Week 9

GUI Lists

This week

- Packages
- ListView
- Opening and closing windows
- Controls
- Change listeners

Packages

Packages

- A package is a collection of related classes.

- Each application or library should be placed in its own package.

- To avoid two programmers using the same package name for their application, we follow a convention:
  - Companies use the reverse of their domain name.
    e.g. For domain name `mycompany.com`, use the package name `com.mycompany`
  - Different applications made by the same company are in sub-packages.
    e.g. `com.mycompany.calculatorapp` and `com.mycompany.studyapp`

Packages

- Complex applications are further divided into sub-packages.

- e.g. in an MVC application, you may have 3 sub-packages:
  - `com.mycompany.studyapp.model` contains the domain model classes
  - `com.mycompany.studyapp.view` contains the views
  - `com.mycompany.studyapp.controller` contains the controller classes

Package declarations

- Declare a class in a package with a package declaration:

```
package com.mycompany.bankapp.model ;

public class Account {
    ...
}
```

- On the file system, sub-packages map onto sub-directories.
  e.g. the `Account` class is stored in the corresponding sub-directory:

```
com/mycompany/bankapp/model/ Account.java
```

Implications for FXML

- In FXML, use a fully-qualified class name to refer to a class in a package:

  fx:controller="com.mycompany.bankapp.controller.AccountController"

- In Java, use an absolute path to refer to an FXML file in a package:

  new FXMLLoader(getClass().getResources("/com/mycompany.bankapp/view/account.fxml"));

ListView

ListView<X>

- A ListVew<X> displays a list of items of type X.

- Items can be either:
  - Strings
  - Objects that have a toString() function

- Create a ListView in FXML:
  ```
  <ListView fx:id="accountsLv"/>
  ```

- Create a ListView in Java:
  ```
  ListView<Account> accountsLv = new ListView<Account>();
  ```
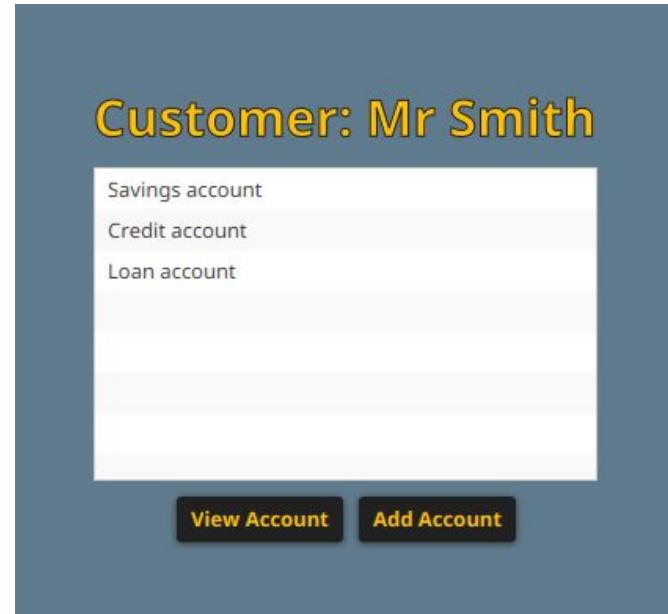
ListView Selection Models

- ListViews support two selection models:
  - SINGLE selection (the default model)
  - MULTIPLE selection

- In both selection models:
  - Clicking an item selects that item
  - The previously selected item is also deselected

- In the multiple selection model:
  - Shift-click or control-click to select multiple items

- If your application requires multiple selection:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```


