



الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 2  
Beginning Android and Java &  
First Contact Java, XML, and the UI designer



# Contents

1. Why Java and Android?
2. How Java and Android work together?
3. Understanding the Android API
4. Setting up Android Studio
5. Building our first Android app
6. Exploring the project Java and the main layout XML
7. Examining the app layout file
8. Adding buttons to the main layout file
9. Coding messages to the user and the developer



# Weekly Learning Outcomes

1. Explain the basic concept of Android & Java.
2. Demonstrate the steps is needed to setup Android Studio.
3. Explain how to build first Android app.



## Required Reading

1. Chapter 1 and Chapter 2 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



# Why Java and Android?



## Why Java and Android?

- Java the popular programming language is used to develop Android applications.
- Java has an extensive set of libraries. It is easy to take advantage of these libraries. Android SDK has many standard Java libraries included. These provide functionalities for data structure, math functions, graphics implantation, and networking functions and much more.
- JAVA help in creating, improving android applications with many libraries and tools of java make Android application developing easier.
- Java helps develop Android applications fast and inefficient manner.
- Android easily implement and fix common problems with other programming languages with the help of Java.

# How Java and Android work together?



## How Java and Android work together- ART system

- JAVA code for android is converted into **bytecode** after **compilation**.
- This bytecode is then converted into machine code by **Android Runtime(ART)** when the application is installed.
- Java is fast for the programmer to program → conversion to machine code is fast for the device.
- ART system : provides hooks into application
- Enhance memory management while the application is running → app run more efficiently.

# Understanding the Android API



# Understanding the Android API

- Android framework API → use by applications to interact with the underlying Android system.
- Android API → Collection of Java code that has already been written to use it as plug and play.
- Java programming language → enable to handle programmer's complexity.
- An API is a software intermediary that allows two applications to talk to each other.
- API is the messenger that delivers the request to the provider delivers the response back.
- APIs enable developers → enable to create complex processes highly reusable with a little bit of code.
- APIs play a vital role in helping developers build out apps faster and more efficiently.

# Java is object-oriented

- An object-oriented language → uses the concept of reusable programming objects.
- Java enables to write java code that can be structured based on real-world things,
- Class is the blueprint of an object in oops concept.
- When a class is transformed into a real working "thing," it is termed as an object or an instance of the class.

# What exactly is Android?

- Android programming uses JAVA code: compiled bytecode and translated into machine code by ART when installed
- Android is linux based operating system → handles the complex and extremely diverse range of hardware of variety of android devices.
- Advanced software called drivers provided by the device manufacturers ensure that the hardware (cpu, gpu, gps receivers, memory chips, hardware interfaces, and so on) can run on the underlying linux operating system.
- Android application package(apk) → bundle of files that consists of bytecode + supporting resources
- Art uses this bundle to execute and prepare app for the user.

# Setting up Android Studio

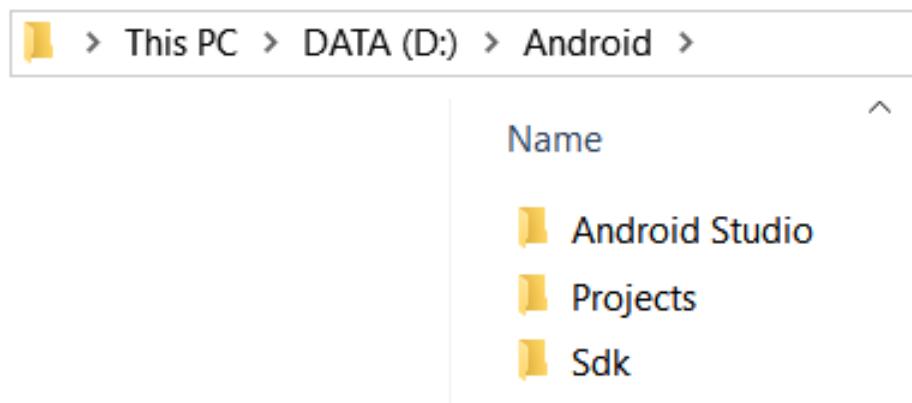


# Android Studio

- Official integrated development environment (IDE) for android app development.
- Two major components:
  - ✓ JDK : To use Java code
  - ✓ Android SDK : Collection of tools for Android development and Android API.
- Tasks by Android Studio
  - ✓ Provides an IDE
  - ✓ compiling code
  - ✓ Linking with the JDK and the Android API

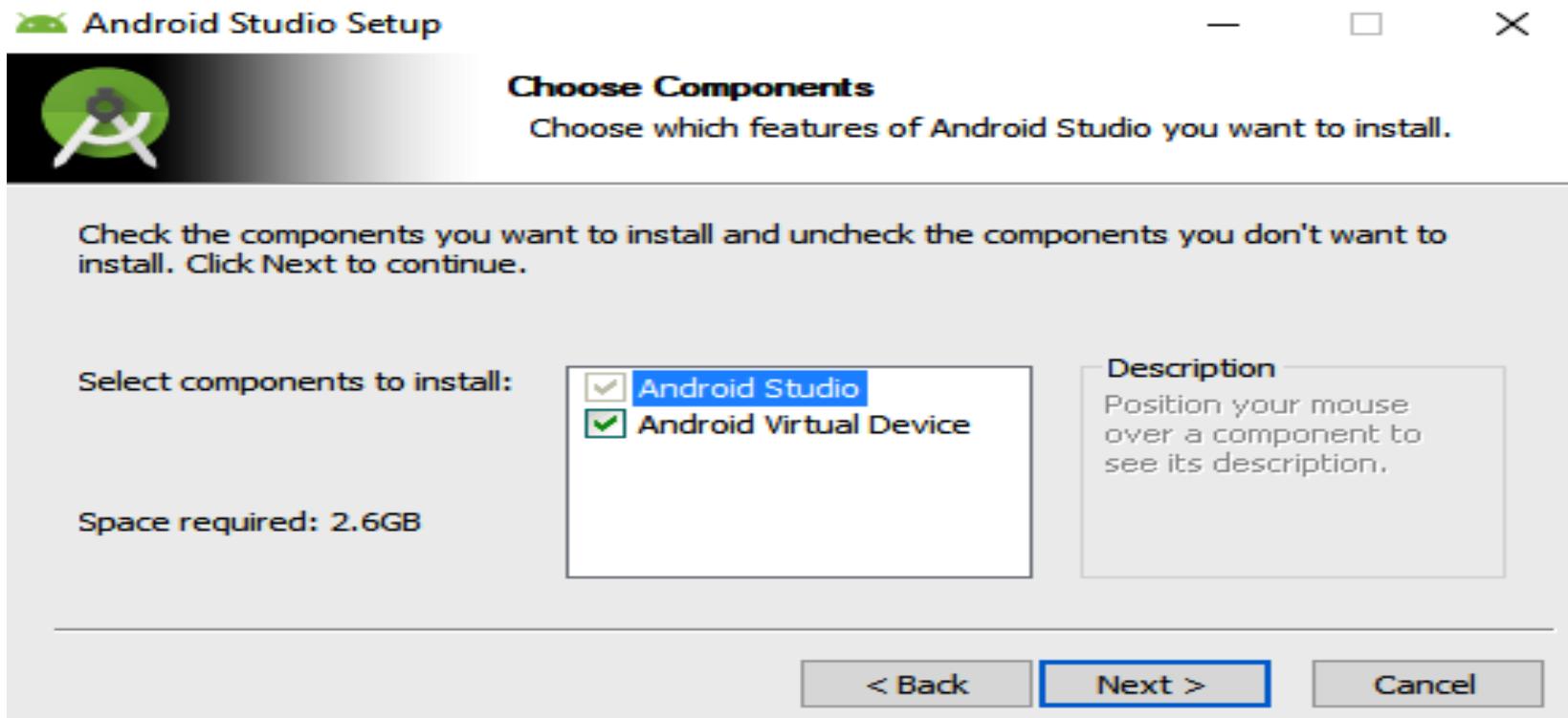
# Setting up Android Studio

1. Download Android Studio from <https://developer.android.com/studio>
2. Create a new folder on the root of C or D drive called Android. Inside the Android folder, create another new folder called Android Studio.
3. Navigate back to the Android folder and create the another new folder named Projects, to keep all the project files.
4. Create another new folder called Sdk, to installer program to install the Android SDK. Should now have a D:\Androidfolder that looks like this:



