

## PASSO 2 – properties



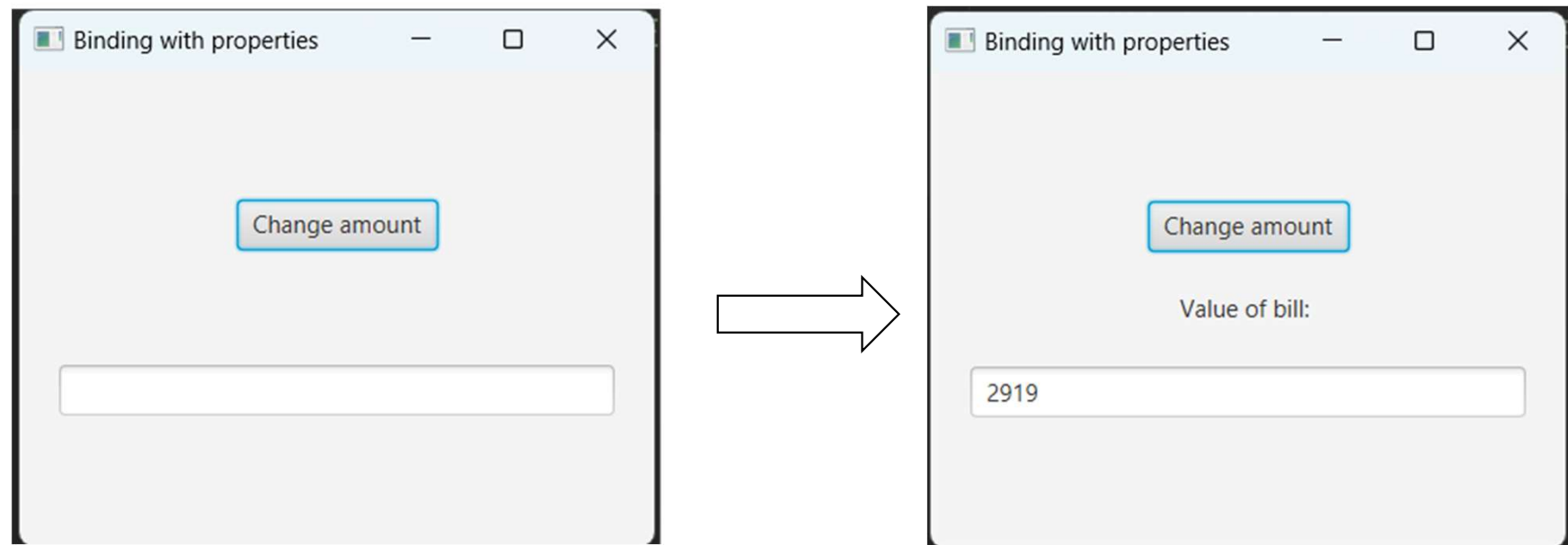
**Uso di java beans e properties per:**

- Rappresentare i dati del model in modo standard, come java beans.
- Legare direttamente i dati del model alla view in modo che la view conosca i valori dei dati e li visualizzi in modo responsive.
- Associare listeners ai dati (properties dei java beans) per effettuare operazioni quando cambiano valore.

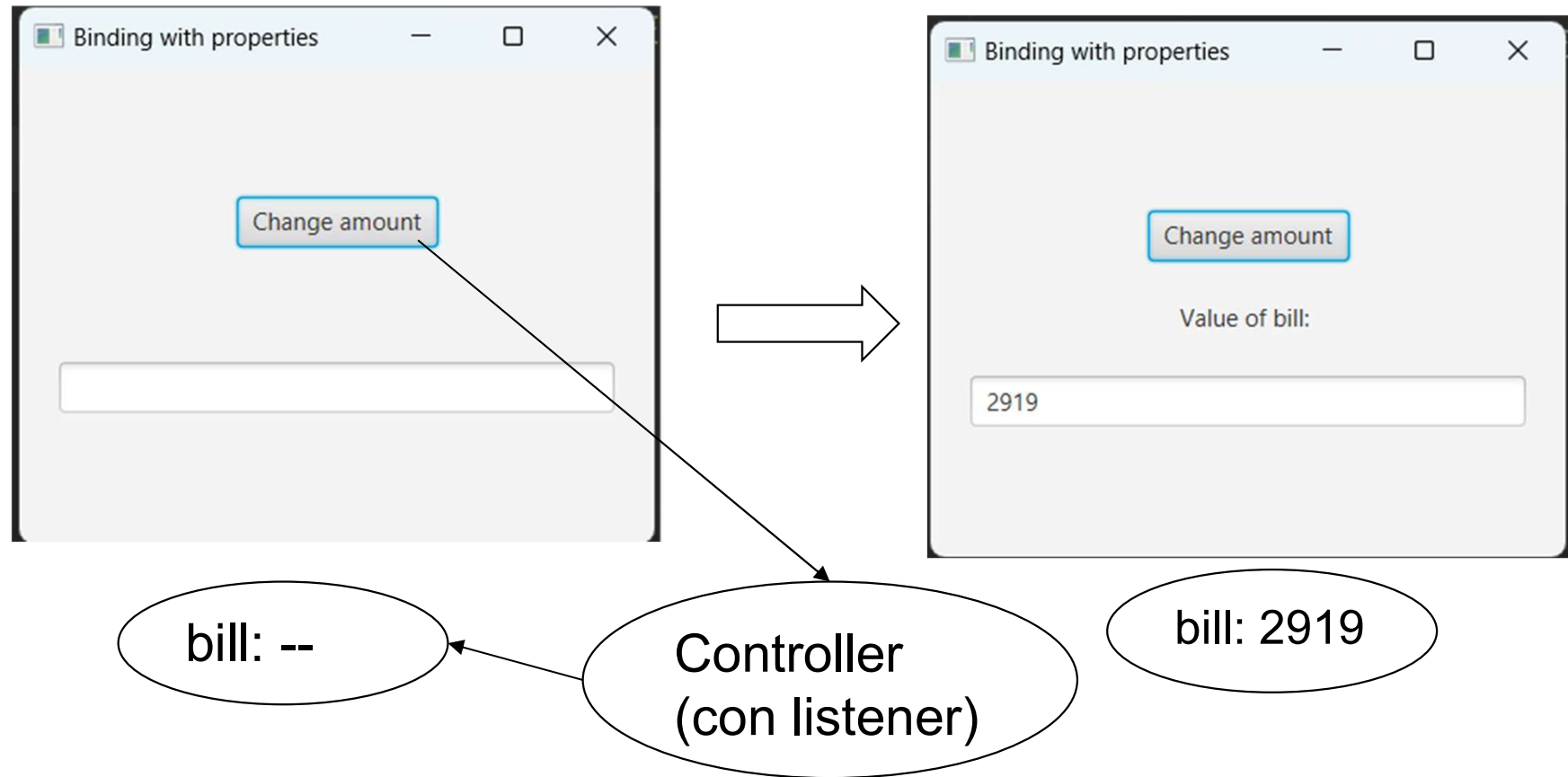
## Esempio – properties - I

In un'app MVC, noi vogliamo che la view sia responsive rispetto ai cambiamenti di stato del model.

