Week 8

Model View Controller
(MVC)

This week

- New technologies:
  - FXML
  - Properties
- Model View Controller (MVC) pattern

FXML

FXML

- Consensus: Programming languages are not good for laying out GUIs.

- Current trend: use a markup language.

- **FXML** is the JavaFX Markup language based on XML.

- Replace this Java code:
  ```
  Label usernameLbl = new Label("Username:");
  ```

  with this FXML code:
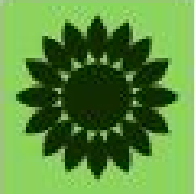 ```
  <Label text="Username:"/>
  ```

# Leaf nodes

Username:

jack

••••••••

Login

`<Label text="Username:"/>`

`<TextField/>`

`<PasswordField/>`

`<Button text="Login"/>`

`<ImageView>`

    `<image><Image url="@flower.png"/></image>`

`</ImageView>`

# VBox



```
<VBox spacing="10" alignment="center">
    <Label text="Username:"/>
    <TextField/>
    <PasswordField/>
    <Button text="Login"/>
    <ImageView>
        <image>
            <Image url="@flower.png"/>
        </image>
    </ImageView>
</VBox>
```

HBox


```
<HBox spacing="10" alignment="center">
    <Label text="Username:"/>
    <TextField/>
    <Button text="Login"/>
    <ImageView>
        <image><Image url="@flower.png"/></image>
    </ImageView>
</HBox>
```
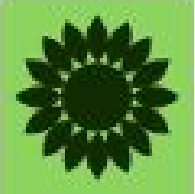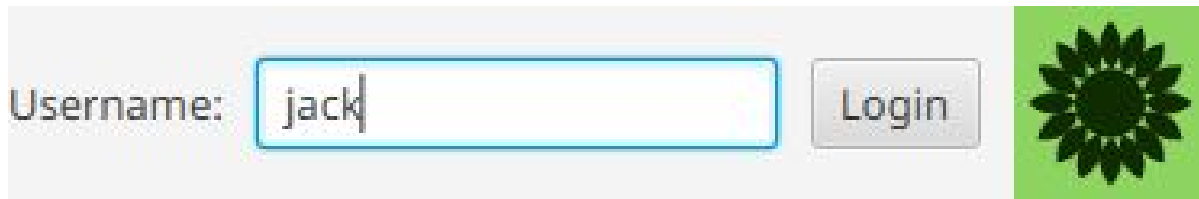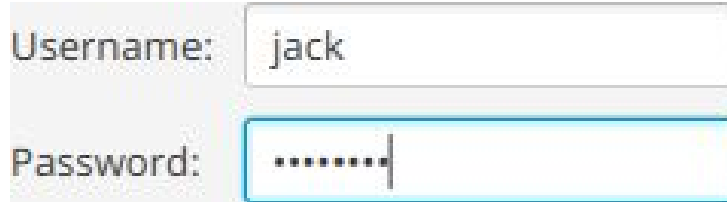
GridPane

Username: jack

Password: ••••••••

```
<GridPane alignment="center" hgap="10" vgap="10">
    <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="0"/>
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
</GridPane>
```

GridPane attributes

Attributes for GridPane:

- hgap sets the horizontal gap between child nodes.
- vgap sets the vertical gap between child nodes.
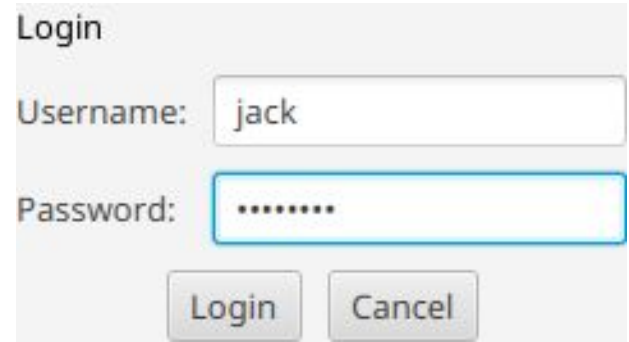
Attributes for children of GridPane:

- GridPane.columnIndex sets the column position of a child.
- GridPane.rowIndex sets the row position of a child.
- GridPane.columnSpan sets how many columns the child occupies.

Nested branches


```
<VBox spacing="10">
  <Text text="Login"/>

  <GridPane alignment="center" hgap="10" vgap="10">
    <Label text="Username:" GridPane.columnIndex="0" GridPane.rowIndex="1"/>
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"/>
    <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="2"/>
    <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="2"/>
  </GridPane>