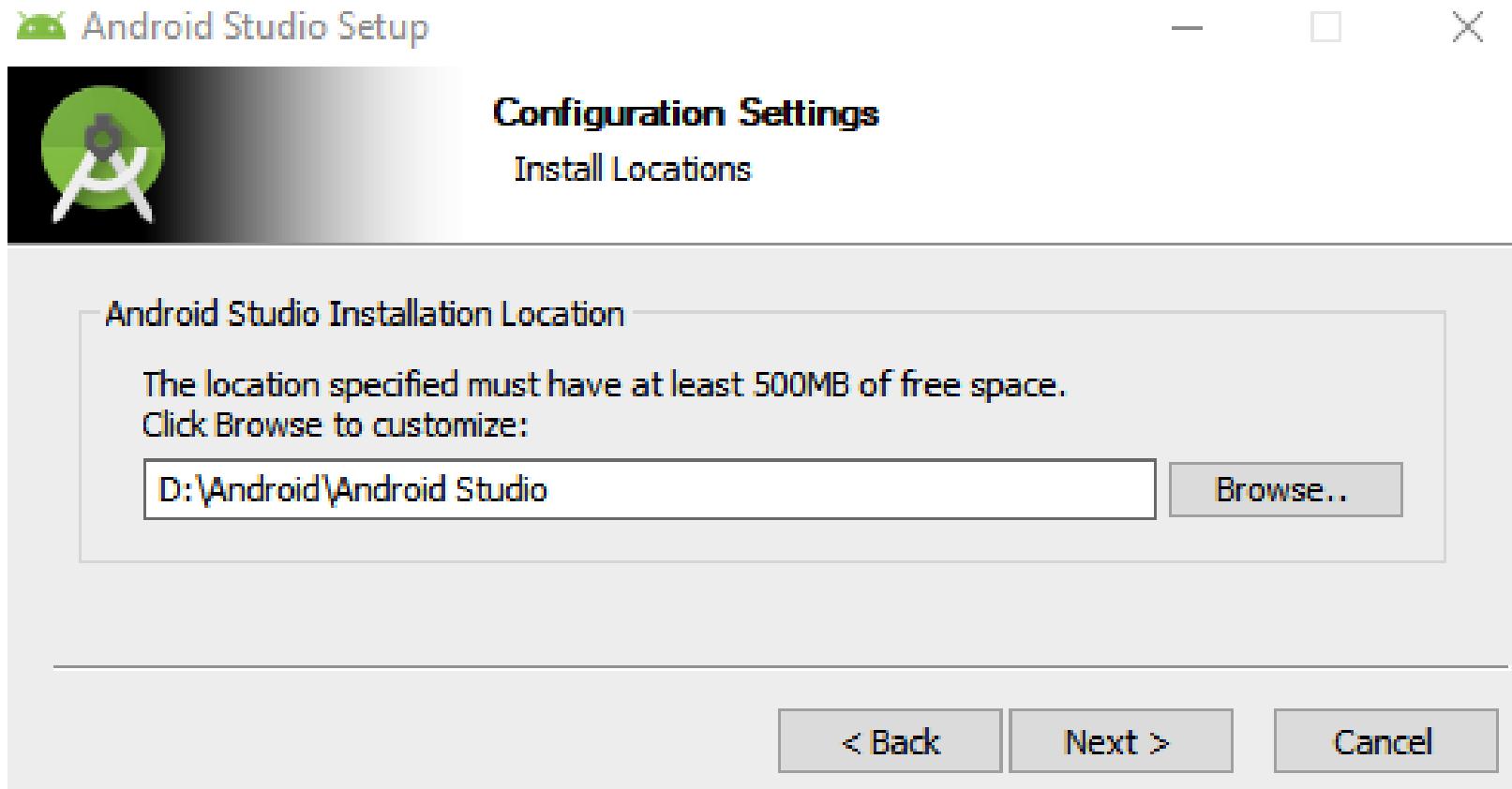
# Setting up Android Studio

3. Once the download is complete → downloaded file – android-studio-ide.
4. Double-left-click the file to run it.
5. Enable the administrative privileges and begin the installation. On the **Choose Components** screen, make sure that both the **Android Studio** and **Android Virtual Device** options are checked, and then left-click the **Next** button:



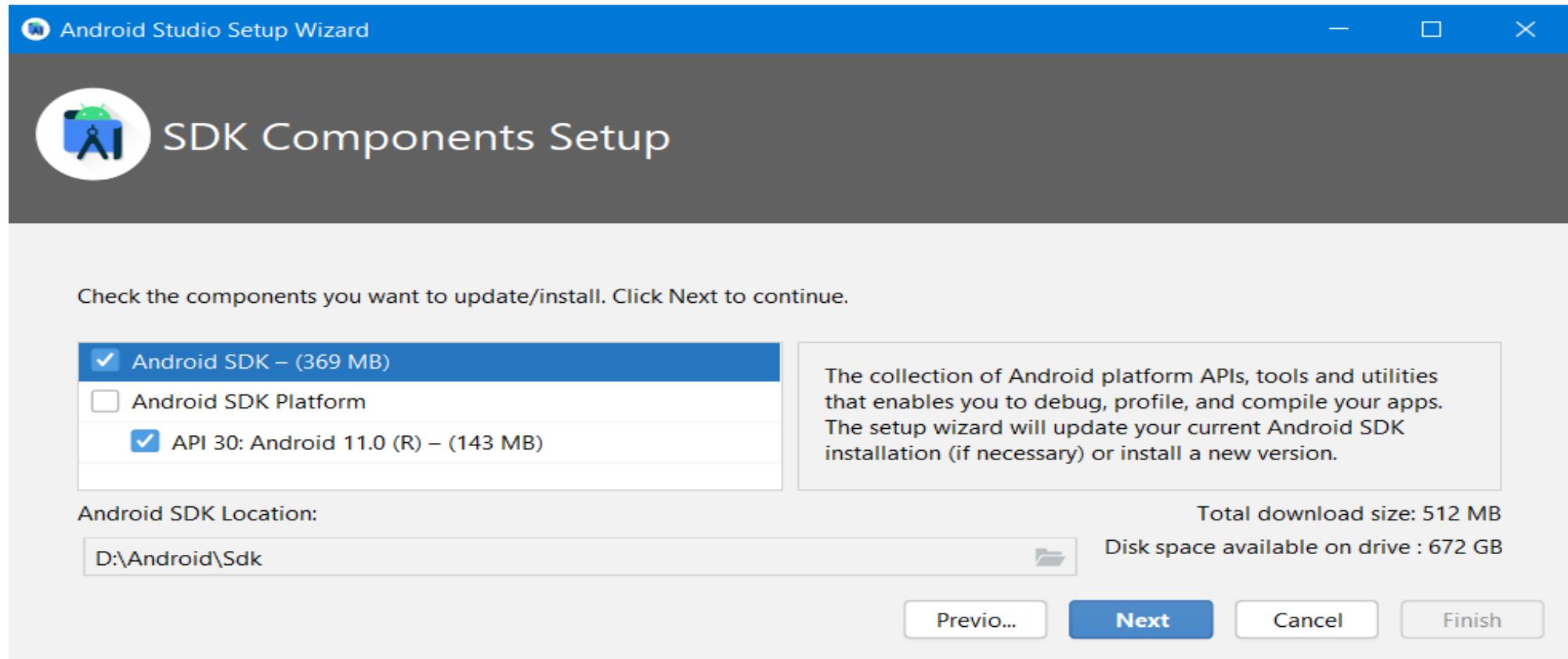
# Setting up Android Studio

5. On the Configuration Settings window, left-click the Browse button and navigate to D:\Android\Android Studio, and then left-click the OK button:



## Setting up Android Studio

6. Left-click the Next button shown in the preceding screenshot. On the Choose Start Menu Folder window, left-click Install to accept the default option. The first part of the installation will now proceed.
7. Once getting the Installation Complete message, left-click the Next button, then left-click the Finish button.
8. It will be prompted that a missing SDK (unless this is not the first time is used Android Studio). Left-click Next to continue.
9. On the SDK Components Setup screen shown next, change the install location. Left-click the Android SDK Location field and browse to D:\Android\ Sdk, as shown in the following screenshot:



10. SDK Components Setup screen Left-click the **Next** button.

11. On the **Verify Settings** window, left-click the **Finish** button. Android Studio will now download some more files and complete the installation. It could take a few minutes or more and will prompt to allow access to the PC.

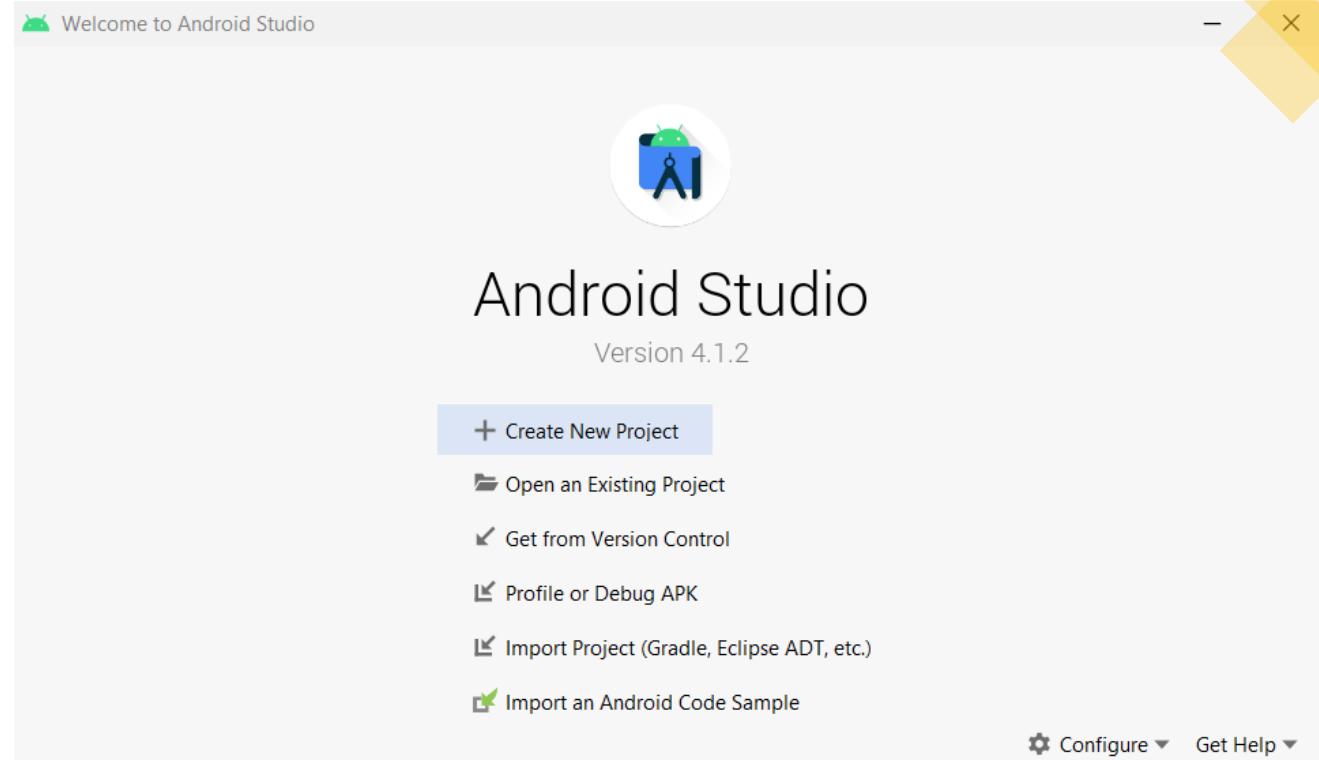
12. When the process is over, left-click the **Finish** button.

