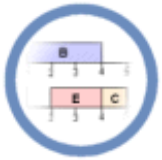




Lecture ”8”
Graphical User Interface (Android, iOS)

<lecturer, date>

Outline

- Graphical User Interface
  - **Overview**
  - Android UI
  - Android Layouts
  - iOS: Designing a UI
  - iOS: Defining the Interaction
  - References

Overview

- Conventional phones: Indirect keypad operations
- Smartphones: Direct operations on a touchscreen

Overview

- Your App’s User Interface (UI): Every thing user can see/interact with
- User needs to interact with app’s interface in the simplest possible way
  - ☞ While design interface have user in mind
  - ☞ Make it efficient, clear and straightforward

Outline

- Graphical User Interface
  - Overview
  - **Android UI**
  - Android Layouts
  - iOS: Designing a UI
  - iOS: Defining the Interaction
  - References

Android UI

- Android provides a variaty of pre-build UI components
  - Structured layout objects
  - UI controls
  - Other UI modules for special interfaces such as
    - Dialogs
    - Notifications
    - Menus
- To build GUI for your app

View & ViewGroup Objects

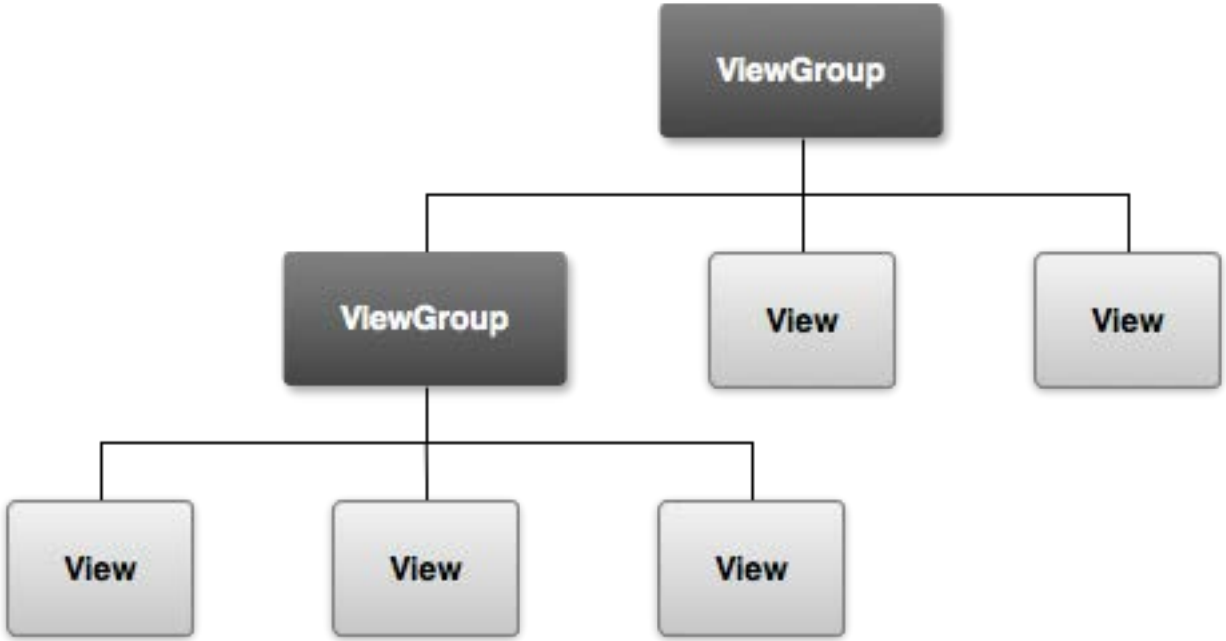
- UI elements in Andoid are built using
  - *View* : An object that drwas some thing on the screen user can interact with
  - *ViewGroup* : An object that holds other *View*/*ViewGroup* objects to define layout of interface
- Android provides a collection of View/ViewGroup subclasses that offer you
  - Common input controls e.g., buttuns, text fields
  - Various layout models e.g., linear/relative layout