  <HBox alignment="center" spacing="10">
    <Button text="Login"/>
    <Button text="Cancel"/>
  </HBox>
</GridPane>
```

Label vs Text

```
import javafx.scene.text.*;
```

- A <Label> is used to label a form input (e.g. a TextField, PasswordField, RadioButton, ...)
- A <Text> is to display free-standing text (e.g. a heading, informative text, error messages, ...)

- The text of a <Label> never changes.
- The text of a <Text> can be changed programmatically via its `setText()` method.

<Text>

<Label>

Login

Username: jack

Password: ••••••••

Login   Cancel
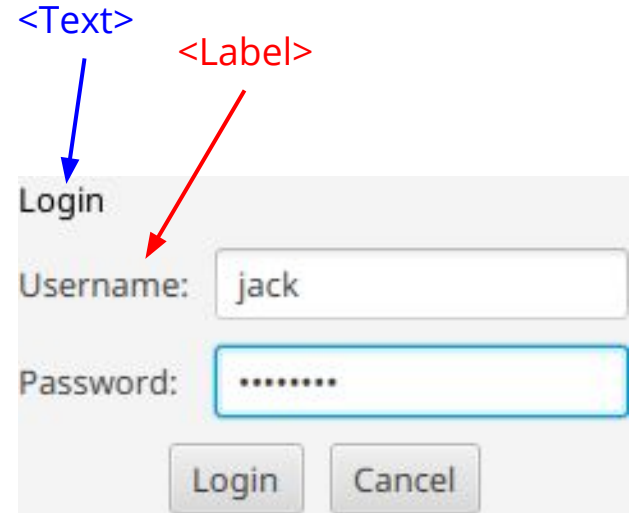
File: login.fxml   - internal structure

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

<?import javafx.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>             ← Imports
<?import javafx.scene.layout.*?>

<VBox spacing="10">          ← Root node
  <Text text="Login" .../>
  <GridPane .../>            ← Child nodes
    ...

</GridPane>
```

CSS - Cascading Style Sheets
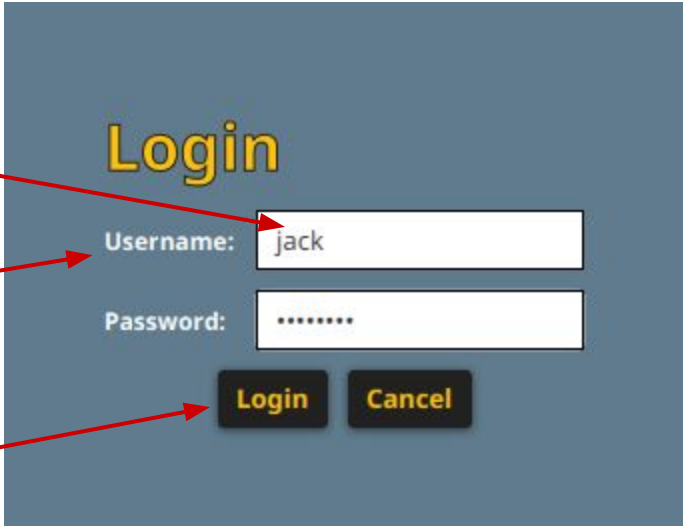
- A CSS file specifies style information. E.g. font sizes, colours, borders.
- The <stylesheets> element is placed within the root node of your scene graph and links to the URL of your CSS file. Prefix the URL with @ if it is in the same directory as the current file.

```
<GridPane alignment="center" hgap="10" vgap="10">
    <Text text="Login" .../>
    ...
    <stylesheets>
        <URL value="@style.css"/>
    </stylesheets>
</GridPane>
```

File: style.css

```css
TextField, PasswordField {
  -fx-background-color: #ffffff;
  -fx-border-color: black;
}
Label {
  -fx-font-size: 12px;
  -fx-font-weight: bold;
  -fx-text-fill: #ffffff;
}
Button {
  -fx-font-family: "Arial";
  -fx-font-weight: bold;
  -fx-background-color: #212121;
  -fx-text-fill: #ffc107;
  -fx-effect: dropshadow( three-pass-box, rgba(0,0,0,0.6), 5, 0.0, 0, 1 );
}
```

For more, see https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Style classes

```
<VBox styleClass=" root">
  <Text text="Login" styleClass=" heading"/>
  ...
</VBox>
```