# Building our first Android app



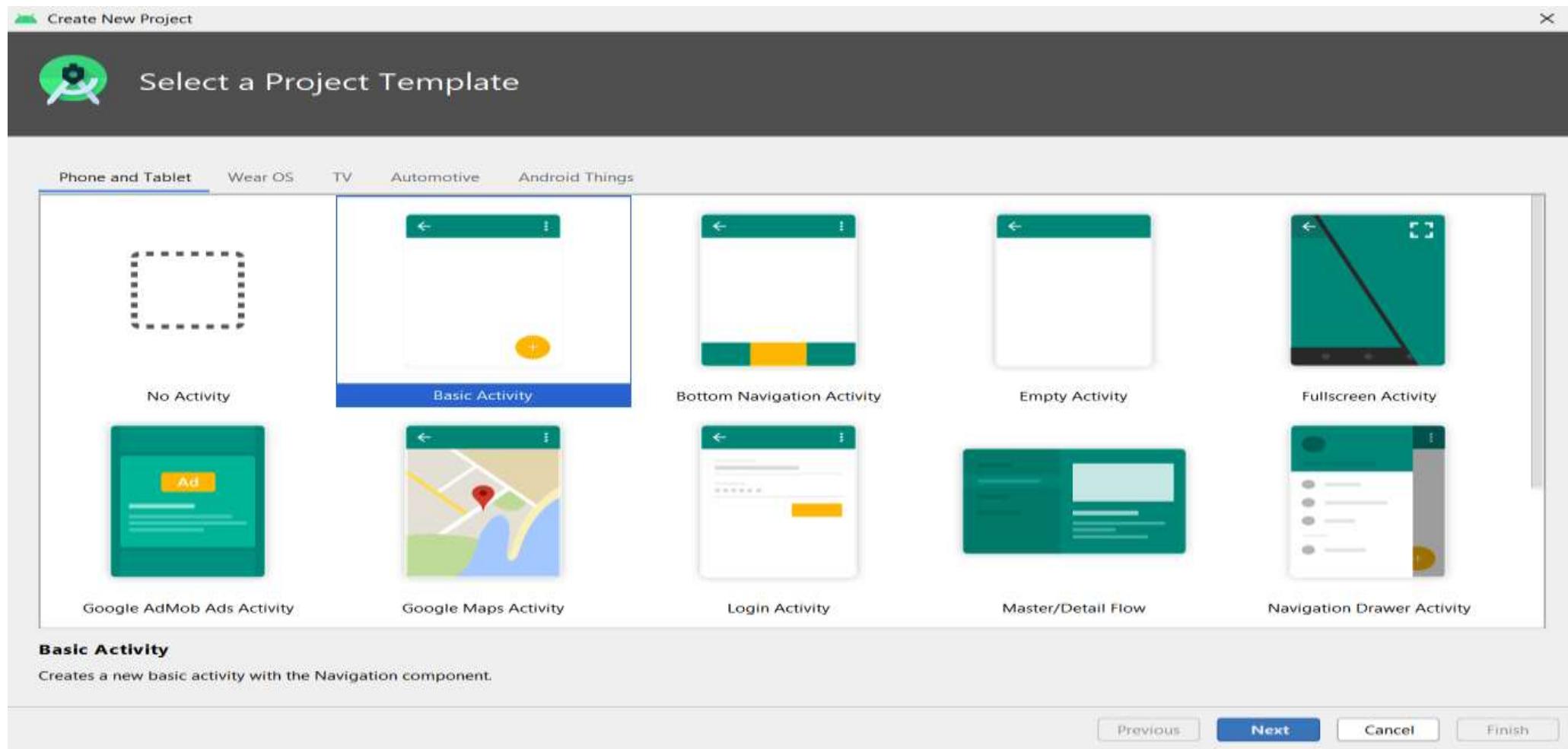
# Building our first Android app

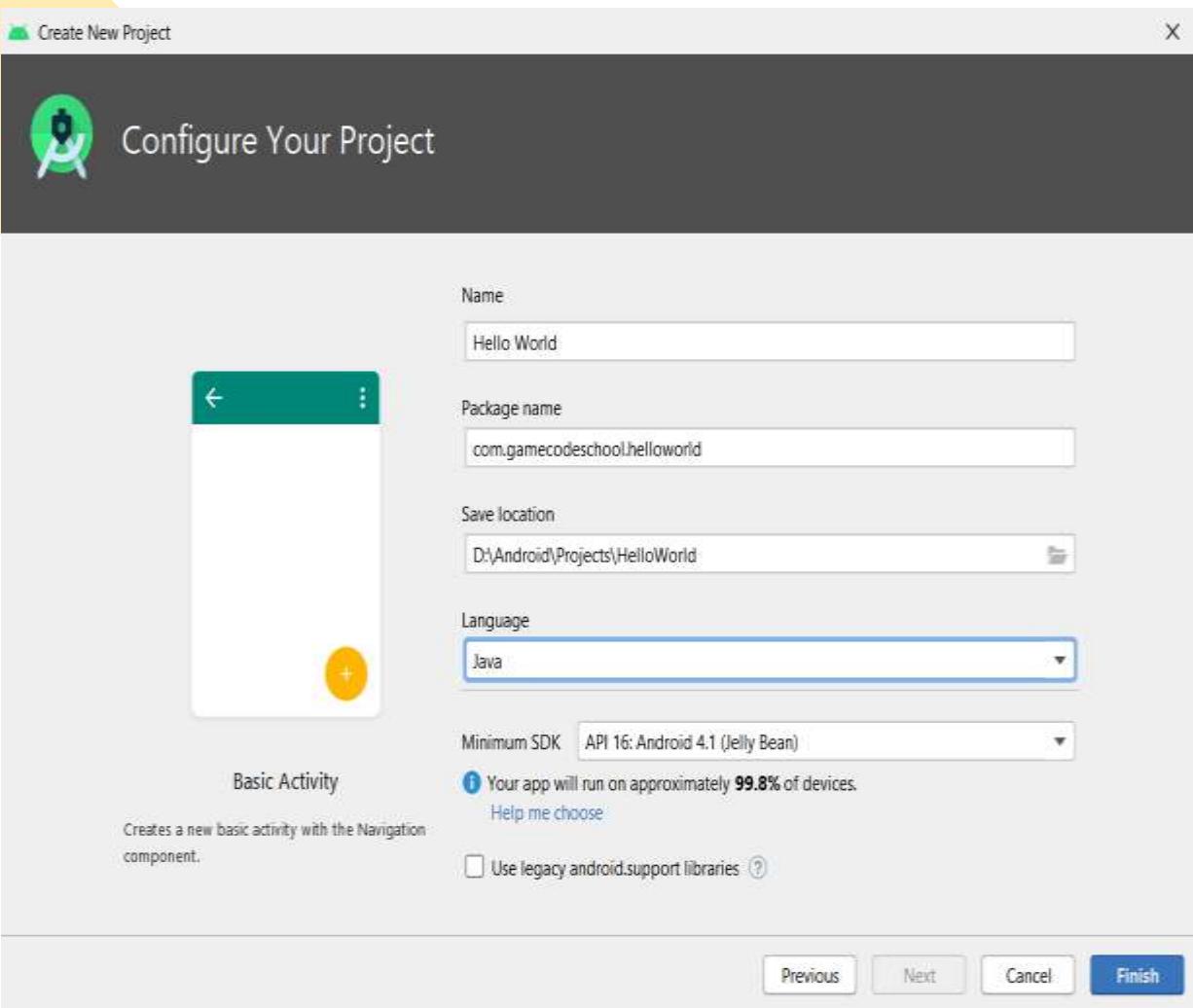
- Steps to start the project:
  1. Run Android Studio. On Windows 10, for example, the launch icon appears in the start menu.
  2. Android Studio welcome screen, as shown in the following screenshot.
  3. Locate the start a new Android Studio project option and left-click it:



3. The window that follows is **Select a Project Template**.

4. Select **Basic Activity**. Here is a picture of the **Select a Project Template** window with the **Basic Activity** option selected:





5. Ensure **Basic Activity** is selected as it is in the preceding screenshot, and then click **Next**.

6. After this, Android Studio will bring up the **Configure Your Project** window: Enter the details of the project here:

- Name the new project.
- Choose the location on the system to save the project files.
- Provide a package name to distinguish our project from any others in case we should ever decide to publish the app on the Play Store.
- Select the programming language we will use.

7. The next screenshot shows the **Configure Your Project** screen once you have entered all the information:

# Exploring the project Java and the main layout XML



# Exploring the project Java and the main layout XML

- Examining the **MainActivity.java** file
- Java code can be viewed by left-clicking on the **MainActivity.java** tab, as shown in the figure below:



Figure 2.2 – MainActivity.java tab

Only an annotated screenshot is required than reproducing the actual code in text form.

9

```
package com.gamecodeschool.helloworld; — 1  
import android.os.Bundle;  
  
import com.google.android.material.floatingactionbutton.FloatingActionButton;  
import com.google.android.material.snackbar.Snackbar;  
  
import androidx.appcompat.app.AppCompatActivity;  
import androidx.appcompat.widget.Toolbar;  
  
import android.view.View;  
import android.view.Menu;  
import android.view.MenuItem;  
  
public class MainActivity extends AppCompatActivity { — 3  
    @Override  
    protected void onCreate(Bundle savedInstanceState) { — 5  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
        setSupportActionBar(toolbar);  
  
        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);  
        fab.setOnClickListener(view) -> {  
            Snackbar.make(view, text: "Replace with your own action", Snackbar.LENGTH_LONG)  
                .setAction(text: "Action", listener: null).show();  
    }; — 6  
    @Override  
    public boolean onCreateOptionsMenu(Menu menu) {...} — 7  
    @Override  
    public boolean onOptionsItemSelected(MenuItem item) {...} — 8  
}
```

4

Resource files

1

2

3

5

7

8

Figure 2.3 – Java code

## The package declaration

- Part **1** is called the **package declaration** and, it is the package name we chose when we created the project preceded by the word package. Every Java file will have a package declaration at the top.

## Importing classes

- Part **2** is eight lines of code that all begin with the word import. After the word import, there are various dot-separated words. The last word of each line is the name of the class that line imports into the project and all the earlier words in each line are the packages and sub-packages that contain these classes.
- For example, this next line imports the AppCompatActivity class from the androidx.appcompat.app package and sub-packages:

```
import androidx.appcompat.app.AppCompatActivity;
```

## The class

- Part **3** of our code is called the **class declaration**.
- public class **MainActivity** extends AppCompatActivity {

- Finally, for part **3**, look at the opening curly brace at the end of the line: {. Now look at the bottom of the figure at part **4** of our code. This closing curly brace } denotes the end of the class. Everything in between the opening and closing curly braces, {...}, is part of the MainActivity class.

## Methods inside the class

- Part **5** of the code: key part for code:

```
protected void onCreate(Bundle savedInstanceState) {
```

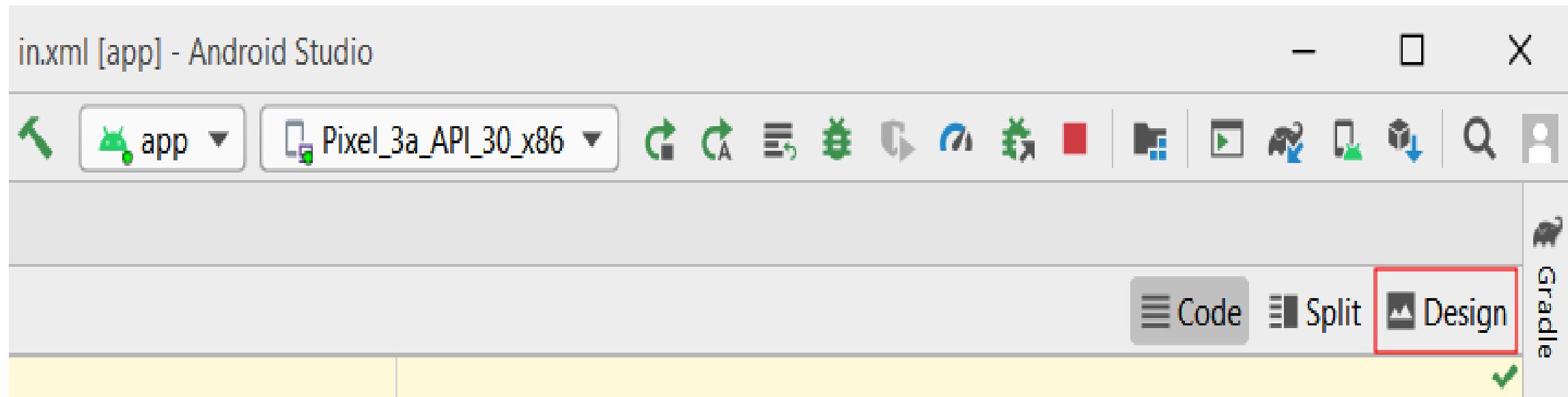
- Now jump to part **6** of code and will see a closing curly brace, }. Everything in between the opening and closing curly braces of the onCreate method is the code that executes when the method is called.
- Parts **7** and **8** are also methods named as onCreateOptionsMenu and onOptionsItemSelected.

