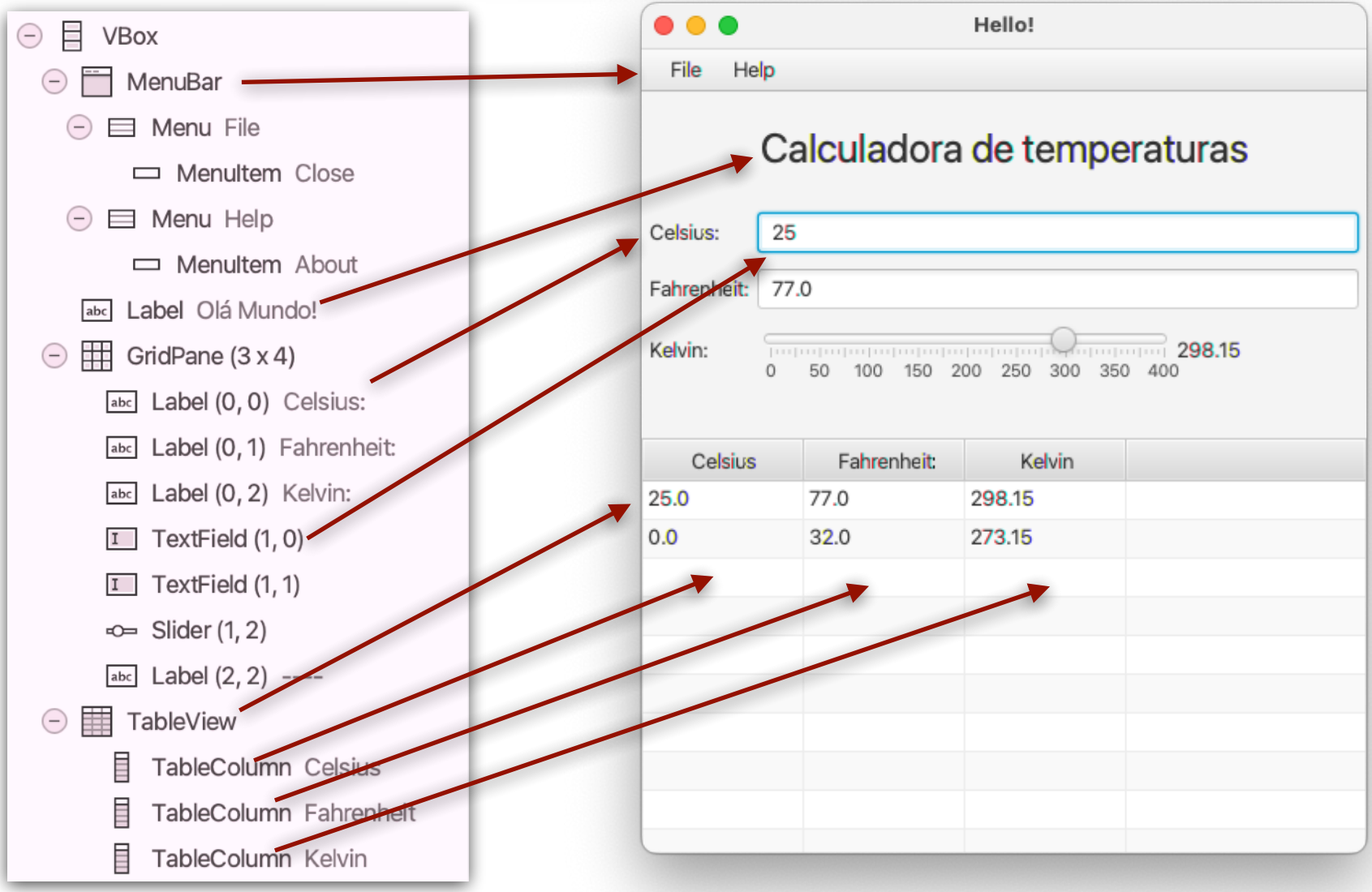
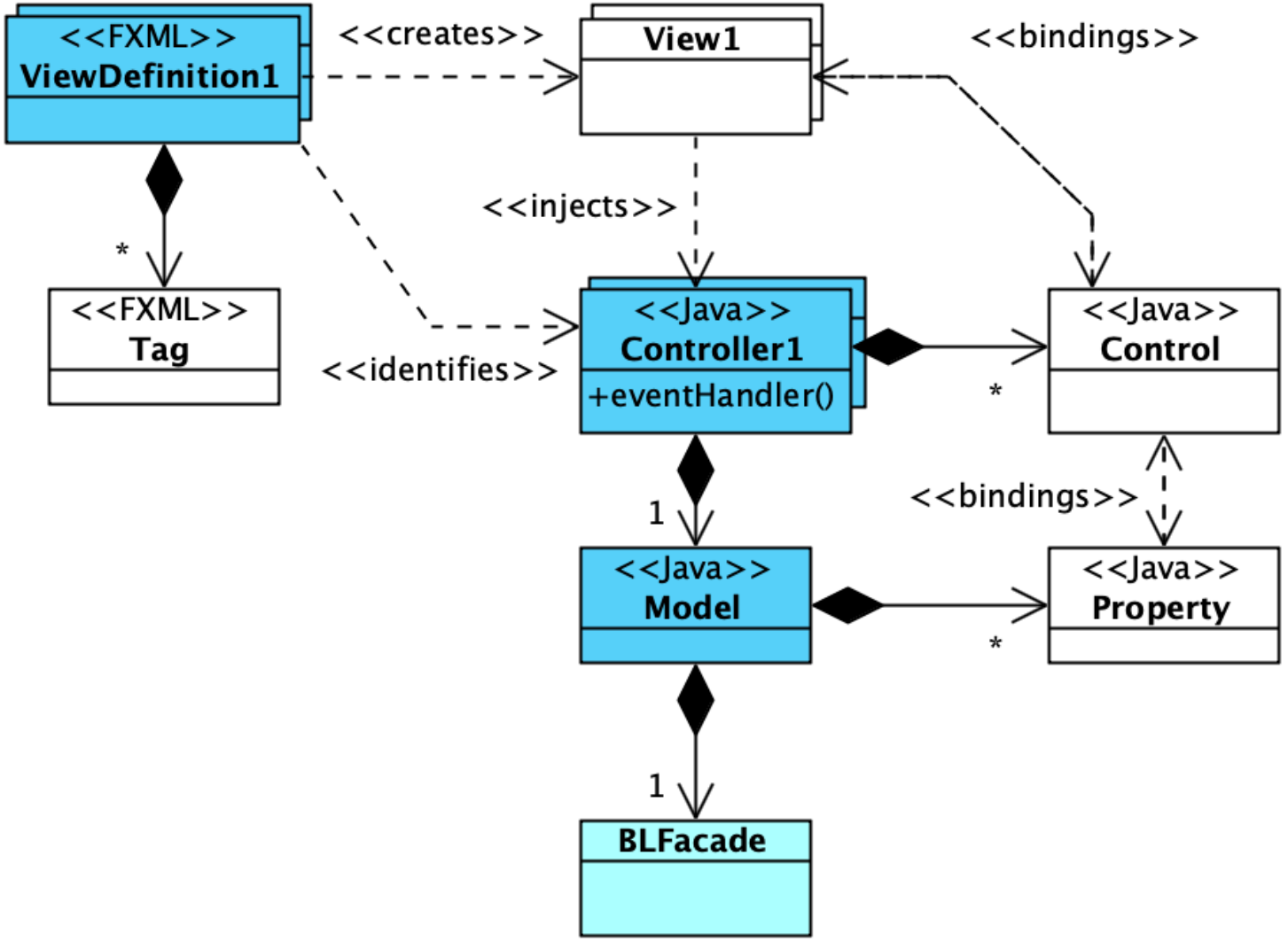
Módulo 9

JAVAFX (CONT.)

