Object-Oriented Programming

Lecturer:

Teaching Assistant:

w

**Lab 05: GUI Programming**

# Swing components

## **AWTAccumulator**

### Createclass{ with the source code as below

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

### Explanation

## **SwingAccumulator**

### Create class SwingAccumulator with the source code as below:

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

### Explanation

* In Swing, the top-level container is JFrame which is inherited by the application class.
* In the constructor, we set up the GUI components in the JFrame object and the event-handling:
* Unlike AWT, the JComponents shall not be added onto the top-level container (e.g., JFrame, JApplet) directly because they are lightweight components. The JComponents must be added onto the so-called content-pane of the top-level container. Content-pane is in fact a java.awt.Container that can be used to group and layout components.
* In line 15, we get the content-pane of the top-level container.
* In line 16, the layout of the content-pane is set as GridLayout
* In line 18, we add the first component to our content-pane, an anonymous JLabel
* In line 20-22, we add a JTextField component to our content-pane, where the user will enter values. We add a listener which takes this JTextField component as the source.
* In line 24, we add another anonymous JLabel to our content-pane
* In line 26 – 28, we add a JTextField component to our content-pane, where the accumulated sum of entered values will be displayed. The component is set to read-only in line 27.
* In line 30 – 32, the title & size of the JFrame is set, and the Frame visibility is set to true, which shows the JFrame to us.
* In the listener class (line 39 - 47), the code for event-handling is exactly like the AWTAccumulator.
* In the main() method, we invoke the SwingAccumulator constructor to set up the GUI

## Compare Swing and AWT elements

Programming with AWT and Swing is quite similar (similar elements including container/components, event-handling). However, there are some differences that you need to note:

* The top-level containers in Swing and AWT
* The class name of components in AWT and corresponding class’s name in Swing

# Organizing Swing components with Layout Managers

**Note**: For this exercise, you will continue using GUIProject, and put all your source code in the package “**hust.soict.globalict.swing**” (for ICT) or “**hust.soict.dsai.swing**” (for DS & AI).

In Swing, there are two groups of GUI classes, the containers and the components. We have worked with several component classes in previous exercises. Now, we will investigate more on the containers.