## Examining the app layout file

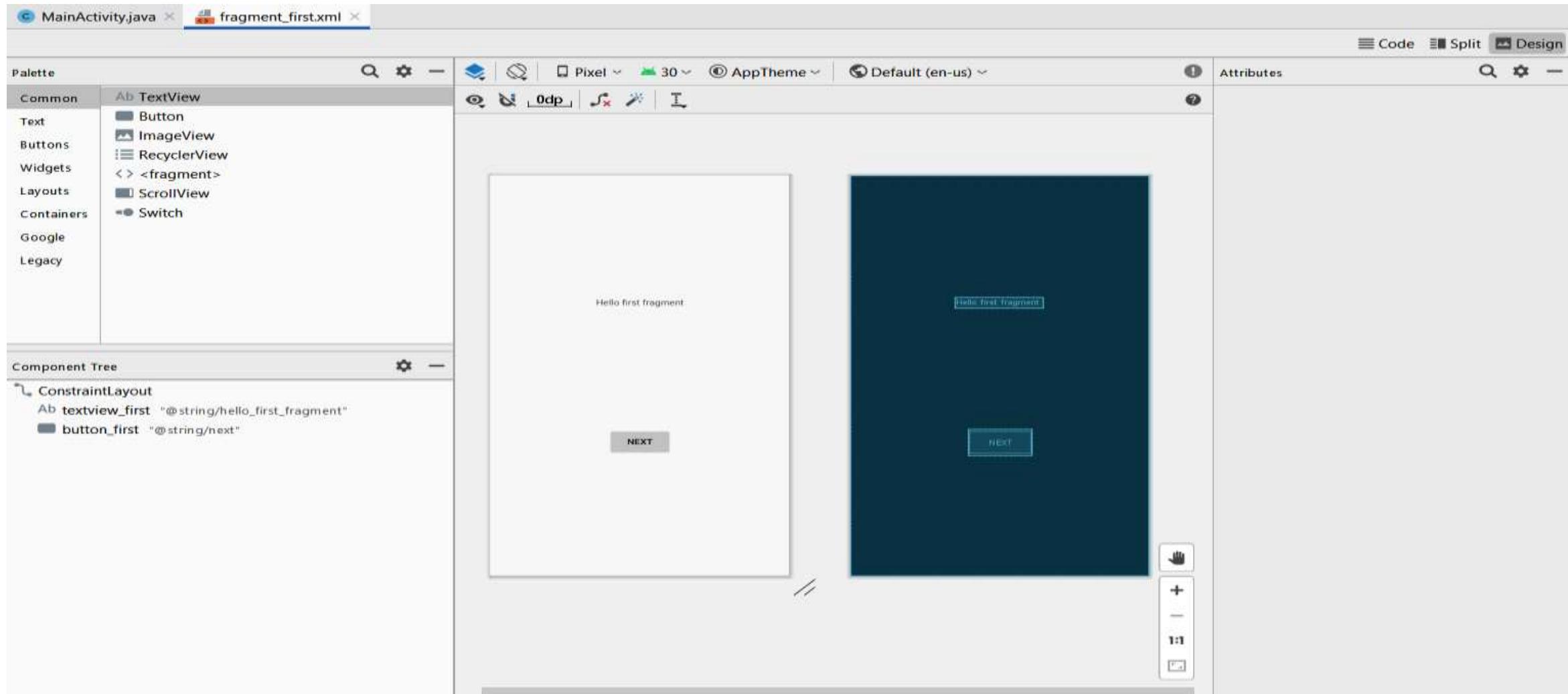


## Examining the app layout file

- In the project explorer window, left-click on the **res** folder and then left-click on the **layout** folder. Now double left-click on the fragment\_first.xml file. The XML code contents of the file is now displayed in the main window of Android Studio.
- Left-click the **Design** button (shown next) to switch to the design view:



- Design view that shows the preview of execution of the XML code when the app is run in the emulator:



The design view is a graphical representation of the XML code contained in the fragment\_first.xml file. Click on the **Code** tab (near the **Design** tab in the previous figure) to see the XML code which forms the layout.

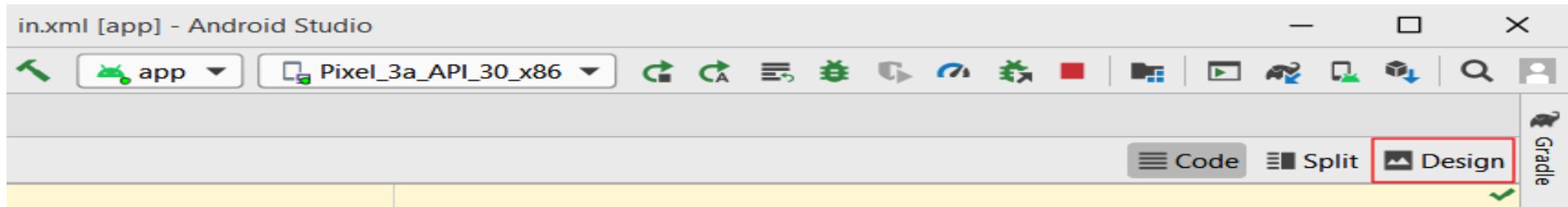
```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    tools:context=".FirstFragment"> — 1b  
  
    <TextView — 3  
        android:id="@+id/textview_first"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Hello first fragment"  
        app:layout_constraintBottom_toTopOf="@+id/button_first"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintTop_toTopOf="parent" /> — 4  
  
    <Button — 5  
        android:id="@+id/button_first"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Next"  
        app:layout_constraintBottom_toBottomOf="parent"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintTop_toBottomOf="@+id/textview_first" /> — 6  
    </androidx.constraintlayout.widget.ConstraintLayout> — 2
```

## Adding buttons to the main layout file

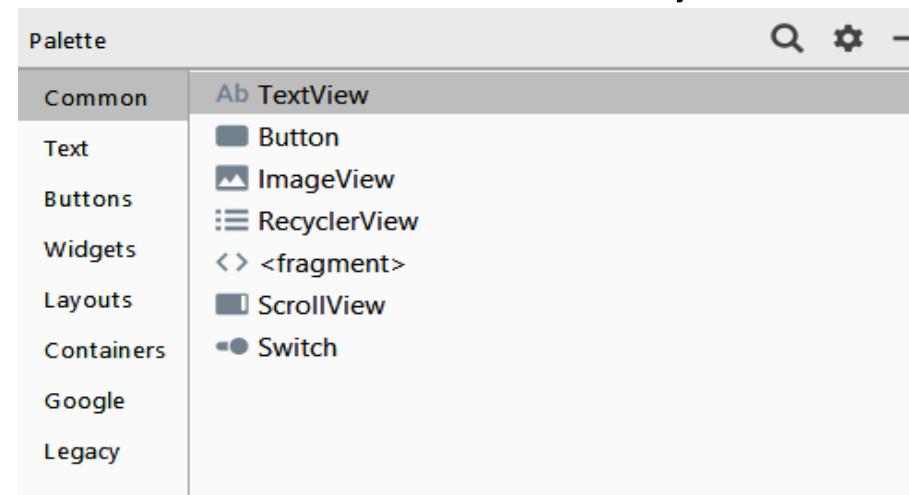


# Adding buttons to the main layout file

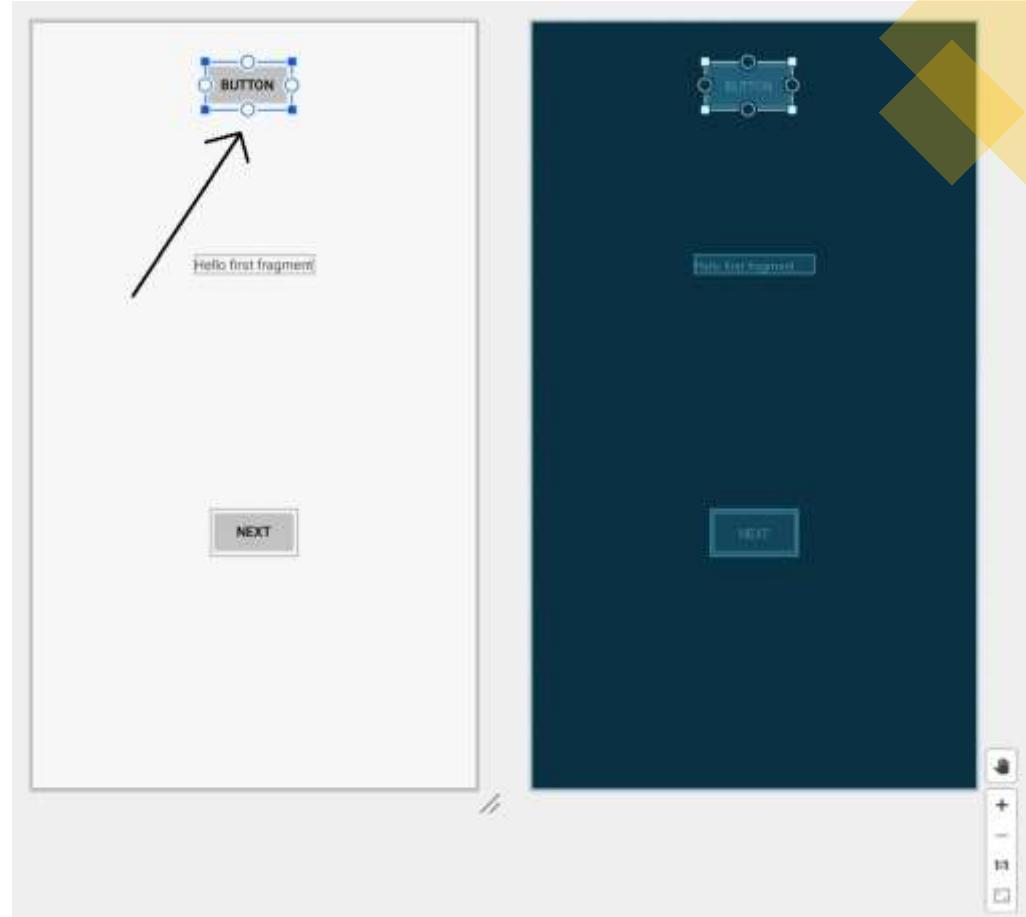
- Button can be added in two different ways: using the visual designer and by editing the XML code directly.
- **Adding a button via the visual designer**
- Open fragment\_first.xml in the editor and switch back to the design view by clicking the **Design** tab



Palette window is opened on the left-hand side of the layout as shown below.