View & ViewGroup Objects

- The UI for each component of your app is defined using a hierarchy of *View* and *ViewGroup* objects

- Each view group: An invisible container that organizes child views

  – Child view: Input controls, widgets that draw some parts of UI



Layout Declaration

- To declare your layout:
  - You can instantiate View objects in code and start building a tree
  - However, the easiest and most effective way is with an XML file
    - XML offers a human-readable structure for the layout, similar to HTML

UI Components

- Don’t need to buid all UIs using *View*/*ViewGroup*
- Android provides several app components offering standard UI layout
- Each UI component has a set of unique APIs e.g., Action Bar, Dialogs, Status Notifications

Outline

- Graphical User Interface
  - Overview
  - Android
  - **Android Layouts**
  - iOS: Designing a UI
  - iOS: Defining the Interaction
  - References

Android Layouts

- Layout defines the visual structure of a user interface e.g., the UI for an *actvity/app widget*

- Can be declared in two ways:

1. Declare UI elements in XML:

    - Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts

2. Instantiate UI elements at runtime:

    - Your app can create *View* and *ViewGroup* objects (and manipulate their properties) programmatically

Android Layouts

- The Android framework gives you the flexibility to use either or both of these methods for declaring and managing your app's UI
  - E.g., you could declare your app's default layouts in XML, including the screen elements that will appear in them and their properties. Then, you could add code in your app that would modify the state of the screen objects, including those declared in XML, at run time

The XML Pros

- Enables you to better separate the presentation of your app from the code that controls its behavior
- Your UI descriptions are external to your app code, i.e., you can modify or adapt it without having to modify your source code and recompile
  - e.g., you can create XML layouts for different screen orientations, different device screen sizes and different languages
- Makes it easier to visualize the structure of your UI, so it's easier to debug problems
- Here, we focus on decalring layout in XML. If you're interested in instantiating View objects at runtime, refer to the *ViewGroup* and *View* class references

Layout XML File

- Using Android XML vocabuary you can quickly design UI layouts and their screen elements similar to creating web pages in HTML
- Each layout file must contain:
  - One root element of type *View*/*ViewGroup* object
  - Then add additional layout objects/widgets as child elements to gradually build a View hierarchy that defines your layout
- After you've declared your layout in XML, save the file as *.xml* in your Android project's *res/layout/* directory, so it will corectly compile.
- More about the syntax for a layout XML file: *Layout Resources* document.

Load the XML File

- When you compile your app, each XML layout file is compiled to a *View* resource

- You should load your layout resource from your app code in your *Activity.onCreate()* callback implementation by:

  - Calling *setContentView()*

  - Passing it the reference to your layout resource in the form of *R.layout.layout_file_name*

Load the XML File

- E.g., load *main_layout.xml* for your Activity as follows:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

- More in Activities document

Attributes

Every *View*/*ViewGroup* object supports their own variaty of XML attributes

- Some attributes are specific to a View object e.g., *textView* supports *textSize* attribute
- Some attributes inherited from the root View object are common to all View objects e.g., *id* attribute
- Some attributes are considered *layout parameters* which descirbe certain layout orientations of the View object

ID

- An integer ID associated to any View object to uniquely identify the View in the tree
- To create views and reference them from the app
  - Define a view/widget in the layout file and assign it to a unique ID
  - Then create an instance of the view object and capture it from the layout
- Define ID for View objects is important when creating a *RelativeLayout*
  - ☞ In a *RelativeLayout* sibling views can define their layout relative to another sibling view which is referenced by the unique ID

Common Layouts

- Each subclass of the *ViewGroup* class provides a unique way to display the views you nest within it

- More common layout types that are built into the Android platform:

  1. Linear layout:

     - Organizes its children into a single horizontal or vertical row

     - Creates a scrollbar if the length of the window exceeds the length of the screen

Common Layouts

2. Relative layout:
   - Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent)

3. Web view: Displays web pages

Common Layouts

- Although you can nest one or more layouts within another layout to acheive your UI design, you should strive to keep your layout hierarchy as shallow as possible