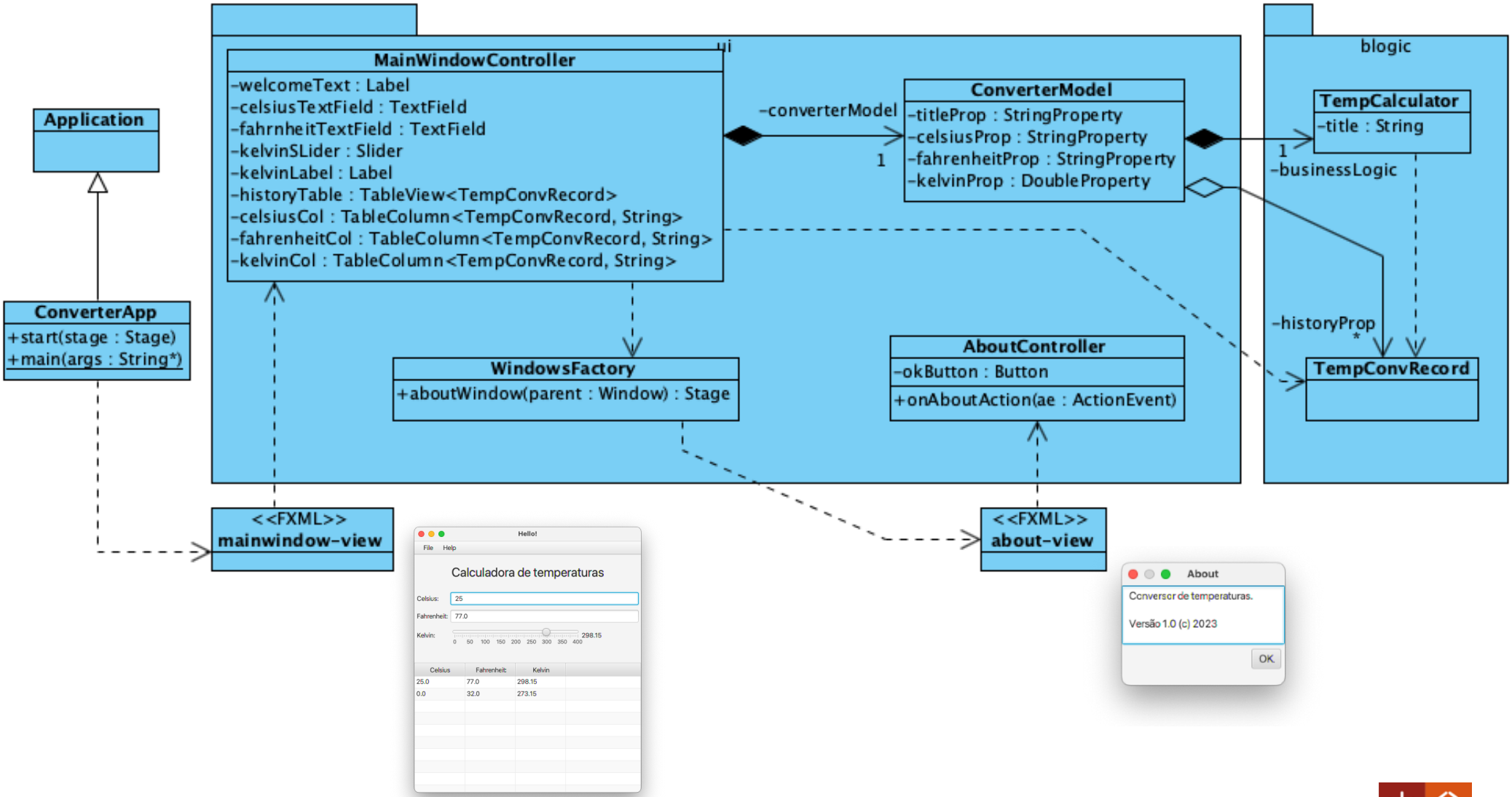
JavaFX GUIs — the example so far…



Generic architecture



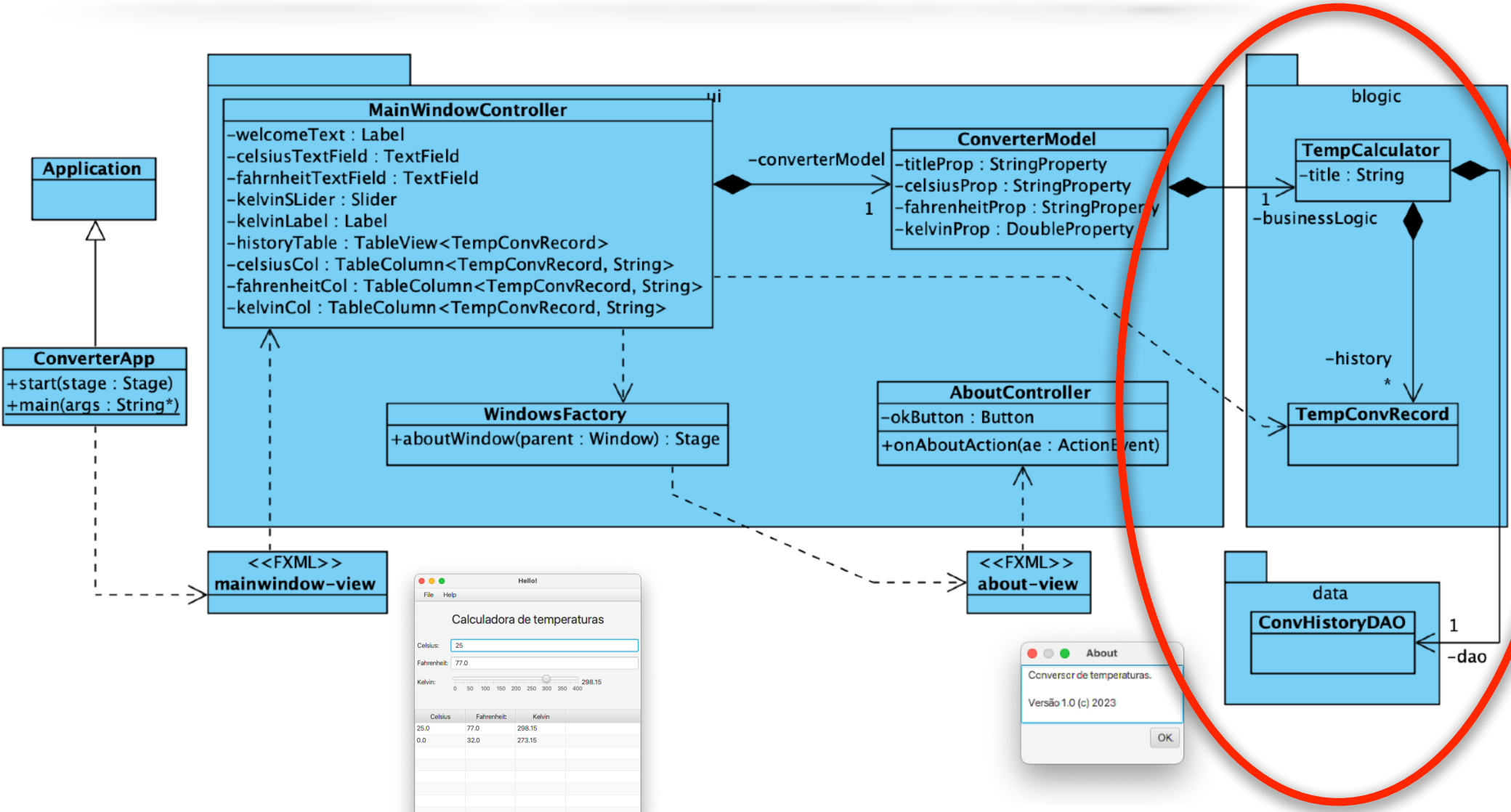
Anatomy of the App



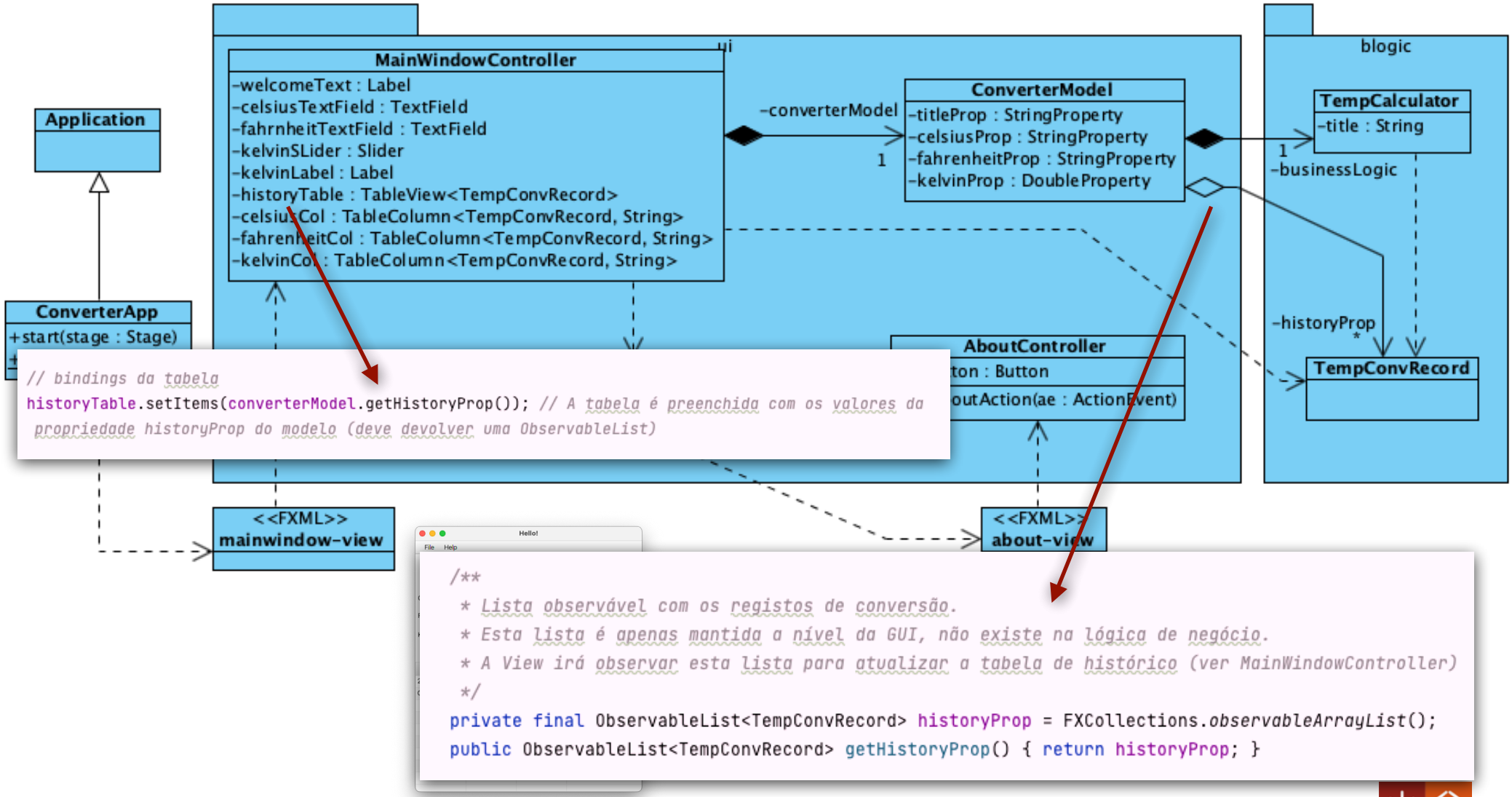
What we want to change

- Add persistence to history table
- Improve handling of numbers
  - Restrict the input to numbers
- Add information about history size

Persistence - Architectural changes

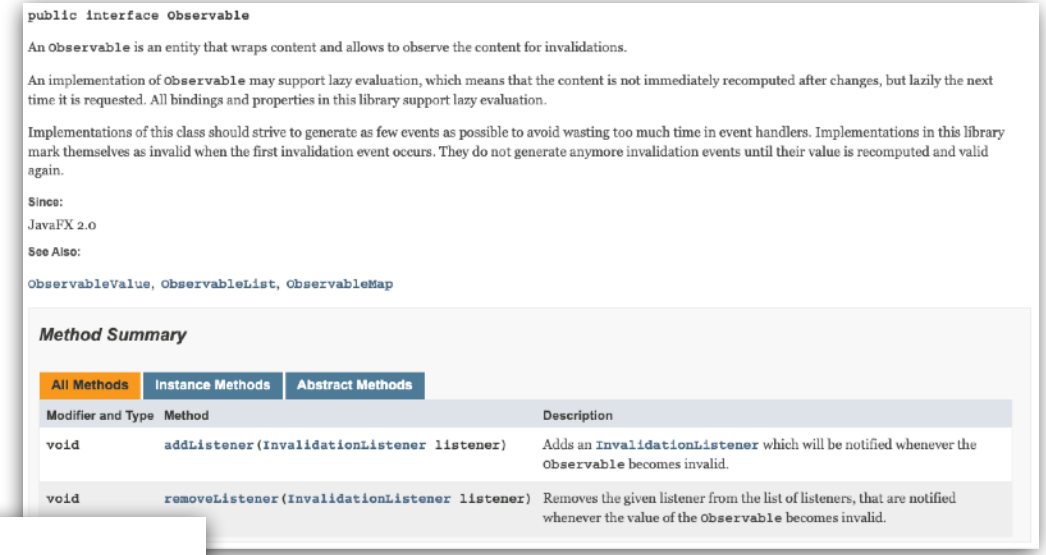


History table - current version



Observables

- changeListeners
- InvalidationListeners



`public interface Observable`

An `Observable` is an entity that wraps content and allows to observe the content for invalidations.

An implementation of `Observable` may support lazy evaluation, which means that the content is not immediately recomputed after changes, but lazily the next time it is requested. All bindings and properties in this library support lazy evaluation.