Customer: Mr Smith

Savings account
Credit account
Loan account

View Account   Add Account
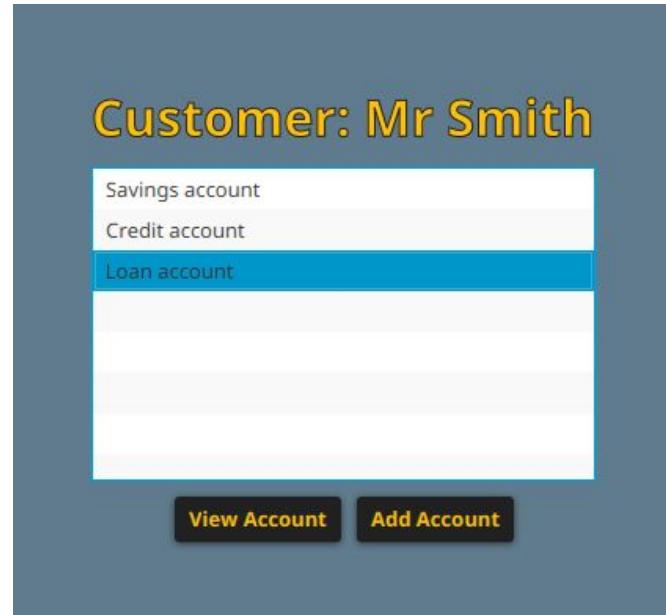
Setting a placeholder

- A placeholder node is shown when the ListView is empty.

- In FXML:
```
<ListView>
    <placeholder>
        <Label text="No accounts"/>
    </placeholder>
</ListView>
```

- In Java:
```
accountsLv.setPlaceholder(new Label("No accounts"));
```

11

Setting preferred dimensions

- In FXML:

```
<ListView prefWidth="300" prefHeight="200"/>
```

- In Java:

```
accountsLv.setPrefWidth(300);
accountsLv.setPrefHeight(200);
```

Linking a ListView to the model

- **Goal**: Whenever the model changes the view is updated.

```
public class Customer {
  private LinkedList<Account> accounts = new LinkedList<Account>();

  public void addAccount(String type) {
    accounts.add(new Account(type));
  }
}
```

We must notify the ListView that
a new Account was added to the list.

Linking a ListView to the model

- **Solution**: Use an ObservableList

```
public class Customer {
  private ObservableList<Account> accounts = FXCollections.
observableArrayList();

  public void addAccount(String type) {
    accounts.add(new Account(type));
  }
}
```

- Observers are notified whenever the list contents changes.

ObservableList "is a" relationships

«interface» Observable‹X›
void addListener() void removeLIstener()

«interface» List‹X›
void add(X element) ...
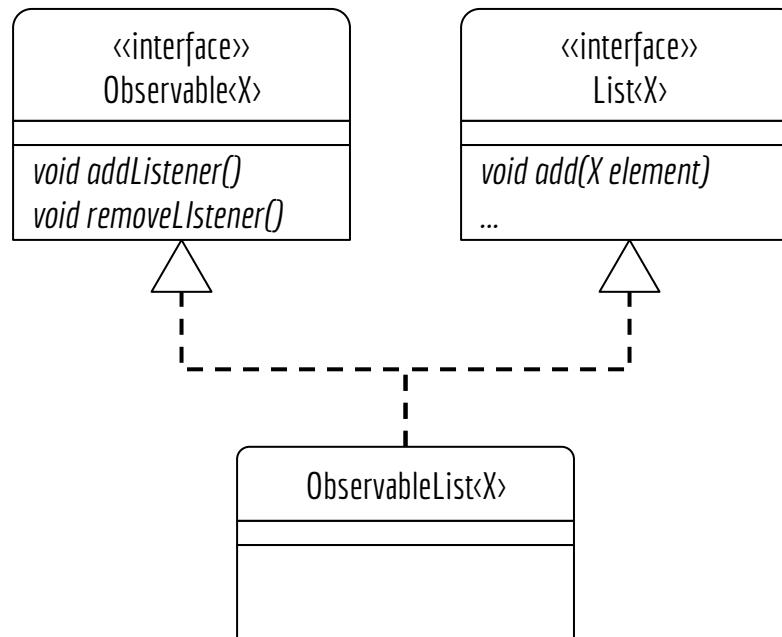
ObservableList‹X›

Linking a ListView to the model

- Define the observable list as an immutable property with mutable state:

```
public class Customer {
    private ObservableList<Account> accounts = ...;
    public final ObservableList<Account> getAccounts() { return accounts;
}
}
```

- Bind the "items" property of ListView to the accounts property of customer
  - In FXML: `<ListView items="${controller.customer.accounts}"/>`
  - In Java: `accountsLv.setItems(customer.getAccounts());`

Selecting a ListView item

Selecting a ListView item

Goal: The user selects an item from a ListView then clicks a button to perform an action on the selected item.

