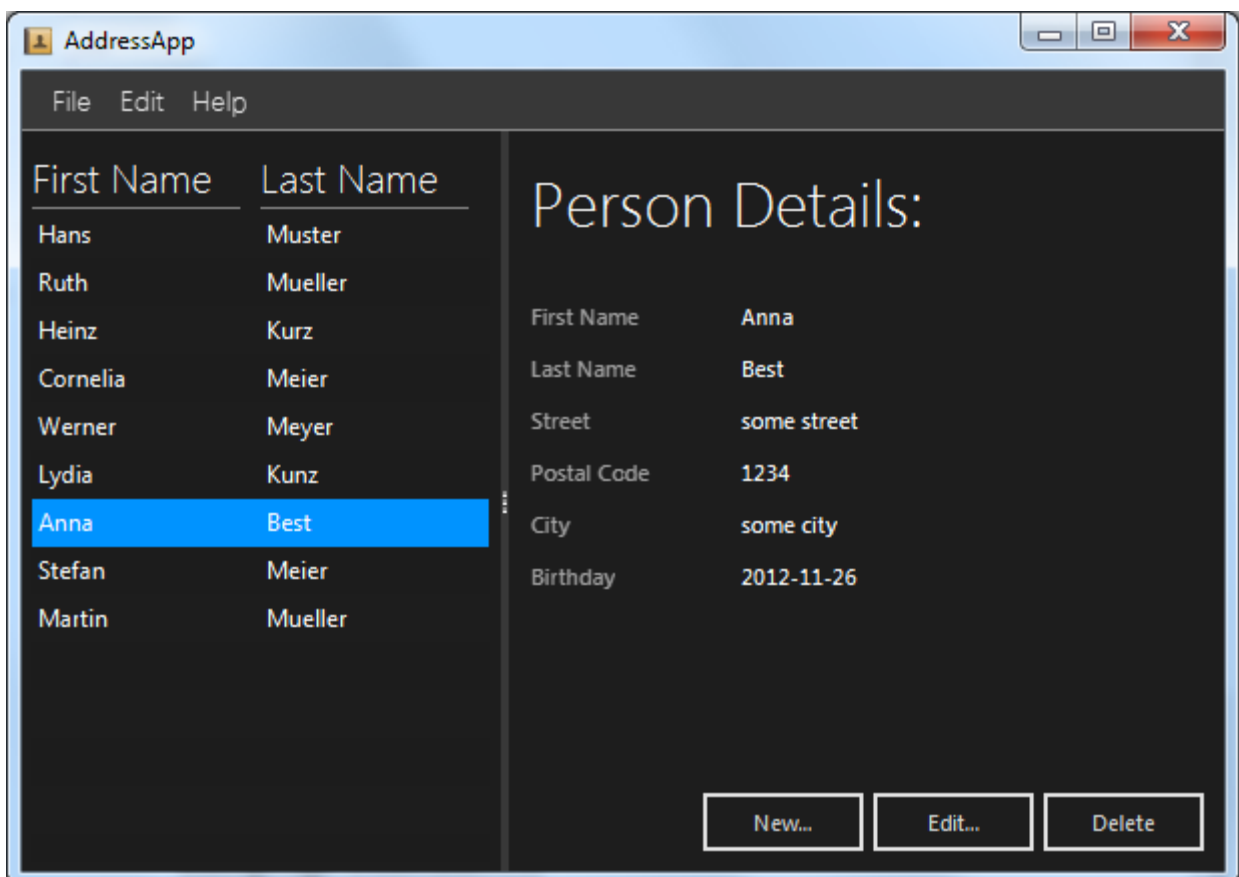
Tutorial JavaFX 8 (Español)

Sep 17, 2014

JavaFX proporciona a los desarrolladores de Java una nueva plataforma gráfica. JavaFX 2.0 se publicó en octubre del 2011 con la intención de reemplazar a Swing en la creación de nuevos interfaces gráficos de usuario (IGU). Cuando empecé a enseñar JavaFX en 2011 era una tecnología muy incipiente todavía. No había libros sobre JavaFX que fueran **adecuados para estudiantes de programación novatos**, así es que empecé a escribir una serie de tutoriales muy detallados sobre JavaFX.



El tutorial te guía a lo largo del diseño, programación y publicación de una aplicación de contactos (libreta de direcciones) mediante JavaFX. Este es el aspecto que tendrá la aplicación final:



Lo que aprenderás

- Creación de un nuevo proyecto JavaFX
- Uso de Scene Builder para diseñar la interfaz de usuario
- Estructuración de una aplicación según el patrón MVC (Modelo, Vista, Controlador)
- Uso de `ObservableList` para la actualización automática de la interfaz de usuario
- Uso de `TableView` y respuesta a cambios de selección en la tabla
- Creación de un diálogo personalizado para editar personas
- Validación de la entrada del usuario
- Aplicación de estilos usando CSS
- Persistencia de datos mediante XML
- Guardado del último archivo abierto en las preferencias de usuario
- Creación de un gráfico JavaFX para mostrar estadísticas
- Despliegue de una aplicación JavaFX nativa

Después de completar esta serie de tutoriales deberías estar preparado para desarrollar aplicaciones sofisticadas con JavaFX.

Como usar este tutorial

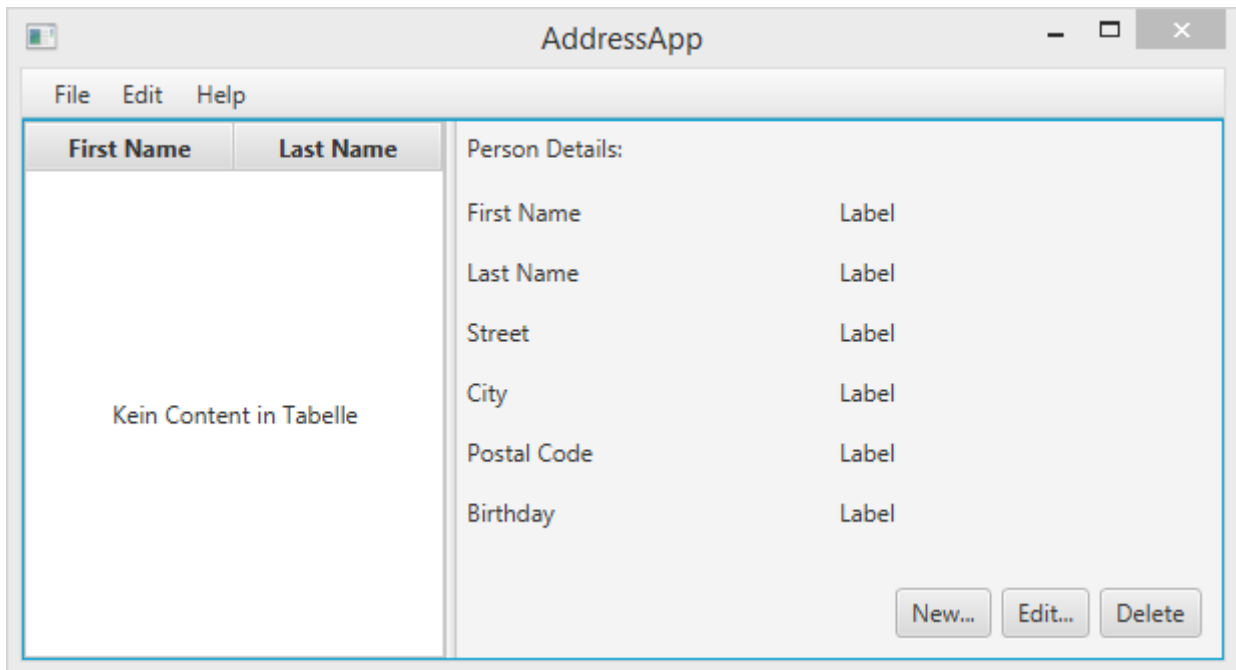
Hay dos formas de utilizarlo

- **máximo-aprendizaje:** Crea tu propio proyecto JavaFX desde cero.
- **máxima-rápidez:** Importa el código fuente de una parte del tutorial en tu entorno de desarrollo favorito (es un proyecto Eclipse, pero puedes usar otros entornos, como Netbeans, con ligeras modificaciones). Después revisa el tutorial para entender el código. Este enfoque también resulta útil si te quedas atascado en la creación de tu propio código.

¡ Espero que te diviertas y aprendas mucho ! Empieza en [Part 1: Scene Builder](#).

Tutorial JavaFX 8 - Parte 1: Scene Builder

Sep 17, 2014



Contenidos en Parte 1

- Familiarizándose con JavaFX
- Crear y empezar un proyecto JavaFX
- Uso de Scene builder para diseñar la interfaz de usuario
- Estructura básica de una aplicación mediante el patrón Modelo Vista Controlador (MVC)

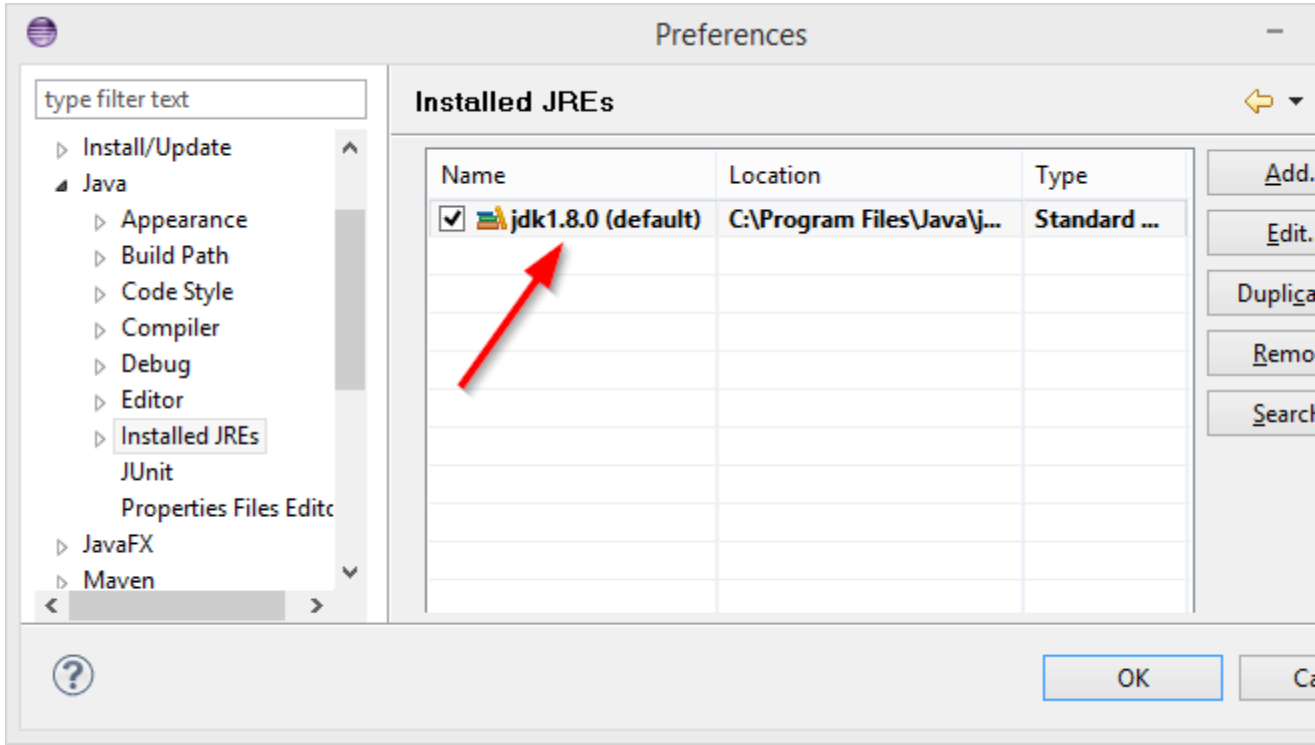
Prerequisitos

- Última versión de [Java JDK 8](#) (includes **JavaFX 8**).
- Eclipse 4.3 o superior con el plugin e(fx)clipse. La forma más sencilla de obtenerlo es descargarse la distribución preconfigurada desde [e\(fx\)clipse website](#). Como alternativa puedes usar un [sitio de actualización](#) para tu instalación de Eclipse.
- [Scene Builder 2.0](#) o superior

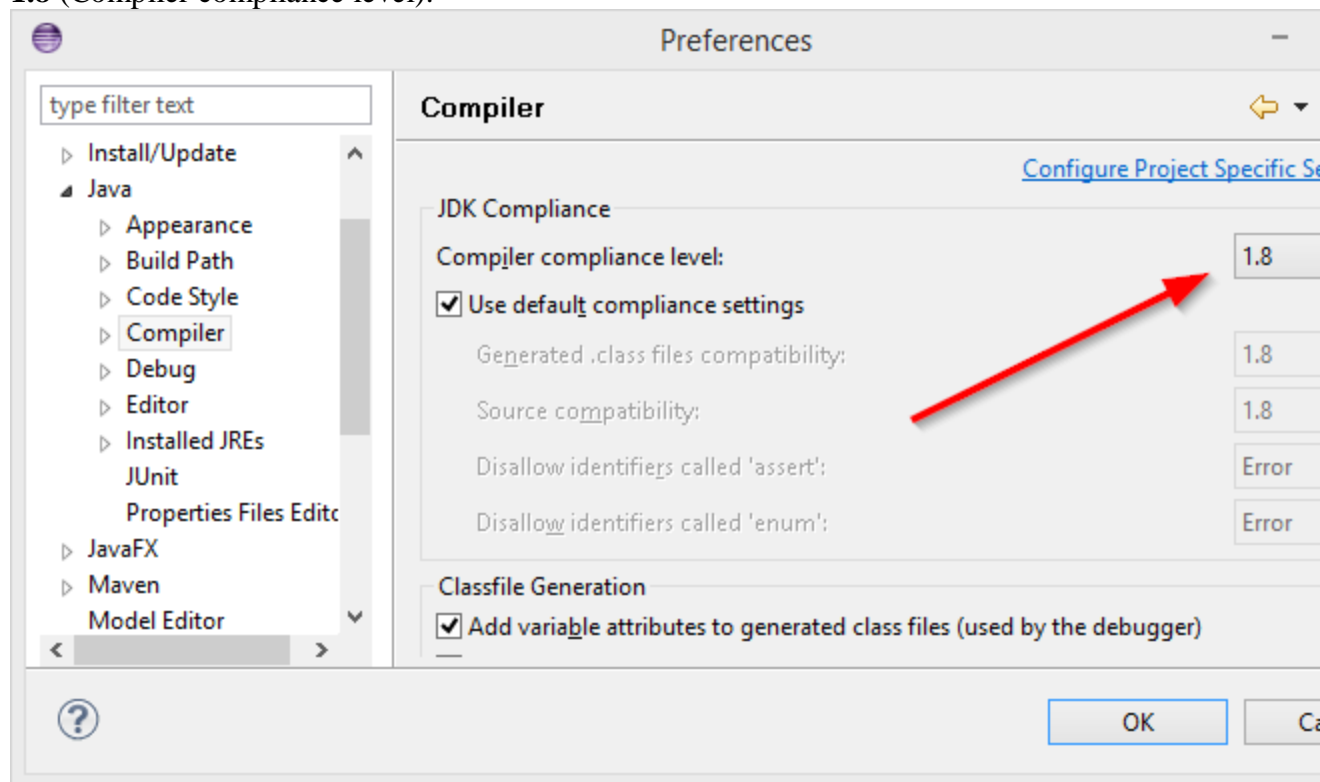
Configuración de Eclipse

Hay que indicarle a Eclipse que use JDK 8 y también dónde se encuentra el ejecutable del Scene Builder:

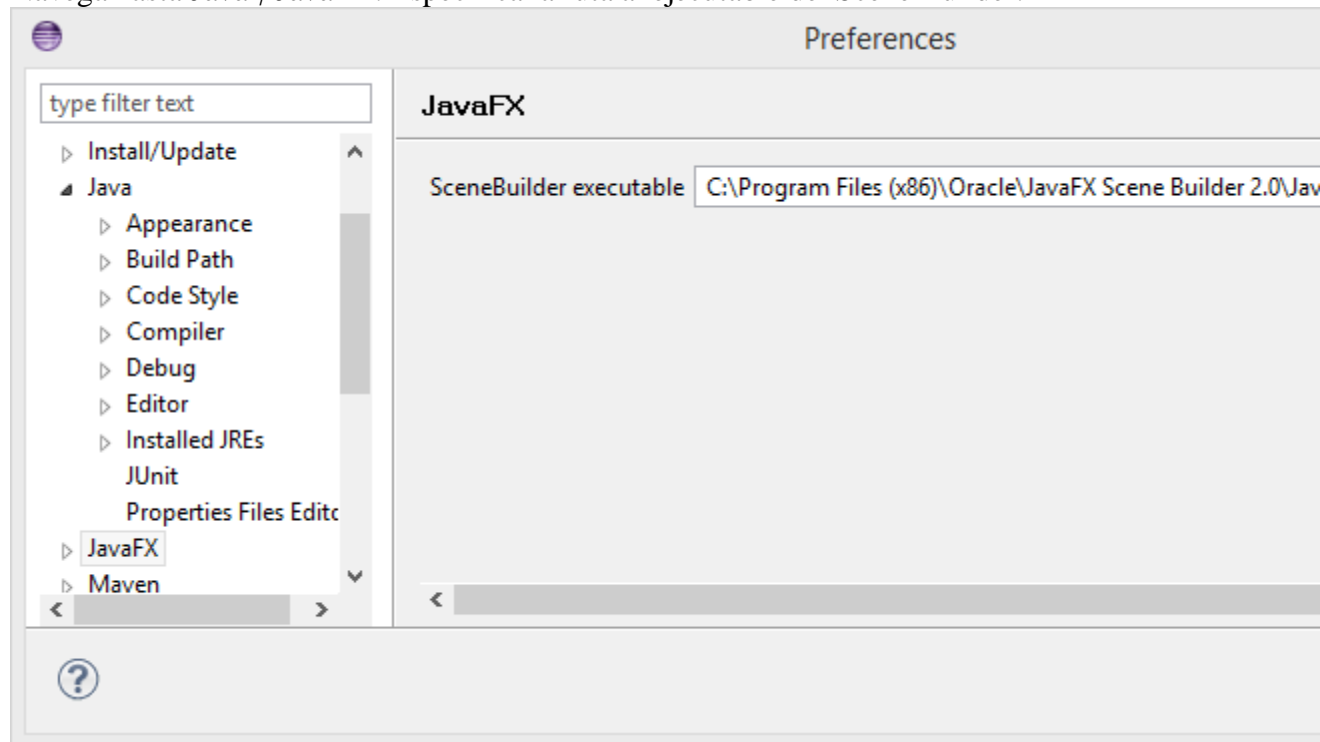
- [illegible]



4. Navega a *Java / Compiler*. Establece el **nivel de cumplimiento del compilador en 1.8** (Compiler compliance level).



5. Navega hasta *Java / JavaFX*. Especifica la ruta al ejecutable del Scene Builder.



Enlaces útiles

Te podría interesar mantener los siguientes enlaces:

- [Java 8 API](#) - Documentación (JavaDoc) de las clases estándar de Java
- [JavaFX 8 API](#) - Documentación de las clases JavaFX
- [ControlsFX API](#) - Documentación para el proyecto [ControlsFX](#), el cual ofrece controles JavaFX adicionales
- [Oracle's JavaFX Tutorials](#) - Tutoriales oficiales de Oracle sobre JavaFX

¡Y ahora, manos a la obra!