- Your layout draws faster if it has fewer nested layouts

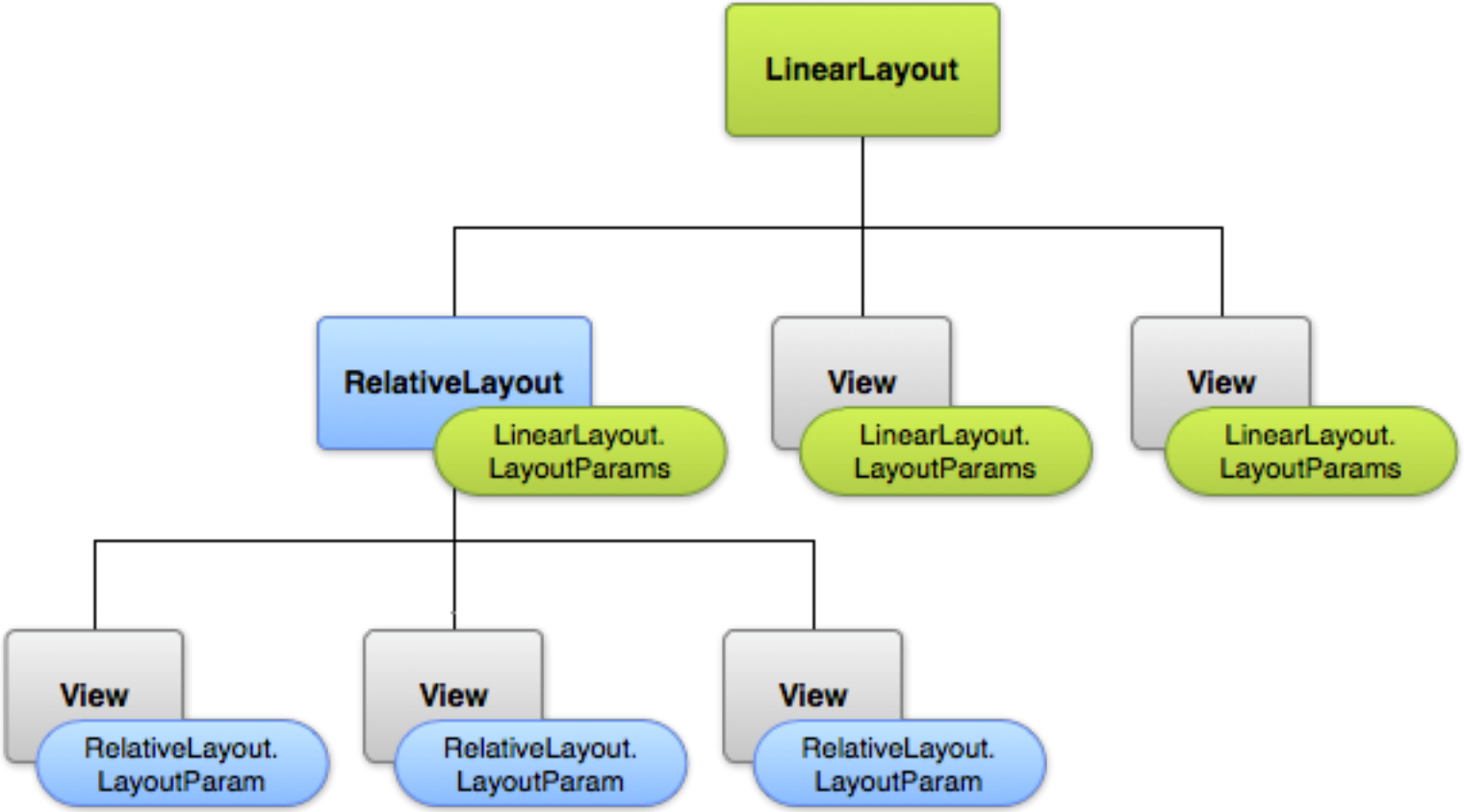
  ☞ A wide view hierarchy is better than a deep view hierarchy