Implementations of this class should strive to generate as few events as possible to avoid wasting too much time in event handlers. Implementations in this library mark themselves as invalid when the first invalidation event occurs. They do not generate anymore invalidation events until their value is recomputed and valid again.

Since:
JavaFX 2.0

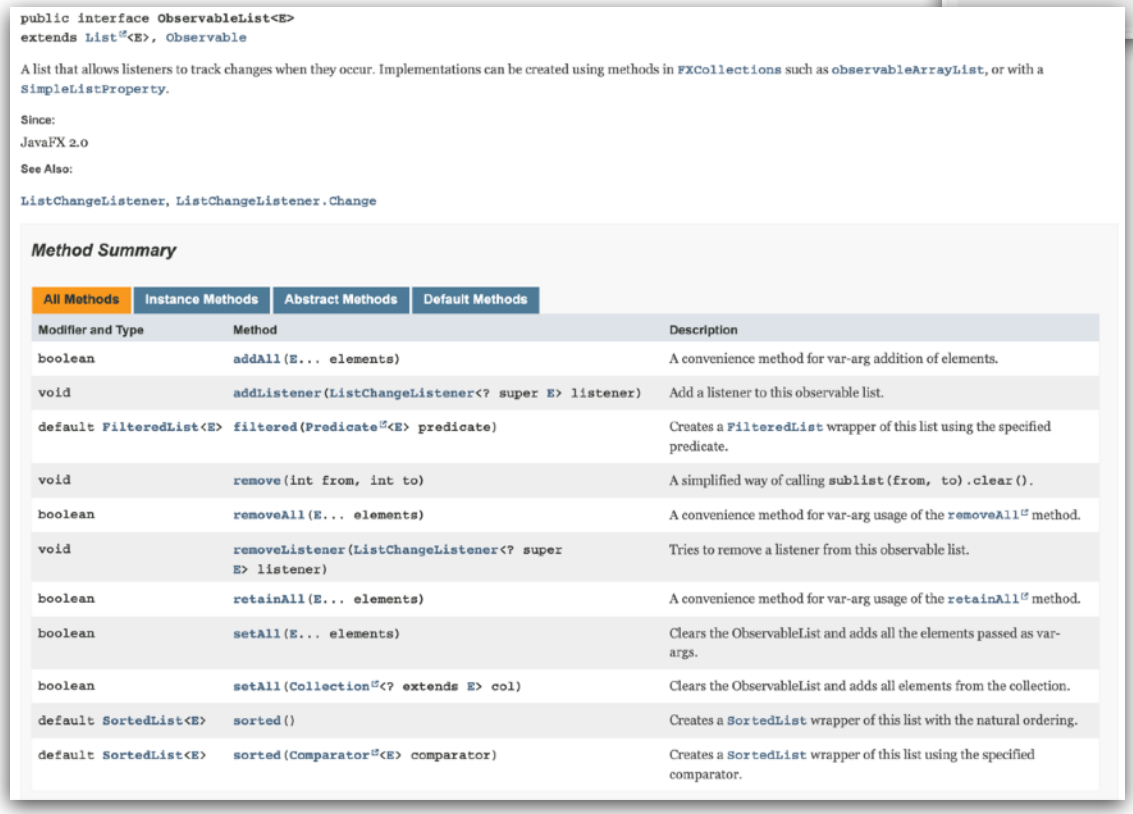
See Also:

`ObservableValue`, `ObservableList`, `ObservableMap`

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
`void`	`addListener(InvalidationListener listener)`	Adds an `InvalidationListener` which will be notified whenever the `Observable` becomes invalid.
`void`	`removeListener(InvalidationListener listener)`	Removes the given listener from the list of listeners, that are notified whenever the value of the `Observable` becomes invalid.



`public interface ObservableList<E>`
`extends List<E>, Observable`

A list that allows listeners to track changes when they occur. Implementations can be created using methods in `FXCollections` such as `observableArrayList`, or with a `SimpleListProperty`.