Solution: Set the onAction handler for the button to perform the following two steps:

1. Get the selected item (pattern)
2. Perform an action on that item

ListView getter pattern

- A `ListView` has a getter that gets the currently selected item.
- It uses the `getSelectedItem()` method of the selection model.

```
public class CustomerController {
    @FXML private ListView<Account> accountsLv;


    private Account getSelectedAccount() {
        return accountsLv.getSelectionModel().getSelectedItem();
    }
}
```

Example: View the selected account

FXML file:

```
<ListView fx:id="accountsLv"/>
<Button text="View Account"  onAction="#handleViewAccount" />
```

Controller:

```
public class CustomerController {
    @FXML private ListView<Account> accountsLv;
    private Account getSelectedAccount() {
        return accountsLv.getSelectionModel().
getSelectedItem();
    }
    @FXML private void handleViewAccount(ActionEvent event) {
        Account account = getSelectedAccount();
        System.out.println("You selected: " + account);
    }
```

If `accountsLv` is clicked when no item is selected, `getSelectedAccount()` returns `null`.

Opening a window

Opening a window

Customer: Mr Smith

- Savings account
- Credit account
- Loan account

View Account   **Add Account**

Click
View Account

Savings account

Type: Savings

Balance: $1000.00

$ 0.00   **Deposit**   **Withdraw**
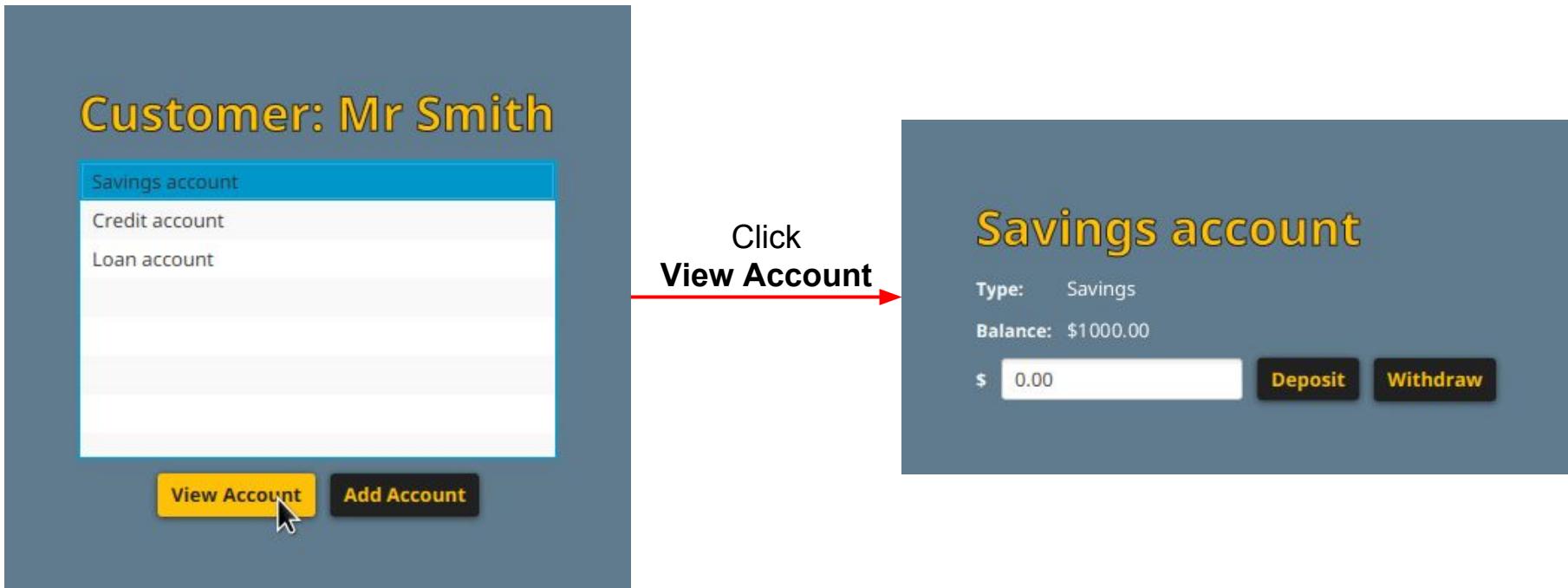
Opening a window