Es: cliccando Change amount, noi generiamo delle fatture (bill) e visualizziamo il nuovo valore della spesa.



## Esempio – properties - II

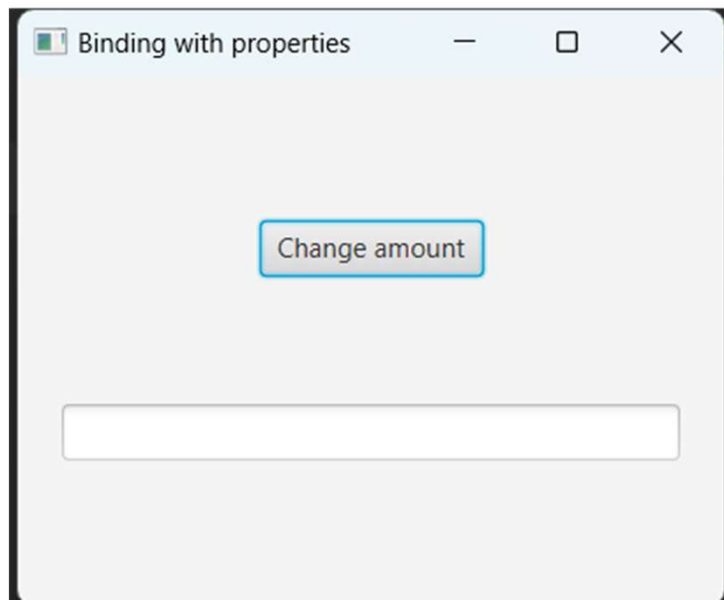


**Il bill è il model dell'applicazione.**

Quando il valore di bill cambia, noi vogliamo che la GUI mostri subito il nuovo valore → **facciamo in modo che l'area di testo della GUI osservi bill!**

## Esempio – properties - III

Obiettivo: evitare di scrivere codice dettagliato per visualizzare nella GUI i valori del model mentre cambiano, e/o per inizializzare i valori del model a partire dai dati acquisiti in input (form) nella GUI.



Le properties dei java beans permettono di fare questo.

# JavaBeans - I



I Java Beans sono classi java che rispettano uno standard di definizione dei metodi e delle loro variabili di istanza (o di stato)

- Per ogni variabile `xxx` di istanza che vogliamo esporre come **property** del javaBean, noi dobbiamo definire il metodo **public Type getXxx()**, per leggere valore della property.
- Se vogliamo permettere di modificare il valore della property, dobbiamo definire anche il metodo **public void setXxx()**, per assegnare un valore alla property.



**Il concetto di property è slegato dall'esistenza di variabili di istanza nel java bean:**

- Il java bean può avere **variabili di istanza private che non sono property** (in quanto prive di metodo getXxx()) e servono internamente per le computazioni.
- **Il nome di una property è determinato dai metodi getXxx() e setXxx(), NON dal nome della variabile di istanza ad essa associata.**

## JavaBeans - III



- **Una property potrebbe derivare dall'esecuzione di un metodo getXxx() del java bean**
  - Noi possiamo definire metodi getYyy() (e quindi properties) utili per eseguire codice applicativo che utilizza lo stato del bean e fa riferimento a più variabili del bean per restituire risultati.
  - Es: se un java bean memorizza due valori numerici in variabili di istanza private, noi possiamo definire una property "prodotto", basata sulla definizione del metodo getProdotto(), che restituisce il prodotto dei due valori.



# JavaBeans – esempio

**NB:** **prop1**, **prop2** e **prodotto** sono **properties**; **var1** e **var2** **NON** sono **property**

```
public class ExampleBean {  
    private int var1; private int var2;  
    public ExampleBean() { }  
    public void setProp1(int n) {  
        this.var1= n; }  
    public int getProp1() {  
        return var1; }  
    public void setProp2(int n) {  
        this.var2 = n; }  
    public int getProp2() {  
        return var2; }  
    public int getProdotto() {  
        return var1*var2;}  
}
```



# Java Beans e Properties



Le properties dei java beans servono per facilitare la definizione di **bindings** tra variabili. I bindings permettono di **cambiare il valore di variabili in dipendenza da altre**: quando si definisce un binding (dipendenza) tra due variabili, se una cambia valore, l'altra viene modificata di conseguenza. I bindings sono utili in varie situazioni. Es:

- Definire una variabile che ha come valore la somma dei valori di altre *n* variabili, con la somma che si aggiorna automaticamente quando una di queste cambia valore
- Nelle interfacce grafiche, legare la componente grafica a un modello, in modo che la componente grafica si aggiorni automaticamente ogni volta che il modello cambia valore

**Strettamente legato a Observer Observable e a MVC**

# public interface ObservableValue<T>



**public interface ObservableValue<T> extends**

**Observable** definisce entità che racchiudono un valore e permettono di osservare i cambiamenti di tale valore. Metodi:

- **T getValue():** restituisce il valore dell'oggetto ObservableValue.
- **void addListener(ChangeListener<? super T> listener):** aggiunge all'ObservableValue un listener di cambiamento.
- **void removeListener(ChangeListener<? super T> listener):** toglie il listener dall'ObservableValue.

Dove **<? super T>** indica che il tipo del ChangeListener deve essere una **superclasse** di T (il contrario di **<? extends T>**)

(qui Observable è javafx.beans.Observable, non l'Observable deprecata)

# public interface ChangeListener<T>



**public interface ChangeListener<T>** definisce i listener di cambiamento di valore delle properties di tipo T

Un ChangeListener viene notificato ogni volta che l'oggetto l'ObservableValue a cui è associato il listener cambia valore.

Le classi che implementano ChangeListener devono implementare il metodo

**void changed(ObservableValue<? extends T> obs,  
T oldValue, T newValue)**

per specificare cosa deve fare il listener quando rileva un evento di cambiamento dell'ObservableValue obs.