Since:
JavaFX 2.0

See Also:

`ListChangeListener`, `ListChangeListener.Change`

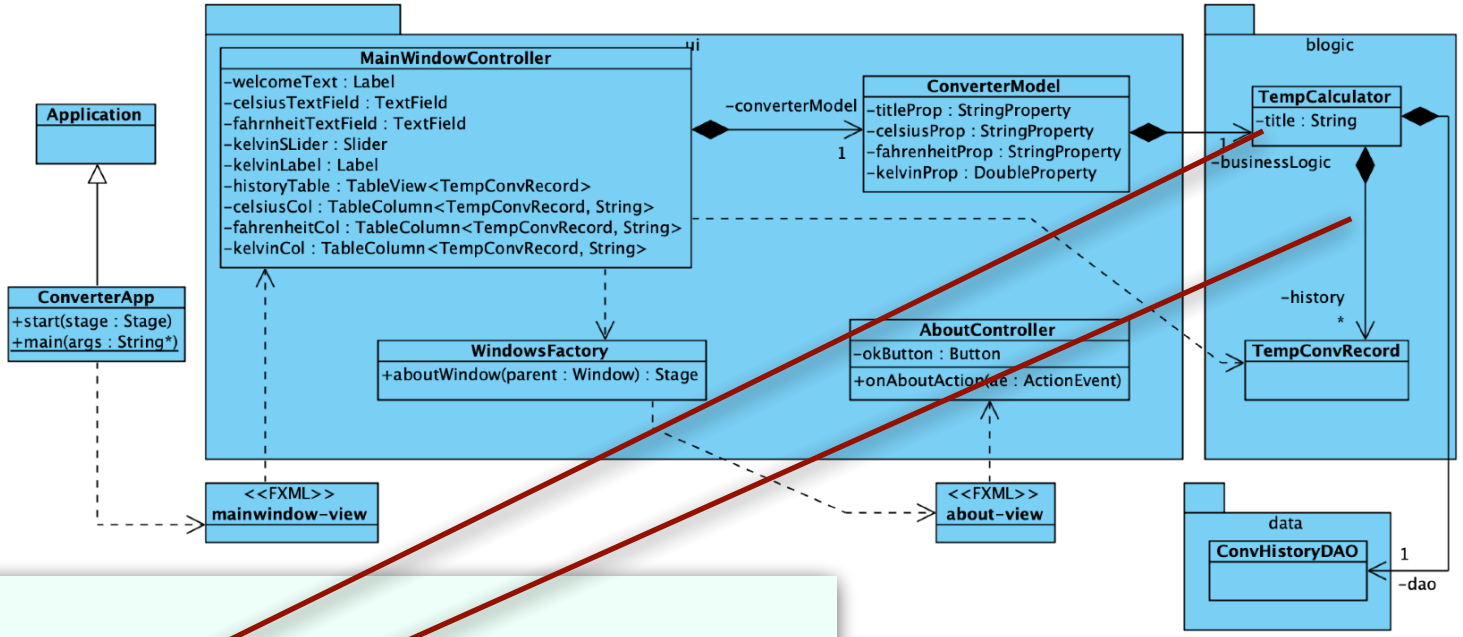
Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
`boolean`	`addAll(E... elements)`	A convenience method for var-arg addition of elements.
`void`	`addListener(ListChangeListener<? super E> listener)`	Add a listener to this observable list.
`default FilteredList<E>`	`filtered(Predicate<E> predicate)`	Creates a `FilteredList` wrapper of this list using the specified predicate.
`void`	`remove(int from, int to)`	A simplified way of calling `sublist(from, to).clear()`.
`boolean`	`removeAll(E... elements)`	A convenience method for var-arg usage of the `removeAll` method.
`void`	`removeListener(ListChangeListener<? super E> listener)`	Tries to remove a listener from this observable list.
`boolean`	`retainAll(E... elements)`	A convenience method for var-arg usage of the `retainAll` method.
`boolean`	`setAll(E... elements)`	Clears the ObservableList and adds all the elements passed as var-args.
`boolean`	`setAll(Collection<? extends E> col)`	Clears the ObservableList and adds all elements from the collection.
`default SortedList<E>`	`sorted()`	Creates a `SortedList` wrapper of this list with the natural ordering.
`default SortedList<E>`	`sorted(Comparator<E> comparator)`	Creates a `SortedList` wrapper of this list using the specified comparator.

Persistence - Architecture changes

- The Facade is not just an API, we also have to decide which parts of its state will be monitores by the GUI (via Properties and Observables)




```
// Variáveis de instância
/**
 * Título da aplicação - Esta variável é usada para definir o título da janela principal da
 * aplicação. É uma propriedade, para que possa ser usada diretamente na definição do título
 * da janela.
 */
private final StringProperty title = new SimpleStringProperty( s: "Conversor de Temperaturas");
private ObservableList<TempConvRecord> history;
private final ConvHistoryDAO dao = ConvHistoryDAO.getInstance();
```

Properties

```
public interface Property<T>
extends ReadOnlyProperty<T>, WritableValue<T>
```

Generic interface that defines the methods common to all (writable) properties, independent of their type.

Since:

JavaFX 2.0

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
void	bind(ObservableValue<? extends T> observable)	Create a unidirection binding for this Property.
void	bindBidirectional(Property<T> other)	Create a bidirectional binding between this Property and another one.
boolean	isBound()	Can be used to check, if a Property is bound.
void	unbind()	Remove the unidirectional binding for this Property.
void	unbindBidirectional(Property<T> other)	Removes a bidirectional binding between this Property and another one.