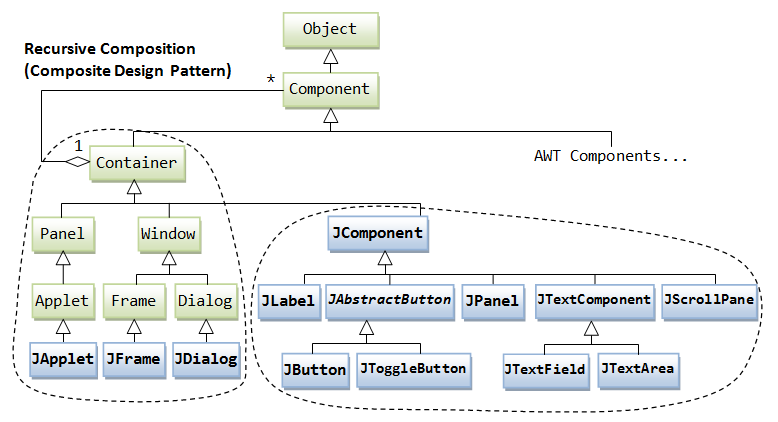


Figure 4. AWT and Swing elements

## Swing top-level and secondary-level containers

## Using JPanel as secondary-level container to organize components

### Create class NumberGrid

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

### Adding buttons

A computer screen shot of a program code

Description automatically generated

### Complete inner class ButtonListener

A computer screen shot of a program code

Description automatically generated

# Create a graphical user interface for AIMS with Swing

## View Store Screen

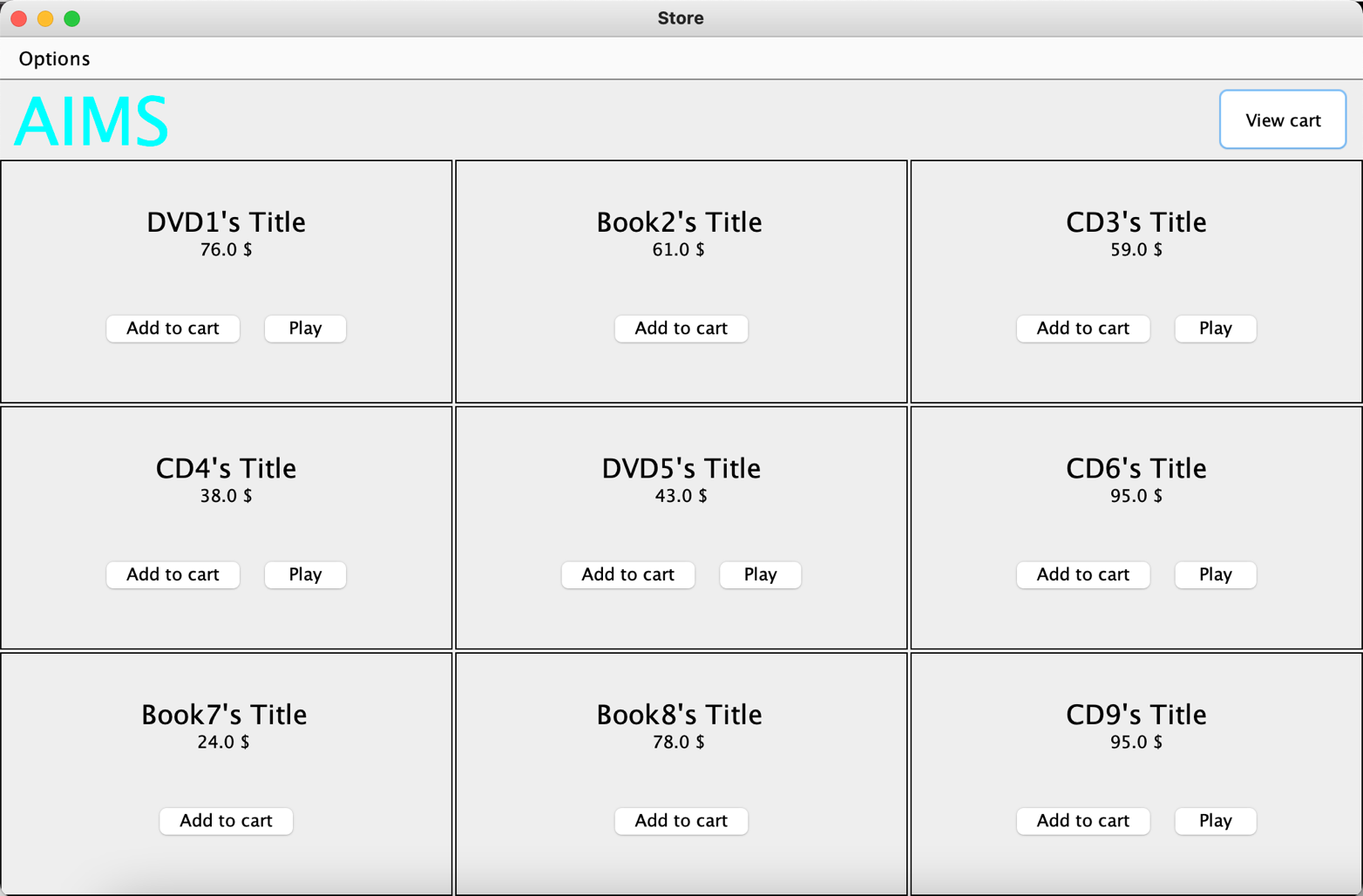


Figure 9. View Store Screen

### Create the StoreScreen class

A screen shot of a computer

Description automatically generated

### The NORTH component

A screen shot of a computer program

Description automatically generated

### The CENTER component

A screen shot of a computer code

Description automatically generated

### The MediaStore class

A computer screen shot of a program code

Description automatically generated

### Putting it all together

A computer screen shot of code

Description automatically generated

A screenshot of a computer program

Description automatically generated

## Adding more user interaction

A screenshot of a computer program

Description automatically generated

# JavaFX API

A computer screen shot of a signature

Description automatically generated

## Create the FXML file

### Create and open the FXML file in Scene Builder from Eclipse

A screen shot of a computer program

Description automatically generated

### Building the GUI

A screenshot of a computer

Description automatically generated

## Create the controller class

A screen shot of a computer program

Description automatically generated

## Create the application

A screen shot of a computer program

Description automatically generated

## Practice exercise

A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

# Setting up the View Cart Screen with ScreenBuilder

A screenshot of a computer