# Properties con JavaFX



JavaFX offre classi di libreria che rappresentano e gestiscono javabeans e properties. Es.: la **abstract class DoubleProperty** rappresenta **una property di tipo Double** di un bean. La property **implementa ObservableValue<Number>**.

```
javafx.beans.property
```

## **Class DoubleProperty**

```
java.lang.Object
```

```
    javafx.beans.binding.NumberExpressionBase
```

```
        javafx.beans.binding.DoubleExpression
```

```
            javafx.beans.property.ReadOnlyDoubleProperty
```

```
                javafx.beans.property.DoubleProperty
```

### **All Implemented Interfaces:**

```
NumberExpression, Observable, Property<Number>, ReadOnlyProperty<Number>,
ObservableDoubleValue, ObservableNumberValue, ObservableValue<Number>,
WritableDoubleValue, WritableNumberValue, WritableValue<Number>
```

# SimpleDoubleProperty



Una implementazione di DoubleProperty è **SimpleDoubleProperty**, che racchiude un valore Double.

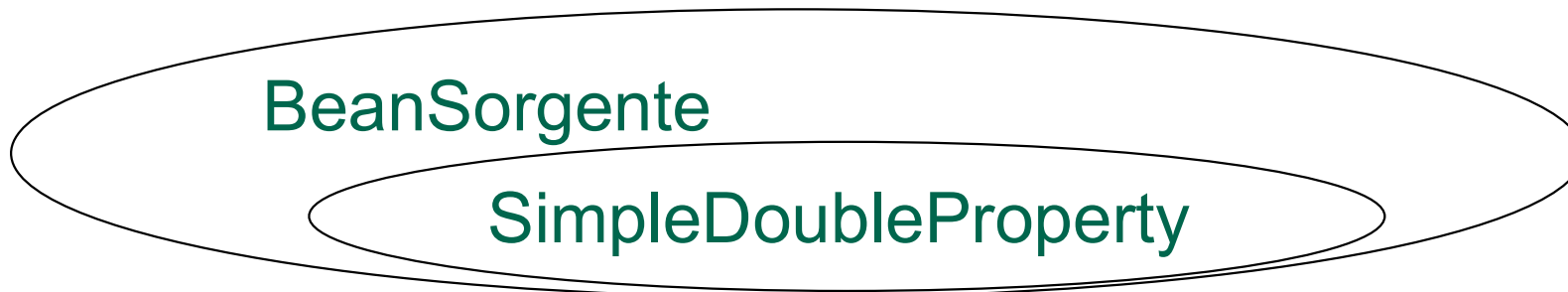
Consideriamo una applicazione JavaFX che gestisce un oggetto con tale property. Vediamo come fare a rilevare automaticamente il cambiamento di valore della property, e a gestire la property facendo delle operazioni.

Per rilevare i cambiamenti di valore della property dobbiamo associare un **listener** (ChangeListener) alla property stessa.

**NB: per ora noi non usiamo JavaFXML e trascuriamo MVC.**

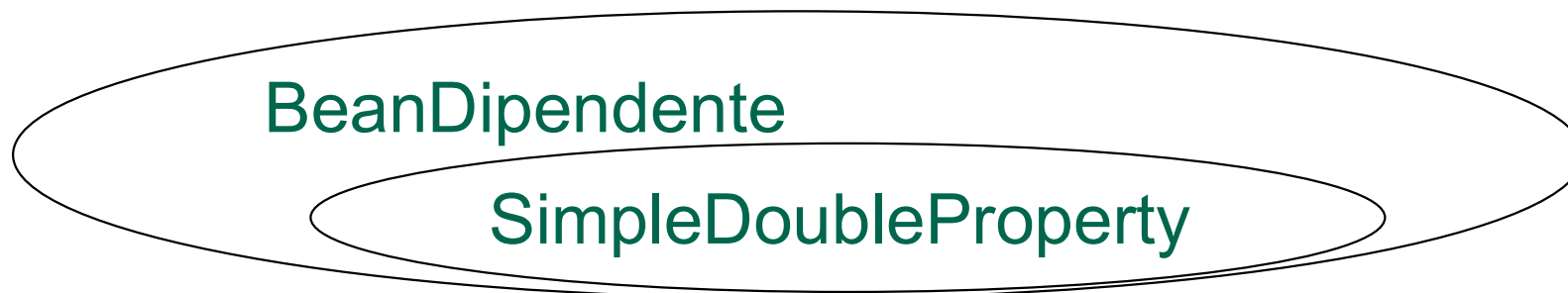
# Listeners di properties – bindingsNoGUI2 - I

```
public class BeanSorgente {  
    // Define a variable to store the property  
    private SimpleDoubleProperty amount = new SimpleDoubleProperty();  
  
    public final double getAmount() { // getter for the property's value  
        return amount.get();  
    }  
    public final void setAmount(double value) { // setter for property's value  
        amount.set(value);  
    }  
    // getter for the property itself  
    public SimpleDoubleProperty amountProperty() {  
        return amount;  
    }  
}
```



# Listeners di properties – bindingsNoGUI2 - II

```
public class BeanDipendente {  
    // Define a variable to store the property  
    private SimpleDoubleProperty amount = new SimpleDoubleProperty();  
  
    public final double getAmount() { // getter for the property's value  
        return amount.get();  
    }  
    public final void setAmount(double value) { // setter for property's value  
        amount.set(value);  
    }  
    // getter for the property itself  
    public SimpleDoubleProperty amountProperty() {  
        return amount;  
    }  
}
```



# Listeners di properties – bindingsNoGUI2 - III

```
public static void main(String[] args) {  
    BeanSorgente b1 = new BeanSorgente();  
    BeanDipendente b2 = new BeanDipendente();  
    SimpleDoubleProperty property = b1.amountProperty();  
    property.addListener(new ChangeListener<Number>() {  
        @Override  
        public void changed(ObservableValue<? extends Number> obs,  
                             Number oldValue, Number newVal) {  
            //b2.setAmount((Double)newVal);  
            System.out.println("b1: " + newVal);  
        }  
    });  
    for (int i=0; i<5; i++) {  
        b1.setAmount(i);  
        System.out.println("Bean sorgente: " + b1.getAmount() +  
            "; Bean dipendente: "+b2.getAmount());  
    }  
}
```