Crea un nuevo proyecto JavaFX

En Eclipse (con e(fx)clipse instalado) ve a *File / New / Other...* y elige *JavaFX Project*. Especifica el nombre del proyecto (ej. *AddressApp*) y aprieta *Finish*.

Borra el paquete *application* y su contenido si ha sido automáticamente creado.

Crea los paquetes

Desde el principio vamos a seguir buenos principios de diseño de software. Algunos de estos principios se traducen en el uso de la arquitectura denominada **Modelo-Vista-Controlador (MVC)**. Esta arquitectura promueve la división de nuestro código en tres apartados claramente definidos, uno por cada elemento de la arquitectura. En Java esta separación se logra mediante la creación de tres paquetes separados.

En el ratón hacemos clic derecho en la carpeta *src*, *New / Package*:

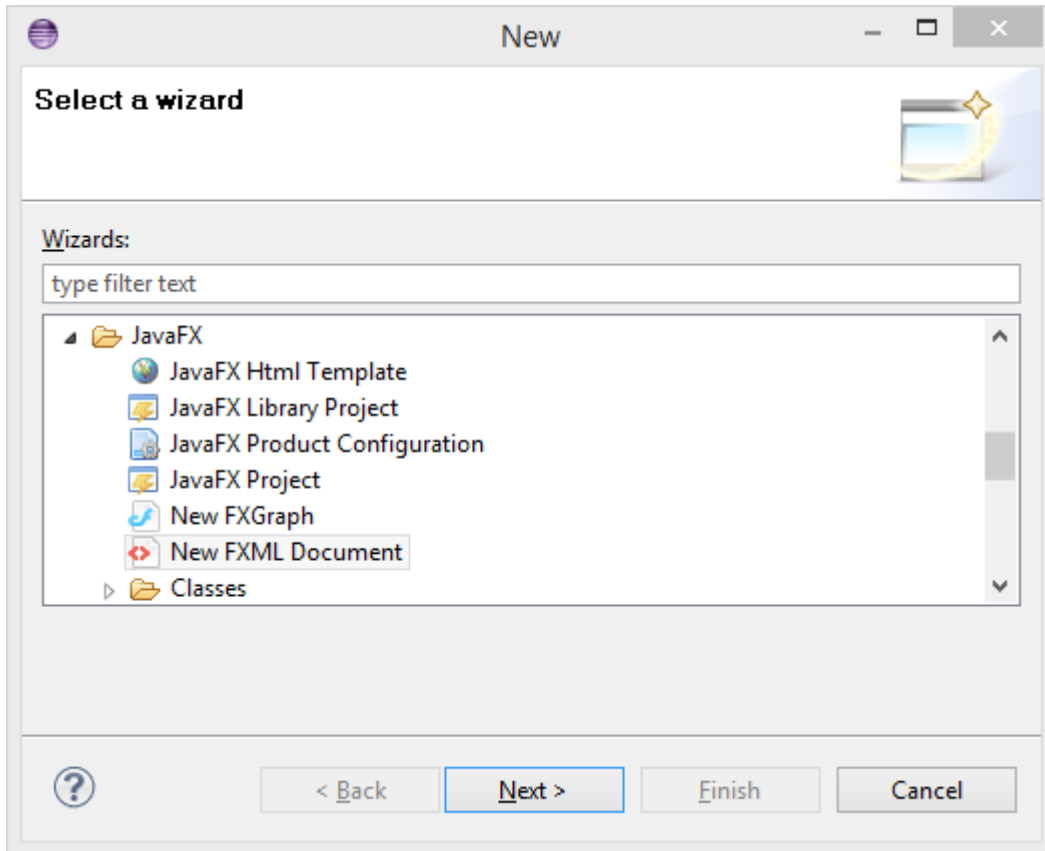
- `ch.makery.address` - contendrá la *mayoría* de clases de control (C)
- `ch.makery.address.model` - contendrá las clases del modelo (M)
- `ch.makery.address.view` - contendrá las vistas (V)

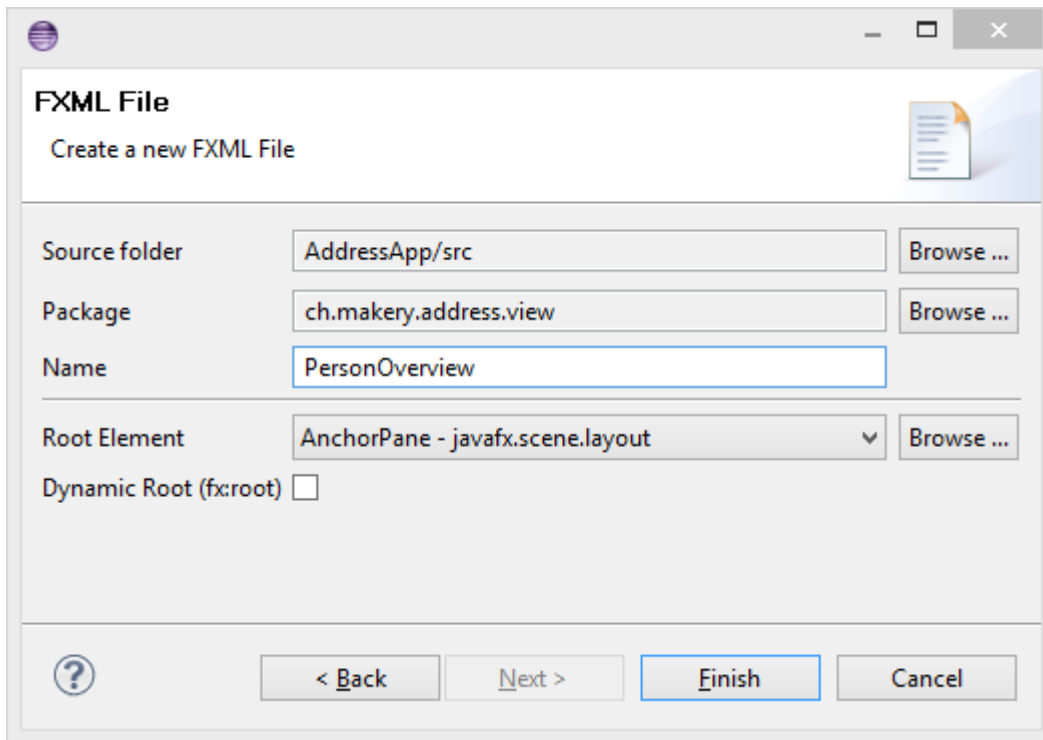
Nota: Nuestro paquete dedicado a las vistas contendrá también algunos controladores dedicados exclusivamente a una vista. Les llamaremos **controladores-vista**.

Crea el archivo FXML de diseño

Hay dos formas de crear la interfaz de usuario. Programándolo en Java o mediante un archivo XML. Si buscas en Internet encontrarás información relativa a ambos métodos. Aquí usaremos XML (archivo con la extensión .fxml) para casi todo. Encuentro más claro mantener el controlador y la vista separados entre sí. Además, podemos usar la herramienta de edición visual Scene Builder, la cual nos evita tener que trabajar directamente con el XML.

Haz clic derecho el paquete *view* y crea un nuevo archivo FXML (*New / Other / FXML / New FXML Document*) llamado **PersonOverview**.



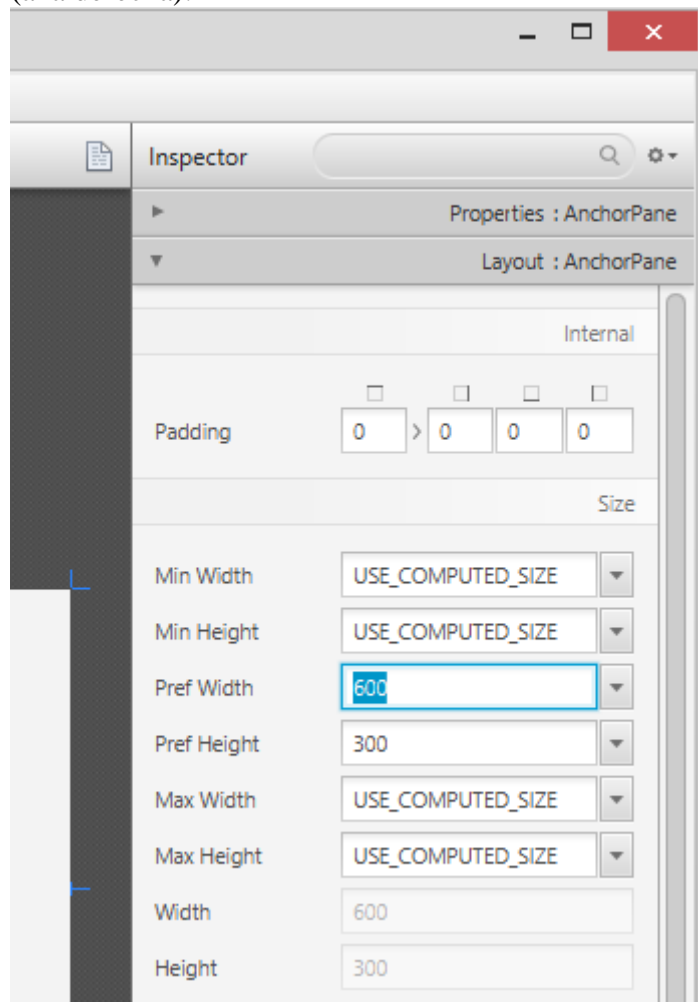


Diseño mediante Scene Builder

Nota: Si no puedes hacerlo funcionar, descarga las fuentes para esta parte del tutorial e inténtalo con el archivo fxml incluido.

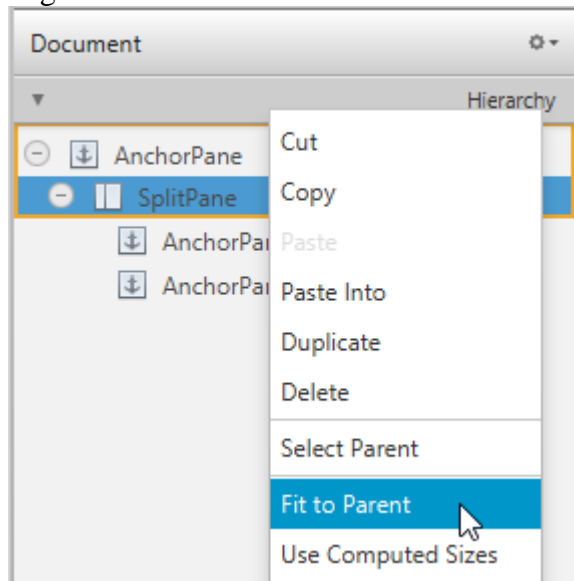
Haz clic derecho sobre `PersonOverview.fxml` y elige *Open with Scene Builder*. Ahora deberías ver el Scene Builder con un *AnchorPane* (visible en la jerarquía de componentes (herramienta Hierarchy) situada a la izquierda).

1. Selecciona el *AnchorPane* en tu jerarquía y ajusta el tamaño en el apartado Layout (a la derecha):

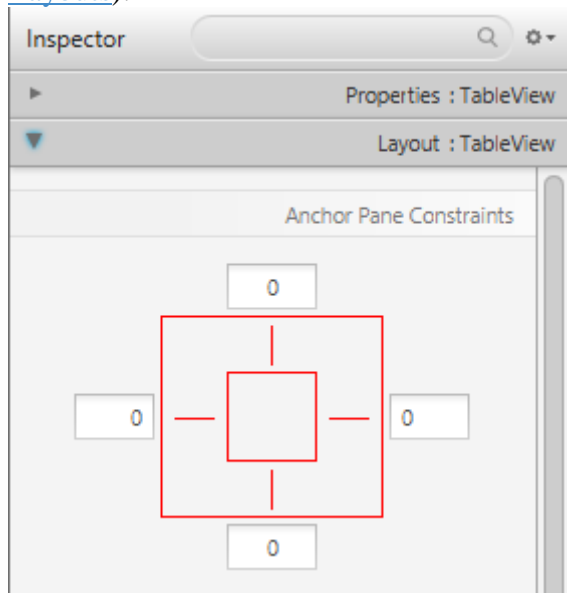


2. Añade un *SplitPane* (*Horizontal Flow*) arrastrándolo desde la librería (Library) al área principal de edición. Haz clic derecho sobre el *SplitPane* en la jerarquía y

elige *Fit to Parent*.

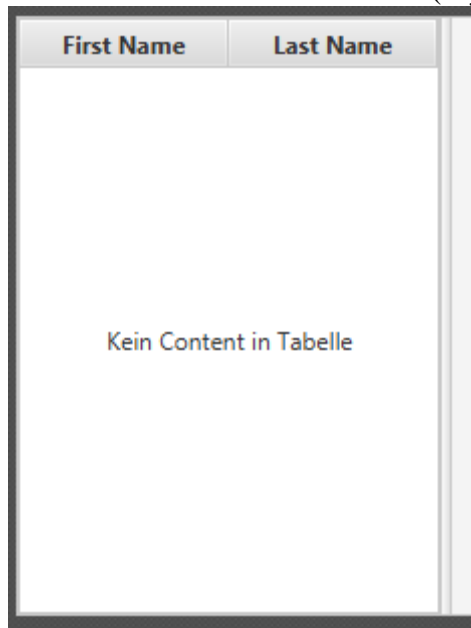


3. Arrastra un *TableView* (bajo *Controls*) al lado izquierdo del *SplitPane*. Selecciona la *TableView* (no una sola columna) y establece las siguientes restricciones de apariencia (Layout) para la *TableView*. Dentro de un *AnchorPane* siempre se pueden establecer anclajes (anchors) para los cuatro bordes ([más información sobre Layouts](#)).

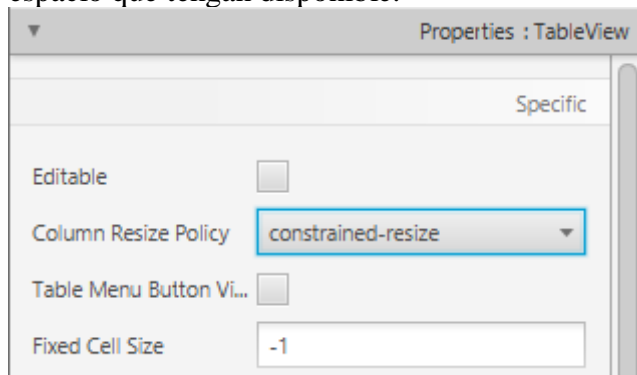


4. Ve al menú *Preview / Show Preview in Window* para comprobar si se visualiza correctamente. Intenta cambiar el tamaño de la ventana. La *TableView* debería ajustar su tamaño al tamaño de la ventana, pues está "anclada" a sus bordes.

5. Cambia el texto de las columnas (bajo Properties) a "First Name" y "Last Name".

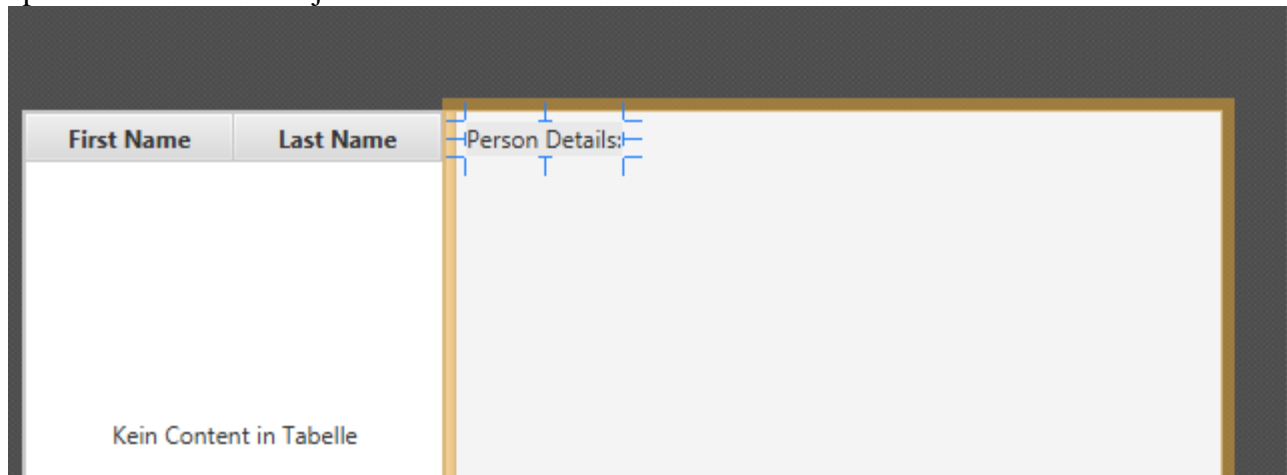


6. Selecciona la *TableView* y elige *constrained-resize* para la *Column Resize Policy* (en Properties). Esto asegura que las columnas usarán siempre todo el espacio que tengan disponible.