Methods declared in interface javafx.beans.Observable

addListener, removeListener

Methods declared in interface javafx.beans.value.ObservableValue

addListener, flatMap, getValue, map, orElse, removeListener

Methods declared in interface javafx.beans.property.ReadOnlyProperty

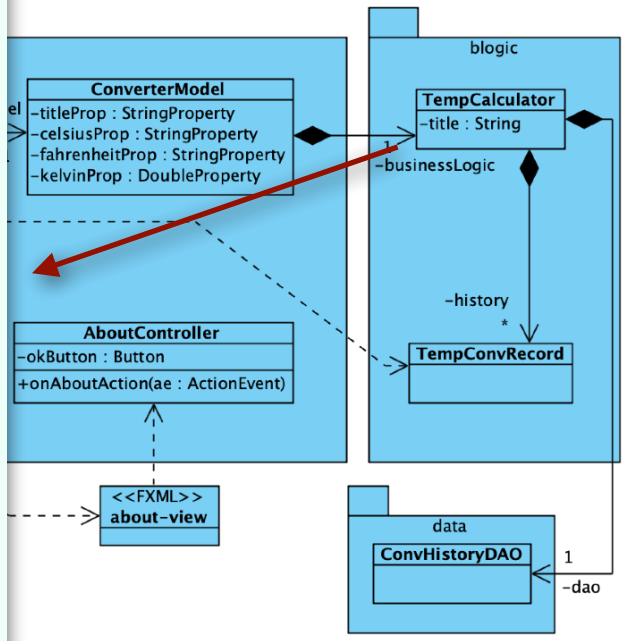
getBean, getName

Methods declared in interface javafx.beans.value.WritableValue

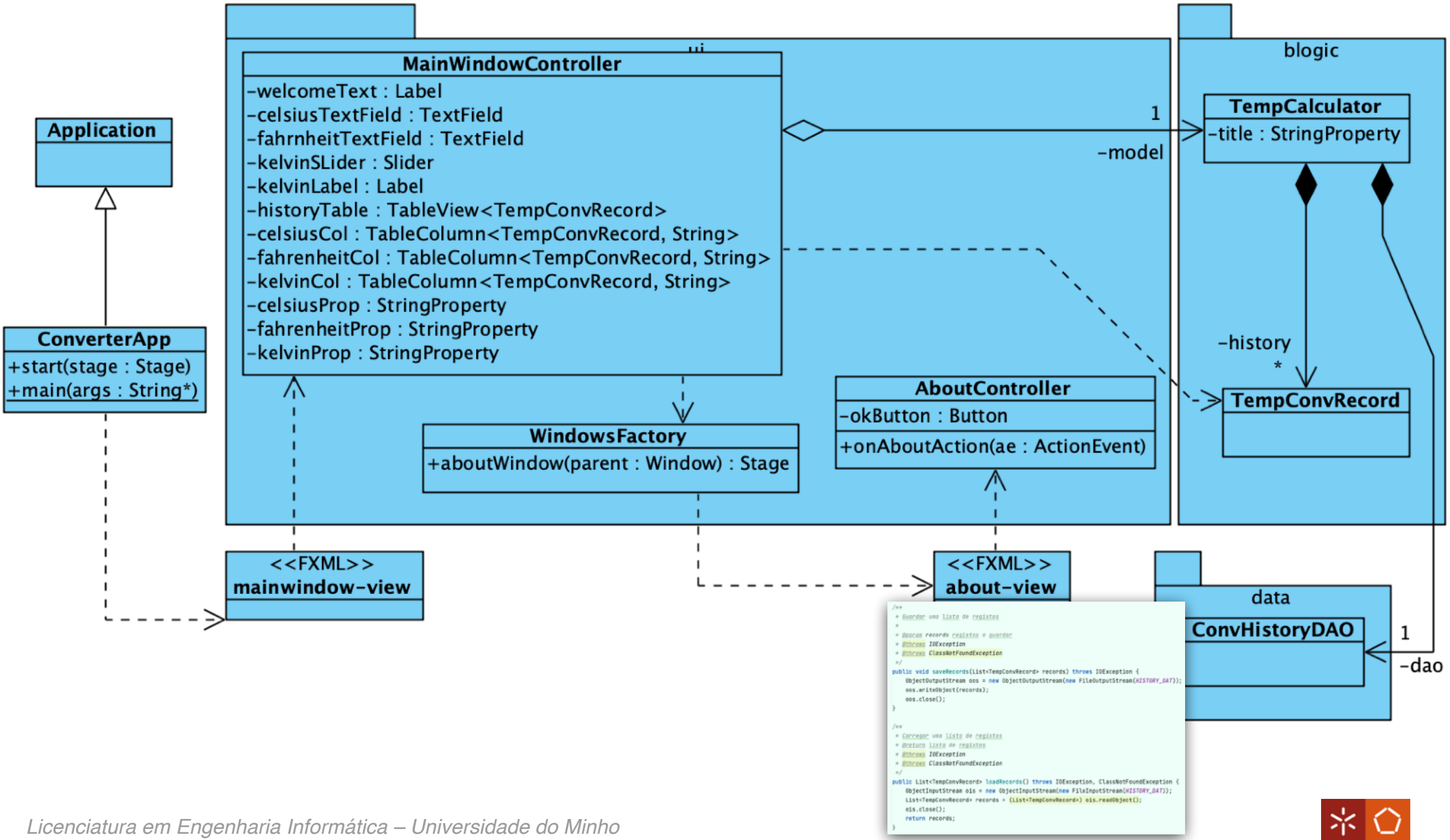
getValue, setValue

Access to Properties — conventions

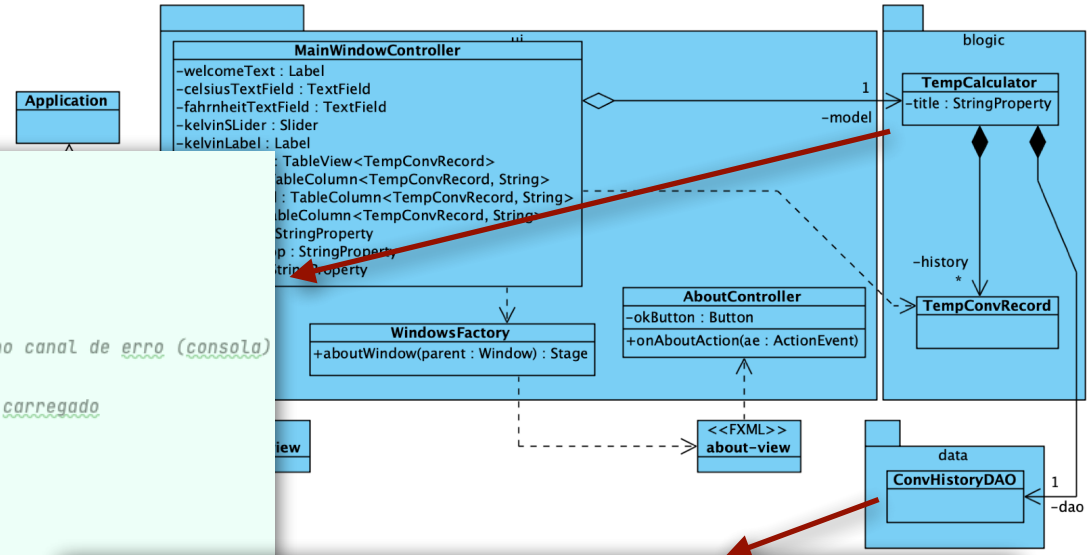
```java
public class TempCalculator {

    // Variáveis de instância

/** Título da aplicação - Esta variável é usada para definir o título da janela principal da ...*/
    private final StringProperty title = new SimpleStringProperty( s: "Conversor de Temperaturas");
    private ObservableList<TempConvRecord> history;
    private final ConvHistoryDAO dao = ConvHistoryDAO.getInstance();

    // Construtores
    /** Construtor por omissão */
    public TempCalculator() {...}

    // Métodos de instância

    /**
     * Obter o título da aplicação como propriedade
     *
     * @return Propriedade com o título da aplicação
     */
    public StringProperty titleProperty() {
        return this.title;
    }

    /**
     * Obter o título da aplicação como String
     *
     * @return String com o título da aplicação
     */
    public String getTitle() {
        return this.title.get();
    }
```



Architecture



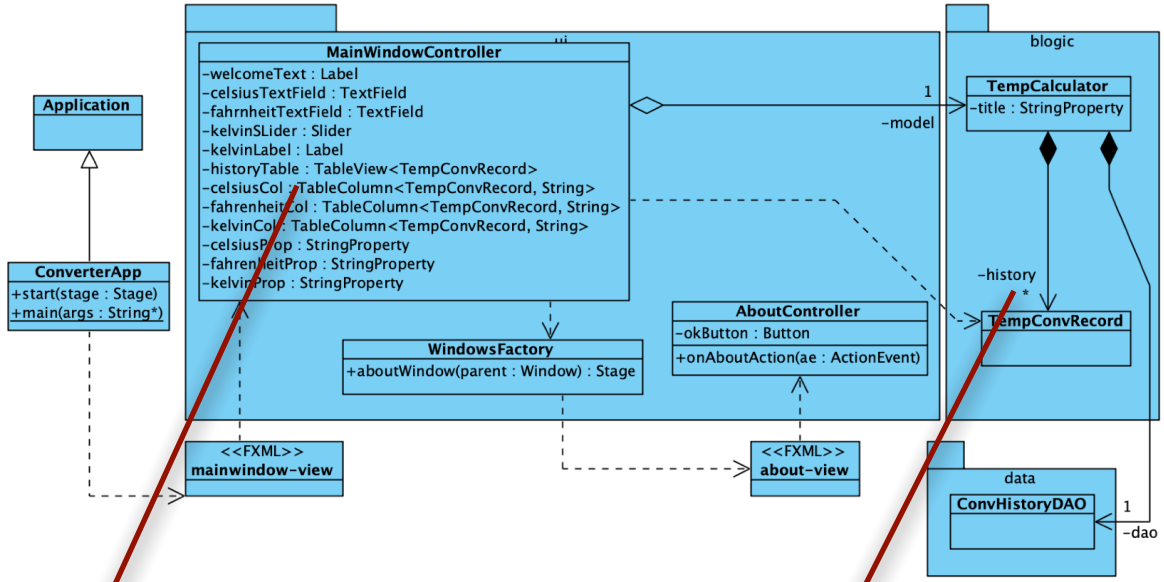
Data layer




```java
/** Construtor por omissão */
public TempCalculator() {
    try {
        this.history = FXCollections.observableArrayList(dao.loadRecords());
    }
    catch (IOException | ClassNotFoundException e) {
        System.err.println("Erro a carregar o histórico de conversões"); // Escreve no canal de erro (consola)
        this.history = FXCollections.observableArrayList();
        // Poderíamos também lançar uma exceção para indicar que o histórico não foi carregado
    }
}

// Métodos de instância

/** Obter o título da aplicação como propriedade ...*/
public StringProperty titleProperty() { return this.title; }

/** Obter o título da aplicação como String ...*/
public String getTitle() { return this.title.get(); }

/** Obter o histórico de conversões ...*/
public ObservableList<TempConvRecord> historyProperty() {...}

/** Converter um valor em Celsius ...*/
public TempConvRecord convertCelsius(double celsius) {...}

/** Converter um valor em Fahrenheit ...*/
public TempConvRecord convertFahrenheit(double fahrenheit) {...}

/** Converter um valor em Kelvin ...*/
public TempConvRecord convertKelvin(double kelvin) {...}

/**
 * Guardar o histórico de conversões
 * @throws IOException
 */
public void saveHistory() throws IOException {
    dao.saveRecords(this.history.stream().collect(Collectors.toList()));
}
```

```java
/**
 * Guardar uma lista de registos
 *
 * @param records registos a guardar
 * @throws IOException
 * @throws ClassNotFoundException
 */
public void saveRecords(List<TempConvRecord> records) throws IOException {
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(HISTORY_DAT));
    oos.writeObject(records);
    oos.close();
}

/**
 * Carregar uma lista de registos
 * @return lista de registos
 * @throws IOException
 * @throws ClassNotFoundException
 */
public List<TempConvRecord> loadRecords() throws IOException, ClassNotFoundException {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(HISTORY_DAT));
    List<TempConvRecord> records = (List<TempConvRecord>) ois.readObject();
    ois.close();
    return records;
}
```