- In MVC, each window requires 3 components:

	View	**Controller**	**Model**
Customer window:	customer.fxml	CustomerController	Customer
Account window:	account.fxml	AccountController	Account
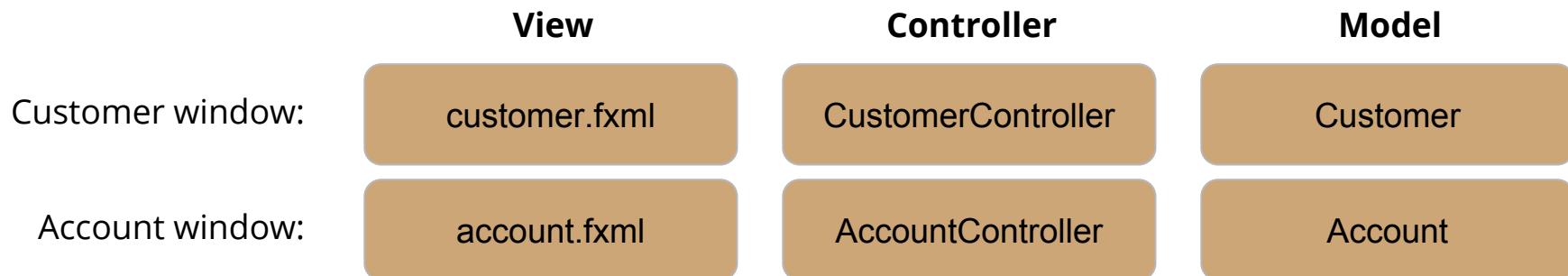
- To open a new window:
  - Load the view from FXML (this in turn creates the controller)
  - Link the controller to the appropriate model
  - Create a new Stage, and show the view's scene graph on the stage

Opening a window

- UTS provides a package to help opening windows:

  ```
  import au.uts.edu.ap.javafx.*;
  ```

- The ViewLoader class simplifies the process of showing a window:

  ```
  ViewLoader.showStage(<model>, <fxml>, <title>, <stage>);
  ```

The model object   **The FXML file**   **The window title**   **The stage**

Source code for ViewLoader

```
package au.edu.uts.ap.javafx;
public class ViewLoader {
  public static <T> void showStage(T model, String fxml, String title, Stage stage)
      throws IOException {
    FXMLLoader loader = new FXMLLoader(Controller.class.getResource(fxml), null, null,
                type -> {
                    try {
                        Controller<T> controller = (Controller<T>)type.newInstance();
                        controller.model = model;
                        controller.stage = stage;
                        return controller;
                    } catch (Exception e) { throw new RuntimeException(e); }
                });
    Parent root = loader.load();
    stage.setTitle(title);
    stage.setScene(new Scene(root));
    stage.sizeToScene();
    stage.show();
  }
}
```

The model and stage are automatically "injected" into the controller.

ViewLoader examples

- Show a customer on the primary stage:

```
public class BankApplication extends Application {
  @Override public void start(Stage stage) throws Exception {
    Customer customer = new Customer("Mr Smith");
    ViewLoader.showStage(customer, "/view/customer.fxml", "Customer",
stage);
  }
}
```

- Show an account on a new stage (new window):

```
ViewLoader.showStage(account, "/view/account.fxml", "Account",  new Stage());
```

Making controllers compatible with ViewLoader

- In the same helper package is an abstract `Controller` class:

```
package au.edu.uts.ap.javafx;

public abstract class Controller<X> {
    protected X model;
    protected Stage stage;
}
```

- If your controller extends this class, it will inherit the `model` and `stage` fields. `<X>` is the type of the model being used.

- `ViewLoader` will try to inject the model and stage into these fields.

Exposing a model property in the controller

- The old way…

```
public class CustomerController {
    private Customer customer = new Customer("Mr Smith");
    public final Customer getCustomer() { return customer; }
}
```

- The new way…

```
public class CustomerController extends Controller<Customer> {
    public final Customer getCustomer() { return model; }
}
```

- The getter returns the model which is now an inherited field.