Layout Parameters

- XML layout attributes named *layout_something* define layout parameters for the *View* that are appropriate for the *ViewGroup* in which it resides
- Every *ViewGroup* class implements a nested class that extends *ViewGroup.LayoutParams*
- This subclass contains property types that define the size and position for each child view, as appropriate for the view group

Layout Parameters

- Parent view group defines layout parameters for each child view (Figure below)



Outline

- Graphical User Interface
  - Overview
  - Android
  - Android Layouts
  - **iOS: Designing a UI**
  - iOS: Defining the Interaction
  - References

iOS: Designing a UI

- Let you design/implement your interface in a graphical environment
- See exactly what you are building
- Immediate feedback about what is/is not working
- Make instantly visible changes to your interface
- You are working with *Views*
  - ☞ *Views* display content to user
  - ☞ Building blocks for constructing your UI
  - ☞ Presenting your content in a clear, elegent and useful way

Views

- Display themselves on screen/react to user input
- Can serve as containers for other views

Views in an app are in a hierarchical structure called View Hierarchy

View Hierarchy

- The View Hierarchy
  - Within the hierarchy:
    - Subviews: Views enclosed within a view
    - Superviews: Parent view encloses a view

- *Window* object: Basic container into which you add your view objects for display onscreen

  ✕ By itself Window does not display any content

  ✓ To display content, add a *content view* object

UIKit Views

- When design your app it is important what kind of view to use for what purpose e.g., view to get user input (text field) different from view to display static text (label)
- *UIKit* views: Apps are easy to create as basic interface can be quickly assembled
- *UIKit* view object is an instance of *UIView* class or one of its subclasses
- UIKits view provides many types of views to present/organize data

UIKit Views General Categories

View Category	Purpose	Examples
Content	Display a particular type of content, such as an image or text	Image view, label
Collections	Display collections or groups of views	Collection view, table view
Controls	Perform actions or display information	Button, slider, switch
Bars	Navigate, or perform actions	Toolbar, navigation bar, tab bar
Input	Receive user input text	Search bar, text view
Containers	Serve as containers for other views	View, scroll view
Modal	Interrupt the regular flow of the app to allow a user to perform an action	Action sheet, alert view

UIKit Views

- Use *Interface Builder* to assemble views graphically
  - Provides a library of standard views/controls and other objects you need to build your interface
- After dragging these objects from the library, you drop them onto the canvas and arrange them in any way you want
- Then, use inspectors to configure those objects before saving them in a storyboard. You see the results immediately, without the need to write code, build, and run your app.

Custom Views

- The UIKit framework provides standard views for presenting many types of content
- However, you can also define your own custom views by subclassing *UIView* (or its descendants).
- A *custom* view is a subclass of UIView in which you handle all of the drawing and event-handling tasks yourself.
- More about implementing a custom view in Defining a Custom View.

Storyboards & Inspectors

- Use *Storyboards* to Lay Out Views
  - To lay out your hierarchy of views in a graphical environment
  - To provide a direct, visual way to work with views and build your interface
- Use *Inspectors* to Configure Views
  - Each inspector provides important configuration options for elements in your interface
  - For an object e.g., view in your storyboard, you can use inspectors t customize different properties of that object

Auto Layout

- Use *Auto Layout* to Position Views
  - When you start positioning views in your storyboard, you need to consider a variety of situations
  - iOS apps run on a number of different devices, with various screen sizes, orientations, and languages
  - You need a dynamic interface that responds seamlessly to changes in screen size, device orientation, localization, and metrics

Outline

- Graphical User Interface
  - Overview
  - Android
  - Android Layouts
  - iOS: Designing a UI
  - **iOS: Defining the Interaction**
  - References

iOS: Defining the Interaction

iOS apps are based on **event-driven programming**

- The flow of app is determined by events
  - System events
  - User actions
- User performs actions on interface ➔ Trigger events in the app
- The events result in execution of the app's logic/manipulation of its data
- App's response to user action is reflected back in the interface
- You define much of your event-handling logic in view controllers

iOS: Defining the Interaction

- Controller are objects that support your views by responding to user actions and populating the views with content.