# Listeners di properties – bindingsNoGUI2 - IV

```
Bean sorgente: 0.0; Bean dipendente: 0.0
```

```
b1: 1.0
```

```
Bean sorgente: 1.0; Bean dipendente: 0.0
```

```
b1: 2.0
```

```
Bean sorgente: 2.0; Bean dipendente: 0.0
```

```
b1: 3.0
```

```
Bean sorgente: 3.0; Bean dipendente: 0.0
```

```
b1: 4.0
```

```
Bean sorgente: 4.0; Bean dipendente: 0.0
```

```
Bean sorgente: 0.0; Bean dipendente: 0.0
```

```
b1: 1.0
```

```
Bean sorgente: 1.0; Bean dipendente: 1.0
```

```
b1: 2.0
```

```
Bean sorgente: 2.0; Bean dipendente: 2.0
```

```
b1: 3.0
```

```
Bean sorgente: 3.0; Bean dipendente: 3.0
```

```
b1: 4.0
```

```
Bean sorgente: 4.0; Bean dipendente: 4.0
```

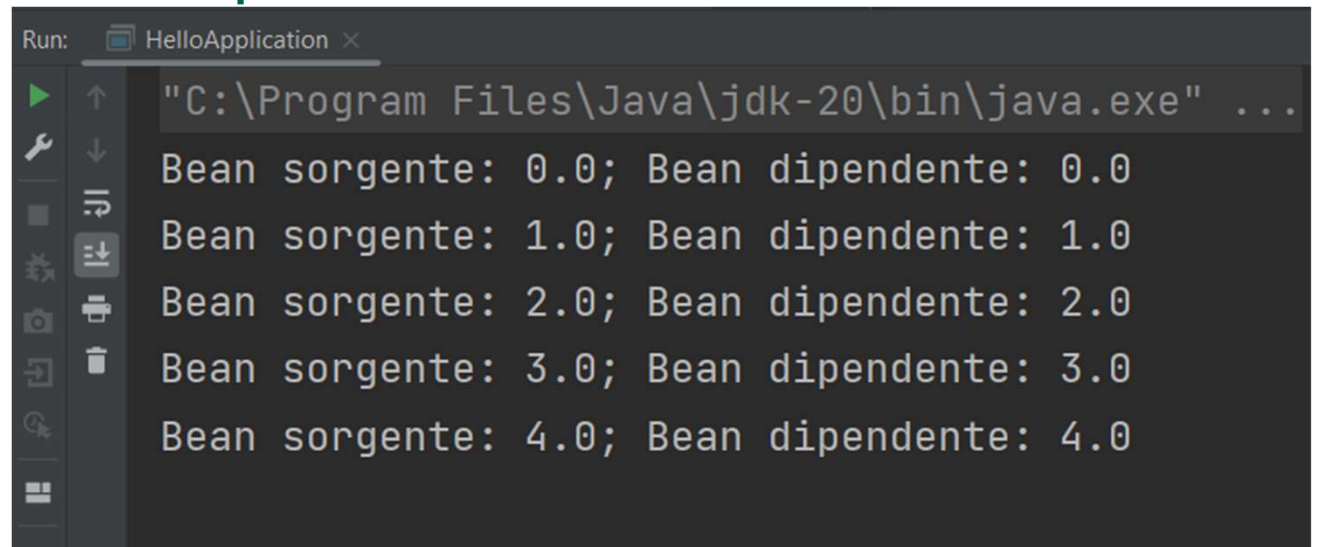
```
Process finished with exit code 0
```

# Esempio – binding di properties - I

Non sempre serve definire un listener perché non ci sono operazioni complesse da fare. Talvolta basta un **binding**!

Sperimentiamo un legame tra properties nell'applicazione **bindingsNoGUI**. Vogliamo che una property rifletta sempre il valore di un'altra property. Lo possiamo ottenere tramite un **binding unidirezionale**:

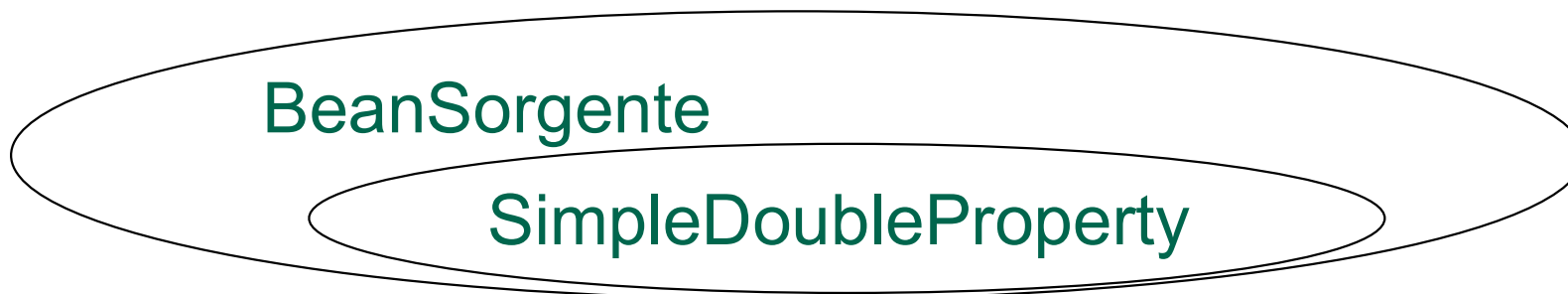
Bean sorgente → Bean dipendente



```
Run: HelloApplication x
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Bean sorgente: 0.0; Bean dipendente: 0.0
Bean sorgente: 1.0; Bean dipendente: 1.0
Bean sorgente: 2.0; Bean dipendente: 2.0
Bean sorgente: 3.0; Bean dipendente: 3.0
Bean sorgente: 4.0; Bean dipendente: 4.0
```

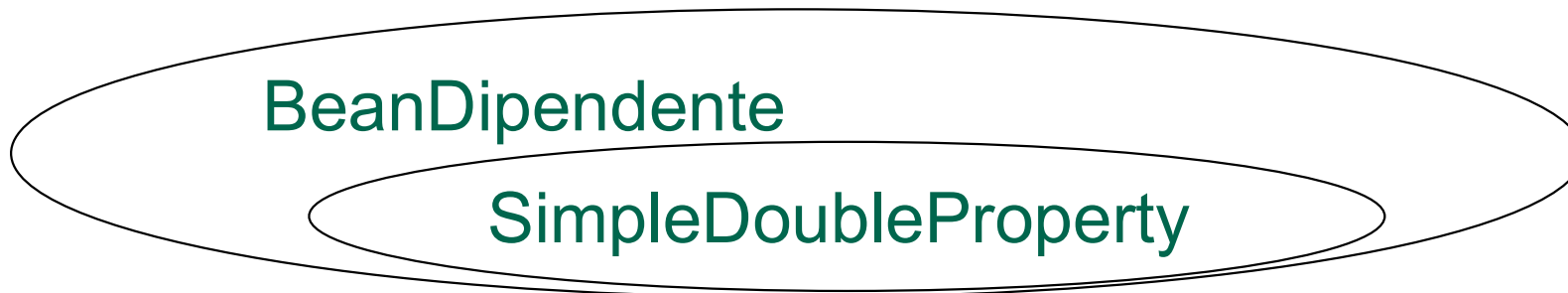
## Esempio – binding di properties - II

```
public class BeanSorgente {  
    // Define a variable to store the property  
    private SimpleDoubleProperty amount = new SimpleDoubleProperty();  
  
    public final double getAmount() { // getter for the property's value  
        return amount.get();  
    }  
    public final void setAmount(double value) { // setter for property's value  
        amount.set(value);  
    }  
    // getter for the property itself  
    public SimpleDoubleProperty amountProperty() {  
        return amount;  
    }  
}
```