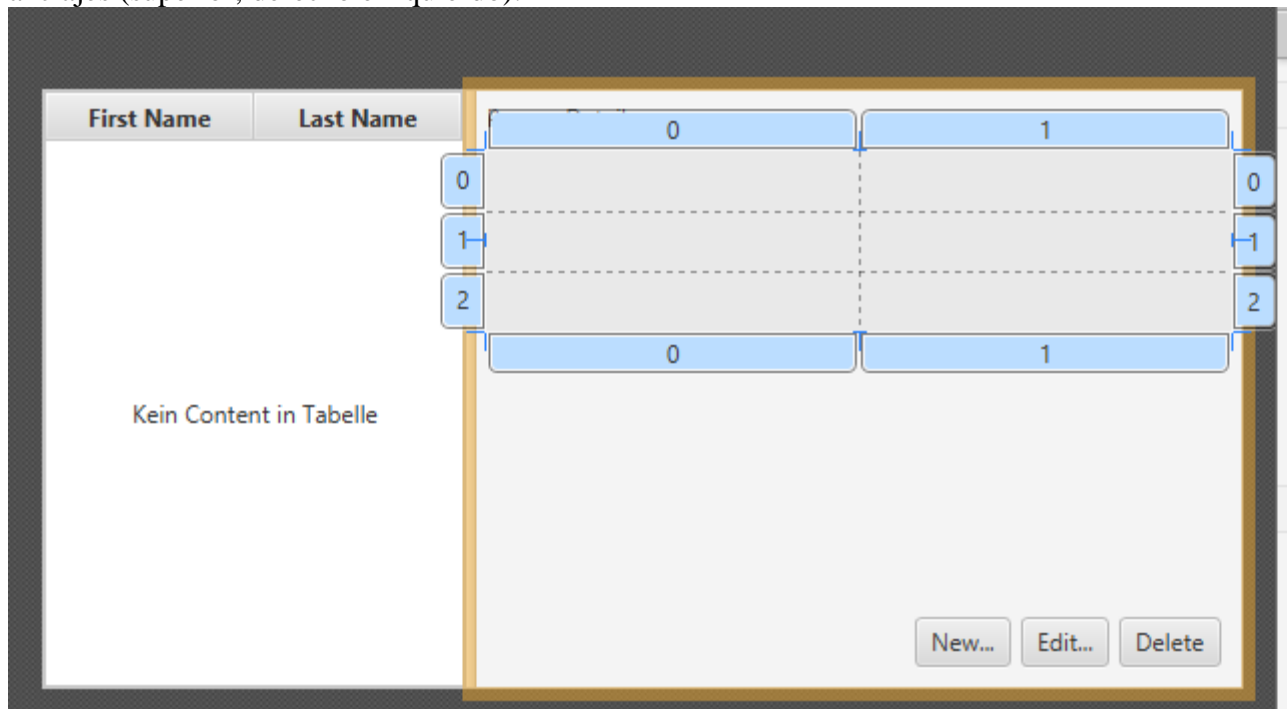


7. Añade una *Label* al lado derecho del *SplitPane* con el texto "Person Details" (truco: usa la búsqueda en la librería para encontrar el componente *Label*). Ajusta su

apariciencia usando anclajes.



8. Añade un GridPane* al lado derecho, selecciónalo y ajusta su apariencia usando anclajes (superior, derecho e izquierdo).



9. Añade las siguientes etiquetas (Label) a las celdas del GridPane .
Nota: Para añadir una fila al GridPane selecciona un número de fila existente (se volverá de color amarillo), haz clic derecho sobre el número de fila y elige "Add"

Row".

First Name	Last Name
Kein Content in Tabelle	

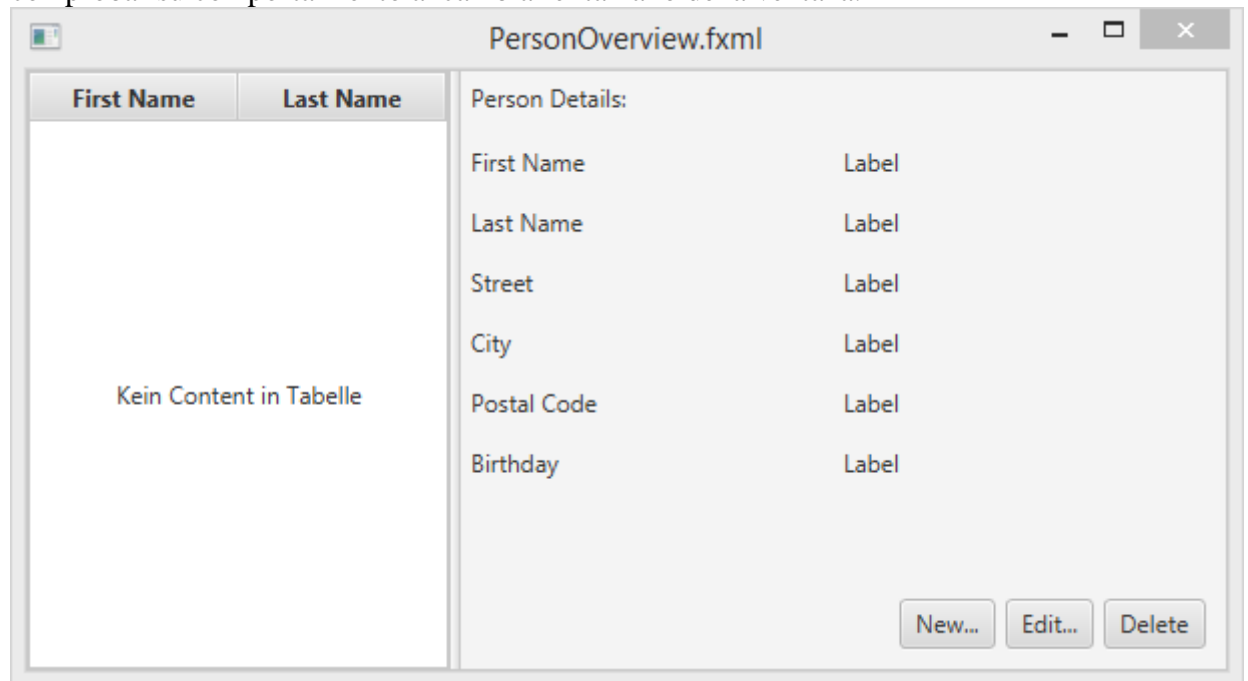
	0	1
0	First Name	Label
1	Last Name	Label
2	Street	Label
3	City	Label
4	Postal Code	Label
5	Birthday	Label

New... Edit... Delete

10. Añade 3 botones a la parte inferior. Truco: Selecciónalos todos, haz click derecho e invoca *Wrap In / HBox*. Esto los pondrá a los 3 juntos en un HBox. Podrías necesitar establecer un espaciado *spacing* dentro del HBox. Después, establece también anclajes (derecho e inferior) para que se mantengan en el lugar correcto.

New... Edit... Delete

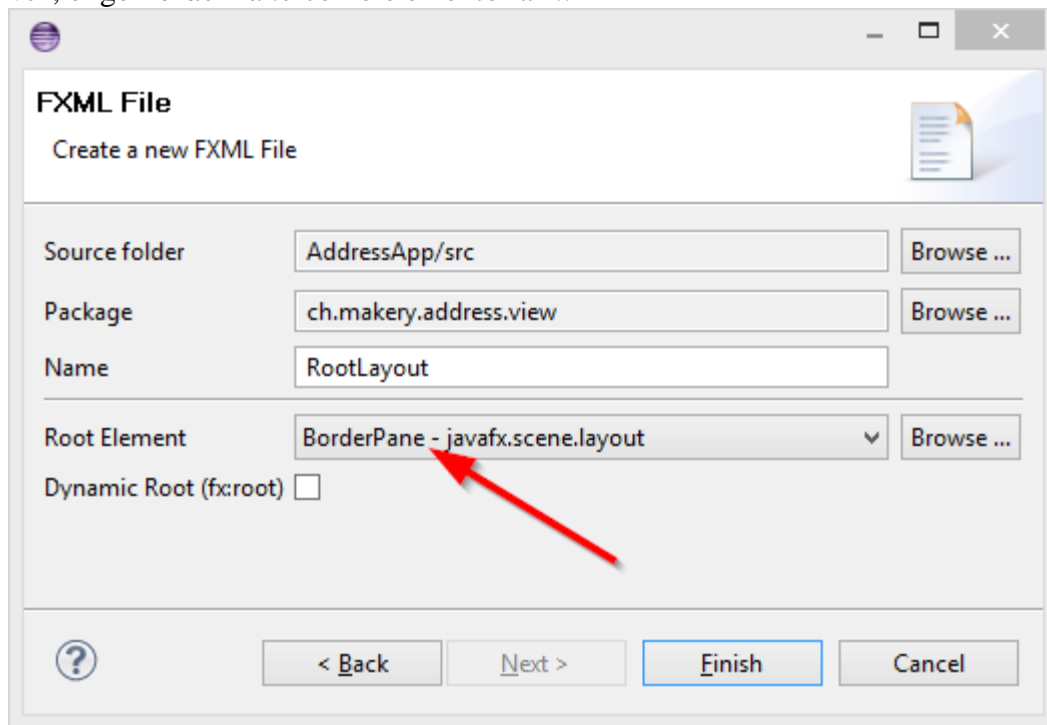
11. Ahora deberías ver algo parecido a lo siguiente. Usa el menú *Preview* para comprobar su comportamiento al cambiar el tamaño de la ventana.



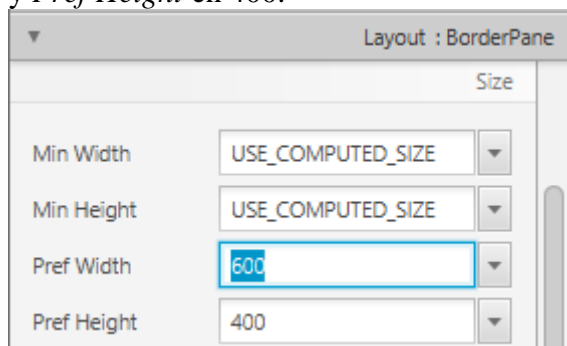
Crea la aplicación principal

Necesitamos otro archivo FXML para nuestra vista raíz, la cual contendrá una barra de menús y encapsulará la vista recién creada `PersonOverview.fxml`.

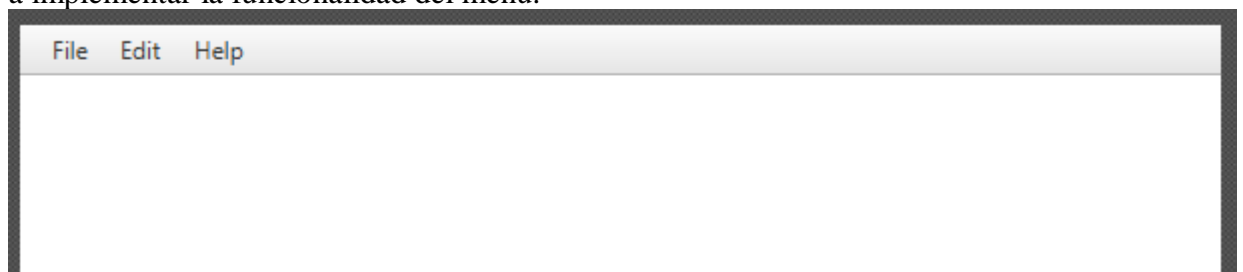
1. Crea otro archivo FXML dentro del paquete view llamado `RootLayout.fxml`. Esta vez, elige `BorderPane` como elemento raíz..



2. Abre `RootLayout.fxml` en el Scene Builder.
3. Cambia el tamaño del `BorderPane` con la propiedad *Pref Width* establecida en 600 y *Pref Height* en 400.



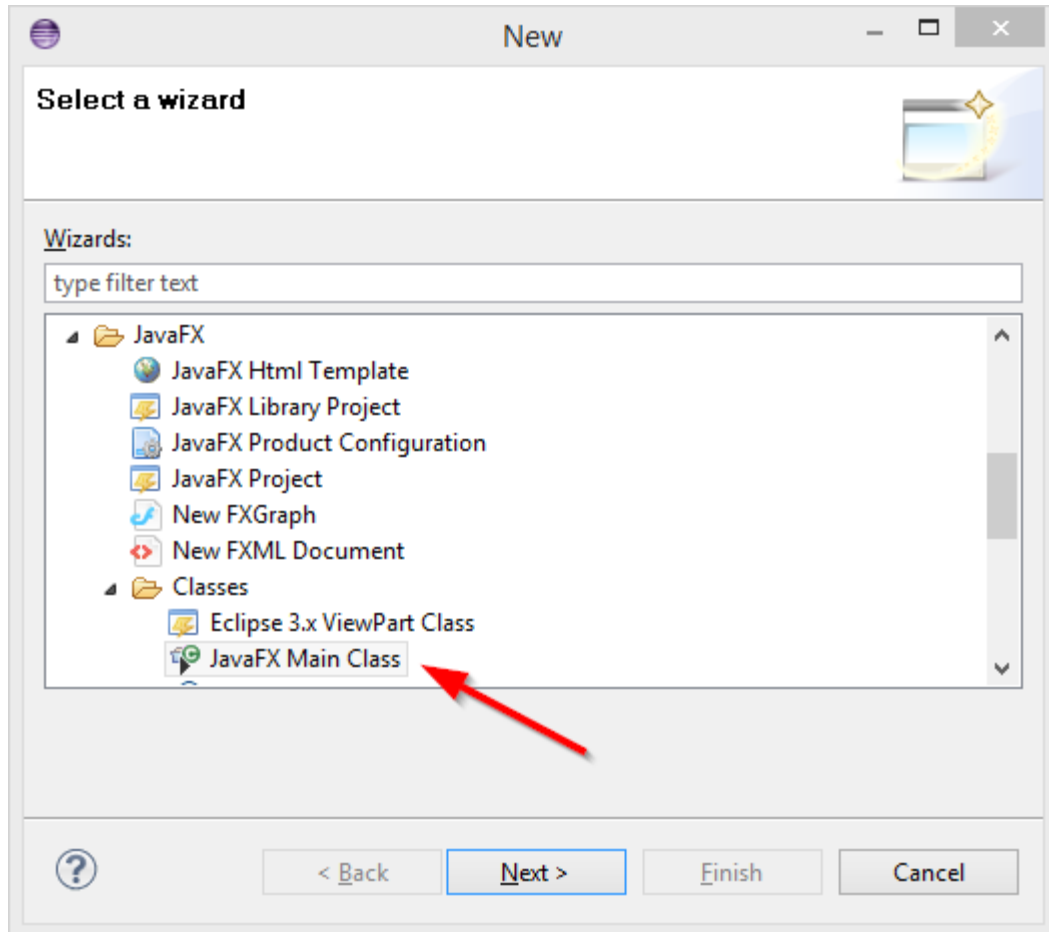
4. Añade una `MenuBar` en la ranura superior del `BorderPane`. De momento no vamos a implementar la funcionalidad del menú.



La clase principal en JavaFX

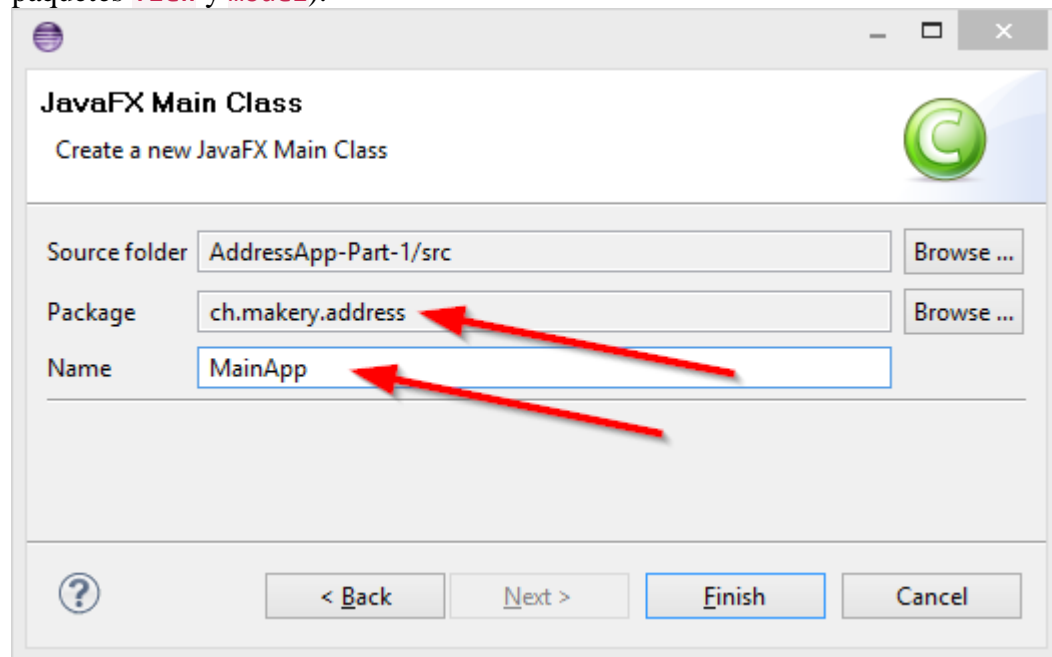
Ahora necesitamos crear la **clase java principal**, la cual iniciará nuestra aplicación mediante `RootLayout.fxml` y añadirá la vista `PersonOverview.fxml` en el centro.

1. Haz clic derecho en el proyecto y elige *New / Other / JavaFX / classes / JavaFX Main Class*.



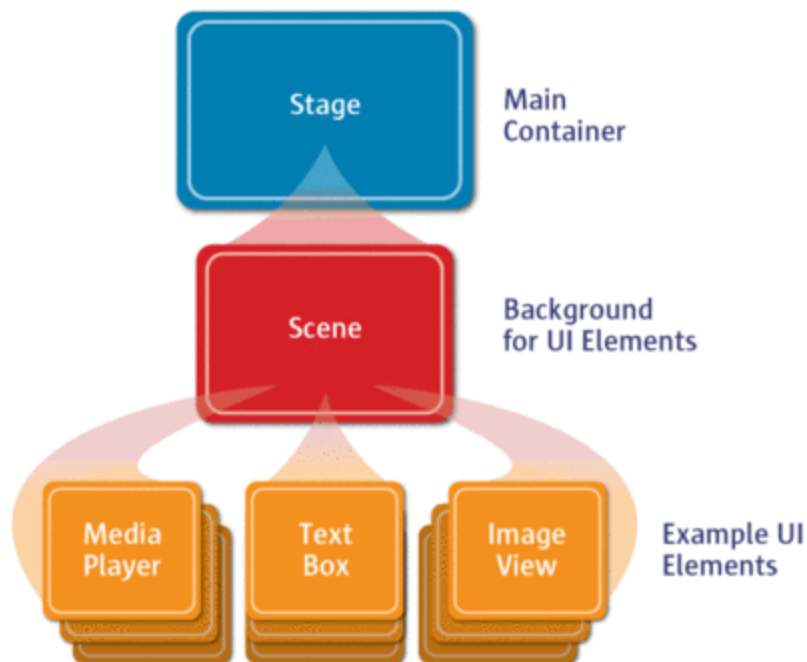
2. Llama a esta clase `MainApp` y ponla en el paquete de controladores `ch.makery.address` (nota: este es el paquete padre de los

paquetes `view` y `model`).