- Assign a style class to a node with `styleClass="xyz"`
- Select nodes with style class "xyz" using the selector `.xyz { ... }`
- Allows you to invent categories for selecting nodes from CSS.

```
.root {
  -fx-background-color: #607b8d;
  -fx-padding: 50px;
}

.heading {
  -fx-font-size: 32px;
  -fx-font-family: "Arial";
  -fx-font-weight: bold;
  -fx-fill: #ffc107;
  -fx-stroke: #212121;
}
```

Style classes

```
<VBox styleClass=" root">
  <Text text="Login" styleClass=" heading"/>
  ...
</VBox>
```

```
.root {
  -fx-background-color: #607b8d;
  -fx-padding: 50px;
}

.heading {
  -fx-font-size: 32px;
  -fx-font-family: "Arial";
  -fx-font-weight: bold;
  -fx-fill: #ffc107;
  -fx-stroke: #212121;
}
```

Loading a scene graph from an FXML file

- Use class FXMLLoader to load an FXML file.

- `import javafx.fxml.*;`

```
@Override public void start(Stage stage) throws Exception {
    FXMLLoader loader = new FXMLLoader(getClass().getResource("login.
fxml"));
    Parent root = loader.load();
    stage.setTitle(title);
    stage.setScene(new Scene(root));
    stage.sizeToScene();
    stage.show();
}
```

Model View Controller

Model-View-Controller (MVC)

The MVC pattern splits a GUI program into 3 layers

- The models are Java objects that represent the data of your application and the operations on that data.
- The views are the components that represent the graphical user interface of your application. Views "observe" data in the models.
- The controllers are the components that handle user interaction. Controllers "observe" events that occur in the views.

MVC Overview

View

```
bank.fxml

customer.fxml

account.fxml
```
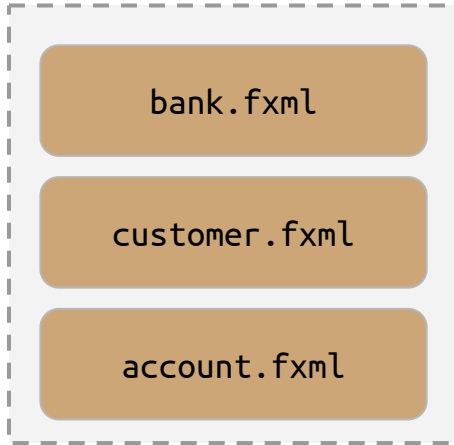
Displays the
Graphical User Interface

Controller

```
BankController

CustomerController

AccountController
```
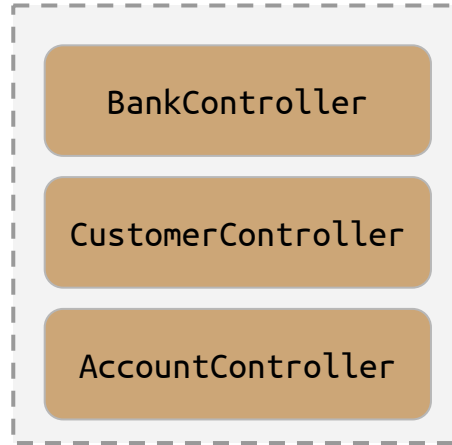
Handles button clicks

Model

```
Bank

Customer

Account
```
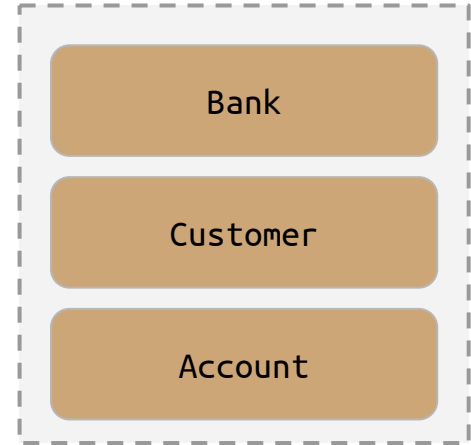
Stores the data

MVC Observers

MVC makes heavy use of the observer pattern:

- **Each view "observes" the model.**
  When model data changes (e.g. bank balance changes), the views are notified so they can update the screen.

- **Each controller "observes" the view.**
  When user interaction (e.g. a button click) happens in the view, the controllers are notified so that they can handle the event.