Description automatically generated

## Setting up the BorderPane

A screenshot of a number

Description automatically generated

## Setting up the TOP area

## Setting up the RIGHT area

A screenshot of a computer

Description automatically generated

# Integrating JavaFX into Swing application – The **JFXPanel** class

A screen shot of a computer program

Description automatically generated

# View the items in cart – JavaFX’s data-driven UI

A screenshot of a computer

Description automatically generated

# Updating buttons based on selected item in **TableView** – **ChangeListener**

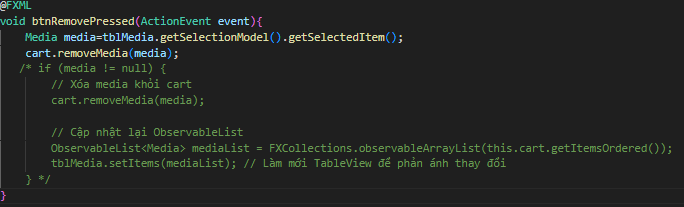
A screen shot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

# Deleting a media



# Filter items in cart – **FilteredList**

This exercise is optional (full credit can still be given for this lab without doing this exercise), but you can do it for extra credit.

We will implement a filter that is re-applied every time the user makes a change in the filter text field. To do this, again, we need references to the text field where the user inputs the filter string, and the two radio buttons (to determine what criteria is being used to filter).

Similar to the above, please add the fx:id property for the components in SceneBuilder and create three corresponding attributes in the controller:

* The TextField: **tfFilter**
* The RadioButton “By ID”: **radioBtnFilterId**
* The RadioButton “By Title”: **radioBtnFilterTitle**

At the end of the initialize() method, put some code to add a ChangeListener to the TextField’s text property (illustrated in Figure 45):

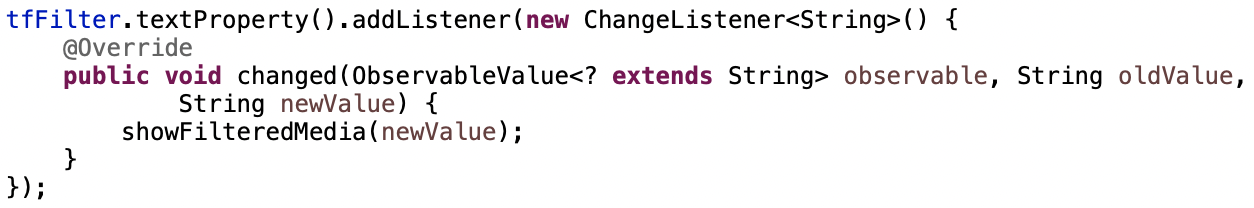


Figure 45. Adding ChangListener for tfFilter in initialize()

Please implement by yourself the showFilteredMedia() method. **Hint**: You might need to change the source code in previous exercises. Wrap the ObservableList in a FilteredList and set a new Predicate for the FilteredList each time you need to apply a new filter.