- The palette is divided into two parts: Categories of UI elements and available UI elements in the currently selected category.
- Select the Common category. Now, left-click and hold on the Button widget and then drag it onto the layout somewhere near the top center.
- if the position of the button need to be changed, then left-click it to select it on the layout and then tap the *Delete* key on the keyboard to get rid of it.
- Now repeat the previous step until a new button is neatly placed



Other operations to the main layout file :

- Editing the button's attributes
- Examining the XML code for the new button
- Adding a button by editing the XML code
- Giving the buttons unique id attributes
- Positioning the two buttons in the layout
- Making the buttons call different methods

Coding messages to the user and the developer



- Two different classes from the Android API are used to output
- **Log**: allows to output messages to the Logcat window.
- **Toast**: produce a toast-shaped pop-up message for app's user to see.
- code to send a message to the Logcat window:

```
Log.i("info","our message here");
```

- code to send a message to the user's screen:

```
Toast.makeText(this,"our message",
```

```
Toast.LENGTH_SHORT).show();
```

# Writing first Java code

- Switch to the MainActivity.java tab in Android Studio.
- onCreate() method is called just before the app starts.
- Copy and paste some code into the onCreate() method
- **Adding message code to the onCreate method**
- Find the closing curly brace } of the onCreate method and add the highlighted code shown next.