MVC observers

View

Controller

Model

bank.fxml

customer.fxml

account.fxml

BankController

CustomerController

AccountController

Bank

Customer

Account

1. Notify ***Deposit*** *button clicked*

2. deposit()

3. Notify ***balance*** *changed*
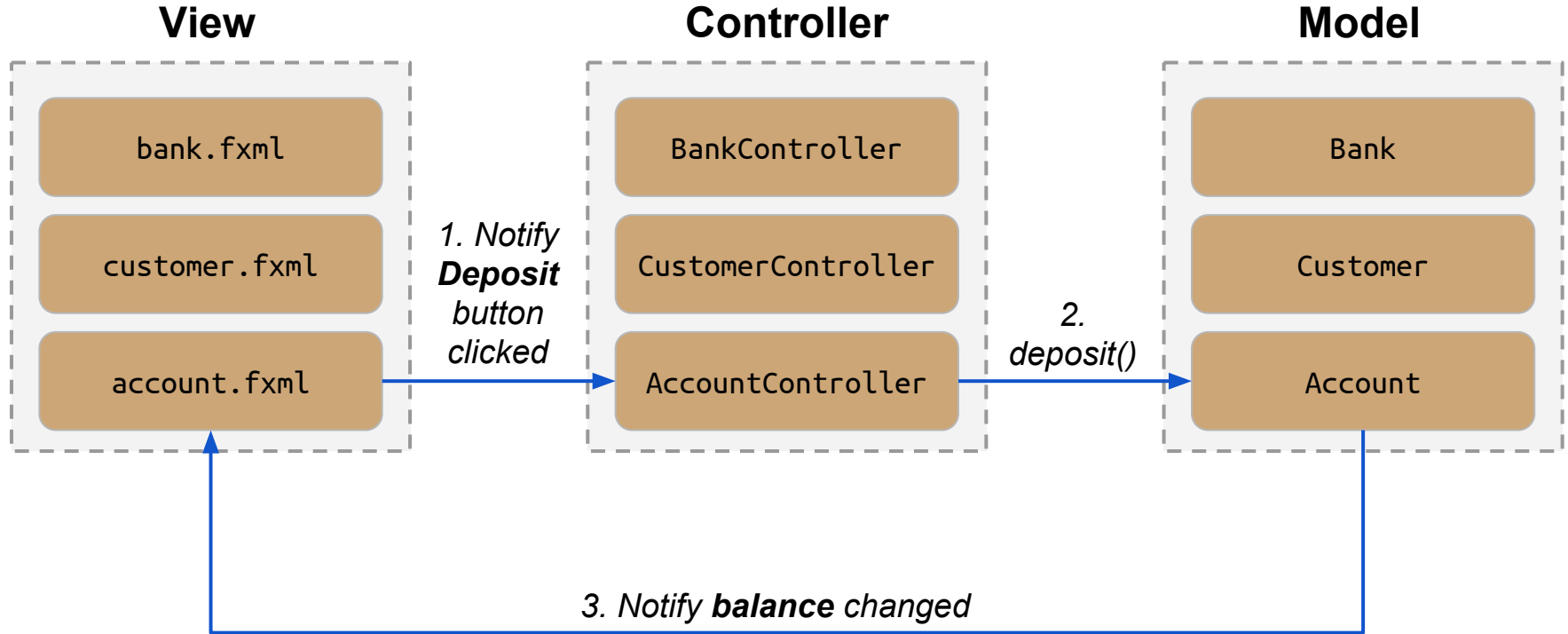
Demo: Bank Account

Model

Model

The model contains:

- Data and related operations.

The model does *not* contain:

- User interface code:
  - No read pattern
  - No menu pattern
  - No output pattern

```
public class Account {
    private String name;
    private double balance;
    public Account(String name, double balance) {
        this.name = name;
        this.balance = balance;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Accessing data from the view

- The GUI needs to display and sometimes update data in the model.
- Therefore, the model should provide getters and setters for data.

```
public class Account {
    ...
    public void getBalance() { return balance; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

- The balance cannot be set directly, so don't provide a setter. Instead, use the deposit and withdraw methods.