La clase generada (`MainApp.java`) extiende a la clase `Application` y contiene dos métodos. Esta es la estructura básica que necesitamos para ejecutar una Aplicación JavaFX. La parte más importante para nosotros es el método `start(Stage primaryStage)`. Este método es invocado automáticamente cuando la aplicación es lanzada desde el método `main`.

Como puedes ver, el método `start(...)` toma un `Stage` como parámetro. El gráfico siguiente muestra la estructura de cualquier aplicación JavaFX:



Fuente de la imagen: <http://www.oracle.com>

Es como una obra de teatro: El **Stage** (escenario) es el contenedor principal, normalmente una ventana con borde y los típicos botones para maximizar, minimizar o cerrar la ventana. Dentro del **Stage** se puede añadir una **Scene** (escena), la cual puede cambiarse dinámicamente por otra **Scene**. Dentro de un **Scene** se añaden los nodos JavaFX, tales como **AnchorPane**, **TextBox**, etc.

Para tener más información puedes consultar [Working with the JavaFX Scene Graph](#).

Abre el archivo **MainApp.java** y reemplaza todo su código con el código siguiente:

```
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);
}
}

```

Los diferentes comentarios deben darte pistas sobre lo que hace cada parte del código.

Si ejecutas la aplicación ahora, verás algo parecido a la captura de pantalla incluida al principio de este artículo.

Problemas frecuentes

Si JavaFX no encuentra un archivo `fxml` puedes obtener el siguiente mensaje de error

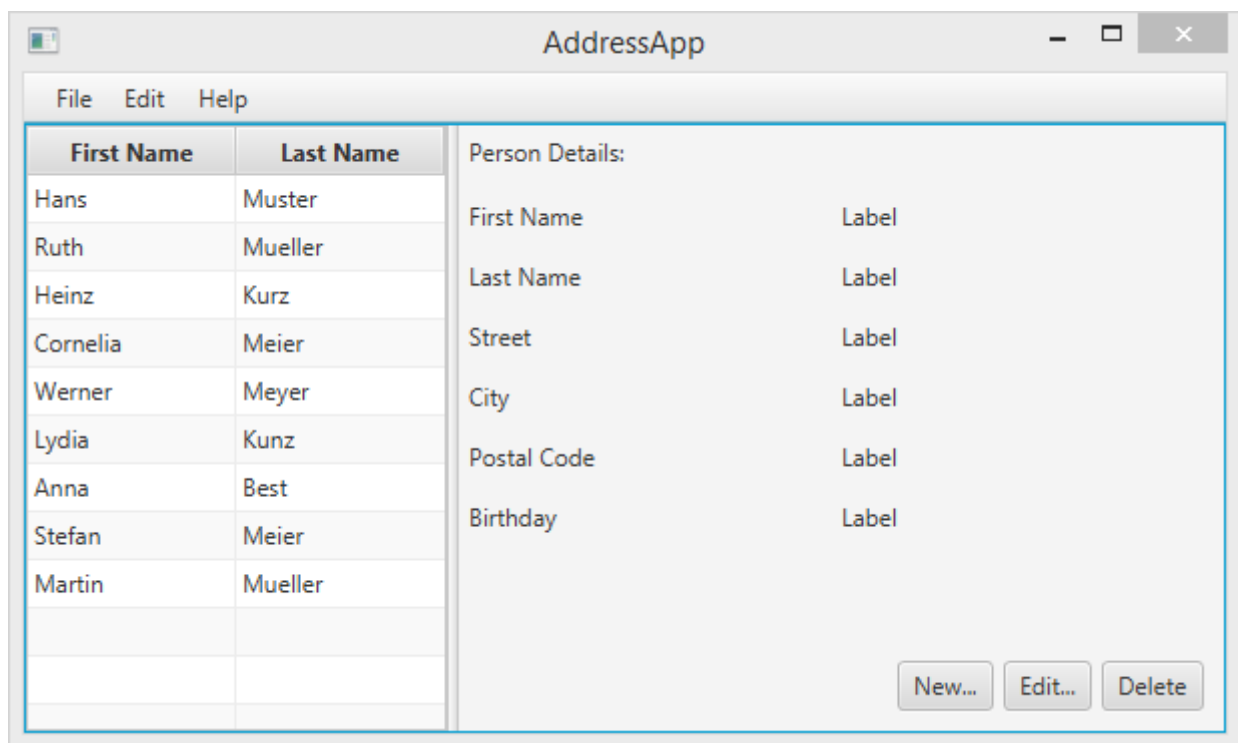
```
java.lang.IllegalStateException: Location is not set.
```

Para resolverlo comprueba otra vez que no hayas escrito mal el nombre de tus archivos `fxml`.

Si todavía no te funciona, descárgate el código de esta parte del tutorial y pruébalo con el fxml proporcionado.

Tutorial JavaFX 8 - Parte 2: Modelo y TableView

Sep 17, 2014



Contenidos en Parte 2

- Creación de una clase para el **modelo**
- Uso del modelo en una **ObservableList**
- Visualización del modelo mediante **TableView** y **Controladores**

Crea la clase para el Modelo

Necesitamos un modelo para contener la información sobre los contactos de nuestra agenda. Añade una nueva clase al paquete encargado de contener los modelos (`ch.makery.address.model`) denominada `Person`. La clase `Person` tendrá atributos (instancias de clase) para el nombre, la dirección y la fecha de nacimiento. Añade el código siguiente a la clase. Explicaré detalles de JavaFX después del código.

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(199
9, 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;
    }
}

```

Explicación del código

- Con JavaFX es habitual usar **Propiedades** para todos los atributos de un clase usada como modelo. Una **Propiedad** permite, entre otras cosas, recibir notificaciones automáticamente cuando el valor de una variable cambia (por ejemplo si cambia **lastName**. Esto ayuda a mantener sincronizados la vista y los datos. Para aprender más sobre **Propiedades** lee [Using JavaFX Properties and Binding](#).
- **LocalDate**, el tipo que usamos para especificar la fecha de nacimiento (**birthday**) es parte de la nueva [API de JDK 8 para la fecha y la hora](#).

Una lista de personas

Los principales datos que maneja nuestra aplicación son una colección de personas. Vamos a crear una lista de objetos de tipo **Person** dentro de la clase principal **MainApp**. El resto de controladores obtendrá luego acceso a esa lista central dentro de **MainApp**.

Lista observable (ObservableList)

Estamos clases gráficas de JavaFX que necesitan ser informadas sobre los cambios en la lista de personas. Esto es importante, pues de otro modo la vista no estará sincronizada con los datos. Para estos fines, JavaFX ha introducido nuevas [clases de colección](#).

De esas colecciones, necesitamos la denominada **ObservableList**. Para crear una nueva **ObservableList**, añade el código siguiente al principio de la clase **MainApp**. También añadiremos un constructor para crear datos de ejemplo y un método de consulta (get) público:

MainApp.java

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

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

/**
 * 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 ...
```

PersonOverviewController

Finalmente vamos a añadir datos a nuestra table. Para ello necesitaremos un controlador específico para la vista `PersonOverview.fxml`.

1. Crea una clase normal dentro del paquete **view** denominado `PersonOverviewController.java`. (Debemos ponerlo en el mismo paquete que `PersonOverview.fxml` o el Scene Builder no lo encontrará, al menos no en la versión actual).
2. Añadiremos algunos atributos para acceder a la tabla y las etiquetas de la vista. Estos atributos irán precedidos por la anotación `@FXML`. Esto es necesario para que la vista tenga acceso a los atributos y métodos del controlador, incluso aunque sean privados. Una vez definida la vista en fxml, la aplicación se encargará de rellenar automáticamente estos atributos al cargar el fxml. Así pues, añade el código siguiente:

Nota: acuérdate siempre de importar `javafx`, NO AWT ó 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().l
astNameProperty());
}

/**
 * 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());
}
}

```

Este código necesita cierta explicación:

- Los campos y métodos donde el archivo fxml necesita acceso deben ser anotados con `@FXML`. En realidad, sólo si son privados, pero es mejor tenerlos privados y marcarlos con la anotación.
- El método `initialize()` es invocado automáticamente tras cargar el fxml. En ese momento, todos los atributos FXML deberían ya haber sido inicializados..
- El método `setCellValueFactory(...)` que aplicamos sobre las columnas de la tabla se usa para determinar qué atributos de la clase `Person` deben ser usados para cada columna particular. La flecha `->` indica que estamos usando una característica de Java 8 denominada *Lambdas*. Otra opción sería utilizar un [PropertyValueFactory](#), pero entonces no ofrecería seguridad de tipo (type-safe).

Conexión de MainApp con PersonOverviewController

El método `setMainApp(...)` debe ser invocado desde la clase `MainApp`. Esto nos da la oportunidad de acceder al objeto `MainApp` para obtener la lista de `Person` y otras cosas. Sustituye el método `showPersonOverview()` con el código siguiente, el cual contiene dos líneas adicionales:

MainApp.java - nuevo método showPersonOverview()

```

/**
 * Shows the person overview inside the root layout.
 */
public void showPersonOverview() {
    try {

```

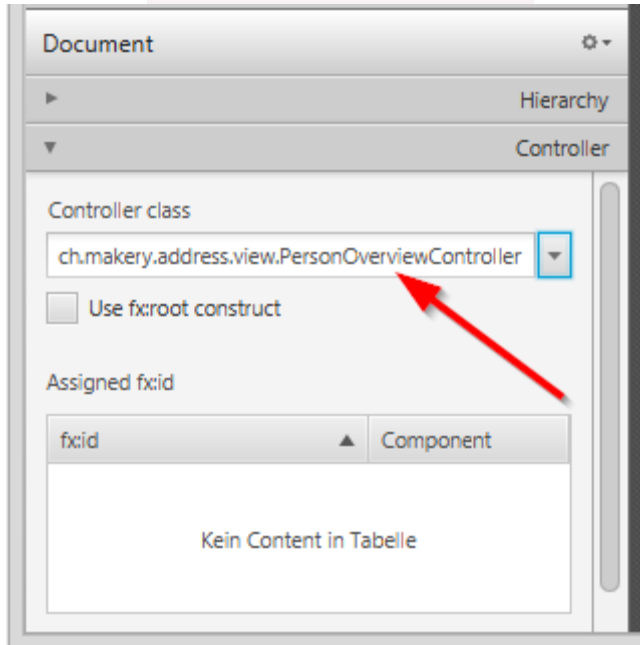
```
// Load person overview.  
FXMLLoader loader = new FXMLLoader();  
loader.setLocation(MainApp.class.getResource("view/PersonOverview.fxml"));  
AnchorPane personOverview = (AnchorPane) loader.load();  
  
// Set person overview into the center of root layout.  
rootLayout.setCenter(personOverview);  
  
// Give the controller access to the main app.  
PersonOverviewController controller = loader.getController();  
controller.setMainApp(this);  
  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}
```

Vincular la vista al controlador

¡Ya casi lo tenemos! Pero falta un detalle: no le hemos indicado a la vista declarada en `PersonOverview.fxml` cuál es su controlador y que elemento hacer corresponder to cada uno de los atributos en el controlador.

1. Abre `PersonOverview.fxml` en *SceneBuilder*.

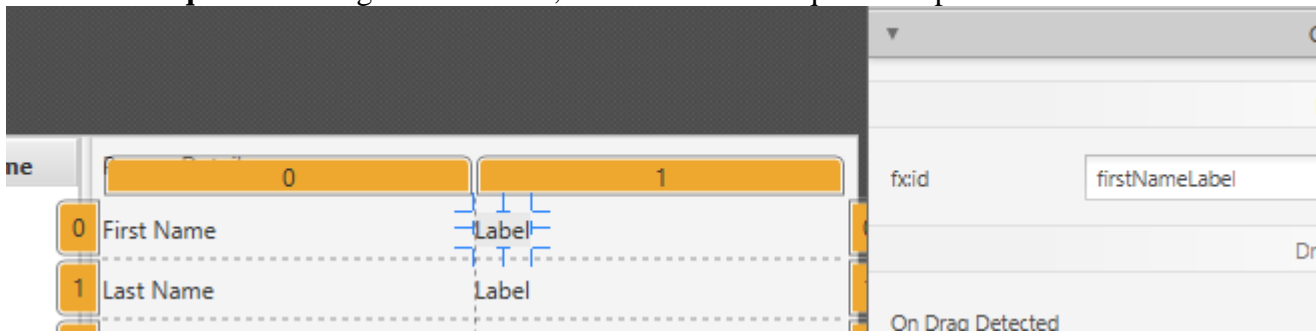
2. Abre la sección *Controller* en el lado izquierdo y selecciona `PersonOverviewController` como **controlador**.



3. Selecciona `TableView` en la sección *Hierarchy* y en el apartado *Code* escribe `personTable` en la propiedad **fx:id**.



4. Haz lo mismo para las columnas, poniendo `firstNameColumn` y `lastNameColumn` como sus **fx:id** respectivamente.
5. Para **cada etiqueta** en la segunda columna, introduce el **fx:id** que corresponda.



6. Importante: En Eclipse **refresca el proyecto AddressApp** (tecla F5). Esto es necesario porque a veces Eclipse no se da cuenta de los cambios realizados desde el Scene Builder.

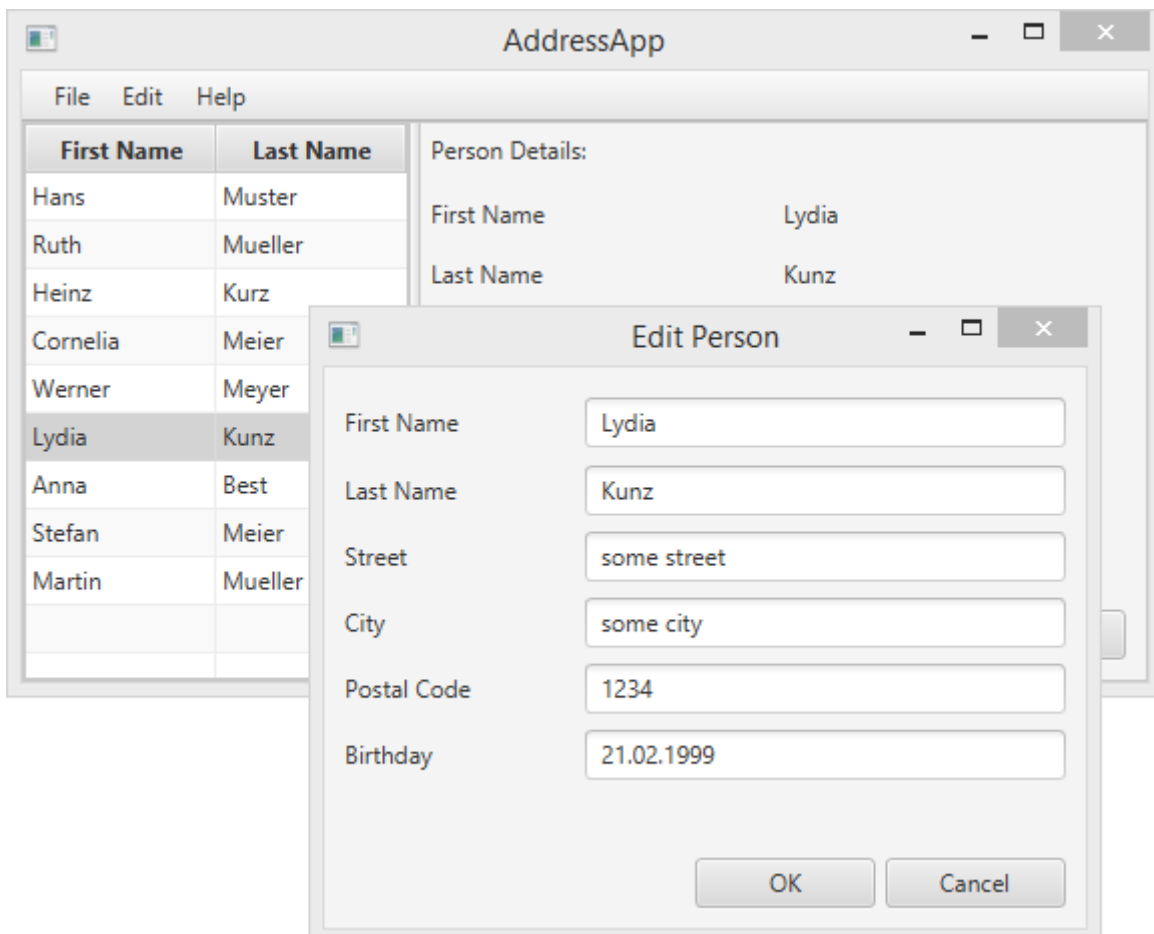
Inicia la aplicación

Ahora, cuando ejecutes la aplicación, deberías obtener algo parecido a la captura de pantalla incluida al principio de este artículo.

Enhorabuena!

Tutorial JavaFX 8 - Parte 3: Interacción con el usuario

Sep 17, 2014



Contenidos en Parte 3

- **Respuesta a cambios en la selección** dentro de la tabla.
 - Añade funcionalidad de los botones **añadir**, **editar**, y **borrar**.
 - Crear un **diálogo emergente** (popup dialog) a medida para editar un contacto.
 - **Validación de la entrada del usuario**.
-

Respuesta a cambios en la selección de la Tabla

Todavía no hemos usado la parte derecha de la interfaz de nuestra aplicación. La intención es usar esta parte para mostrar los detalles de la persona seleccionada por el usuario en la tabla.

En primer lugar vamos a añadir un nuevo método dentro de `PersonOverviewController` que nos ayude a rellenar las etiquetas con los datos de una sola persona.

Crea un método llamado `showPersonDetails(Person person)`. Este método recorrerá todas las etiquetas y establecerá el texto con detalles de la persona usando `setText(...)`. Si en vez de una instancia de `Person` se pasa `null` entonces las etiquetas deben ser borradas.

`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("");
    }
}

```

Convierte la fecha de nacimiento en una cadena

Te darás cuenta de que no podemos usar el atributo `birthday` directamente para establecer el valor de una `Label` porque se requiere un `String`, y `birthday` es de tipo `LocalDate`. Así pues necesitamos convertir `birthday` de `LocalDate` a `String`.