- Controller objects are a conduit through which views interact with the data model.

- Views are notified of changes in model data through the app's controllers, and controllers communicate user-initiated changes, e.g., text entered in a text field, to model objects.

- Controllers implement your app's behavior whether they are responding to user actions or defining navigation.

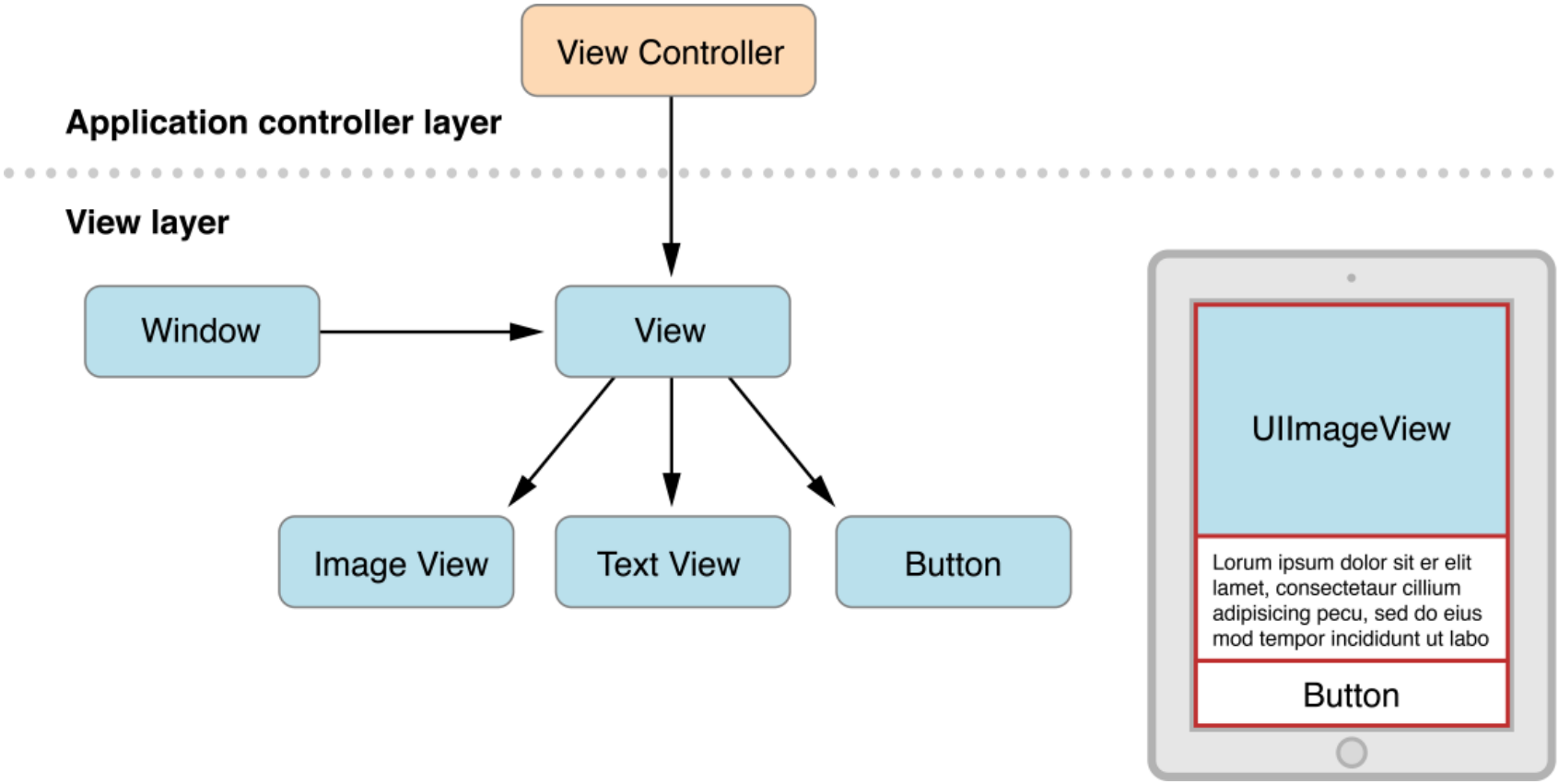
View Controllers

- After you built a basic view hierarchy, your next step is to control the visual elements/respond to user input
- In an iOS app, you use a view controller (*UIViewController*) to manage a content view with its hierarchy of subviews
- Not part of the view hierarchy
- Not an element in your interface
- Instead, it manages the view objects in the hierarchy and provides them with behavior
- Each content view hierarchy that you build in your storyboard needs a corresponding view controller, responsible for managing the interface elements and performing tasks in response to user interaction

View Controllers

- Coordinate the flow of information between the app’s data model and the views that display that data
- Manage the life cycle of their content views
- Handle orientation changes when the device is rotated
- Their most obvious role is to respond to user input
- You also use view controllers to implement transitions from one type of content to another

View Controllers



Navigation Controllers

- If your app has more than one content view hierarchy, you need to be able to transition between them
- You will use a specialized type of view controller called a navigation controller (*UINavigationController*)
- A navigation controller manages transitions backward and forward through a series of view controllers
- Navigation stack: The set of view controllers managed by a particular navigation controller
- Navigation controller is also responsible for presenting custom views of its own
- Use Storyboards to define navigation

Actions

- A piece of code that is linked to an event that can occur in your app
- When the event takes place, the code gets executed
- To accomplish anything e.g., manipulating a piece of data/updating the user interface
- To drive the flow of your app in response to user/system events

Outlets

- Provide a way to reference interface objects - the objects you added to your storyboard - from source code files

- To create an outlet, Control-drag from a particular object in your storyboard to a view controller file. This operation creates:

  - ✓ A property for the object in your view controller file

  - ✓ And lets you access and manipulate that object from code at runtime.

- Outlets are defined as *IBOutlet* properties:

  - – *@property (weak, nonatomic) IBOutlet UITextField *textField;*

  - – The *IBOutlet* keyword tells Xcode that you can connect to this property from Interface Builder.

- More about how to connect an outlet from a storyboard to source code in Tutorial: Storyboards.

Controls

- A user interface object such as a button/slider/switch that users manipulate to interact with content, provide input, navigate within an app, and perform other actions that you define
- Controls enable your code to receive messages from the user interface
- When a user interacts with a control, a control event is created
- A control event represents various physical gestures that users can make on controls e.g., lifting a finger from a control, dragging a finger onto a control, and touching down within a text field

Controls

- Three categories of control events

1. Touch and drag events: User interacts with a control with a touch/drag
2. Editing events: User edits a text field
3. Value-changed events: User maipulates a control

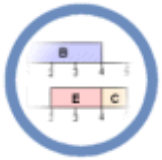
Outline

- Graphical User Interface
  - Overview
  - Android
  - Android Layouts
  - iOS: Designing a UI
  - iOS: Defining the Interaction
  - **References**

References

- Android UI: http://developer.android.com/guide/topics/ui/index.html
- Android Layouts: http://developer.android.com/guide/topics/ui/declaring-layout.html
- iOS UI:
  https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/DesigningaUserInterface.html
- Apple Tutorial, Storyboards:
  https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/SecondTutorial.html#//apple_ref/doc/uid/TP40011343-CH8-SW1





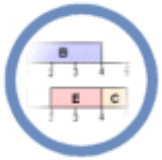
Lab ”8”
User Interface

<lecturer, date>

Lab ”8”

- Use your Lab ”7” layout and make it such that it responds when the button is pressed by sending the content of the text field to another activity.





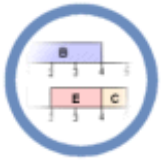
Seminar ”8”
User Interface

<lecturer, date>

Seminar ”8”

- In Lab ”8”, you made an app that shows an activity (a single screen) with a text field and a button. Discuss how it can start a new activity when the user clicks the *Send* button.





Mini-Project ”8”
User Interface

<lecturer, date>

Mini-Project ”8”

- Discuss the steps to build a dynamic UI with Android fragments