JavaBeans properties

- A pair of getter/setter methods is called a JavaBeans property:

```
public String getName() {          public void setName(String name) {
    return name;                        this.name = name;
}                                  }
```

- A read-only JavaBeans property is a getter without a setter. e.g.

```
public double getBalance() {
    return balance;
}
```

- A boolean property getter uses the prefix "is" instead of "get":

```
public boolean isFixedInterestRate() {
    return fixedInterestRate;
}
```

JavaFX properties

- JavaFX introduces "observable" properties following the observer pattern.
- JavaFX properties are backwards compatible with JavaBeans properties.
- `import javafx.beans.property.*;`

Simple properties	Properties	Read-only properties
`SimpleIntegerProperty`	`IntegerProperty`	`ReadOnlyIntegerProperty`
`SimpleDoubleProperty`	`DoubleProperty`	`ReadOnlyDoubleProperty`
`SimpleBooleanProperty`	`BooleanProperty`	`ReadOnlyBooleanProperty`
`SimpleStringProperty`	`StringProperty`	`ReadOnlyStringProperty`
`SimpleObjectProperty`	`ObjectProperty`	`ReadOnlyObjectProperty`

For more, see: https://docs.oracle.com/javafx/2/api/javafx/beans/property/package-summary.html

"is a" relationships

- An Observable can add and remove listeners (observers).
- A ReadOnlyIntegerProperty is an Observable, and also has a get() method.
- An IntegerProperty is everything above, plus also has a set() method.
- A SimpleIntegerProperty is a concrete implementation of the abstract class IntegerProperty.


```
‹‹interface››
Observable

void addListener(…)
void removeListener(…)
```

```
‹‹abstract››
ReadOnlyIntegerProperty

int get()
```

```
‹‹abstract››
IntegerProperty

void set(int value)
```

```
SimpleIntegerProperty


```
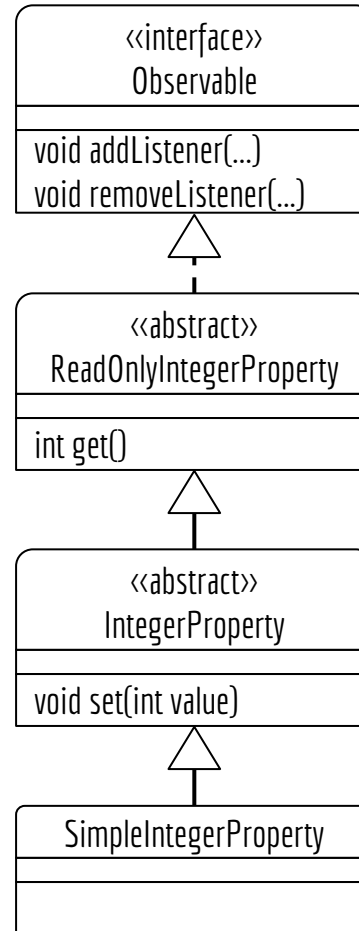
JavaFX property patterns

JavaFX properties fall into the following patterns:

1. Immutable Property
2. Read Write Property
3. Read Only Property
4. Immutable Property with Mutable State

Pattern #1: Immutable Property

- A property that never changes.
- Final getter. No setter.

```
public class SomeClass {
    private final int value;
    public SomeClass(int value) {
        this.value = value;
    }
    public final int getValue() { return value; }
}
```

Pattern #2: Read Write Property

- A property that is readable, writable and observable.
- Encapsulate the value in a property object.
- Final getter and setter.
- Property method called xProperty (where x is the name of the property).

```java
public class SomeClass {
    private IntegerProperty value = new SimpleIntegerProperty();
    public SomeClass(int value) {
        this.value.set(value) ;
    }
    public final int getValue() { return  value.get(); }
    public final void setValue(int value) {  this.value.set(value) ; }
    public IntegerProperty valueProperty() { return value; }
}
```

Pattern #3: Read Only Property

- A property that is readable and observable. Can be written by the class.
- Encapsulate the value in a property object.
- Final getter and optional **private** setter.
- Property method returns a read only property.

```java
public class SomeClass {
    private IntegerProperty value = new SimpleIntegerProperty();
    public SomeClass(int value) {
        this.value.set(value);
    }
    public final int getValue() { return value.get(); }
    private final void setValue(int value) { this.value.set(value); }
    public ReadOnlyIntegerProperty valueProperty() { return value; }
}
```

Pattern #4: Immutable Property, Mutable State

- A property that is a reference to an object.
- The reference doesn't change, but the properties of the object can.
- Final getter. No setter.

```
public class Customer {
    private Account account;
    public Customer() {
        account = new Account("Mr Smith");
    }
    public final Account getAccount() { return account; }
}
```

- Not possible: customer.~~setAccount~~(new Account("Dr Smith"));
- Still possible: customer.getAccount().setName("Dr Smith");