Example: CustomerController

```java
public class CustomerController extends Controller<Customer> {
  @FXML private ListView<Account> accountsLv;
  public final Customer getCustomer() { return model; }
  private Account getSelectedAccount() {
    return accountsLv.getSelectionModel().getSelectedItem();
  }
  @FXML private void handleViewAccount(ActionEvent event) {
    Account account = getSelectedAccount();
    ViewLoader.showStage(account, "/view/account.fxml", "Account", new Stage());
  }
}
```

- Upon clicking the button, show `/view/account.fxml` and inject the selected account into the account controller.

Example: AccountController

```java
public class AccountController extends Controller<Account> {
  @FXML private TextField amountTf;
  public final Account getAccount() { return model; }
  private int getAmount() { return Integer.parseInt(amountTf.getText()); }
  private void setAmount(int amount) { amountTf.setText("" + amount); }
  @FXML private void handleDeposit(ActionEvent event) {
    getAccount().deposit(getAmount());
    setAmount(0);
  }
  @FXML private void handleWithdraw(ActionEvent event) { ... }
}
```

- The model field holds the selected account.
- Use getAccount() to access the account.

Closing a window

Closing a window
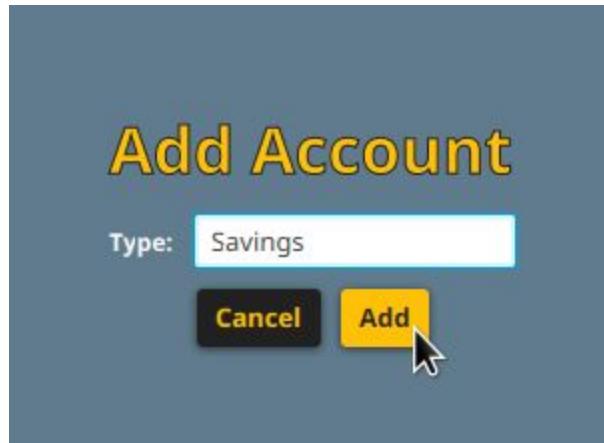
Closing a window

- Close a window with `stage.close()`.
- Every subclass of `Controller<X>` inherits a `stage` field. Therefore, any method within a controller can call `stage.close()`.

```java
public class AddAccountController extends Controller<Customer> {
    @FXML private String typeTf;
    public final Customer getCustomer() { return model; }
    private String getType() { return typeTf.getText(); }
    @FXML public void handleAdd(ActionEvent event) {
        getCustomer().addAccount(getType());
        stage.close();
    }
}
```

JavaFX controls

JavaFX controls
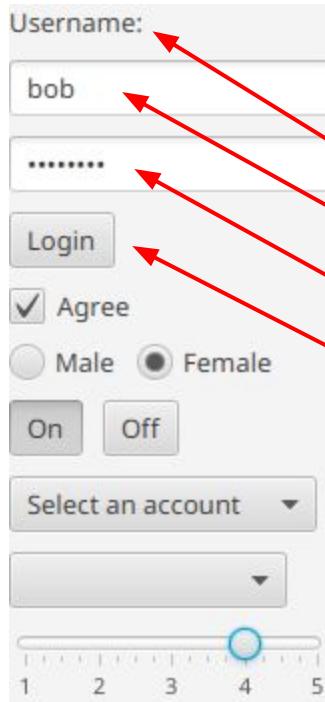
- JavaFX controls are nodes that tend to appear in forms.
- We have already seen:

  - `Label`
  - `TextField`
  - `PasswordField`
  - `Button`

More controls

Username:

bob

••••••••

Login

☑ Agree

○ Male ● Female

On   Off

Select an account ▼

▼

|1 2 3 4 5
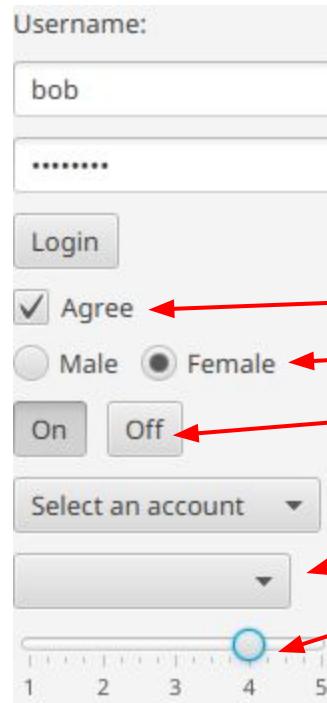
- JavaFX also supports:

  - CheckBox
  - RadioButton
  - ToggleButton
  - ComboBox
  - ChoiceBox
  - Slider

For more, see: https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm

CheckBox

- A `CheckBox` is either selected or not.