## Esempio – binding di properties - III

```
public class BeanDipendente {  
    // Define a variable to store the property  
    private SimpleDoubleProperty amount = new SimpleDoubleProperty();  
  
    public final double getAmount() { // getter for the property's value  
        return amount.get();  
    }  
    public final void setAmount(double value) { // setter for property's value  
        amount.set(value);  
    }  
    // getter for the property itself  
    public SimpleDoubleProperty amountProperty() {  
        return amount;  
    }  
}
```



## Esempio – binding di properties - IV

```
public class HelloApplication {  
    public static void main(String[] args) {  
        BeanSorgente b1 = new BeanSorgente();  
        BeanDipendente b2 = new BeanDipendente();  
        bindProperties(b1, b2);  
        for (int i=0; i<5; i++) {  
            b1.setAmount(i);  
            System.out.println("Bean sorgente: " + b1.getAmount() +  
                               "; Bean dipendente: "+b2.getAmount());  
        }  
    }  
  
    public static void bindProperties(BeanSorgente b1, BeanDipendente b2){  
        b2.amountProperty().bind(b1.amountProperty());  
    }  
}
```

In seguito al binding il valore di b2 dipende dal valore di b1.

# Binding di properties con GUI - I



**abstract class StringProperty**

**class SimpleStringProperty** racchiude un valore String

Sviluppiamo una semplice applicazione JavaFXML in cui leghiamo due campi di testo della GUI in modo che il secondo prenda sempre il valore che viene inserito nel primo:



← inserimento  
← visualizza

# Come legare i due campi di testo della GUI - I

Sia il textfield «inserimento» che il textfield «visualizza» hanno una SimpleString property predefinita che mantiene il valore che viene visualizzato a video → basta legare le due properties in un metodo del controller della vista (per ora, il binding è attivato cliccando il bottone):

## Applicazione JavaFXML2

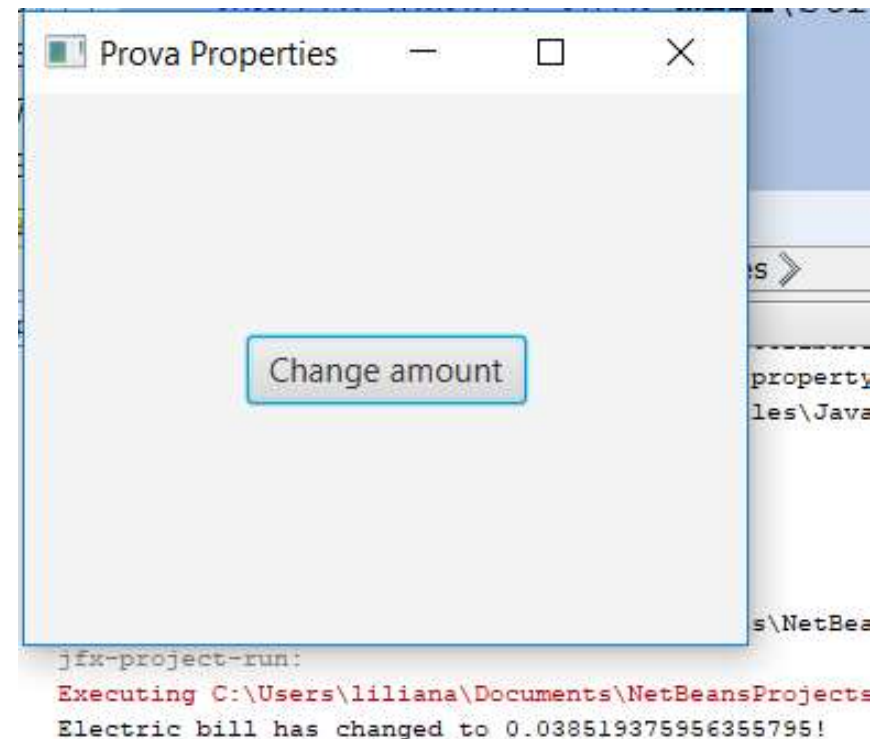
```
public class HelloController {  
    @FXML  
    private TextField inserimento;  
    @FXML  
    private TextField visualizza;  
    @FXML  
    private void onHelloButtonClick() { bindProperties(); }  
    @Override  
    public void bindProperties() {  
        visualizza.textProperty().bind(inserimento.textProperty());  
    }  
}
```

# Properties dei Java Beans



Da: <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

Definiamo una applicazione JavaFX che crea un oggetto Bill e offre un pulsante per assegnare un quantitativo di soldi random alla property dell'oggetto.



Cliccando sul pulsante «Change amount» (btn nel codice) cambia il valore della property di Bill: vd. la stampa su output standard: «Electric bill has changed to ....»



# Properties dei Java Beans – Esempio



Da: <https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm>

Definiamo un JavaBean Bill che rappresenta una fattura.