Account class with JavaFX Properties

```
public class Account {
    private StringProperty name = new SimpleStringProperty();
    private DoubleProperty balance = new SimpleDoubleProperty();
    public Account(String name, double balance) {
        this.name.set(name);
        this.balance.set(balance);
    }
    public final String getName() { return name.get(); }
    public final void setName(String name) { this.name.set(name); }
    public StringProperty nameProperty() { return name; }
    public final double getBalance() { return balance.get(); }
    public ReadOnlyDoubleProperty balanceProperty() { return balance; }
    public void deposit(double amount) { balance.set(getBalance() + amount); }
    public void withdraw(double amount) { balance.set(getBalance() - amount);
}
```

Customer class with JavaFX properties

```
public class Customer {
    private final Account account;
    public Customer(String name) {
        account = new Account(name);
    }
    public final Account getAccount() {
        return account;
    }
}
```

The Payoff

- Each property implements the observer pattern.
- The view can be notified whenever a property changes.

View + Controller

View + Controller

The view is an FXML file.
It contains the scene graph.

```
<VBox>
    <Label text="Name:"/>
    ...
</VBox>
```

The controller is a Java class.
It contains the event handlers.

```
public class AccountController {
    ...
}
```

Each view has an associated controller class to handle user interaction.

Link the view to the controller

- The root node of a scene specifies the controller:

```
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="AccountController">
    <Label text="Name:"/>
    <TextField/>
    <Label text="Balance:"/>
    <TextField/>
    <Label text="Transaction amount ($):"/>
    <TextField text="0.00"/>
    <Button text="Deposit"/>
    <Button text="Withdraw"/>
</VBox>
```

- **AccountController** is the class name for this view's controller.

Node IDs

- Assign an ID to each node that the controller will need to access:

```
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="AccountController">
    <Label text="Name:"/>
    <TextField fx:id="nameTf"/>
    <Label text="Balance:"/>
    <TextField fx:id="balanceTf"/>
    <Label text="Transaction amount ($):"/>
    <TextField fx:id="amountTf" text="0.00"/>
    <Button text="Deposit"/>
    <Button text="Withdraw"/>
</VBox>
```

The controller

```
import javafx.fxml.*;

public class AccountController {
    @FXML private TextField nameTf;
    @FXML private TextField balanceTf;
    @FXML private TextField amountTf;
}
```

- For each node with an `fx:id`, declare a field in the controller with the same name and annotate it with `@FXML`.
- Each node will be "injected" into the corresponding field.

Define TextField getters/setters as needed

```
public class AccountController {
  @FXML private TextField nameTf;
  @FXML private TextField balanceTf;
  @FXML private TextField amountTf;

  private String getName() { return nameTf.getText(); }
  private double getBalance() { return Double.parseDouble(balanceTf.getText());
}
  private double getAmount() { return Double.parseDouble(amountTf.getText()); }
  private void setAmount(double amount) { amountTf.setText(“” + amount); }
}
```

Set the event handlers on the buttons in FXML

```
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="AccountController">
    <Label text="Name:"/>
    <TextField fx:id="nameTf"/>
    <Label text="Balance:"/>
    <TextField fx:id="balanceTf"/>
    <Label text="Transaction amount ($):"/>
    <TextField fx:id="amountTf" text="0.00"/>
    <Button text="Deposit"  onAction="#handleDeposit"/>
    <Button text="Withdraw"  onAction="#handleWithdraw"/>
</VBox>
```

- **handleDeposit** and **handleWithdraw** name methods in the controller to handle action events for each button.

Define the event handlers in the controller

```java
public class AccountController {
    ...
    private Account account = new Account("Mr. Smith");

    @FXML private void handleDeposit(ActionEvent event) {
        account.deposit(getAmount());
        setAmount(0);
    }

    @FXML private void handleWithdraw(ActionEvent event) {
        account.withdraw(getAmount());
        setAmount(0);
    }
}
```

The @FXML initialize method

Nodes annotated with `@FXML` are injected by `FXMLLoader`. BUT, only AFTER the constructor has been called.

Dereferencing a node from the constructor results in a `NullPointerException`.

Solution: Put initialisation code into the `@FXML`-annotated **initialize** method. The `FXMLLoader` calls this after injecting the nodes.

```java
public class AccountController {
  @FXML private TextField nameTf;
  @FXML private TextField balanceTf;
  @FXML private TextField amountTf;

  public AccountController() {
    amountTf.setText("0.00");
  }


  @FXML private void initialize() {
   amountTf.setText("0.00");
  }
}
```

Property bindings

Goal: Property p1 is updated whenever property p2 changes.
      i.e. p1 observes p2.

```
p1.bind(p2);
```

Goal: Properties p1 and p2 are both updated whenever the other changes.
      i.e. p1 observes p2 *and* p2 observes p1

```
p1.bindBidirectional(p2);
```

Link the view to the model

```
public class AccountController {
    ...
    @FXML private void initialize() {
        nameTf.textProperty().bindBidirectional(account.nameProperty());
        balanceTf.textProperty().bind(account.balanceProperty());
    }
}
```

- Bidirectional bind:
  - When the account name changes, nameTf is updated.
  - When the user edits nameTf, the account name is updated.
- Unidirectional bind:
  - When the bank balance changes, balanceTf is updated.
  - The user can't edit balanceTf.