En la práctica vamos a necesitar convertir entre `LocalDate` y `String` en varios sitios y en ambos sentidos. Una buena práctica es crear una clase auxiliar con métodos estáticos (`static`) para esta finalidad. Llamaremos a esta clase `DateUtil` y la ubicaremos una paquete separado denominado `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_P
ATTEN}
 * 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;
    }
}

```

Truco: Puedes cambiar el formato de la fecha cambiando el patrón `DATE_PATTERN`. Para conocer los diferentes tipos de formato consulta [DateTimeFormatter](#).

Utilización de la clase `DateUtil`

Ahora necesitamos utilizar la nueva clase `DateUtil` en el método `showPersonDetails` de `PersonOverviewController`. Sustituye el *TODO* que habíamos añadido con la línea siguiente:

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

Detecta cambios de selección en la tabla

Para enterarse de que el usuario ha seleccionado a una persona en la tabla de contactos, necesitamos escuchar los cambios. Esto se consigue mediante la implementación de un interface de JavaFX que se llama `ChangeListener` with one method called `changed(...)`. Este método solo tiene tres parámetros: `observable`, `oldValue`, y `newValue`.

En Java 8 la forma más elegante de implementar una interfaz con un único método es mediante una *lambda expression*. Añadiremos algunas líneas al método `initialize()` de `PersonOverviewController`. El código resultante se asemejará al siguiente:

`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))
;
}

```

Con `showPersonDetails(null);` borramos los detalles de una persona.

Con `personTable.getSelectionModel...` obtenemos la *selectedItemProperty* de la tabla de personas, y le añadimos un *listener*. Cuando quiera que el usuario seleccione a una persona en la table, nuestra *lambda expression* será ejecutada: se toma la persona recién seleccionada y se le pasa al método `showPersonDetails(...)` method.

Intenta **ejecutar tu aplicación** en este momento. Comprueba que cuando seleccionas a una persona, los detalles sobre esta son mostrados en la parte derecha de la ventana.

Si algo no funciona, puedes comparar tu clase `PersonOverviewController` con [PersonOverviewController.java](#).

El botón de borrar (Delete)

Nuestro interfaz de usuario ya contiene un botón de borrar, pero sin funcionalidad. Podemos seleccionar la acción a ejecutar al pulsar un botón desde el *Scene Builder*. Cualquier método de nuestro controlador anotado con `@FXML` (o declarado como *public*) es accesible desde *Scene Builder*. Así pues, empecemos añadiendo el método de borrado al final de nuestra clase `PersonOverviewController`:

`PersonOverviewController.java`

```

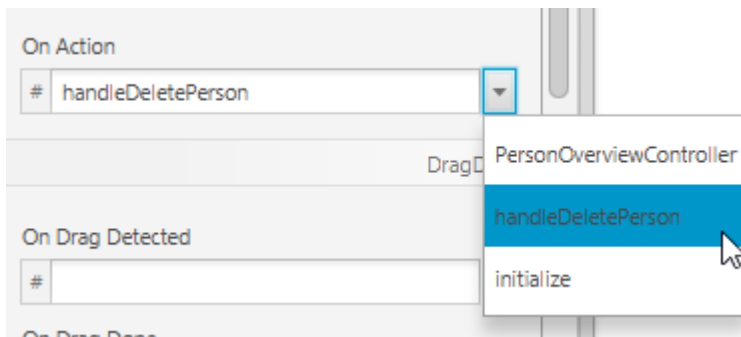
/**
 * Called when the user clicks on the delete button.
 */
@FXML

```



```
private void handleDeletePerson() {
    int selectedIndex = personTable.getSelectionModel().getSelectedIndex();
    personTable.getItems().remove(selectedIndex);
}
```

Ahora, abre el archivo `PersonOverview.fxml` en el *SceneBuilder*. Selecciona el botón *Delete*, abre el apartado *Code* y pon `handleDeletePerson` en el menú desplegable denominado **On Action**.



Gestión de errores

Si ejecutas tu aplicación en este punto deberías ser capaz de borrar personas de la tabla. Pero, ¿qué ocurre si **pulsas el botón de borrar sin seleccionar a nadie** en la tabla.

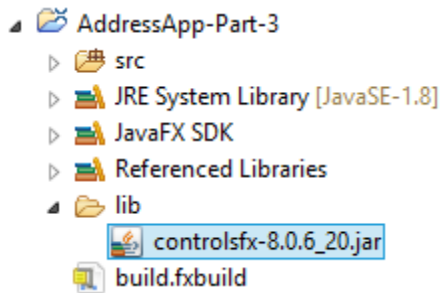
Se produce un error de tipo `ArrayIndexOutOfBoundsException` porque no puede borrar una persona en el índice `-1`, que es el valor devuelto por el método `getSelectedIndex()` - cuando no hay ningún elemento seleccionado.

Ignorar semejante error no es nada recomendable. Deberíamos hacerle saber al usuario que tiene que seleccionar una persona previamente para poderla borrar (incluso mejor sería deshabilitar el botón para que el usuario ni siquiera tenga la oportunidad de realizar una acción incorrecta).

Vamos a añadir un diálogo emergente para informar al usuario. Desafortunadamente no hay componentes para diálogos incluidos en JavaFX 8. Para evitar tener que crearlos manualmente podemos **añadir una librería** que ya los incluya ([Dialogs](#)):

1. Descarga este [controlsfx-8.0.6_20.jar](#) (también se puede obtener de la [página web de ControlsFX](#)).
Importante: La versión de ControlsFX debe ser la `8.0.6_20` o superior para que funcione con `JDK 8u20` debido a un cambio crítico en esa versión.
2. Crea una subcarpeta **lib** dentro del proyecto y coloca dentro del archivo jar.

3. Añade la librería al **classpath** de tu proyecto: En Eclipse se puede hacer mediante *clic-derecho sobre el archivo jar | Build Path | Add to Build Path*. Ahora Eclipse ya sabe donde encontrar esa librería.



Con algunos cambios en el método `handleDeletePerson()` podemos mostrar una simple ventana de diálogo emergente en el caso de que el usuario pulse el botón Delete sin haber seleccionado a nadie en la tabla de contactos:

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();
    }
}
```

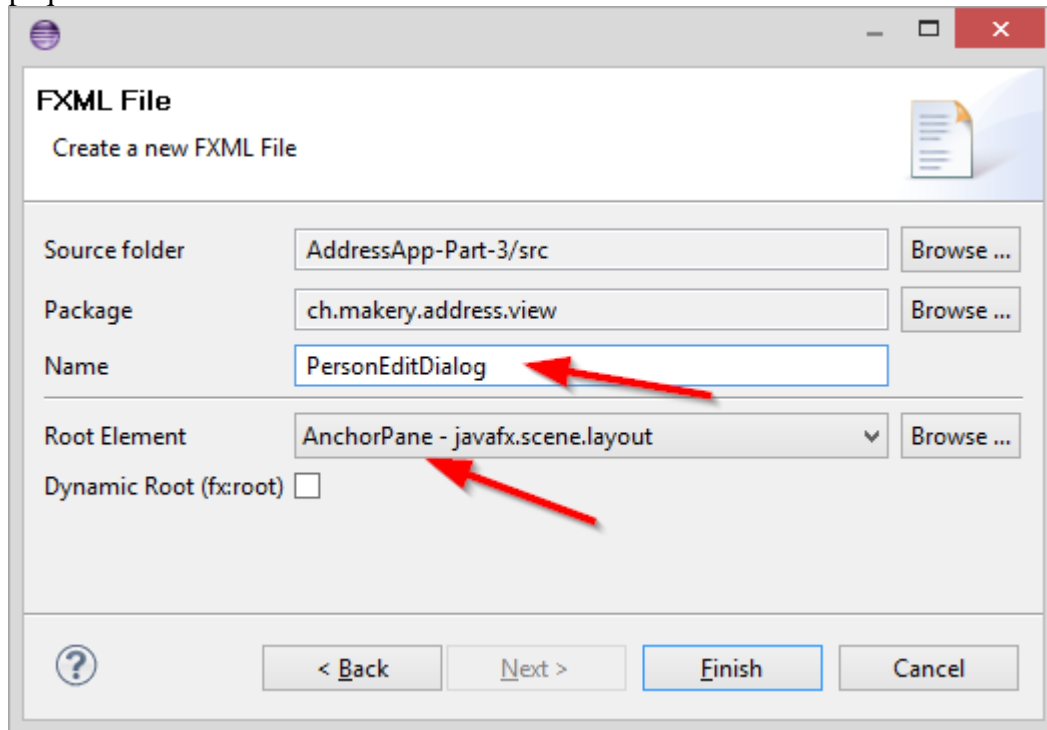
Para ver más ejemplos de utilización de ventanas de diálogo, consulta [JavaFX 8 Dialogs](#).

Diálogos para crear y editar contactos

Las acciones de editar y crear nuevo contacto necesitan algo más de elaboración: vamos a necesitar una ventana de diálogo a medida (es decir, un nuevo **stage**) con un formulario para preguntar al usuario los detalles sobre la persona.

Diseña la ventana de diálogo

1. Crea un nuevo archivo fxml llamado **PersonEditDialog.fxml** dentro del paquete *view*.



2. Usa un panel de rejilla (**GridPane**), etiquetas (**Label**), campos de texto (**TextField**) y botones (**Button**) para crear una ventana de diálogo como la

siguiente:

The image shows a JavaFX dialog box with a title bar containing two buttons labeled '0' and '1'. The main content area is a table-like structure with six rows. Each row has a label on the left, a text input field in the center, and a small button on the right. The labels are 'First Name', 'Last Name', 'Street', 'City', 'Postal Code', and 'Birthday'. The buttons on the left are labeled '0' through '5' respectively. The buttons on the right are also labeled '0' through '5' respectively. At the bottom of the dialog are two buttons labeled 'OK' and 'Cancel'.

Si quieres puedes descargar el código desde [PersonEditDialog.fxml](#).

Create the Controller

Crea el controlador para la ventana de edición de personas y llámalo `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.validateDate(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;
}
}
}

```

Algunas cuestiones relativas a este controlador:

- El método `setPerson(...)` puede ser invocado desde otra clase para establecer la persona que será editada.
- Cuando el usuario pulsa el botón OK, el método `handleOk()` es invocado. Primero se valida la entrada del usuario mediante la ejecución del método `isInputValid()`. Sólo si la validación tiene éxito el objeto persona es modificado con los datos introducidos por el usuario. Esos cambios son aplicados directamente sobre el objeto pasado como argumento del método `setPerson(...)`!
- El método boolean `okClicked` se utiliza para determinar si el usuario ha pulsado el botón OK o el botón Cancel.

Enlaza la vista y el controlador

Una vez creadas la vista (FXML) y el controlador, necesitamos vincular el uno con el otro:

1. Abre el archivo `PersonEditDialog.fxml`.
2. En la sección *Controller* a la izquierda selecciona `PersonEditDialogController` como clase de control.
3. Establece el campo **fx:id** de todas los `TextField` con los identificadores de los atributos del controlador correspondientes.
4. Especifica el campo **onAction** de los dos botones con los métodos del controlador correspondientes a cada acción.

Abriendo la ventana de diálogo

Añade un método para cargar y mostrar el método de edición de una persona dentro de la clase `MainApp`:

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;
    }
}

```

Añade los siguientes métodos a la clase `PersonOverviewController`. Esos métodos llamarán al método `showPersonEditDialog(...)` desde `MainApp` cuando el usuario pulse en los botones *new* o *edit*.

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();
    }
}

```

Abre el archivo `PersonOverview.fxml` mediante Scene Builder. Elige los métodos correspondientes en el campo *On Action* para los botones *new* y *edit*.

¡Ya está!

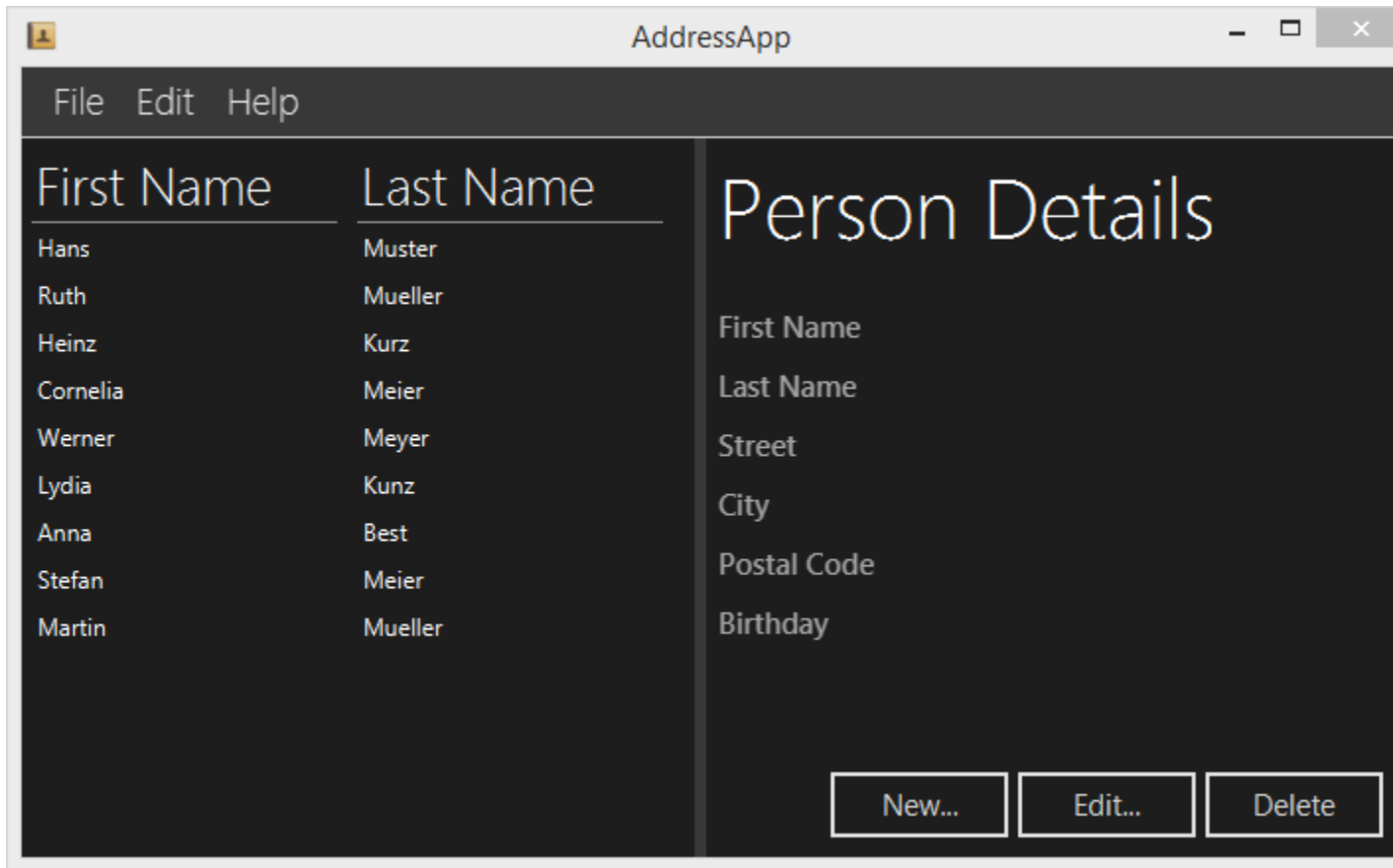
Llegados a este punto deberías tener una aplicación de *libreta de contactos* en funcionamiento. Esta aplicación es capaz de añadir, editar y borrar personas. Tiene incluso algunas capacidades de validación para evitar que el usuario introduzca información incorrecta.

Espero que los conceptos y estructura de esta aplicación te permitan empezar tu propia aplicación JavaFX. ¡ Disfruta !

Tutorial JavaFX 8 - Parte 4:

Hojas de estilo CSS

Sep 17, 2014



Contenidos en Parte 4

- Estilos mediante CSS
- Añadiendo un Icono de Aplicación

Estilos mediante CSS

En JavaFX puedes dar estilo al interfaz de usuario utilizando hojas de estilo en cascada (CSS). ¡ Esto es estupendo ! Nunca había sido tan fácil personalizar la apariencia de una aplicación Java.

En este tutorial vamos a crear un tema oscuro (*DarkTheme*) inspirado en el diseño de Windows 8 Metro. El código CSS de los botones está basado en el artículo de blog [JMetro - Windows 8 Metro controls on Java](#) by Pedro Duque Vieira.

Familiarizándose con CSS

Para poder aplicar estilos a una aplicación JavaFX application debes tener una comprensión básica de CSS en general. Un buen punto de partida es este tutorial: [CSS tutorial](#).

Para información más específica de CSS y JavaFX puedes consultar:

- [Skinning JavaFX Applications with CSS](#) - Tutorial by Oracle
- [JavaFX CSS Reference](#) - Official Reference

Estilo por defecto en JavaFX

Los estilos por defecto de JavaFX 8 se encuentran en un archivo denominado `modena.css`. Este archivo CSS se encuentra dentro del archivo `jfxrt.jar` que se encuentra en tu directorio de instalación de Java, en la ruta `/jdk1.8.x/jre/lib/ext/jfxrt.jar`.

Puedes descomprimir `jfxrt.jar` o abrirlo como si fuera un zip. Encontrarás el archivo `modena.css` en la ruta `com/sun/javafx/scene/control/skin/modena/`

Este estilo se aplica siempre a una aplicación JavaFX. Añadiendo un estilo personal podemos reescribir los estilos por defecto definidos en `modena.css`.

Truco: Ayuda consultar el archivo CSS por defecto para ver qué estilos necesitas sobrescribir.

Vinculando hojas de estilo CSS

Añade un archivo CSS denominado `DarkTheme.css` al paquete `view`.