Binding…



MainWindowController::initialise()

```
// bindings da tabela
historyTable.setItems(model.historyProperty()); // A tabela é preenchida
 com os valores da propriedade historyProp do modelo (deve devolver uma
 ObservableList)
```

```
/**
 * Obter o histórico de conversões
 *
 * @return histórico de conversões
 */
public ObservableList<TempConvRecord> historyProperty() {
    return FXCollections.unmodifiableObservableList(this.history);
}
```

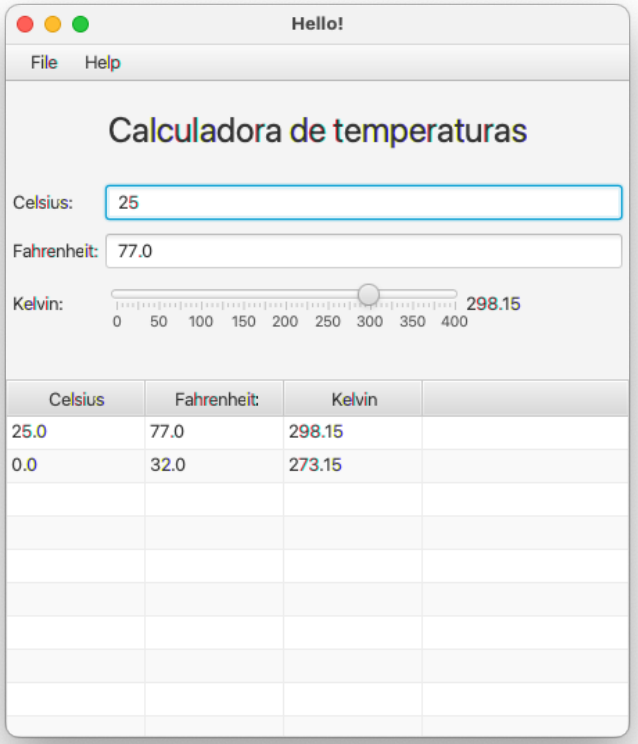
Platform.runLater()

- Window title is not right!
- Controller initialise() method:

MainWindowController::initialise()

```
((Stage)welcomeText.getScene().getWindow()).setTitle(model.getTitle());
```

```
Platform.runLater(() -> {
    ((Stage)welcomeText.getScene().getWindow()).setTitle(model.getTitle());
});
```




```
Platform.runLater(() -> {
    ((Stage)welcomeText.getScene().getWindow()).titleProperty().bind(model.titleProperty());
}); // Binding unidireccional - o título da janela é atualizado sempre que o título muda no modelo
```

Binding TextFields to doubles…

```java
/**
 * Atualiza as propriedades com base no valor em Celsius
 */
public void updateForCelsius() {
    try {
        TempConvRecord record = businessLogic.convertCelsius(Double.parseDouble(celsiusProp.get()));
        fahrenheitProp.set(String.valueOf(record.getFahrenheit()));
        kelvinProp.set(record.getKelvin());
        historyProp.add( index: 0,record);
    } catch (NumberFormatException e) {  // Em caso de erro, repõe os valores anteriores
        celsiusProp.set(String.valueOf(historyProp.get(0).getCelsius()));
        fahrenheitProp.set(String.valueOf(historyProp.get(0).getFahrenheit()));
        kelvinProp.set(historyProp.get(0).getKelvin());
    }
}
```

Use of StringProperties complicates handling of input values in event handlers.

```java
public class MainWindowController {
    // Variáveis de instância - Model
    private final TempCalculator model = new TempCalculator();

    // Variáveis de instância - propriedades auxiliares para controlar a View
    private final DoubleProperty celsiusProp = new SimpleDoubleProperty();
    private final DoubleProperty fahrenheitProp = new SimpleDoubleProperty();
    private final DoubleProperty kelvinProp = new SimpleDoubleProperty();

    // Variáveis de instância - Controlos da View
    @FXML
    private Label welcomeText;
    @FXML
    private TextField celsiusTextField;
    @FXML
    private TextField fahrnheitTextField;
```

TextFormatters and Converters

Class TextFormatter<V>

java.lang.Object
javafx.scene.control.TextFormatter<V>

Type Parameters:
V - The type of the value

```
public class TextFormatter<V>
extends Object
```

A Formatter describes a format of a `TextInputControl` text by using two distinct mechanisms:

- A filter (`getFilter()`) that can intercept and modify user input. This helps to keep the text in the desire
  provide the initial text.
- A value converter (`getValueConverter()`) and value (`valueProperty()`) can be used to provide spe
  control is editable and the text is changed by the user, the value is then updated to correspond to the text.

It's possible to have a formatter with just a filter or a value converter. If a value converter is not provided, setting
`IllegalStateException` and the value is always `null`.

Since `Formatter` contains a value that represents the state of the `TextInputControl` to which it is currently
only in one `TextInputControl` at a time.

Since:
JavaFX 8u40

Constructor Summary

Constructors

Constructor	Description
`TextFormatter(UnaryOperator<TextFormatter.Change> filter)`	Creates a new Formatter with the provided filter.
`TextFormatter(StringConverter<V> valueConverter)`	Creates a new Formatter with the provided value converter.
`TextFormatter(StringConverter<V> valueConverter, V defaultValue)`	Creates a new Formatter with the provided value converter and default value.
`TextFormatter(StringConverter<V> valueConverter, V defaultValue, UnaryOperator<TextFormatter.Change> filter)`	Creates a new Formatter with the provided filter, value converter and default

MainWindowController::initialise()

```
// celsiusTextField e farhrenheitTextField devem conter valores numéricos
UnaryOperator<TextFormatter.Change> numberFilter = change -> { // Filtro para o texto do TextField - o
 valor introduzido deve ser um número real
    String newText = change.getControlNewText();
    // if proposed change results in a valid value, return change as-is:
    if (newText.matches( regex: "-?([1-9][0-9]*\\.?[0-9]*)?")) {
        return change;
    }
    // invalid change, veto it by returning null:
    return null;
};
```

MainWindowController::initialise()

```
// celsiusTextField e celsiusProp devem estar sempre sincronizados
TextFormatter<Number> formatter = new TextFormatter<>(new NumberStringConverter(), v: 0, numberFilter);
celsiusTextField.setTextFormatter(formatter); // Formatação do texto do TextField - o valor introduzido
 deve ser um número real
celsiusProp.bindBidirectional(formatter.valueProperty()); // Binding bidireccional - o valor da
 propriedade celsiusProp e o valor no formatter do TextField ficam sempre iguais
```

Binding TextFields to doubles…

MainWindowController::initialise()

- More on Bindings

```
// celsiusTextField e farhrenheitTextField devem conter valores numéricos
UnaryOperator<TextFormatter.Change> numberFilter = change -> { // Filtro para o texto do TextField - o
 valor introduzido deve ser um número real
    String newText = change.getControlNewText();
    // if proposed change results in a valid value, return change as-is:
    if (newText.matches( regex: "-?([1-9][0-9]*\\.?[0-9]*)?")) {
        return change;
    }
    // invalid change, veto it by returning null:
    return null;
};

// celsiusTextField e celsiusProp devem estar sempre sincronizados
TextFormatter<Number> formatter = new TextFormatter<>(new NumberStringConverter(), v: 0, numberFilter);
celsiusTextField.setTextFormatter(formatter); // Formatação do texto do TextField - o valor introduzido
 deve ser um número real
celsiusProp.bindBidirectional(formatter.valueProperty()); // Binding bidireccional - o valor da
 propriedade celsiusProp e o valor no formatter do TextField ficam sempre iguais

TextFormatter<Number> formatter2 = new TextFormatter<>(new NumberStringConverter(), v: 0, numberFilter);
fahrnheitTextField.setTextFormatter(formatter2);
fahrenheitProp.bindBidirectional(formatter2.valueProperty());
```

Binding TextFields to doubles…

```java
/**
 * Acção do TestField dos graus Celsius (tecla Enter)
 * @param actionEvent the event
 */
@FXML
protected void onCelsiusAction(ActionEvent actionEvent) {
    TempConvRecord record = model.convertCelsius(celsiusProp.get());
    fahrenheitProp.set(record.getFahrenheit());
    kelvinProp.set(record.getKelvin());
}

/**
 * Acção do TestField dos graus Fahrenheit (tecla Enter)
 * @param actionEvent the event
 */
@FXML
protected void onFahrenheitAction(ActionEvent actionEvent) {
    TempConvRecord record =  model.convertFahrenheit(fahrenheitProp.get());
    celsiusProp.set(record.getCelsius());
    kelvinProp.set(record.getKelvin());
}

/**
 * Acção do Slider dos graus Kelvin (mouse released)
 * @param mouseEvent the event
 */
@FXML
protected void onKelvinMouseReleased(MouseEvent mouseEvent) {
    TempConvRecord record = model.convertKelvin(kelvinProp.get());
    celsiusProp.set(record.getCelsius());
    fahrenheitProp.set(record.getFahrenheit());
}
```

```java
/**
 * Atualiza as propriedades com base no valor em Celsius
 */
public void updateForCelsius() {
    try {
        TempConvRecord record = businessLogic.convertCelsius(Double.parseDouble(celsiusProp.get()));
        fahrenheitProp.set(String.valueOf(record.getFahrenheit()));
        kelvinProp.set(record.getKelvin());
        historyProp.add( index: 0,record);
    } catch (NumberFormatException e) {  // Em caso de erro, repõe os valores anteriores
        celsiusProp.set(String.valueOf(historyProp.get(0).getCelsius()));
        fahrenheitProp.set(String.valueOf(historyProp.get(0).getFahrenheit()));
        kelvinProp.set(historyProp.get(0).getKelvin());
    }
}
```

Simpler event handlers.