Expression bindings in FXML

FXML supports unidirectional property bindings through ${...} notation.

- **In Java**: `balanceTf.textProperty().bind(controller.getAccount().balanceProperty());`
- **In FXML**: `<TextField fx:id="balanceTf" text="${controller.account.balance}"/>`

Binding expressions in FXML understand property naming conventions:

- Given the expression `foo.bar`, FXML looks for the a `getBar()` method or `isBar()` method or `barProperty()` method inside of object foo.

FXML does not support bidirectional bindings. They must be done in Java.

For more, see: https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html#expression_binding

Link the view to the model in FXML

- Bind `balanceTf` to the balance inside the account inside the controller:

```
<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="AccountController">
    ...
    <Label text="Balance:"/>
    <TextField fx:id="balanceTf"  text="${controller.account.balance}" />
</VBox>
```

- Expose an `account` property in the controller.
  Property Pattern #4: Immutable property with mutable state:

```
public class AccountController {
    private Account account = new Account("Mr. Smith");
    public final Account getAccount() { return account; }
}
```

More complex bindings in Java

- Bind a text field to a double property formatted as currency:
  ```
  balanceTf.textProperty().bind(account.balanceProperty().asString("$%.2f"));
  ```
- Bind a text field to a double property concatenated with a string:
  ```
  balanceTf.textProperty().bind(account.balanceProperty().asString().concat(" dollars"));
  ```
- Bind a property to another property to another property times a number:
  ```
  public class Account {
      private static final double INTEREST_RATE = 0.07;
      private DoubleProperty balance = new SimpleDoubleProperty();
      private DoubleProperty interest = new SimpleDoubleProperty();
      public Account() {
          interest.bind(balance.multiply(INTEREST_RATE));
      }
  ```
  Now interest changes whenever balance changes.

For more, see: https://docs.oracle.com/javase/8/javafx/properties-binding-tutorial/binding.htm
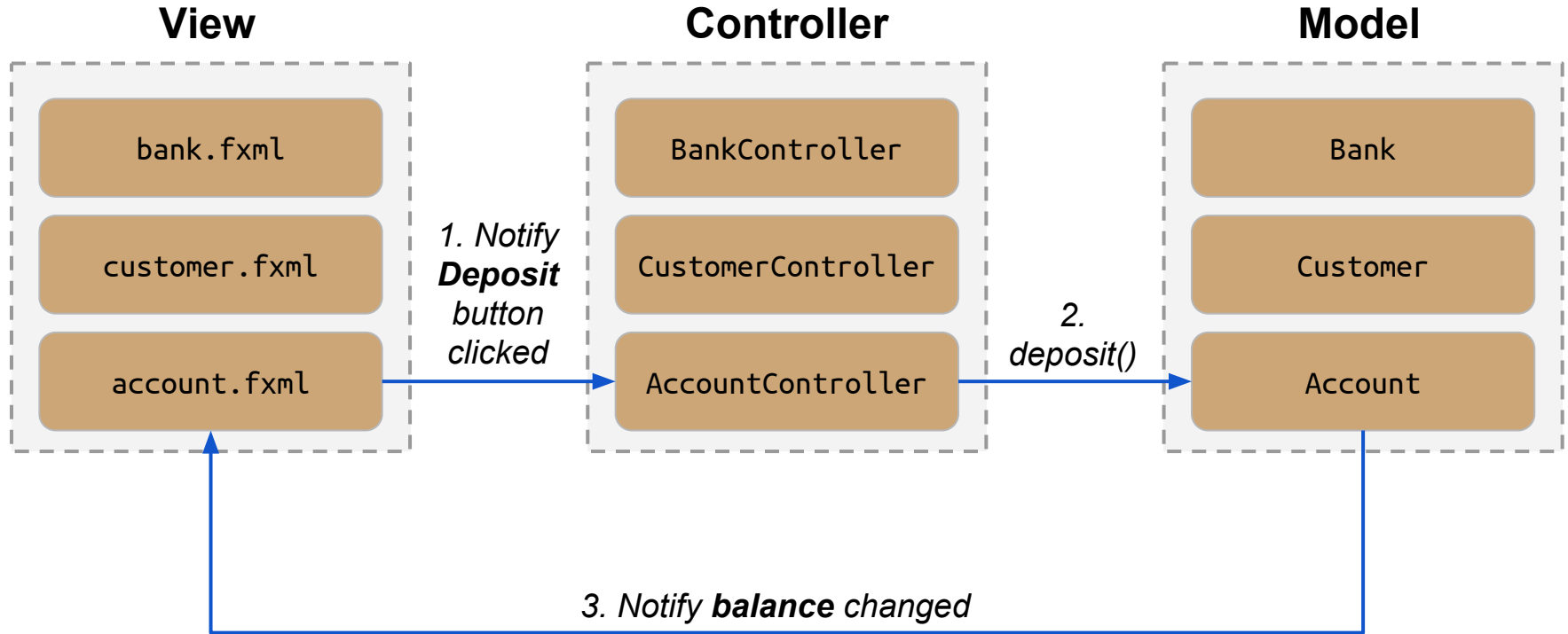
More complex bindings in FXML