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    ...  
    ...  
    // Your code goes here  
    Toast.makeText(this, "Can you see me?",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "Done creating the app");  
}
```

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 3  
Exploring Android Studio and the Project Structure



# Contents

1. Project Explorer and project anatomy.
2. Exploring the file and folder structure of the Empty Activity project template.
3. Exploring the file and folder structure of the Basic Activity project template.
4. Exploring the Android emulator.



# Weekly Learning Outcomes

1. Demonstrate the Project Explorer and project anatomy.
2. Explain the Android emulator.



## Required Reading

1. Chapter 3 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



Project Explorer and project anatomy.



# Project Explorer and project anatomy

- Create new Android project – using a project template
- Template is used to determine the exact selection and contents of files that Android Studio will generate.
- Build two project templates and examine the files and their contents,
- Create **Empty Activity** project – to analyse the concept of Linking two project templates together through the code (**Extensible Markup Language (XML)** and Java).
- **Empty Activity** project template: simplest project type with an autogenerated UI
- Android Studio autogenerated the Java code to display the UI.
- When it is added to the empty UI, it is ready to be displayed.

Exploring the file and folder structure of the Empty Activity project template.



➤ Select **Start a new Android Studio project**.

- ✓ On the **Select a Project Template** window, select **Empty Activity**.
- ✓ In the **Configure your project** screen, change the **Name** field to **Empty Activity App**.
- ✓ The rest of the settings can be left at their defaults, so just click **Finish**.

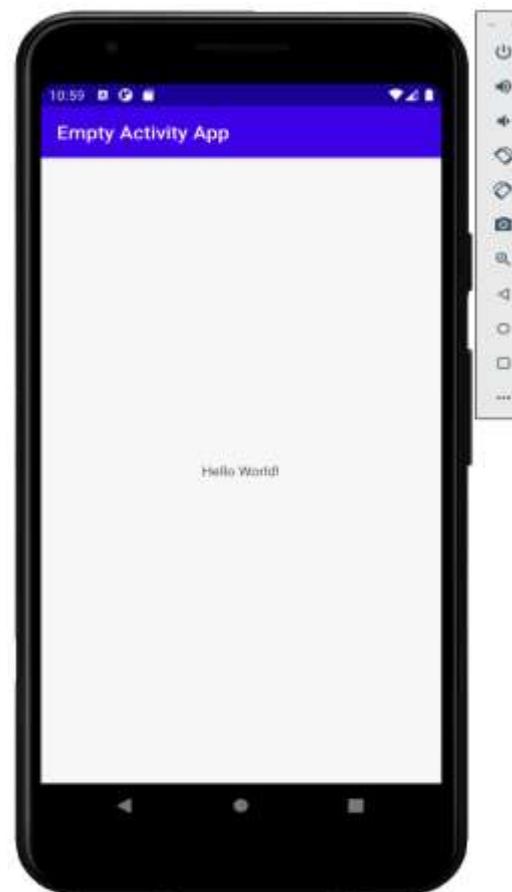
➤ Android Studio will generate all the code and the other project resources.

➤ Cross examine and verify if the generated details match with the contents that appear in the **Project Explorer** window.

➤ Launch the emulator it by selecting **Tools | AVD Manager** in the **Your Virtual Devices** window.

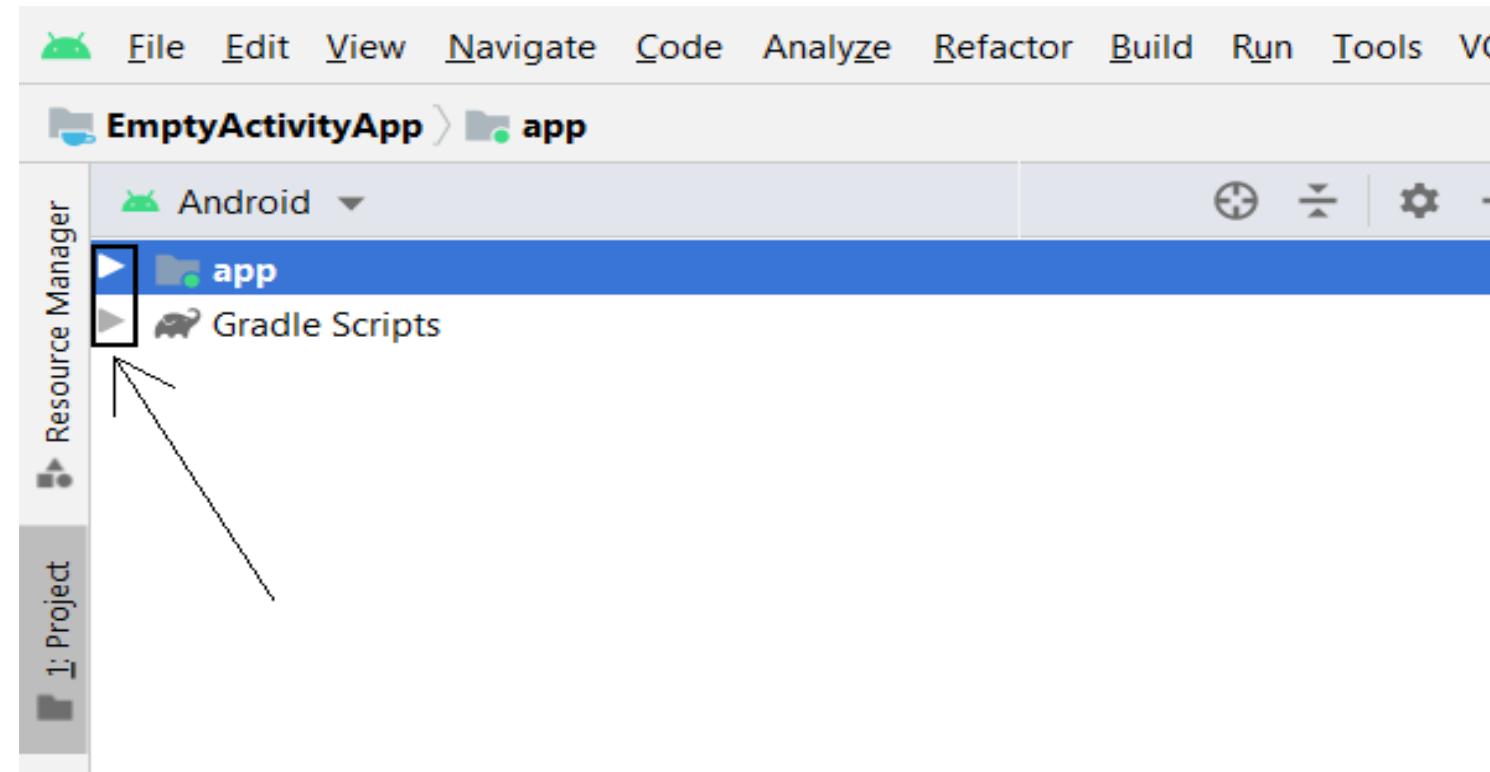
➤ Run the app on the emulator by clicking the play button in the quick launch bar

- The output appears as shown in the figure below: It is— well—empty → no menu at the top; no floating button at the bottom.
- Only **Hello World!**, text is displayed in the screen.

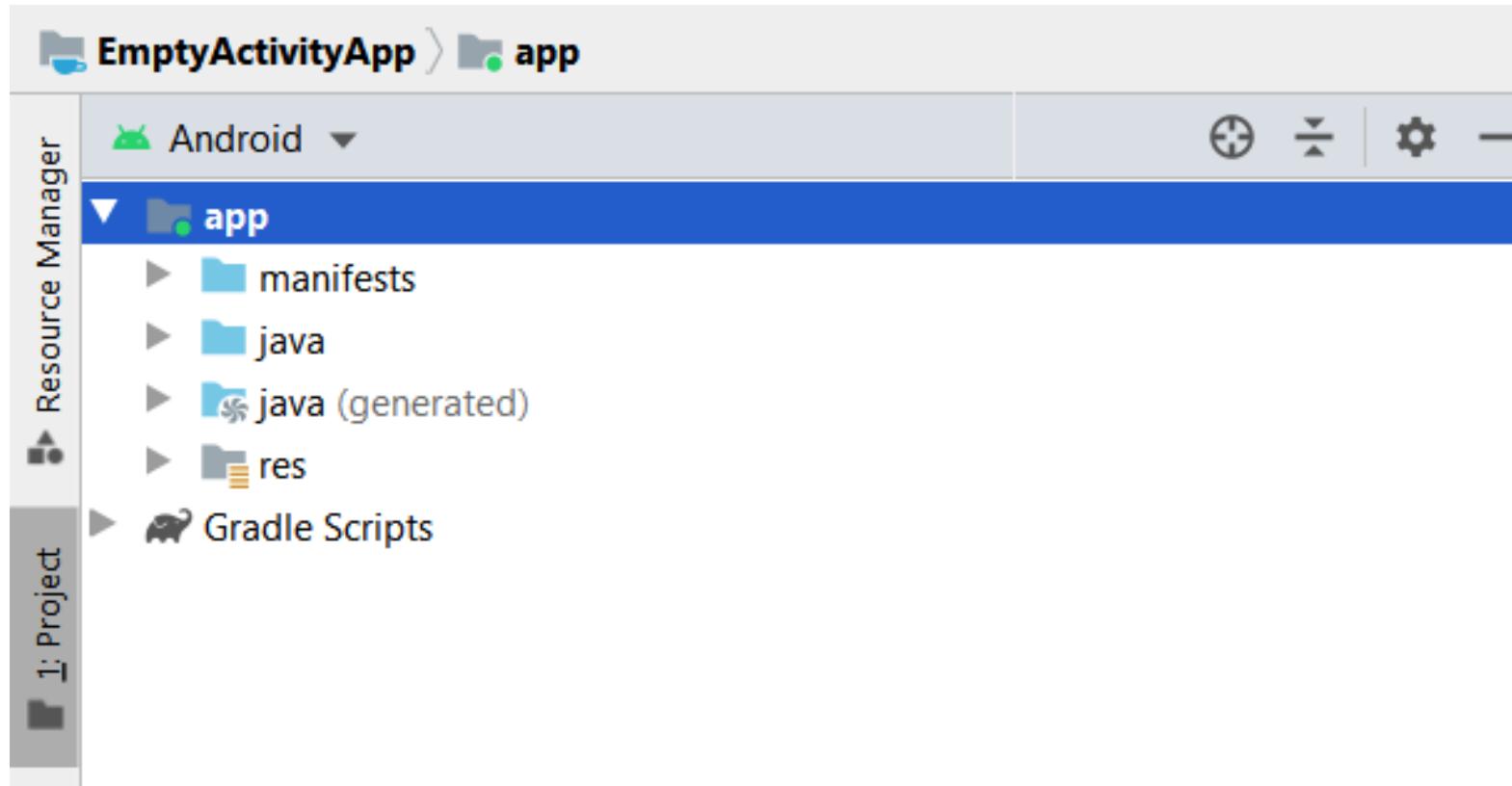


# Exploring an Empty Activity project

- Project Explorer window → after the project is created:
- two arrows → Enable to expand the app and Gradle Scripts folders.
- 



- Click the arrow next to the app folder to expand its contents.
- The first level of contents is shown in the following screenshot:



Has generated four more folders: manifests, java, java(generated), and res.

## Manifests folder

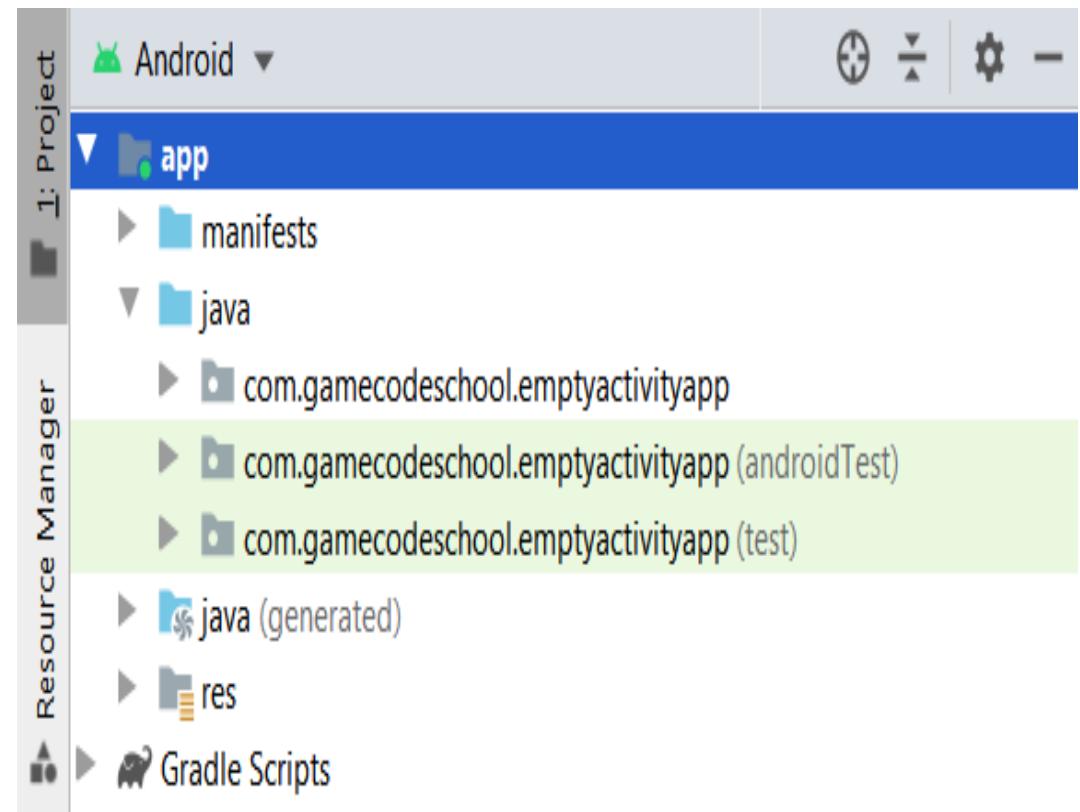
- Contains only one file inside it:  
**AndroidManifest.xml**

## Java folder

- Consists of only one file initially and three folders
- Gets populated with more files as the project is developed
- Expand the java folder and you will find three more folders

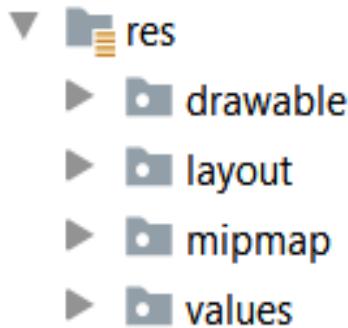
## Java (generated) folder

- This folder contains code generated by Android Studio.



## The res folder

- Consists of all the resources

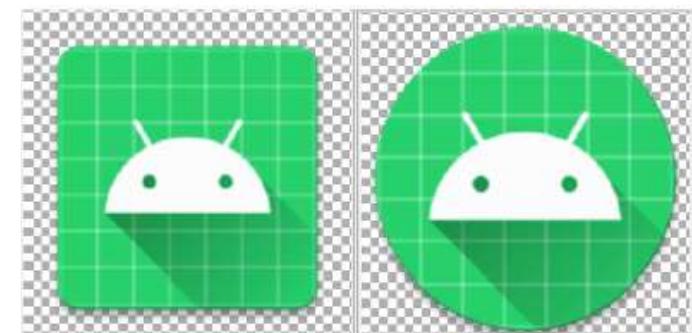
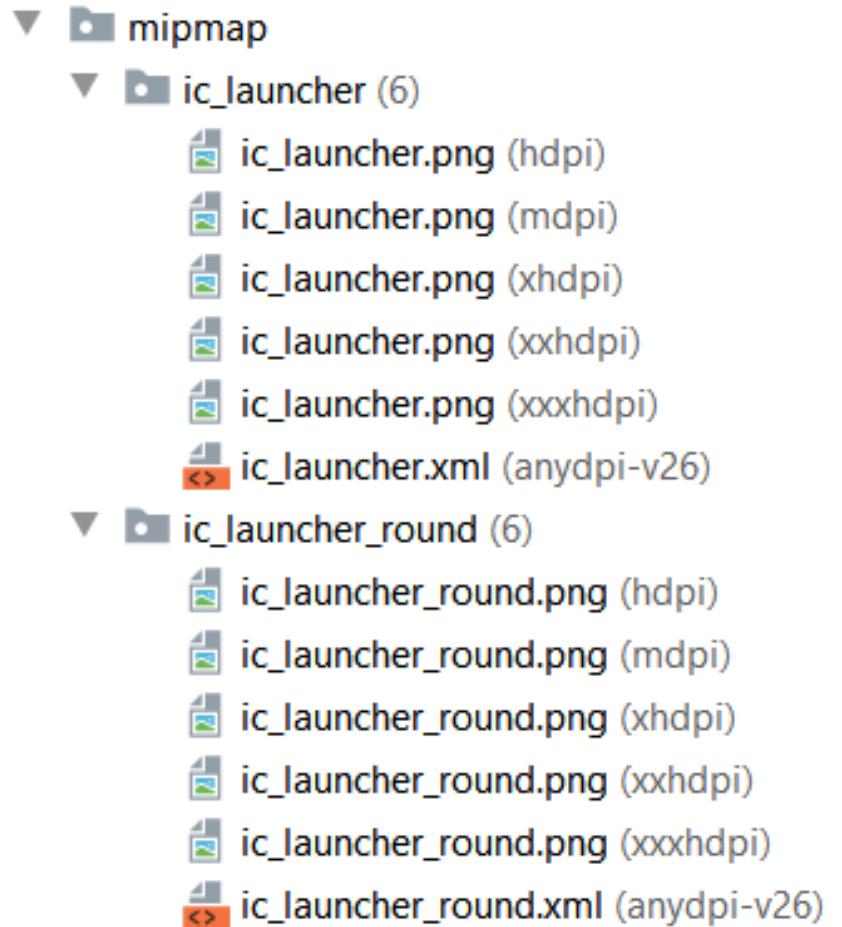


**res/drawable folder** : holds just two files(ic\_launcher\_foreground and ic\_launcher\_background) along with the graphics added to the project

**The res/layout folder** : Contains all the resources for app → icons, layouts (XML files), sounds, and strings.

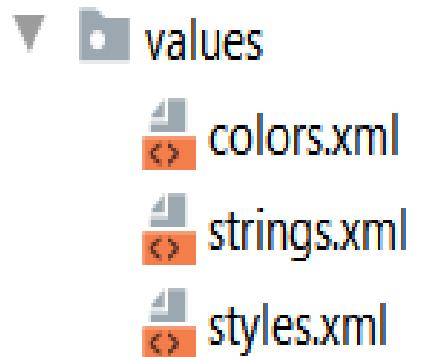
# The res/mipmap folder

- Consists of 2 subfolders: ic\_launcher and ic\_launcher\_round
- ic\_launcher → graphics for the app launcher icon
- ic\_launcher\_round → graphics for devices that use the round icons.



## The res/values folder:

- Consists of three files: colors.xml, string.xml, style.xml
- All these files interlink/refer to each other and/or other files
- Colors.xml → defines the color to be displayed on the screen.
- String.xml → defines the name of the app created
- Styles.xml → themes of android project is defined here



Exploring the file and folder structure of the Basic Activity project template.



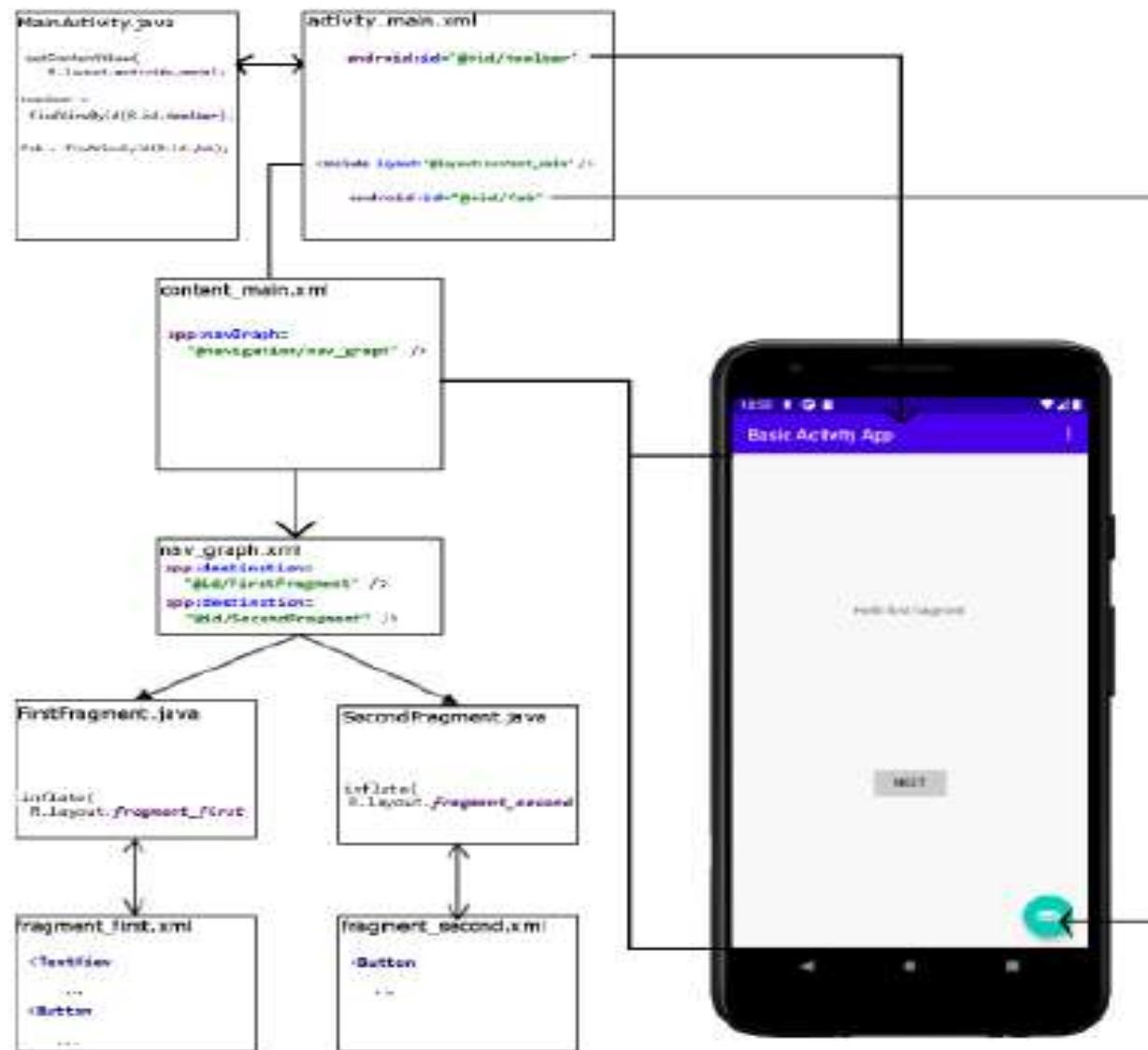
The next-simplest project type with an autogenerated UI is a **Basic Activity** project.

1. Run Android Studio and left-click the **Start a new Android Studio project** option.
2. The window that follows is the **Select a Project Template** window. Select **Basic Activity** and click **Next**.
3. In the **Configure Your Project** window, set up the project as follows:

Option	Value entered
Name:	Basic Activity App
Package name:	com.gamecodeschool. basicactivityapp
Save location:	D:\Android\Projects\ BasicActivityApp
Language:	Java
Minimum SDK:	Leave this and any other options at their default settings

4. Click the **Finish** button, and run the app.

# Basic Activity Template



# Exploring a Basic Activity project

## MainActivity.java file

- The code refers to two resources →
  - Toolbar resource : R.id.toolbar, FloatingActionBar resource :R.id.fab.

## activity\_main.xml file

- Java code : setting up the toolbar and the floating action bar ready for use.
- XML files can refer to other XML files.
- Java can refer to XML files

## The extra methods in MainActivity.java

- Enhances the menu that is defined in the menu\_main.xml file

onCreateOptionsMenu method is an overridden method that is called by the operating system directly.

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    // Inflate the menu; this adds items to the  
    // action bar if it is present.  
    getMenuInflater().inflate(R.menu.menu_main, menu);  
    return true;  
}
```