Values only change with enter!

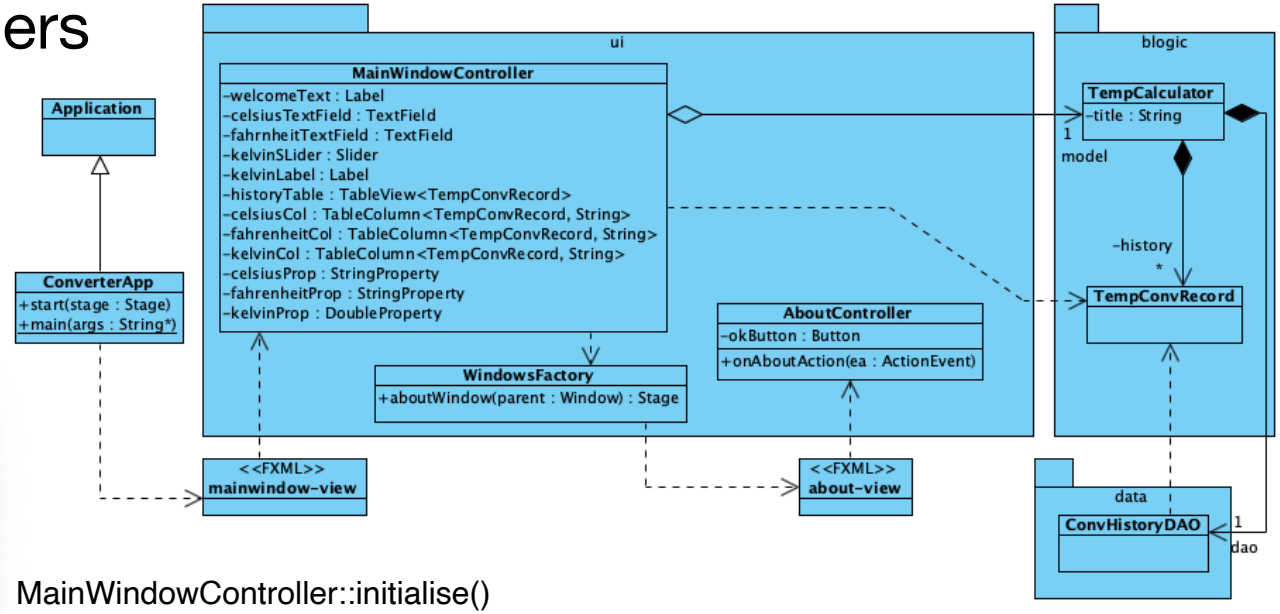
- Change them also when GUI Control looses focus (Tab key, clicking somewhere else…)
- Event handler:

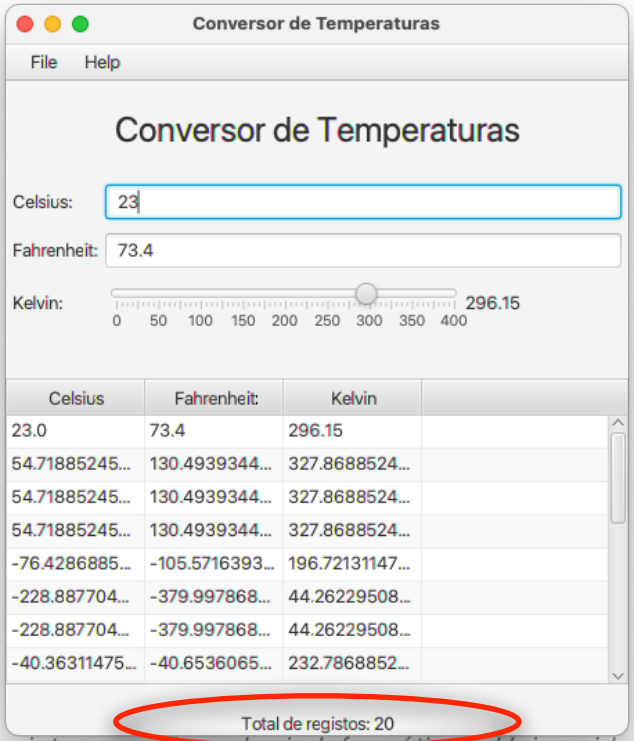
MainWindowController::initialise()

```
// Listener para o foco do TextField - quando o foco é perdido, o valor do TextField é processado
celsiusTextField.focusedProperty().addListener((observable, oldVal, newValue) -> {
    if (!newValue && celsiusProp.get()!=historyTable.getItems().get(0).getCelsius()) {
        // Se o foco foi perdido e o valor do TextField é diferente do valor na tabela
        onCelsiusAction( actionEvent: null);
    }
});
```

Presenting the history size

- with ChangeListeners





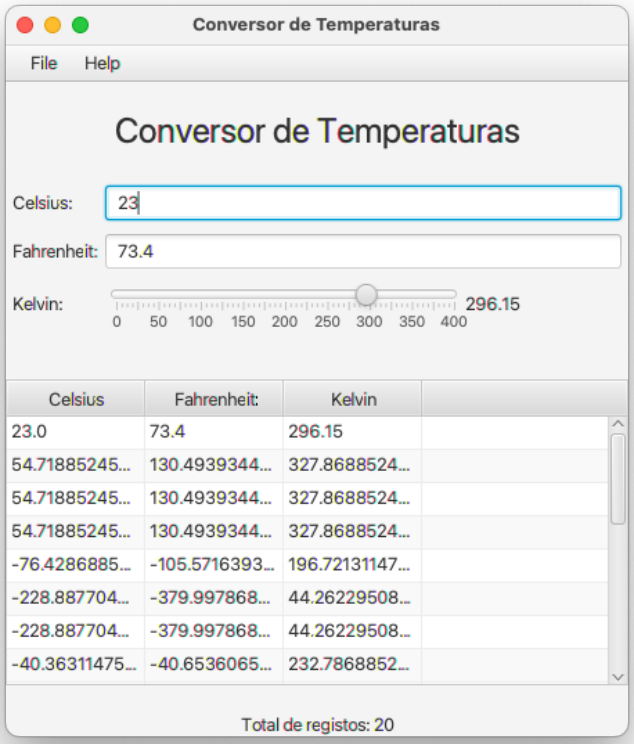
MainWindowController::initialise()

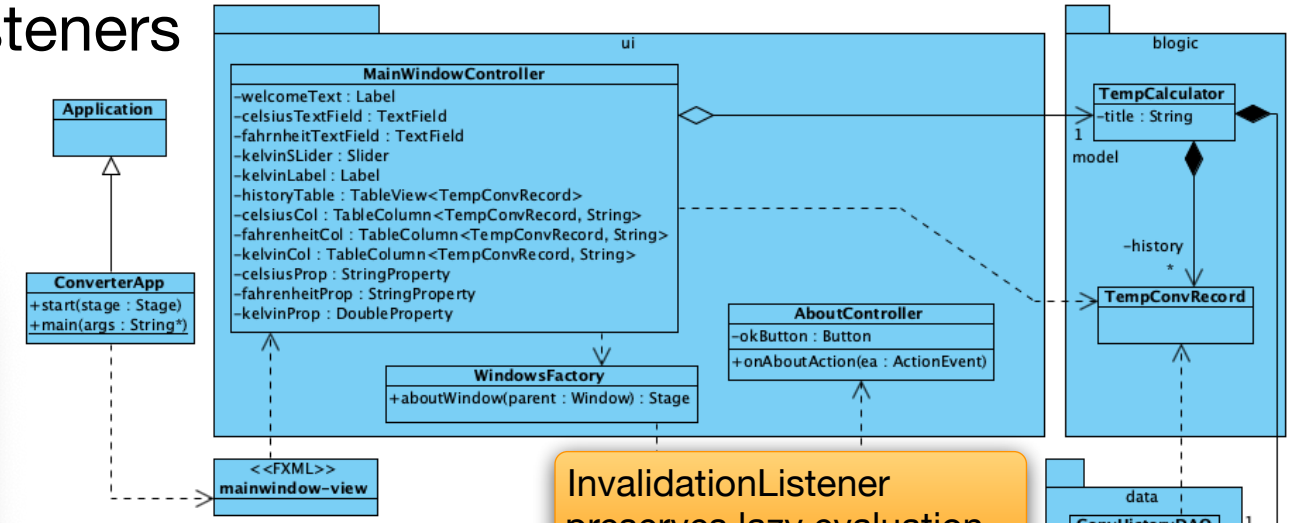
```
historyTable.getItems().addListener(new ListChangeListener<TempConvRecord>() {
    @Override
    public void onChanged(Change<? extends TempConvRecord> change) {
        messagesTextField.setText("Total de registos: " + historyTable.getItems().size());
    }
});
```

ChangeListener makes lazy evaluation impossible!