- FXML: `<CheckBox text="Agree" fx:id="agreeCb"/>`

- Controller: `@FXML private CheckBox agreeCb;`

- `CheckBox` properties:
  - `selected`: indicates whether the `CheckBox` is currently selected.

- e.g.
  ```
  if (agreeCb.isSelected())
      System.out.println("User agrees");
  ```

RadioButton

- Related `RadioButton`s are placed in a `ToggleGroup`. Only one `RadioButton` in the group can be selected at a time.
- FXML:
  ```
  <fx:define><ToggleGroup fx:id="genderTg"/></fx:define>
  <RadioButton text="Male" userData="m" toggleGroup="$genderTg"/>
  <RadioButton text="Female" userData="f" toggleGroup="$genderTg"/>
  ```
- Controller: `@FXML private ToggleGroup genderTg;`
- Properties of `RadioButton`
  - `text`: the text to display
  - `userData`: the raw data (e.g. to store in the model, or a database)
- Properties of `ToggleGroup`
  - `selectedToggle`: the `RadioButton` that is currently selected
- e.g. `String s = genderTg.getSelectedToggle().getUserData().toString();`
 `System.out.println("The user selected: " + s);`

ToggleButton

- Behaves the same way as a `RadioButton`.

- FXML:
  ```
  <fx:define><ToggleGroup fx:id="musicTg"/></fx:define>
  <ToggleButton text="On" userData="on" toggleGroup="$musicTg"/>
  <ToggleButton text="Off" userData="off" toggleGroup="$musicTg"/>
  ```

ComboBox
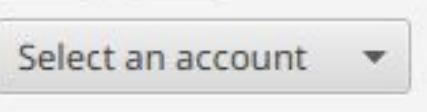
Select an account ▼

- A `ComboBox` shows a popup list of items with a prompt.

- FXML: `<ComboBox fx:id="accountsCmb" promptText="Select an account"`
  `items="${controller.customer.accounts}"/>`

- Controller: `@FXML private ComboBox accountsCmb;`

- `ComboBox` properties are similar to `ListView` properties:
  - `items`: defines the list of items to display.
  - `selectionModel`: represents the selection state.

- e.g. `Account account = accountsCmb.getSelectionModel().getSelectedItem();`

ChoiceBox

- A `ChoiceBox` shows a popup list of items *without* a prompt.

- FXML: `<ComboBox fx:id="accountsChb" items="${controller.customer.accounts}"/>`

- Works the same way as a `ComboBox`.

Slider

- A `Slider` shows a thumb that can be slid on a track.

- FXML: `<Slider min="1" max="5" showTickMarks="true" showTickLabels="true"`
          `majorTickUnit="1"/>`

For more, see:
https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm

Getter/Setter patterns for controls

- It is good practice to define getters and setters to wrap the contolled value. E.g.

```
private String getGender() {
    return genderTg.getSelectedToggle().getUserData().toString();
}
private boolean isAgree() {
    return agreeCb.isSelected();
}
private Account getAccount() {
    return accountsCmb.getSelectionModel().getSelectedItem();
}
```

Change listeners

Change listeners

- **Goal**: Be notified whenever an observable value changes.
- **Examples**:
  - Be notified when the text in a `TextField` changes.
  - Be notified when the selected item of a `ComboBox` changes.
  - Be notified when the selected toggle of a `ToggleGroup` changes.
- **Solution**: Register an observer that implements the `ChangeListener<X>` interface, where `<X>` is the type of value observed.

```
import javafx.beans.value.*;
public interface ChangeListener<X> {
    void changed(ObservableValue<? extends X> observable,
                 X oldValue, X newValue);
}
```