`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:focus {
    -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%);
}

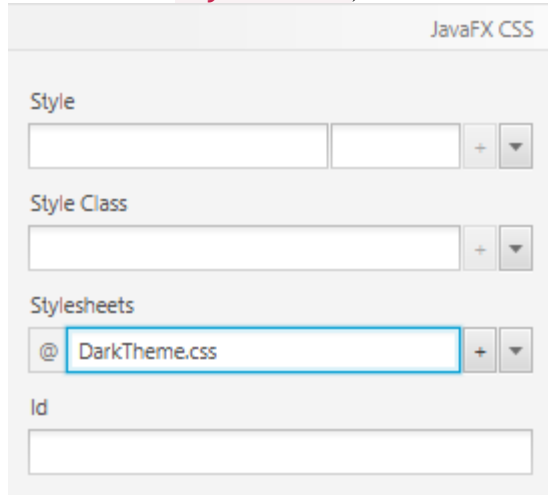
```

A continuación necesitamos vincular el CSS a nuestra escena. Podemos hacer esto programáticamente, mediante código Java, pero en esta ocasión vamos a utilizar Scene Builder para añadirlo a nuestros archivos FXML:

Añade el CSS a RootLayout.fxml

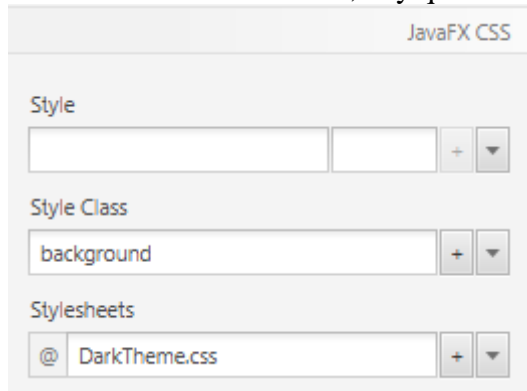
1. Abre el archivo `RootLayout.fxml` en Scene Builder.

2. Selecciona el **BorderPane** raíz en la sección *Hierarchy*. En la vista *Properties* añade el archivo **DarkTheme.css** como hoja de estilo (campo denominado **Stylesheets**).



Añade el CSS a *PersonEditDialog.fxml*

1. Abre el archivo **PersonEditDialog.fxml** en Scene Builder. Selecciona el **AnchorPane** raíz e incluye **DarkTheme.css** como hoja de estilo en la sección *Properties*.
2. El fondo todavía es blanco, hay que añadir la clase **background** al **AnchorPane** raíz.



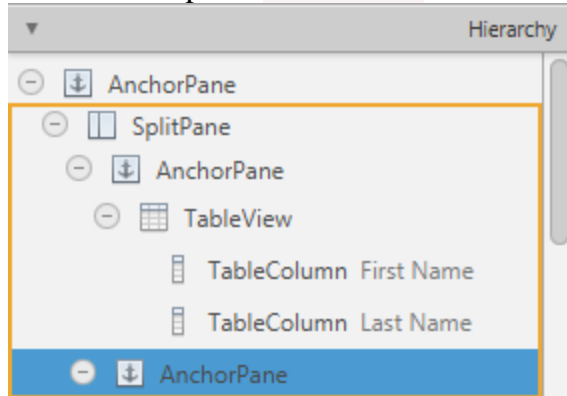
3. Selecciona el botón OK y elige *Default Button* en la vista *Properties*. Eso cambiará su color y lo convertirá en el botón "por defecto", el que se ejecutará si el usuario aprieta la tecla *enter*.

Añade el CSS a *PersonOverview.fxml*

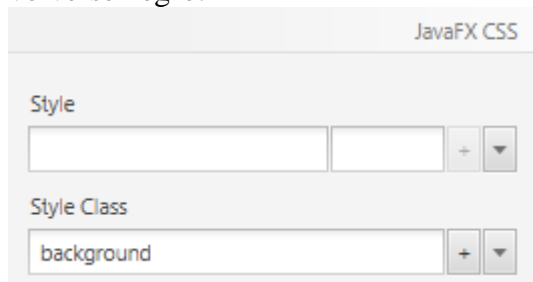
1. Abre el archivo **PersonOverview.fxml** en Scene Builder. Selecciona el **AnchorPane** raíz en la sección *Hierarchy* y añade **DarkTheme.css** a sus **Stylesheets**.
2. A estas alturas deberías haber observado algunos cambios: La tabla y los botones son negros. Las clases de estilo **.table-view** y **.button** de **modena.css** se aplican

automáticamente a la tabla y los botones. Ya que hemos redefinido (sobreescribió) algunos de esos estilos en nuestro propio CSS, los nuevos estilos se aplican automáticamente.

3. Posiblemente tengas que ajustar el tamaño de los botones para que se muestre todo el texto.
4. Selecciona el panel **AnchorPane** de la derecha, dentro del **SplitPane**.



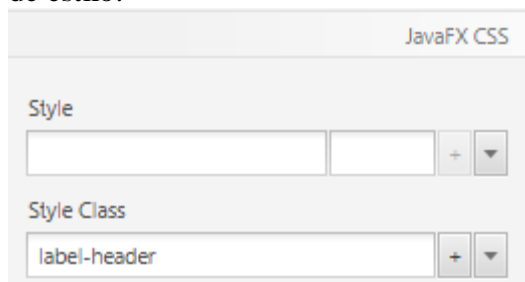
5. Ves a la vista *Properties* y elige **background** como clase de estilo. El fondo debería volverse negro.



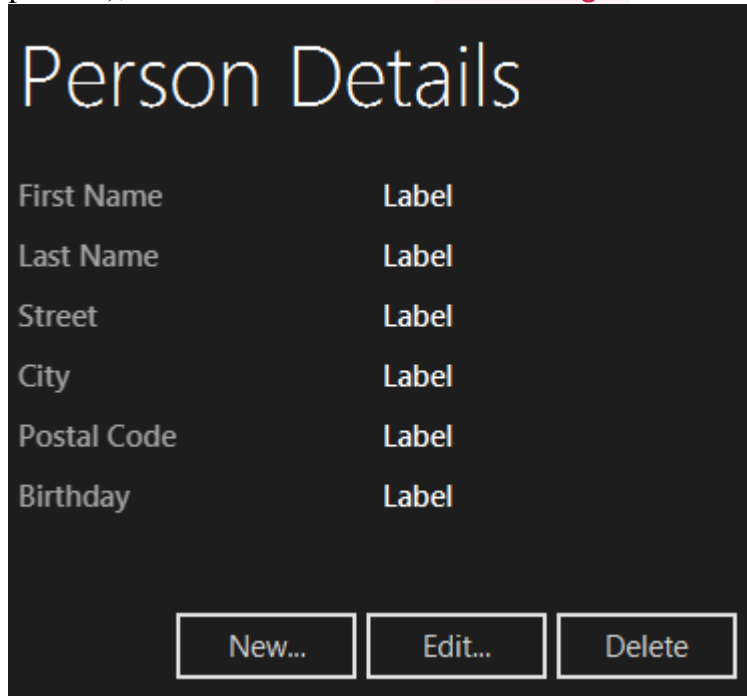
Etiquetas con un estilo diferente

En este momento, todas las etiquetas en el lado derecho tienen el mismo tamaño. Ya tenemos definidos en el CSS unos estilos denominados **.label-header** y **.label-bright** que vamos a usar para personalizar la apariencia de las etiquetas.

1. Selecciona la etiqueta (**Label**) *Person Details* y añade **label-header** como clase de estilo.



2. A cada etiqueta en la columna de la derecha (donde se muestran los detalles de una persona), añade la clase de estilo `label-bright`.



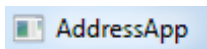
Person Details

First Name	Label
Last Name	Label
Street	Label
City	Label
Postal Code	Label
Birthday	Label

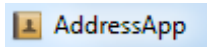
New... Edit... Delete

Añadiendo un icono a la aplicación

Ahora mismo nuestra aplicación utiliza el icono por defecto para la barra de título y la barra de tareas:



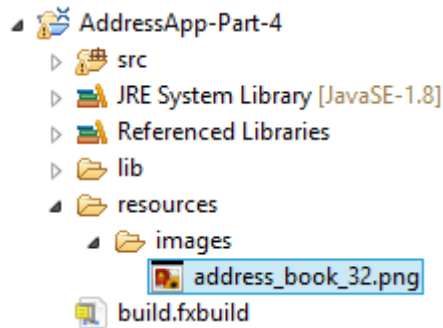
Quedaría mucho mejor con un icono propio:



El archivo de icono

Un posible sitio para obtener iconos gratuitos es [Icon Finder](#). Yo por ejemplo descargué este [icono de libreta de direcciones](#).

Crea una carpeta dentro de tu aplicación llamado **resources** y añádele una subcarpeta para almacenar imágenes, llámala **images**. Pon el icono que hayas elegido dentro de la carpeta de imágenes. La estructura de directorios de tu carpeta debe tener un aspecto similar a este:



Establece el icono de la escena principal

Para establecer el icono de nuestra escena debemos añadir la línea de código siguiente al método `start(...)` dentro de `MainApp.java`

`MainApp.java`

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

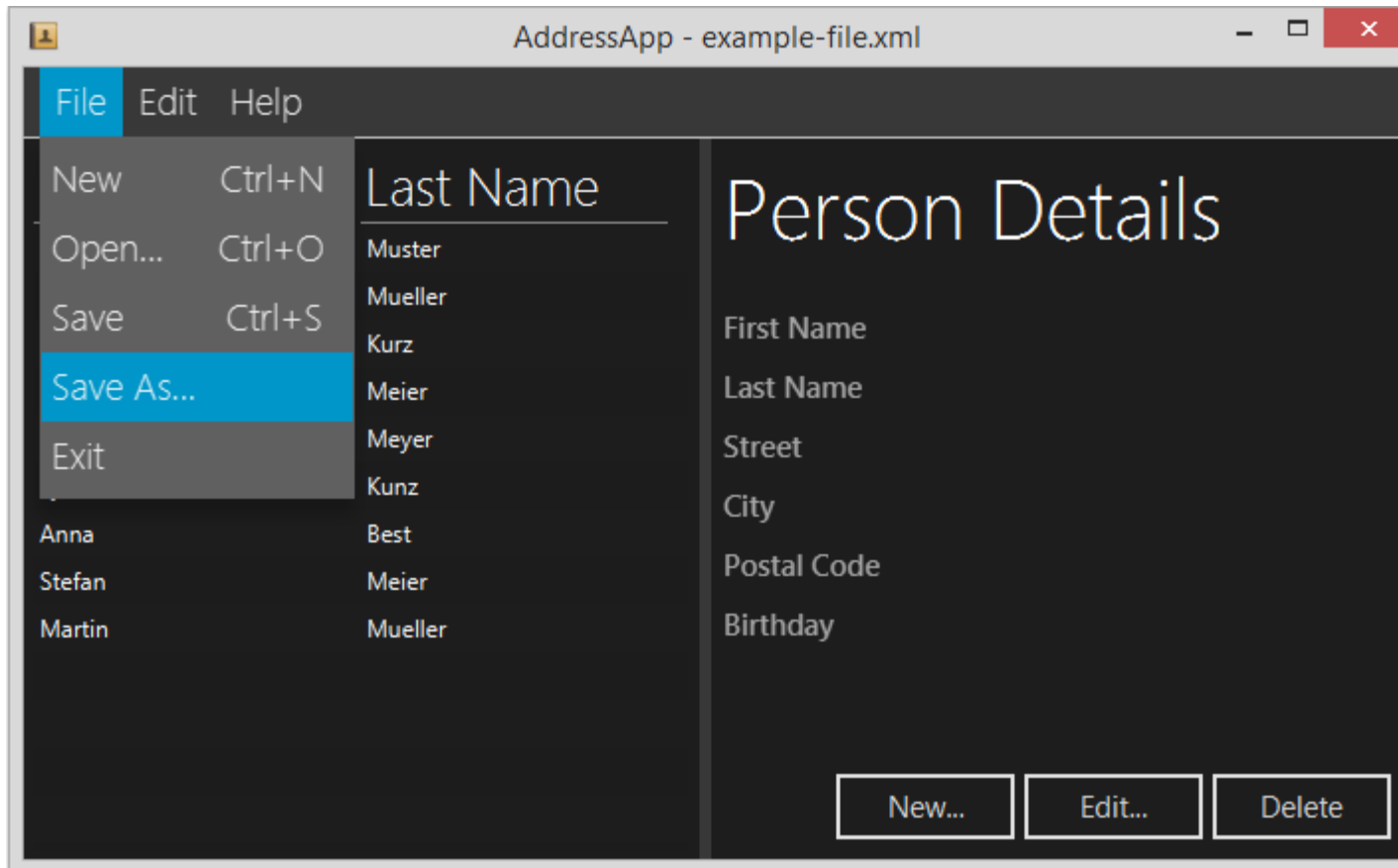
The whole `start(...)` method should look something like this now:

```
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();  
}
```

También puedes añadir un icono a la escena que contiene la venta de edición de los detalles de una persona.

Tutorial JavaFX 8 - Parte 5: Persistencia de datos con XML

Sep 17, 2014



Contenidos en Parte 5

- **Persistencia de datos en XML**
 - Utilización de **FileChooser**
 - Utilización de **Menu**
 - Guardando la ruta al último archivo abierto en las **preferencias de usuario**
-

Actualmente, los datos de nuestra aplicación de libreta de direcciones reside únicamente en memoria. Cada vez que cerramos la aplicación los datos se pierden. Así pues, ha llegado la hora de pensar en como guardar los datos de forma persistente.

Guardando preferencias del usuario

Java nos permite guardar cierta información mediante una clase llamada `Preferences`, pensada para guardar las preferencias de usuario de una aplicación. Dependiendo del sistema operativo, estas preferencias son guardadas en un sitio u otro (por ejemplo el registro de Windows).

No podemos usar un archivo de `Preferences` para guardar nuestra libreta de direcciones completa, pero nos sirve para guardar **información de estado muy simple**. Un ejemplo del tipo de cosas que podemos guardar en estas preferencias es **la ruta al último archivo abierto**. Con esta información podemos recuperar el último estado de la aplicación cuando el usuario vuelva a ejecutar la aplicación.

Los siguientes dos métodos se encargan de guardar y recuperar las `Preferences`. Añádelos al final de la clase `MainApp`:

`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");
    }
}
}

```

Persistencia de datos mediante XML

¿Por qué XML?

Una de las formas más habituales de almacenar datos es mediante una base de datos. Las bases de datos típicamente contienen algún tipo de datos relacionales (tablas relacionadas mediante índices), mientras que los datos que tenemos que guardar. A este problema se le denomina *desadaptación de impedancias objeto-relacional* ([object-relational impedance mismatch](#)). Cuesta bastante trabajo adaptar objetos a tablas de una base de datos relacional. Aunque existen algunas soluciones para ayudarnos a realizar esta adaptación (ej. [Hibernate](#), la más popular), todavía cuesta bastante trabajo de configuración.

Para nuestro sencillo modelo de datos es mucho más fácil usar XML. Usaremos una librería llamada [JAXB](#) (**J**ava **A**rchitecture for **X**ML **B**inding). Con apenas unas pocas líneas de código JAXB nos permitirá generar una salida en XML como esta:

Ejemplo de salida en XML

```
<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>
```

Utilización de JAXB

JAXB viene incluido en el JDKm. Eso significa que no necesitamos añadir ninguna librería adicional.

JAXB proporciona dos funcionalidades principales: la capacidad de convertir objetos Java en XML (**marshalling**), y a la inversa, la capacidad de convertir XML en objetos Java (**unmarshalling**).

Para que JAXB sea capaz de hacer la conversión, necesitamos preparar nuestro modelo.

Preparando el modelo para JAXB

Los datos que queremos guardar se hallan en la variable `personData` dentro de la clase `MainApp`. JAXB requiere que la clase raíz (la que contenga a todo el árbol XML) sea anotada con `@XmlElement`. Sin embargo `personData` es de clase `ObservableList`, que no se puede utilizar en JAXB. De ahí que necesitemos crear otra clase para contener nuestra lista de personas (`Person`) de cara a ser adaptada a XML por JAXB

La nueva clase que creamos se llama `PersonListWrapper` y la ponemos en el paquete `ch.makery.address.model`.

`PersonListWrapper.java`

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

import java.util.List;

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

/**
 * 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;
    }
}

```

Fíjate en las dos anotaciones.

- `@XmlRootElement` define el nombre del elemento raíz del XML.
- `@XmlElement` es un nombre opcional que podemos especificar para el elemento (usado en su representación XML).

Leyendo y escribiendo datos con JAXB

Haremos a nuestra clase `MainApp` responsable de leer y escribir los datos XML. Añade la siguiente pareja de métodos al final de la clase `MainApp.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);
}
}

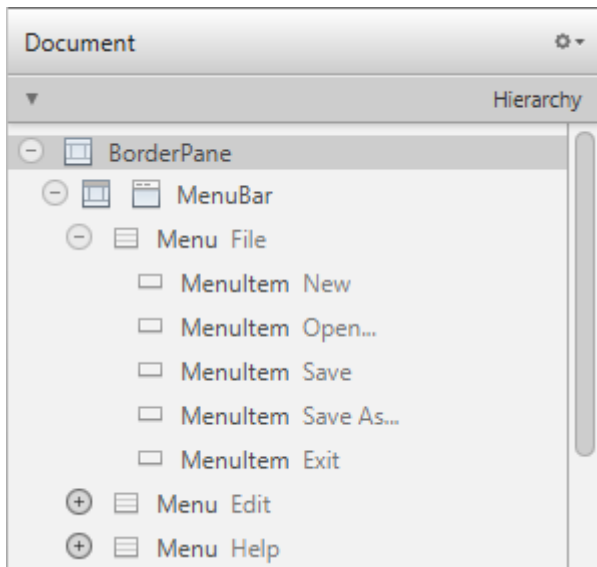
```

Los métodos de escritura (marshalling) y lectura (unmarshalling) ya están listos. Ahora crearemos unas opciones de menú para poder utilizar esos métodos.

Gestión de acciones de menú

En nuestro `RootLayout.fxml` ya hay un menú, pero todavía no lo hemos utilizado. Antes de añadir acciones al menú crearemos todos los ítems del menú.

Abre el archivo `RootLayout.fxml` en Scene Builder y arrastra los ítems de menú necesarios desde la sección *library* a la barra de menús (componente `MenuBar` en la *hierarchy*). Crea los siguientes ítems: **New**, **Open...**, **Save**, **Save As...**, y **Exit**.



Truco: Mediante el uso de la opción *Accelerator* en la vista *Properties* se pueden establecer atajos de teclado para lanzar las acciones asociadas a los ítems del menú.

Controlador para las acciones de menú: RootLayoutController

Para implementar las acciones del menú necesitaremos una nueva clase de control. Crea una nueva clase `RootLayoutController` dentro de `ch.makery.address.view`.

Añade el siguiente contenido al controlador recién creado.