Presenting the history size

- with InvalidationListeners





InvalidationListener preserves lazy evaluation

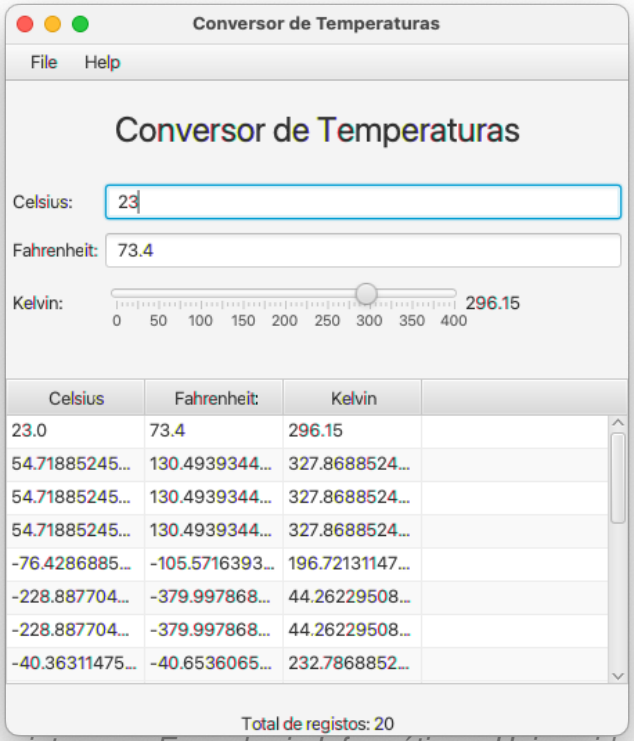
MainWindowController::initialise()

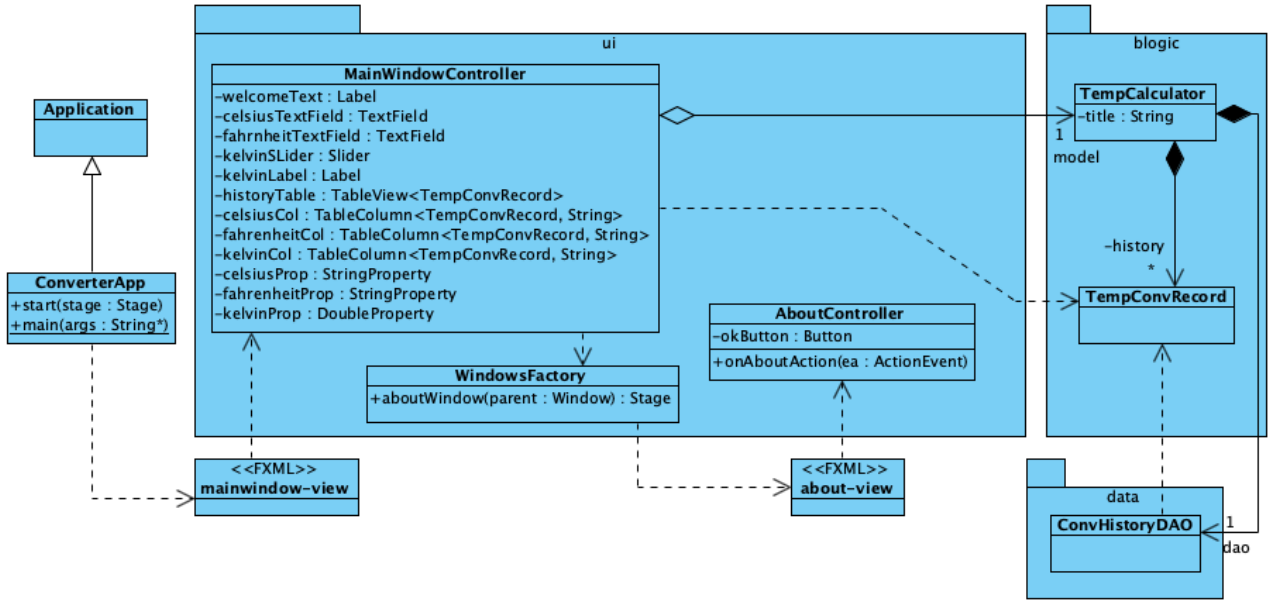
```
historyTable.getItems().addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable observable) {
        messagesTextField.setText("Total de registos: " + historyTable.getItems().size());
    }
});
```

```
historyTable.getItems().addListener((InvalidationListener) observable -> {
    messagesTextField.setText("Total de registos: " + historyTable.getItems().size());
});
```

Presenting the history size

- with Bindings





MainWindowController::initialise()

```
messagesTextField.textProperty()
                 .bind(Bindings.size(historyTable.getItems())
                               .asString( s: "Total de registos: %d"));
```

Bindings

Class Bindings

java.lang.Object
    javafx.beans.binding.Bindings

```
public final class Bindings
extends Object
```

Bindings is a helper class with a lot of utility functions to create simple bindings.

Interface Binding<T>

Type Parameters:

`T` - the type of the wrapped value

All Superinterfaces:

`Observable`, `ObservableValue<T>`

All Known Subinterfaces:

`NumberBinding`

All Known Implementing Classes:

`BooleanBinding`, `DoubleBinding`, `FloatBinding`, `IntegerBinding`, `ListBinding`, `LongBinding`, `MapBinding`, `ObjectBinding`, `SetBinding`, `StringBinding`

```
public interface Binding<T>
extends ObservableValue<T>
```

A `Binding` calculates a value that depends on one or more sources. The sources are usually called the dependency of a binding. A binding observes its dependencies for changes and updates its value automatically.

While a dependency of a binding can be anything, it is almost always an implementation of `ObservableValue`. `Binding` implements `ObservableValue` allowing to use it in another binding. With that one can assemble very complex bindings from simple bindings.

All bindings in the JavaFX runtime are calculated lazily. That means, if a dependency changes, the result of a binding is not immediately recalculated, but it is marked as invalid. Next time the value of an invalid binding is requested, it is recalculated.

It is recommended to use one of the base classes defined in this package (e.g. `DoubleBinding`) to define a custom binding, because these classes already provide most of the needed functionality. See `DoubleBinding` for an example.

Since:

JavaFX 2.0

See Also:

`DoubleBinding`

MainWindowController::initialise()

```
messagesTextField.textProperty()
        .bind(Bindings.when(Bindings.equal(Bindings.size(historyTable.getItems()), i: 1)) When
                .then( s: "Sem histórico de registos!") StringConditionBuilder
                .otherwise(Bindings.size(historyTable.getItems()).asString( s: "Total de registos: %d")));
```


A final(?!) problem… save on exit?!

- Another example with event handlers…

Event handler for menu button

```
/**
 * Acção do botão "Close" do menu "File"
 * @param actionEvent the event
 */
@FXML
protected void onCloseAction(ActionEvent actionEvent) {
    saveHistory();
    Platform.exit();
}

/** Guarda o histórico. ...*/
private void saveHistory() {
    try {
        model.saveHistory();
        System.err.println("Histórico gravado!");
    } catch (Exception e) {
        System.err.println("Erro ao gravar o histórico: " + e.getMessage());
    }
}
```

MainWindowController::initialise()

```
// event handler para o fecho da janela - usa runLater para garantir que o controlador
// está inicializado quando o código for executado
Platform.runLater(() -> {
    welcomeText.getScene().getWindow().setOnCloseRequest(event -> {
        this.saveHistory();
    });
});
```

Event handler for window close request

In summary

- Properties
- Observables
- Bindings
- TextFormatters
- Converters
- Filters
- Event handling