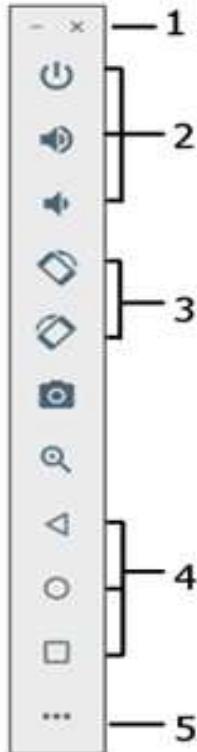
Exploring the Android emulator.



# Emulator control panel

The mini control panel that appears beside the emulator when executed.

1. Window controls : minimize or close the emulator window.
2. Power off the emulator, simulating powering off the actual device. Two icons to increase and decrease the volume.
3. Two buttons → to rotate the emulator left and right. Helps to visualize the appearance of the app in different orientations.
4. Icons simulate the back button and home button, and viewing running apps.
5. This button is used launch the **ADVANCED SETTINGS** menu, to interact with hardware such as sensors, the **Global Positioning System (GPS)**, the battery, the fingerprint reader, and more.



# Using the emulator as a real device

- The emulator can emulate every feature of a real phone

## Accessing the app drawer

- Hold the mouse cursor on the bottom of the home screen and drag upward to access the app drawer (all the apps). The following screenshot shows this action halfway through:



- Now, it can run any app installed on the emulator.
- When it runs one of the apps through Android Studio, it remains installed on the emulator and is therefore runnable from the app drawer.
- Every change made to the app in Android Studio will require to run/install the app again by clicking the play button on the Android Studio quick launch bar.

## Viewing active apps and switching between apps

- To view active apps, it can be used the emulator control panel, the square labeled as number **4** on the screenshot of the emulator control panel
- To access the same option using the phone screen swipe up, just as when accessing the app drawer.
- This process is illustrated in the following screenshot:



Swipe left and right through recent apps, swipe an app up to close it, or tap the back button to return to what were doing before viewed this option.

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448

Mobile Application Development

Week 4

Getting Started with Layouts the Project Structure

&

Beautiful Layouts with CardView and ScrollView



# Contents

1. Finding out about Material Design
2. Exploring Android UI design
3. Introducing layouts
4. Building a precise UI with ConstraintLayout
5. Laying out data with TableLayout
6. Linking back to the main menu
7. Attributes quick summary
8. Building a UI with CardView and ScrollView
9. Creating a tablet emulator



# Weekly Learning Outcomes

1. Explain the Android UI design and layouts.
2. Demonstrate the UI with CardView and ScrollView



## Required Reading

1. Chapter 4 and Chapter 5 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



# Finding out about Material Design



# Finding out about Material Design

- Objective of material design : simple to create appealing uis.
- Material design is taken directly from the design principles .
- Material design uses the concept of layers of materials – similar to photo-editing app
- Set of principles, rules, and guidelines → consistency

# Exploring Android UI design



- Android UI design is context-sensitive. The way that a given widget's x attribute will influence its appearance might depend on a widget's y attribute or even on an attribute on another widget.
- Example :dragging and dropping widgets onto the design
- **Extensible Markup Language (XML)** code that is generated will vary quite considerably, depending upon type of the layout used.
- Different layout types use different techniques to identify the widget's position—for example, the *LinearLayout* is very differently from the *ConstraintLayout*
- Three types of layouts:
  - *LinearLayout*
  - *ConstraintLayout*
  - *TableLayout*

# Introducing layouts



- Layouts are the building blocks that group together the other UI elements.
- Layouts can also contain other layouts.
- Layouts can be activated by `setContentView` method to work.

## Creating and exploring a layout project

- **Exploring Layouts** project is the first app of this type,
1. Create a new project in Android Studio, select **File | New Project**. When prompted, choose **Open in same window**.
  2. Select the **Empty Activity** project template, click the **Next** button.
  3. Enter Exploring Layouts for the application name and then click the **Finish** button.

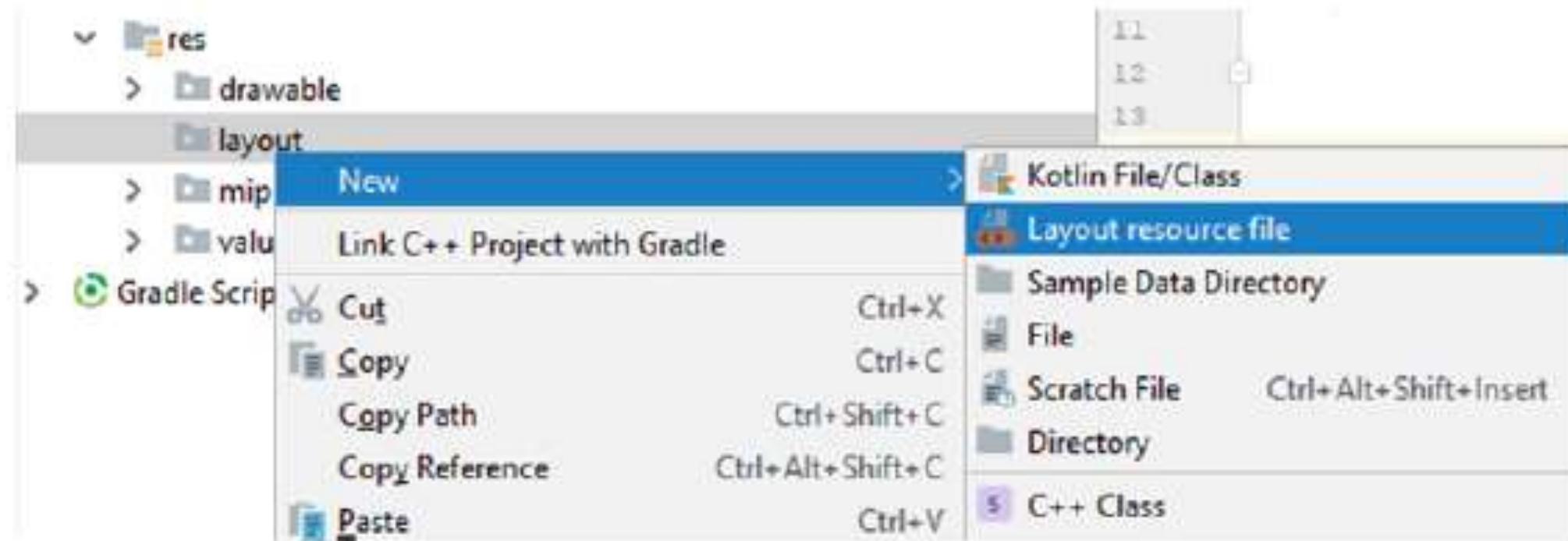
# Building a layout with LinearLayout

- LinearLayout is the simplest layout.
- All the UI items within it are laid out linearly.
- There are only two choices: vertical and horizontal.