`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.ExtensionFil
ter(
            "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.ExtensionFil
ter(
        "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);
}
}

```

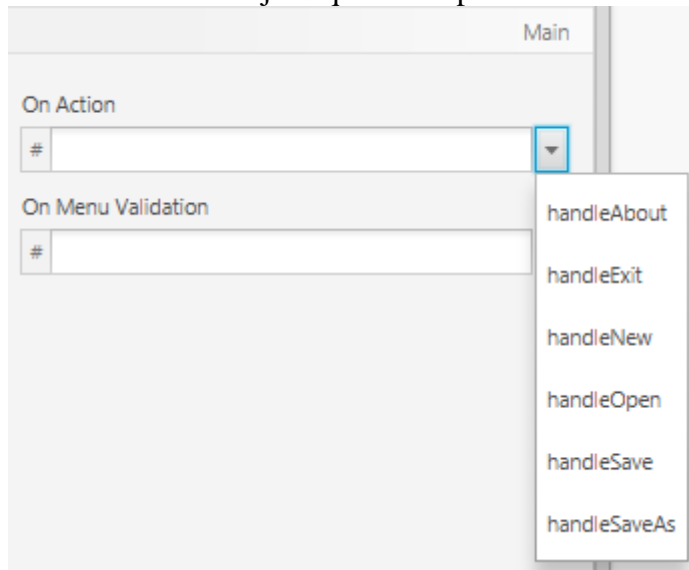
FileChooser

Fíjate en los métodos que usan la clase `FileChooser` dentro de `RootLayoutController`. Primero, se crea una nueva instancia de la clase `FileChooser`. A continuación, se le añade un filtro de extensión para que sólo se muestren los archivos terminados en `.xml`. Finalmente, el objeto `FileChooser` se muestra justo encima de la escena principal.

Si el usuario cierra la ventana del `FileChooser` sin escoger un archivo, se devuelve `null`. En otro caso, se obtiene el archivo seleccionado, y se lo podemos pasar al método `loadPersonDataFromFile(...)` o al método `savePersonDataToFile(...)` de la clase `MainApp`.

Conectando el FXML con el controlador

1. Abre `RootLayout.fxml` en Scene Builder. En la sección *Controller* selecciona `RootLayoutController` como controlador.
2. Vuelve a la sección *Hierarchy* y elige un ítem del menú. En el campo **On Action** de la sección *Code* debes tener como opciones todos los métodos disponibles en la clase de control. Elige el que corresponda a cada uno de los ítems del menú.



3. Repite el paso 2 para todos y cada uno de los ítems del menú.
4. Cierra Scene Builder y refresca el proyecto (pulsas **Refresh (F5)** sobre la carpeta raíz de tu proyecto). Esto hará que Eclipse se entere de los cambios realizados en Scene Builder.

Conectando la clase MainApp y el controlador RootLayoutController

En varios sitios, el controlador `RootLayoutController` necesita una referencia a la clase `MainApp`. Todavía no hemos pasado esa referencia al `RootLayoutController`.

Abre la clase `MainApp` y sustituye el método `initRootLayout()` por el código siguiente:

```
/**
```

```

    * 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);
    }
}

```

```
}
```

Fíjate en los 2 cambios introducidos: Las líneas que *dan acceso a MainApp*" y las últimas tres líneas para cargar el último archivo abierto*.

Pruebas

Si pruebas ahora tu aplicación deberías ser capaz de usar los menús para grabar los datos de los contactos en un archivo XML.

Si abres el archivo XML resultante en un editor, notarás que la fecha de nacimiento no se guarda correctamente, aparece una etiqueta `<birthday/>` vacía. La razón es que JAXB no sabe como convertir `LocalDate` a XML. Debemos proporcionar un adaptador a medida para realizar esta conversión.

Dentro de `ch.makery.address.util` crea una nueva clase denominada `LocalDateAdapter` con el contenido siguiente:

`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();
    }
}

```

A continuación abre la clase `Person` y añade la siguiente anotación al método `getBirthday()`:

```

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

```

Ahora prueba a guardar los datos de nuevo y abre el archivo XML otra vez. Debería abrir automáticamente el último archivo abierto durante la ejecución previa.

Como funciona

Ahora veamos como funciona todo junto

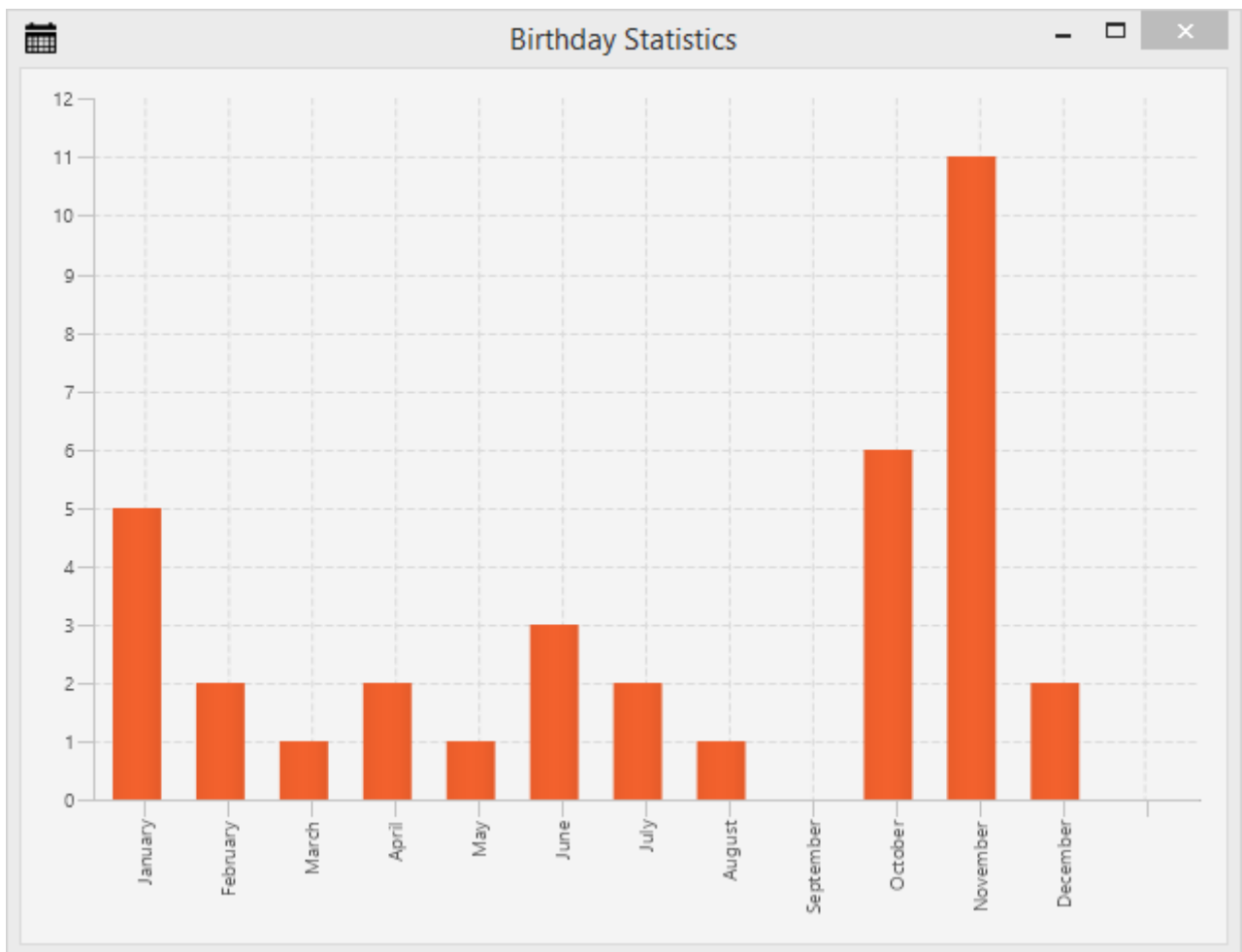
1. La aplicación se inicia con la ejecución del método `main(...)` de la clase `MainApp`.
2. El constructor `public MainApp()` es invocado y añade algunos datos de ejemplo.
3. El método `start(...)` de la clase `MainApp` es invocado, el cual a su vez invoca a `initRootLayout()` para inicializar la vista principal utilizando el archivo `RootLayout.fxml`. El archivo FSML tiene información sobre qué controlador utilizar y enlaza la vista con su controlador `RootLayoutController`.
4. `MainApp` obtiene el controlador `RootLayoutController` del cargador FXML y le pasa a ese controlador una referencia a sí mismo. con esta referencia el controlador podrá después acceder a los métodos (públicos) de `MainApp`.

5. Al final del método `initRootLayout()` se intenta obtener el *último archivo de direcciones abierto* desde las `Preferences`. Si existe esa información en `Preferences` entonces se leen los datos del XML. Estos datos sobrescribirán los datos de ejemplo generados en el constructor.

Tutorial JavaFX 8 - Parte 6:

Gráficos estadísticas

Sep 17, 2014



Contenidos en Parte 6

- Creando un **Gráfico estadístico** para mostrar la distribución de meses de nacimiento.

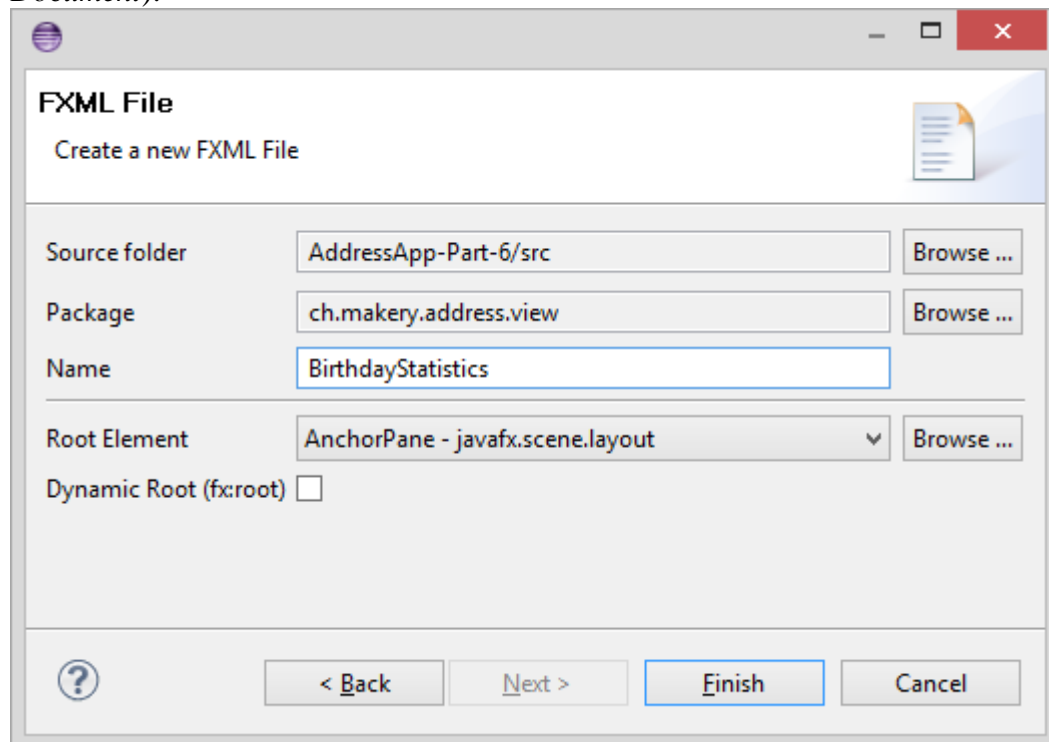
Estadísticas sobre fecha de nacimiento

Las personas en nuestra libreta de direcciones tienen una fecha de nacimiento. ¿No sería bonito tener algunas estadísticas sobre los meses en que celebran el cumpleaños?

Usaremos un **Gráfico de barras** conteniendo una barra para cada mes. Cada barra mostrará cuantas personas en nuestra libreta cumplen años ese mes en particular.

La vista FXML de estadísticas

1. Empezamos creando un archivo `BirthdayStatistics.fxml` dentro del paquete `ch.makery.address.view` (*Clic derecho / New / Other... / New FXML Document*).



2. Abre el archivo `BirthdayStatistics.fxml` en Scene Builder.
3. Elige el panel raíz `AnchorPane`. En la sección de *Layout* establece la anchura y altura preferidas (*Pref Width*, *Pref Height*) en 620 y 450 respectivamente.
4. Añade un componente `BarChart` al `AnchorPane`.

5. Clic derecho sobre el elemento `BarChart` y elige *Fit to Parent*.
6. Guarda el archivo FXML, vuelve a Eclipse y refresca el proyecto (F5).

Antes de volver a Scene Builder vamos a crear el controlador y a realizar las conexiones necesarias en nuestra `MainApp`.

El controlador para las estadísticas

En el paquete de vistas `ch.makery.address.view` crea una nueva clase Java denominada `BirthdayStatisticsController.java`.

Veamos primero el controlador completo antes de explicarlo:

`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.observableArra
yList();

    /**
     * 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).getM
onths();
        // 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);
}
}

```

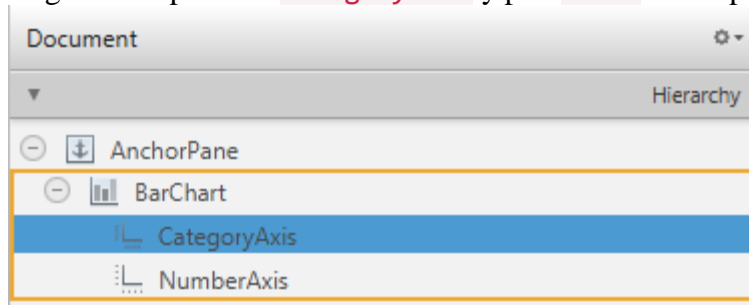
Como funciona el controlador

1. El controlador necesitará acceso a dos elementos de nuestro archivo FXML:
 - o **barChar**: Tiene los tipos **String** e **Integer**. El tipo **String** se utiliza para el mes en el eje-X y el tipo **Integer** se utiliza para el número de personas en un mes determinado.
 - o **xAxis**: Lo usaremos para añadir las nombres de los meses
2. El método **initialize()** rellena el eje-X con la lista de todos los meses.

3. El método `setPersonData(...)` será accedido por la clase `MainApp` para establecer los datos de personas. Este método recorre todas las personas y cuenta los nacimientos por mes. Entonces añade `XYChart.Data` para cada mes a las series de datos. Cada objeto `XYChart.Data` representará una barra en el gráfico.

Conectando vista y controlador

1. Abre `BirthdayStatistics.fxml` en Scene Builder.
2. En la sección *Controller* pon `BirthdayStatisticsController` como controlador.
3. Selecciona el componente `BarChart` y pon `barChart` en la propiedad `fx:id` (sección *Code*).
4. Elige el componente `CategoryAxis` y pon `xAxis` como propiedad `fx:id`.



5. Puedes añadir un título al componente `BarChart` (en *Properties*) para mejorar la apariencia.

Conectando vista y controlador con MainApp

Para el gráfico de estadísticas usaremos el mismo mecanismo que utilizamos para el diálogo de edición de personas, una ventana de diálogo emergente.

Añade el siguiente método a tu clase `MainApp`:

```
/**
 * 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();
    }
}

```

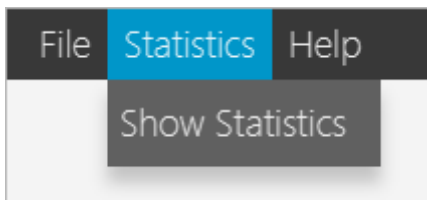
Todo está dispuesto, pero todavía no tenemos nada para invocar al nuevo método `showBirthdayStatistics()`. Afortunadamente ya tenemos un menú en la vista `RootLayout.fxml` que puede ser usado para estos fines.

Muestra el menú de estadísticas de cumpleaños

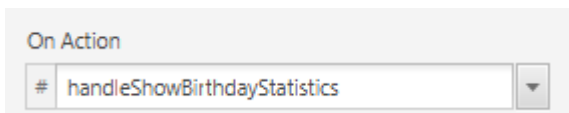
En tu `RootLayoutController` añade el siguiente método para gestionar las pulsaciones sobre el ítem de menú *Show Statistics*:

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

Ahora, abre `RootLayout.fxml` en Scene Builder. Crea el menú *Statistics* con un ítem denominado *Show Statistics*:



Elige el ítem de menú *Show Statistics* y establece `handleShowBirthdayStatistics` en su campo **On Action** (sección *Code*).



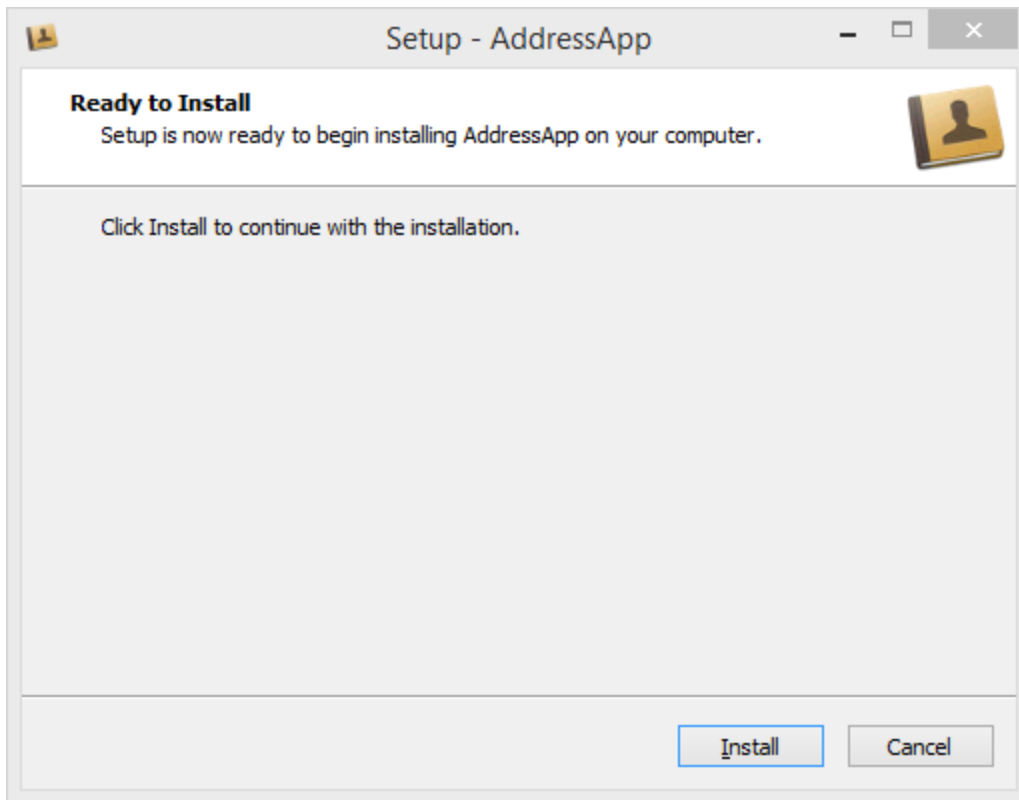
Vuelve a Eclipse, refresca el proyecto y **pruébalo**.

Más información sobre gráficos JavaFX

Un buen sitio para obtener más información es el tutorial oficial de Oracle: [Working with JavaFX Charts](#).

Tutorial JavaFX 8 - Parte 7: Despliegue

Sep 17, 2014



He pensado escribir una última parte de este tutorial para mostrar como desplegar (es decir empaquetar y publicar) la aplicación de libreta de direcciones.

Contenidos en Parte 7

- Desplegando nuestra aplicación JavaFX como un **Paquete nativo** con e(fx)clipse
-

Qué es el despliegue

El despliegue es el proceso de empaquetar y distribuir o hacer llegar una aplicación al usuario. Es una fase crucial del desarrollo porque es el primer contacto del usuario final con nuestro software.

Java se anuncia con el slogan **Escribe una vez, ejecuta donde sea** para ilustrar sobre los beneficios *multi-plataforma* del lenguaje Java. Idealmente, esto significa que nuestra aplicación Java puede ser ejecutada en cualquier dispositivo equipado con una *Máquina Virtual Java*(JVM).

En el pasado, la experiencia de usuario instalando una aplicación Java no ha sido siempre agradable. Si el usuario no tenía en su sistema la versión requerida de Java Runtime (JRE), debía ser guiado para su instalación previa. Esto originaba ciertas dificultades, como problemas de privilegios (había que ser administrador) o problemas de compatibilidad.

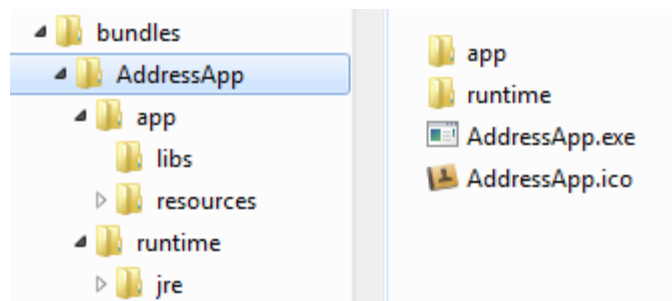
Afortunadamente, JavaFX ofrece una nueva opción para el despliegue de una aplicación denominada **Native Packaging** (también llamada Self-Contained Application Package). Un paquete nativo es un lote que contiene tanto la aplicación como la JRE específica requerida.

Oracle en su documentación oficial sobre JavaFX ofrece una guía extensiva de todas las opciones de despliegue en su [JavaFX deployment options](#).

En esta parte del tutorial mostraré como crear un **paquete nativo** con Eclipse y el **plugin e(fx)clipse**.

Crea una paquete nativo

El objetivo es crear una aplicación auto-contenida en una carpeta. Esta es la apariencia que tendrá la aplicación de libreta de direcciones (en Windows):



La carpeta **app** contiene los datos de nuestra aplicación y el **runtime** de Java específico de la plataforma.

Para hacérselo incluso más fácil al usuario, vamos a proporcionar un instalador:

- Un archivo de instalación **.exe** para Windows.
- Un archivo de instalación **.dmg** (drag & drop) para MacOS.

El plugin e(fx)clipse nos ayudará a generar el paquete nativo y el instalador.

Paso 1 - Edita build.fxbuild

El archivo `build.fxbuild` es utilizado por e(fx)clipse para generar otro archivo que a su vez será usado por la herramienta Ant. (Si no tuvieras el archivo `build.fxbuild`, crea un nuevo proyecto JavaFX en Eclipse y copia el archivo generado)

1. Abre `build.fxbuild` desde la raíz de tu proyecto.
2. Rellena todos los campos que contengan una estrella. *Para usuarios de MacOS: no uses espacios en el título de la aplicación porque puede ocasionar un problema.*

FX Build Configuration

▼ **Build & Package Properties**
The following properties are needed to build the JavaFX-Application

Build Directory*:

Vendor name*: ←

Application title*: ←

Application version*: ←

Application class*: ←

Preloader class:

Splash:

Manifest-Attributes:

Name	Value

Toolkit Type:

Packaging Format: ←

☐ automatic Proxy Resolution

☐ Convert CSS into binary form

☐ Enable verbose build mode (Not recommended)

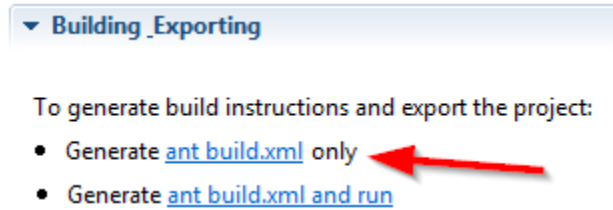
▼ **Building_Exporting**

To generate build instructions and export:

- Generate [ant build.xml](#) only
- Generate [ant build.xml and run](#)

3. Como **Packaging Format** elige entre `exe` para Windows, `dmg` para MacOS, o `rpm` para Linux.

4. Pincha en el enlace **Generate ant build.xml only** (se encuentra en el lado derecho).



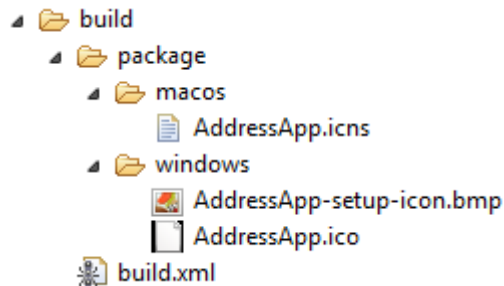
5. Verifica que se ha creado una nueva carpeta denominada **build** así como un archivo **build.xml**.

Paso 2 - Añade iconos de instalación

Nos gustaría tener algunos iconos para nuestro instalador:

- [AddressApp.ico](#) para el icono de la aplicación una vez instalada
- [AddressApp-setup-icon.bmp](#) para la pantalla del programa de instalación
- [AddressApp.icns](#) para el instalador de Mac

1. Crea las siguientes subcarpetas en la carpeta **build**:
 - **build/package/windows** (sólo para Windows)
 - **build/package/macOS** (sólo para MacOS)
2. Copia los iconos indicados arriba en las subcarpetas que correspondan. Debería tener esta apariencia ahora:

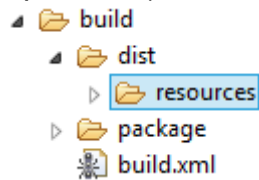


3. **Importante:** El nombre de los iconos debe coincidir exactamente con el título de la aplicación tal y como ha sido especificada en la propiedad **Application title** del archivo **build.fxbuild**:
 - **YourAppTitle.ico**
 - **YourAppTitle-setup-icon.bmp**
 - **YourAppTitle.icns**

Paso 3 - Añade recursos

Nuestra carpeta **resources** no se copia automáticamente, debemos añadirla nosotros a la carpeta **build**:

1. Crea una subcarpeta denominada `dist` dentro de la carpeta `build`.
2. Copia la carpeta `resources` (conteniendo las imágenes de nuestra aplicación) dentro de `build/dist`.



Paso 4 - Edita build.xml para incluir iconos

E(fx)clipse ha generado un archivo `build/build.xml` que está listo para ser ejecutado por **Ant**. Sin embargo los iconos y las imágenes que tenemos en la carpeta `resources` no se incluirán en el paquete nativo.

Como todavía no se le puede indicar a e(fx)clipse que incluya recursos adicionales como nuestra carpeta `resources` y los iconos anteriores, tenemos que editar el archivo `build.xml` manualmente:

Abre para edición el archivo `build.xml` y encuentra la ruta `fxant`. Añade una línea para indicar el directorio base `${basedir}` (esto hará que nuestros iconos estén disponibles):

build.xml - añade "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>
```

Encuentra el bloque `fx:resources id="appRes"` más abajo en el archivo y añade una línea para nuestros `resources`:

build.xml - añade "resources"

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

```
<fx:fileset dir="dist" includes="resources/**"/>
</fx:resources>
```

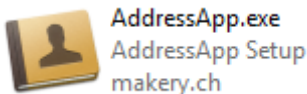
Por alguna razón el número de versión se añade a la aplicación (propiedad `fx:application`) lo que hace que el instalador adopte siempre el valor por defecto `1.0` (tal y como han indicado varios comentarios). Para corregir esto, hay que añadir el número de versión de forma manual (gracias a Marc por [descubrir como hacerlo](#)):

[build.xml](#) - añade número de "version"

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

Ya podemos ejecutar `build.xml` con *Ant*. Esto generará un jar ejecutable del proyecto. Pero queremos ir un paso más allá y crear un práctico instalador.

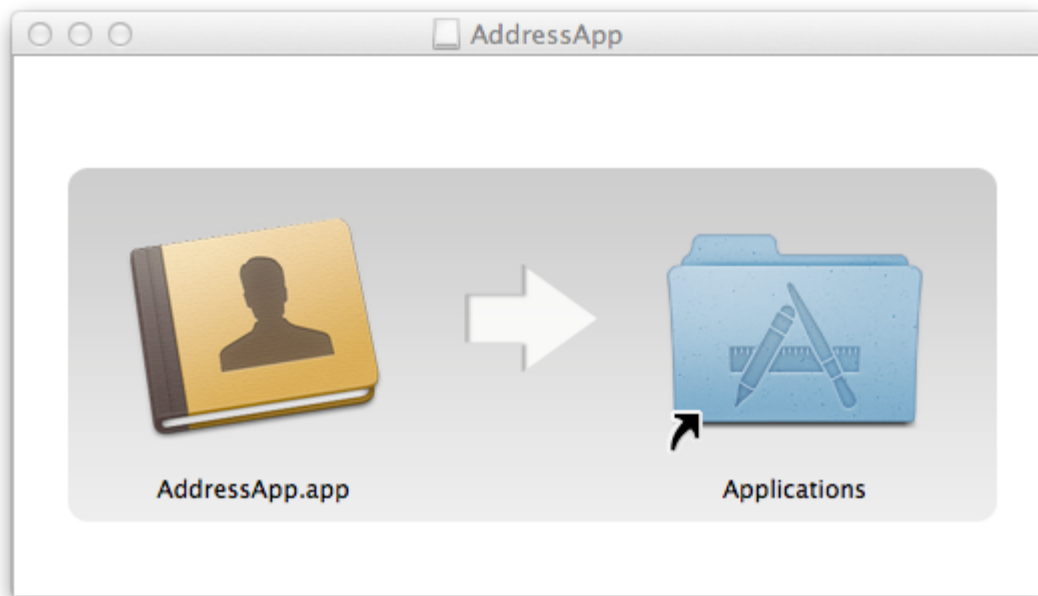
Paso 5 (WINDOWS) - Instalador de Windows (exe)



Con el programa **Inno Setup** podemos crear un instalador Windows de nuestra aplicación como un único archivo `.exe`. El archivo resultante realizará una instalación a nivel de usuario (no se requieren permisos de administrador). Además se creará un enlace (menú de inicio o escritorio).

1. Descarga [Inno Setup 5 o posterior](#). Instala Inno Setup en tu computadora. Nuestro script Ant lo usará automáticamente para generar el instalador.
2. Publica la ruta donde está instalado Inno Setup (e.g. `C:\Program Files (x86)\Inno Setup 5`). Para hacerlo, añade esa ruta a la variable `Path` en tus variables de entorno de Windows. Si no sabes donde encontrarlas, lee [How to set the path and environment variables in Windows](#).
3. Reinicia Eclipse y continúa con el Paso 5.

Paso 5 (MAC) - Instalador de MacOS (dmg)



Para crear un instalador de Mac OS (**dmg**) no se requiere ninguna herramienta adicional.

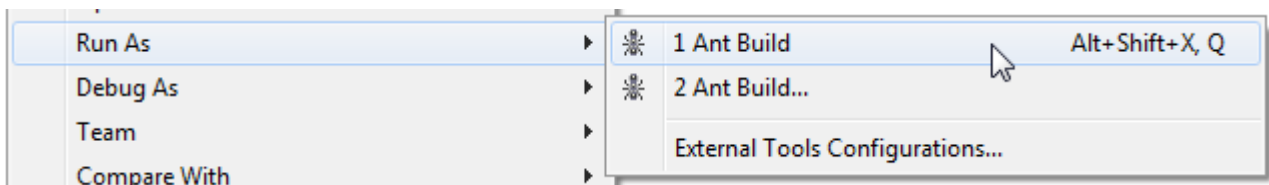
Nota: Para que la imagen del instalador funcione debe tener exactamente el mismo nombre que la aplicación.

Paso 5 (LINUX etc.) - Instalador de Linux (rpm)

Para otras opciones de instalación (ej. **msi** para Windows o **rpm** para Linux) consulta [este artículo](#) o la [documentación de Oracle](#).

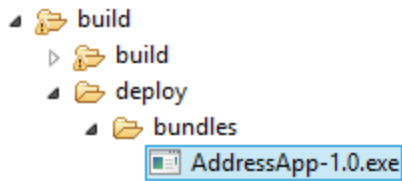
Paso 6 - Ejecuta build.xml

Como último paso, tenemos que ejecutar **build.xml** con Ant: *Clic derecho sobre **build.xml** | Run As | Ant Build.*



La ejecución de **build.xml** **puede tardar un poco** (dependiendo de la máquina que lo ejecute).

Si el proceso concluye con éxito, deberías encontrar un lote de instaladores nativos en la carpeta `build/deploy/bundles`. Este es el aspecto que tiene la versión de Windows:



El archivo `AddressApp-1.0.exe` puede ser como archivo de instalación independiente. Este instalador copiará el lote en `C:/Users/[yourname]/AppData/Local/AddressApp`.

¿Qué es lo siguiente?

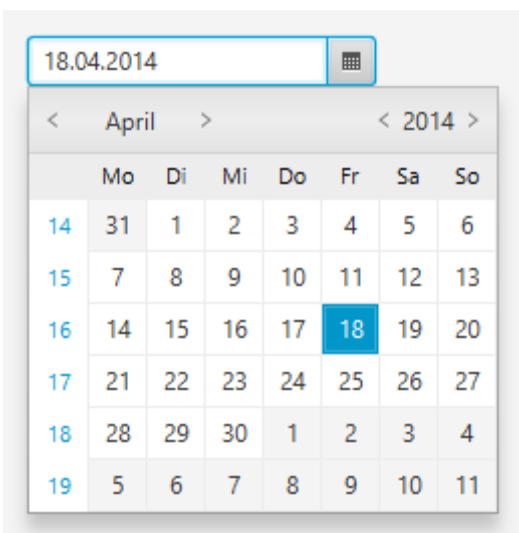
Espero que este tutorial te haya resultado de ayuda iniciarte en JavaFX y a partir de aquí seas capaz de desarrollar tu propio proyecto.

Cualquier feedback es bienvenido. No dudes en comentar si tienes sugerencias que hacer o algo no te ha quedado claro.

JavaFX 8 Date Picker

Finally, with JavaFX 8 a `DatePicker` control was added (for JavaFX 2 we had to [create our own](#))!

The `DatePicker` works well in combination with the new **Java 8 Date and Time API**. Those two things together provide a much better experience when working with dates in Java!



Basic Usage

Using the DatePicker is straight forward:

```
// Create the DatePicker.
DatePicker datePicker = new DatePicker();

// Add some action (in Java 8 lambda syntax style).
datePicker.setOnAction(event -> {
    LocalDate date = datePicker.getValue();
    System.out.println("Selected date: " + date);
});

// Add the DatePicker to the Stage.
StackPane root = new StackPane();
root.getChildren().add(datePicker);
stage.setScene(new Scene(root, 500, 650));
stage.show();
```

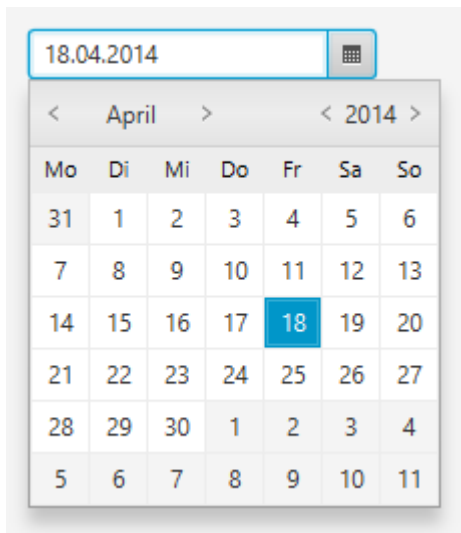
Using FXML

I usually prefer keeping as much of the view in `fxml` instead of instantiating the controls in Java code as above. With [Scene Builder 2.0](#) and above you can even drag-and-drop the DatePicker into your `fxml`.

Options

Hide Week Numbers

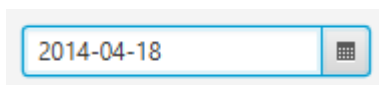
We can set the `showWeekNumbersProperty` to true/false to show/hide a column showing week numbers. Note: The default value depends on the country of the current locale.



```
datePicker.setShowWeekNumbers(false);
```

Date Converter

The date in the text field is automatically converted to the local format (in my case, it's `dd.MM.yyyy`). By setting a `StringConverter` we can change this, for example, to `yyyy-MM-dd`.



```
String pattern = "yyyy-MM-dd";

datePicker.setPromptText(pattern.toLowerCase());

datePicker.setConverter(new StringConverter<LocalDate>() {
    DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern(pattern);

    @Override
    public String toString(LocalDate date) {
        if (date != null) {
            return dateFormatter.format(date);
        }
    }
});
```

```

        } else {
            return "";
        }
    }

@Override
public LocalDate fromString(String string) {
    if (string != null && !string.isEmpty()) {
        return LocalDate.parse(string, dateFormatter);
    } else {
        return null;
    }
}

});

```

Other Calendar Systems

If you need a different chronology, e.g. the Japanese Imperial calendar system, you can change it as follows:

```

// Japanese calendar.
datePicker.setChronology(JapaneseChronology.INSTANCE);

// Hijrah calendar.
datePicker.setChronology(HijrahChronology.INSTANCE);

// Minguo calendar.
datePicker.setChronology(MinguoChronology.INSTANCE);

// Buddhist calendar.

```

```
datePicker.setChronology(ThaiBuddhistChronology.INSTANCE);
```

Here is a screenshot of the Hijrah calendar:

15.7.1435 AH

< Mai > < 2014 >

Rajab - Sha'ban 1435

	Mo	Di	Mi	Do	Fr	Sa	So
18	28 28	29 29	30 1	1 2	2 3	3 4	4 5
19	5 6	6 7	7 8	8 9	9 10	10 11	11 12
20	12 13	13 14	14 15	15 16	16 17	17 18	18 19
21	19 20	20 21	21 22	22 23	23 24	24 25	25 26
22	26 27	27 28	28 29	29 30	30 1	31 2	1 3
23	2 4	3 5	4 6	5 7	6 8	7 9	8 10