Bill ha la DoubleProperty amountDue che rappresenta il quantitativo di soldi da pagare:

```
class Bill {  
    private DoubleProperty amountDue = new SimpleDoubleProperty();  
    public final double getAmountDue(){return amountDue.get();}  
    public final void setAmountDue(double value){amountDue.set(value);}  
    // Getter della property come oggetto  
    public DoubleProperty amountDueProperty() {return amountDue;}  
}
```

# Properties – esempio – I



```
public class Main extends Application {
```

```
    @Override
```

```
    public void start(Stage primaryStage) {
```

```
        Bill electricBill = new Bill();
```

```
        final Random r = new Random();
```

```
// assegno il ChangeListener alla property amountDue per rilevare i suoi cambiamenti
```

```
        electricBill.amountDueProperty().addListener(new ChangeListener() {
```

```
            @Override
```

```
            public void changed(ObservableValue o, Object oldV, Object newV) {
```

```
                double n = ((Double) newV).doubleValue();
```

```
                System.out.println("Electric bill has changed to " + n + "!");
```

```
            }
```

```
        });
```

```
    ... Continua ...
```

# Properties – esempio – II



```
Button btn = new Button();  
btn.setText("Change amount");  
// definisco il bottone e l'EventListener del bottone, per modificare il  
// valore della property «amountDueProperty»  
btn.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        double d = r.nextDouble();  
        electricBill.setAmountDue(d);  
    }  
});  
// Continua ...
```

# Properties – esempio – III



```
StackPane root = new StackPane();
root.getChildren().add(btn);
Scene scene = new Scene(root, 300, 250);
primaryStage.setTitle("Prova Properties");
primaryStage.setScene(scene);
primaryStage.show();
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    launch(args);
}
}
```

## PASSO 3



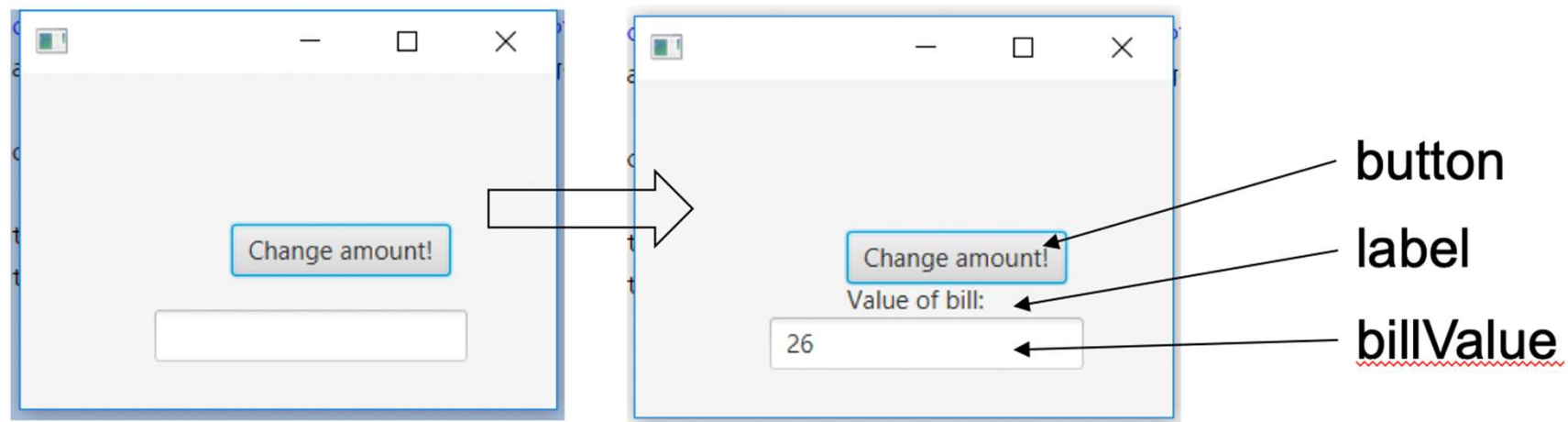
- **Uso di java beans e BINDING delle properties ai componenti della GUI** per semplificare il codice della GUI – in combinazione con la specifica XML della GUI