# Complete the Aims GUI application

A screenshot of a computer

Description automatically generated

Total thay đổi khi remove dvd

A screenshot of a computer

Description automatically generated

Place Order  
A screenshot of a computer

Description automatically generated

Sau khi đặt hàng thành công sẽ xóa hết khỏi giỏ hàng

A screenshot of a computer

Description automatically generated

Thêm hàm trong store screen để khi ấn view cart, sẽ chuyển giao diện từ store sang cart

A screen shot of a computer code

Description automatically generated

# Check all the previous source codes to catch/handle/delegate runtime exceptions

Trongbook.java  
A computer screen shot of text

Description automatically generated

Sửa lỗi nếu không tìm thấy tác giả vẫn được liệt kê

# Create a class which inherits from **Exception**

The **PlayerException** class represents an exception that will be thrown when an exceptional condition occurs during the playing of a media in your **AimsProject**.

## Create new class named **PlayerException**

- Enter the following specifications in the New Java Class dialog:

* Name: **PlayerException**
* Package: **hust.soict.[globalict||dsai].aims.exception**
* Access Modifier: **public**
* Superclass: **java.lang.Exception**
* Constructor from Superclass: checked
* **public static void main(String [] args):** do not check
* All other boxes: do not check

- Finish

## Raise the **PlayerException** in the **play()** method

- Update **play()** method in **DigitalVideoDisc** and **Track**

* For each of **DigitalVideoDisc** and **Track**, update the **play()** method to first check the object's length using **getLength()** method. If the length of the **Media** is less than or equal to zero, the **Media** object cannot be played.
* At this point, you should output an error message using **System.err.println()** method and the **PlayerException** should be raised.

- The example of codes and results for the **play**() of **DigitalVideoDisc** are illustrated in the following figures.



Figure 47. Sample code for method play() of DigitalVideoDisc

**- Save your changes and make the same with the play() method of Track.**

## Update **play()** in the **Playable** interface

- Change the method signature for the **Playable** interface's **play()** method to include the throws **PlayerException** keywords.

## Update **play()** in **CompactDisc**

- The **play**() method in the **CompactDisc** is more interesting because not only it is possible for the **CompactDisc** to have an invalid **length** of 0 or less, but it is also possible that as it iterates through the tracks to play each one, there may have a track of length 0 or less

- First update the **play**() method in **CompactDisc** class to check the length using **getLength**() method as you did with **DigitalVideoDisc**

- Raise the **PlayerException**. Be sure to change the method signature to include **throws** **PlayerException** keywords.

- Update the **play()** method to catch a **PlayerException** raised by each **Track** using block **try-catch.**

The code example is shown as follows.



Figure 48. Sample code for method play() of CompactDisc

- You should modify the above source code so that if any track in a **CD** can’t play, it throws a **PlayerException** exception.

# Update the **Aims** class

- The **Aims** class must be updated to handle any exceptions generated when the **play()** methods are called. What happens when you don’t update for them to catch?

- Try to use **try-catch** block when you call the **play()** method of **Media's** objects.

With all these steps, you have practiced with User-defined Exception (**PlayerException**), **try**-**catch** block and also **throw**. The **try-catch** block is used in the main method of class **Aims.java** and in the **play()** method of the **CompactDisc.java**. Print all information of the exception object, e.g. **getMessage()**, **toString()**, **printStackTrace()**, display a dialog box to the user with the content of the exception.

The example of codes and results for the **play**() of **DigitalVideoDisc** in **Swing** are illustrated in the following figure.



Figure 49.Swing dialog showing error

# Modify the **equals()** method of **Media** class

- Two medias are equal if they have the same **title**

- Please remember to check for **NullPointerException** and **ClassCastException** if applicable.

You may use **instanceof** operator to check if an object is an instance of a **ClassType**.

# Reading Document

Please read the following links for better understanding.

* Exception-handling basics:

<https://developer.ibm.com/tutorials/j-perry-exceptions/>

* Basic guidelines: Although the examples are in C++, the ideas are important.

<https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2019#basic-guidelines>

# Update Aims class diagram

Make an exception hierarchical tree for all self-defined exceptions in Aims Project. Use class diagram in Astah to draw this tree, export it as a png file, and save them in design directory.