The JavaFX expression binding language supports:

- Dot notation for properties. e.g. controller.account.balance
- Literals: "a string", 'a string', 3.45, 27, null, true, false
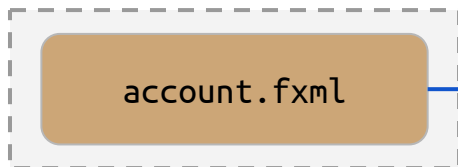- Operators: +, -, *, /, !, &&, ||, …..

Examples:

- `${account.balance * 0.07}` binds to the account balance times 0.07
- `${account.balance + ' dollars'}` binds to a string "1000.0 dollars" where 1000.0 is updated whenever the account balance changes.

MVC summary

View	Controller	Model
bank.fxml	BankController	Bank
customer.fxml	CustomerController	Customer
account.fxml	AccountController	Account

*1. Notify **Deposit** button clicked*

2. deposit()

*3. Notify **balance** changed*

MVC Summary: #1

View

Controller

account.fxml

*1. Notify **Deposit** button clicked*

AccountController

```
<TextField fx:id="amountTf"/>
<Button text="Deposit"
    onAction="#handleDeposit"/>
```
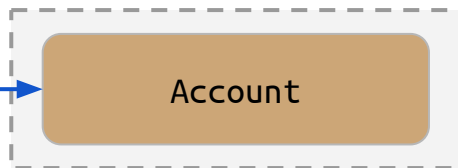
```
public class AccountController {
  @FXML
  private void handleDeposit(ActionEvent e) {
    account.deposit(getAmount());
    setAmount(0);
  }
}
```

MVC Summary: #2

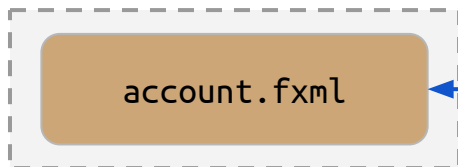**Controller**   **Model**

```
┌─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐                                        ┌─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
│  ┌──────────────────┐  │          2. deposit()                  │  ┌──────────────────┐  │
│  │ AccountController│  │ ──────────────────────────────────────▶│  │     Account      │  │
│  └──────────────────┘  │                                        │  └──────────────────┘  │
└─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘                                        └─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
```

```java
public class AccountController {
  @FXML
  private void handleDeposit(ActionEvent e) {
    account.deposit(getAmount());
    setAmount(0);
  }
}
```

```java
public class Account {
    private DoubleProperty balance = ...;

    public void deposit(double amount) {
      balance.set(getBalance() + amount);
    }
}
```
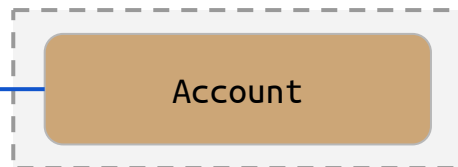
MVC Summary: #3

View

Model

account.fxml

Account

*3. notify **balance** changed*

```
<TextField
    fx:id="balanceTf"
    text="${controller.account.balance}"/>
```

```
public class Account {
    private DoubleProperty balance = ...;

    public void deposit(double amount) {
        balance.set(getBalance() + amount);
    }
}
```

Download the lecture code
for the full program