Mostriamo l'uso dei binding con l'applicazione  
JavaFXML4-properties

# JavaFXML4-properties

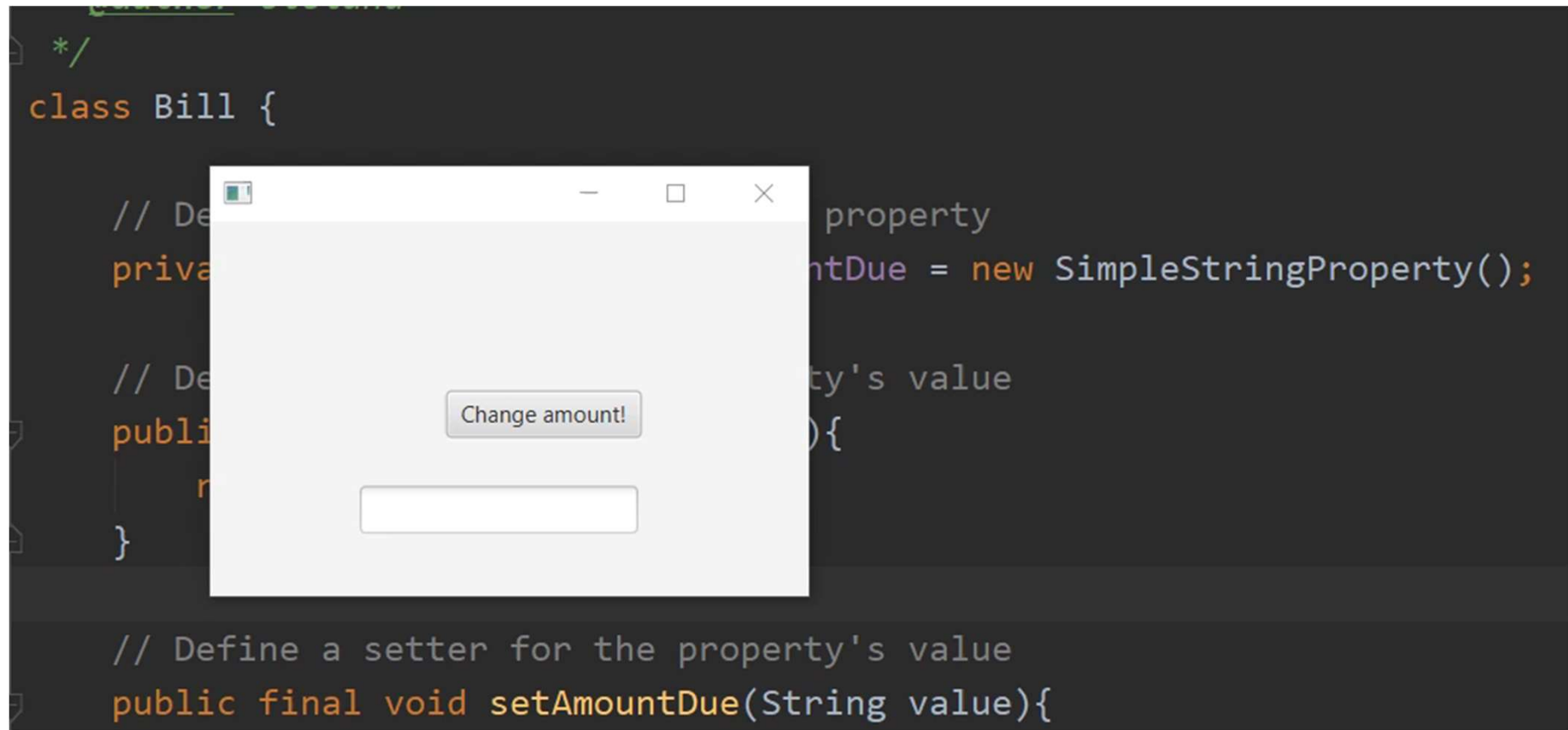


La seguente applicazione visualizza il valore del "bill" ad ogni suo cambiamento. Il bill cambia valore quando l'utente clicca sul bottone. Come prima, il click assegna al bill un numero intero casuale. Bill rappresenta una fattura.



NB: i TextField hanno una **StringProperty** che mantiene il loro valore → **billValue.textProperty()** restituisce tale property.

# JavaFXML4-properties - esecuzione





```
class Bill {
```

```
    // Definiamo la property «amountDue» come StringProperty affinché sia  
    compatibile con la text property del campo della GUI
```

```
    private SimpleStringProperty amountDue = new SimpleStringProperty();
```

```
    // Define a getter for the property's value
```

```
    public final String getAmountDue(){return amountDue.get();}
```

```
    // Define a setter for the property's value
```

```
    public final void setAmountDue(String value){amountDue.set(value);}
```

```
    // Define a getter for the property itself
```

```
    public SimpleStringProperty amountDueProperty() {return amountDue;}
```

```
}
```



# JavaFX4-properties – View (hello-view.fxml)

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

```
<?import java.lang.*?>...<?import javafx.scene.layout.*?>
```

```
<\ VBox ....
```

```
  <Button text="Change amount" onAction="#onHelloButtonClick"/>
```

```
  <Label fx:id="ris" layoutX="126" layoutY="120"
```

```
                                minHeight="16" minWidth="69" />
```

```
  <TextField fx:id="billValue" layoutX="80.0" layoutY="141.0" />
```

```
</VBox>
```

# JavaFXML4-properties – Controller – I



```
public class Controller implements Initializable {
```

```
    private Bill electricBill = new Bill();           //variabile del controller
```

```
    private final Random r = new Random();           //costante del controller
```

```
    @FXML
```

```
    private Label ris;
```

```
    @FXML
```

```
    private TextField billValue;
```

```
    @FXML
```

```
protected void onHelloButtonClick() {
```

```
    ris.setText("Value of bill:"); // brutto...
```

```
    int i = r.nextInt(10000);
```

```
    electricBill.setAmountDue(new StringBuilder().append(i).toString());
```

```
}
```

```
// continua...
```

# JavaFXML4-properties – Controller - II



```
public void bindProperties() {
```

```
    // Questa istruzione associa la SimpleStringProperty
```

```
    // del textField billValue (billValue.textProperty())
```

```
    // alla SimpleStringProperty amountDueProperty di electricBill
```

```
    // → ogni volta che amountDueProperty viene modificata,
```

```
    // si aggiorna la visualizzazione di billValue nella GUI
```

```
    billValue.textProperty().bind(electricBill.amountDueProperty());
```

```
    // continua ...
```

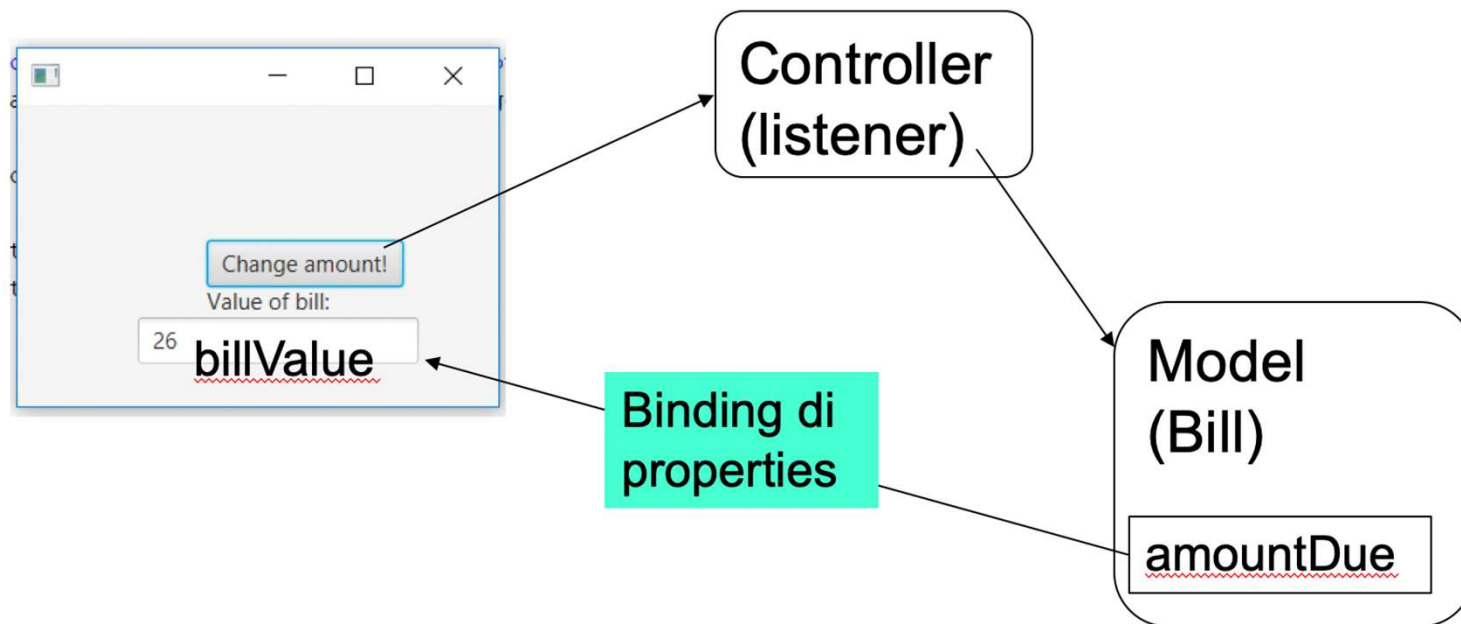
```
    // il binding tra queste due property evita di definire un listener di  
    // cambiamento esplicito (ChangeListener) → semplifica il codice del controller
```

# JavaFX- binding



```
billValue.textProperty().bind(electricBill.amountDueProperty());
```

lega la textProperty della GUI alla SimpleStringProperty del model



# JavaFXML-properties – Controller - III

/\* continua:



se vogliamo visualizzare il valore della property su standard output dobbiamo definire anche il listener di cambiamento della property (vedi sotto). Se non ci serve questa visualizzazione, il codice sotto può essere rimosso. In ogni caso, non bisogna assegnare esplicitamente il nuovo valore a billValue perché l'assegnamento viene fatto in automatico grazie al binding delle due property.