- Register the observer with `observable.addListener(observer);`

Any property can be observed for changes

- **Print the account balance whenever it changes:**
```
account.balanceProperty().addListener((obs, oldBal, newBal) ->
    System.out.println("Balance changed from "+oldBal+" to "+newBal)
);
```
- **Print the text of a TextField whenever it changes:**
```
nameTf.textProperty().addListener((obj, oldText, newText) ->
    System.out.println("Text updated to " + newText)
);
```
- **Print the selected account whenever it is selected:**
```
accountsLv.getSelectionModel().addListener((o, oldAcct, newAcct) -> {
    if (newAcct != null) // if an account was selected
        System.out.println("You selected " + newAcct);
});
```

Example: Enable button when account is selected
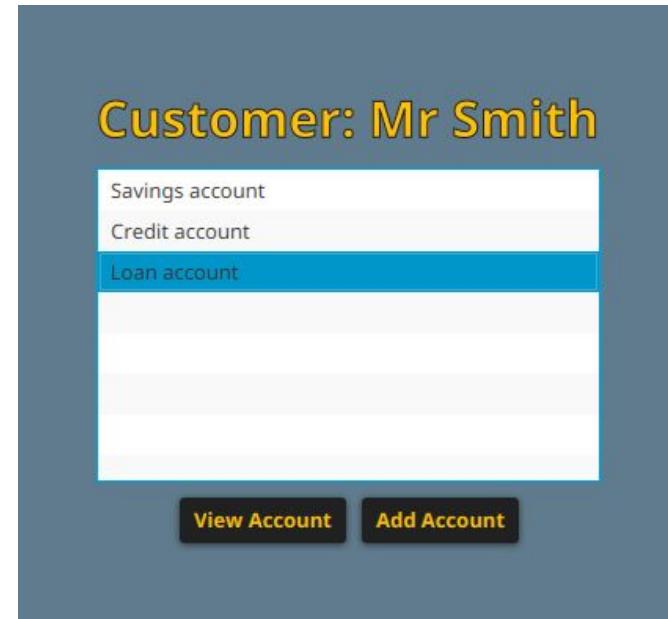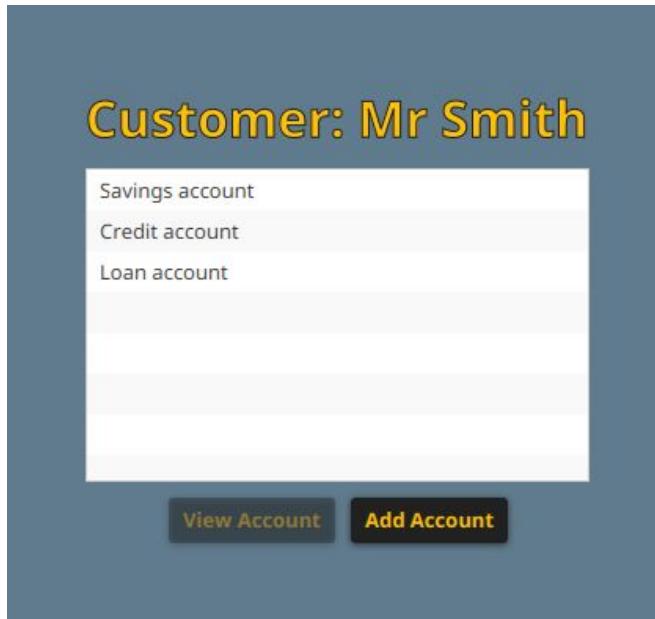
Solution

- FXML:
```
<ListView fx:id="accountsLv" items="${controller.customer.accounts}"/>
<Button fx:id="viewAccountBtn" text="View Account"  disable="true"
        onAction="#handleViewAccount"/>
```
- Controller:
```
@FXML private void initialize() {
  accountsLv.getSelectionModel(). selectedItemProperty ().addListener(
    (observable, oldAccount, newAccount) ->
      viewAccountBtn. setDisable (getAccount() == null));
}
```
- This makes use of two properties:
  - Register a ChangeListener to the **selectedItem** property of the selection model.
  - When the selection changes, update the **disable** property of the Button.