\*/

```
electricBill.amountDueProperty().addListener(new  
ChangeListener<String>(){  
    public void changed(ObservableValue<? extends String> o,  
        String oldVal, String newVal) {  
        System.out.println("Electric bill has changed to " + newVal + "!");  
        //billValue.setText((new Double(newVal)).toString()); // no – binding!  
    }  
    });  
}}
```

# JavaFXML-properties – Main



```
public class Main extends Application {  
    @Override  
    public void start(Stage stage) throws IOException {  
        FXMLLoader fxmlLoader = new  
            FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));  
        Scene scene = new Scene(fxmlLoader.load(), 320, 240);  
        stage.setTitle("Binding with properties");  
        stage.setScene(scene); // invoco il metodo bindProperties nel main  
        HelloController contr = fxmlLoader.getController();  
        contr.bindProperties();  
        stage.show();  
    }  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

# JavaFX – note tecniche



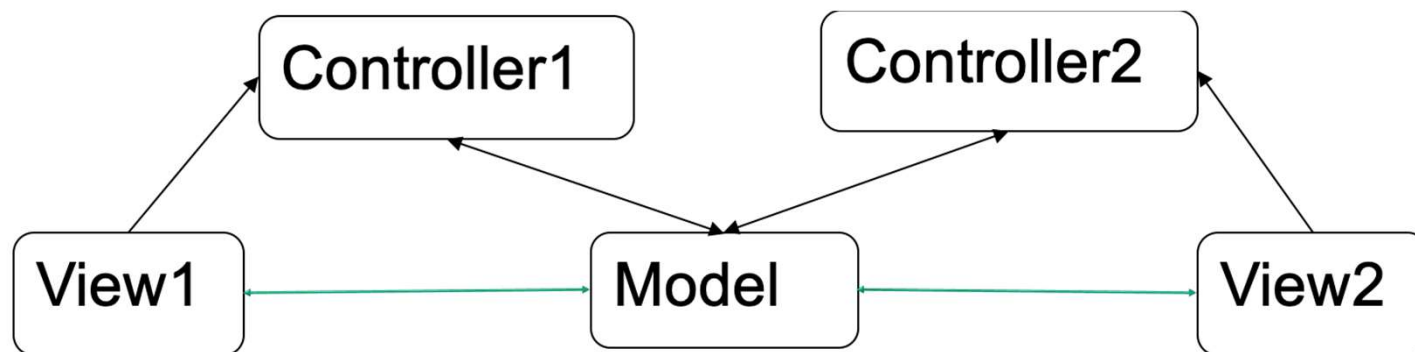
- Quando dovete risolvere i nomi di classi che non appartengono al core Java (es. GridPane, Button, etc.), risolvetele sempre con la versione del pacchetto javafx. Non scegliete le altre versioni delle classi, se presenti, altrimenti non funzioneranno le applicazioni JavaFX.
- Ricordate che se l'applicazione utilizza file (es., immagini, o fogli stile) e a runtime si blocca per via di una NullPointerException, il tutto potrebbe essere dovuto alla mancanza dei file nella cartella del codice eseguibile (sotto build/classes). Controllate, e se necessario caricate i file manualmente (può accadere, anche se raramente).

# JavaFX – applicazioni con view e controller multipli



Se un'applicazione JavaFXML ha viste multiple necessita di più di un controller (uno per ogni view). In tal caso normalmente i controller devono condividere il model dell'applicazione per sincronizzarsi attraverso il model. Quindi, voi dovete passare il riferimento delle variabili a tutti i controller. Per fare questo, nel metodo `start(Stage)` dell'applicazione JavaFXML eseguite le seguenti operazioni:

- Create il Model *m* (e inizializzatelo).
- Create i controller con `FXMLLoader` e prendete i loro riferimenti.
- ...





## File

Jacob Smith

Isabella Johnson

Ethan Williams

Emma Jones

Michael Brown

First Name:

Last Name:

Email:

# JavaFXML-TUTORIAL-MVC – I



```
public class Main extends Application { // codice non aggiornato a ultima versione di JavaFXML
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        DataModel model = new DataModel();      // creo e inizializzo il model
        // poi creo i controller e prendo i riferimenti a ciascuno di essi
        FXMLLoader listLoader = new FXMLLoader(getClass().getResource("list.fxml"));
        root.setCenter(listLoader.load());
        ListController listController = listLoader.getController();
        FXMLLoader editorLoader = new FXMLLoader(getClass().getResource("editor.fxml"));
        root.setRight(editorLoader.load());
        EditorController editorController = editorLoader.getController();
        listController.initModel(model); editorController.initModel(model); // passo il model ai
controller
        Scene scene = new Scene(root, 800, 600); primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

# JavaFXML-TUTORIAL-MVC – II



```
FXMLLoader listLoader =  
    new FXMLLoader(getClass().getResource("list.fxml"));  
root.setCenter(listLoader.load());  
ListController listController = listLoader.getController();
```

Per ciascun controller, prendete il riferimento al controller  $c_i$  interrogando il loader (metodo `getController()` del loader) e invocate `cj.initModel(m)` per dargli il model. A quel punto tutti i controller hanno il riferimento allo stesso model.



```
public class ListController {  
  
    @FXML  
    private ListView<Person> listView;  
  
    private DataModel model; // variabile globale del ListController, definita nella classe  
  
    public void initModel(DataModel model) {  
        if (this.model != null) { // ensure model is only set once:  
            throw new IllegalStateException("Model can only be initialized once");  
        }  
        this.model = model ;  
        ... continua il metodo di inizializzazione con le altre operazioni da fare  
    } ...
```

# JavaFX – collezioni osservabili



Tutorial per usare **ObservableList** etc.:

<https://docs.oracle.com/javafx/2/collections/jfxpub-collections.htm>

Per esempio:

API delle ObservableList (gli osservabili):

<https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html>

API delle ListView (gli osservatori):

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ListView.html>