## Adding a LinearLayout layout type to a project

- Expand the res folder in the project window.
- Right-click the layout folder and select **New**.
- Select **Layout resource file** and the **New Resource File** dialog window will pop up.
- In the **Root element:** field, enter LinearLayout.
- In the **File name:** field, enter main\_menu. Click the **OK** button, and Android Studio will generate a new LinearLayout in an XML file called main\_menu and place it in the layout folder, ready to build our new main menu UI. The new file is automatically opened in the editor, ready for us to get designing.

# Adding LinearLayout to project



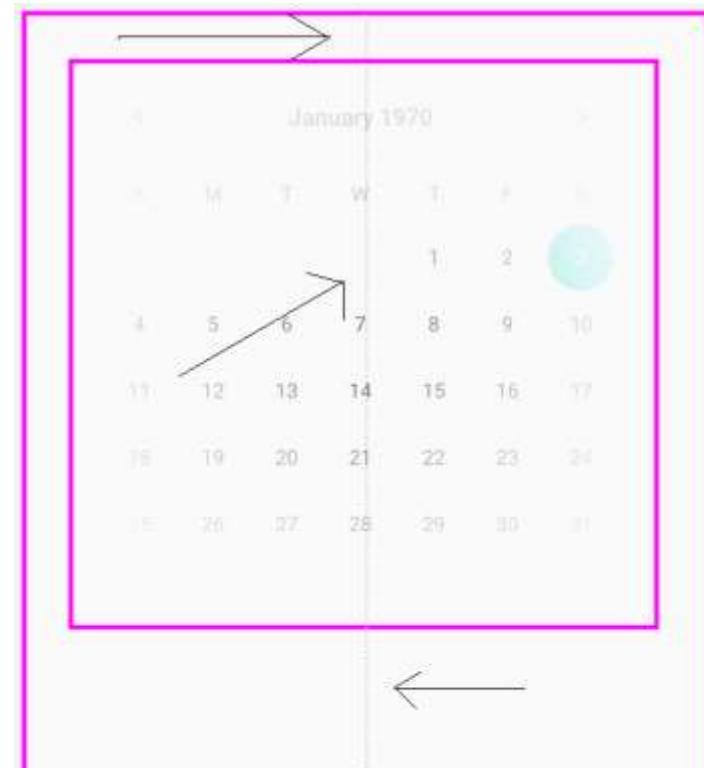
# Building a precise UI with ConstraintLayout



- Open the ConstraintLayout that was autogenerated when project is created.
- It will be in the res/ layout folder. The filename is activity\_main.xml.
- Examine the XML in the **Code** tab : a TextView that says Hello World.
- Switch back to the Design tab, left-click the TextView to select it, and tap the *Delete* key to get rid of it.

## Adding a CalendarView

- Drag and drop the CalendarView from the widgets category of the palette, near the top and horizontally central.



# Resizing a view in a ConstraintLayout

- Left-click and hold one of the corner squares that are revealed when let go of the CalendarView, and drag inward to decrease the size of the CalendarView, as illustrated in the following screenshot:



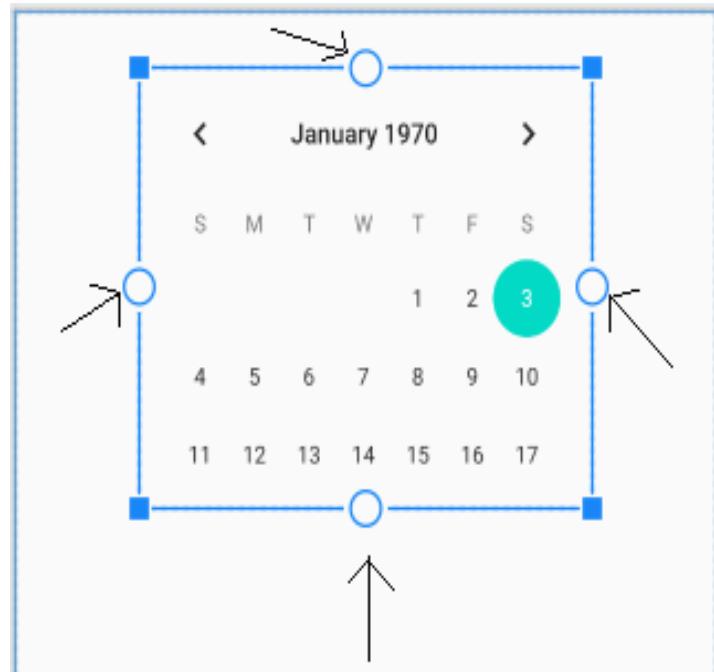
Reduce the size by about half and leave the CalendarView near the top and horizontally central. It might be needed to reposition the widget resizing it. The result should look like this:

## Using the Component Tree window

- **Component Tree** window, the window to the left of the visual designer and below the palette. A component tree is a way of visualizing the layout of the XML but without all the details.

## Adding constraints manually

- Select the CalenderView, the four small circles on the top, bottom, left, and right, as illustrated in the following screenshot:



# Laying out data with TableLayout



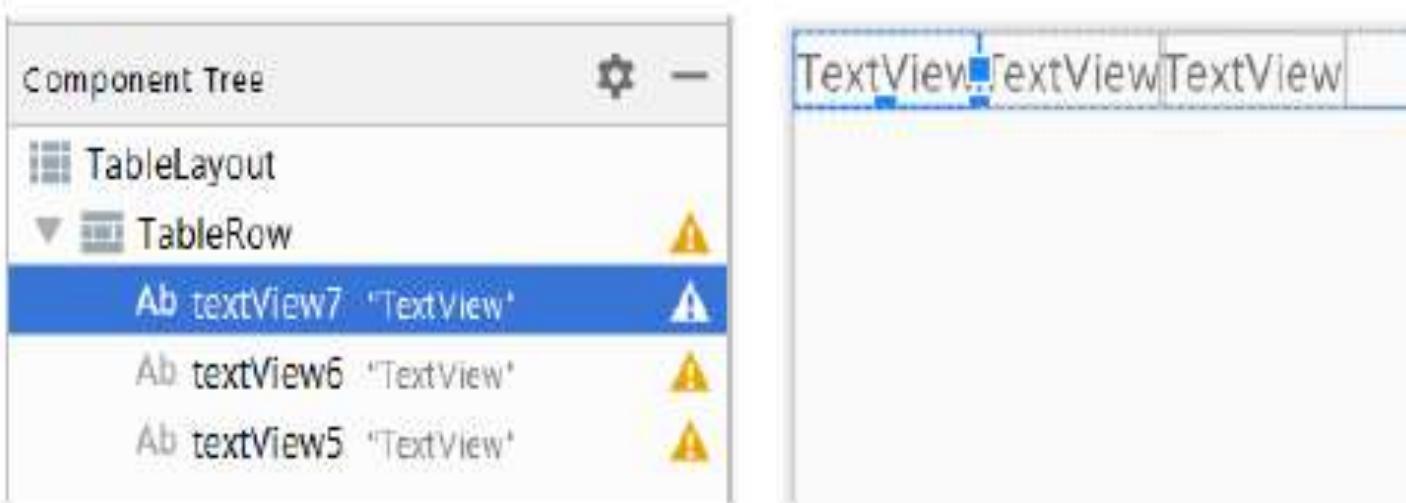
Building a layout that resembles a spreadsheet with neatly aligned cells with titles and data.

Steps:

1. In the project window, expand the res folder. Now, right-click the layout folder and select **New**.
2. Select **Layout resource file**, the **New Resource File** dialog window will be seen.
3. In the **File name:** field, enter my\_table\_layout. This is the same name which is used in the call to setContentView within the loadTableLayout method.
4. In the **Root element:** field that it has selected by default ...**ConstraintLayout** as the option. Delete ...ConstraintLayout and type TableLayout.
5. Click the **OK** button, and Android Studio will generate a new TableLayout in an XML file called my\_table\_layout and place it in the layout folder, ready to build new table-based UI. Android Studio will also open the UI designer with the palette on the left and the **Attributes** window on the right.
6. Now uncomment the loadTableLayout method in the MainActivity.java file

- **Adding a TableRow element to TableLayout**

- Drag and drop the TableRow widget from the component tree

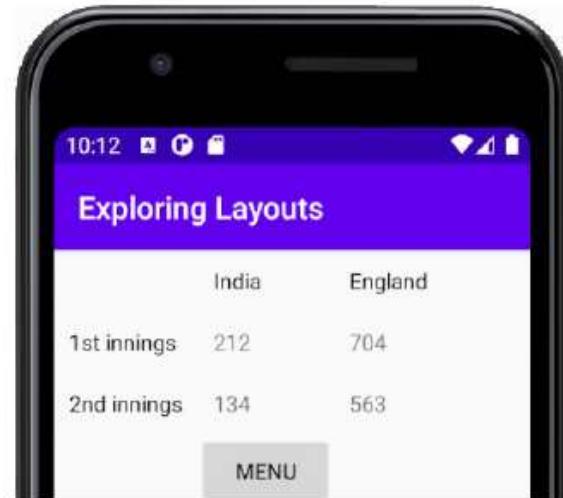


Linking back to the main menu



## Linking the table back to main menu

1. Add another TableRow via the **Component Tree**.
2. Drag a button onto the new TableRow.
3. Edit its layout\_column attribute to 1 so that it is in the middle of the row.
4. Edit its text attribute to Menu and edit its onClick attribute to match already existing loadMenuLayout method



# Beautiful Layouts with CardView and ScrollView(Chapter 5)

## Attributes quick summary



# Chapter Objective

Learning of  
different  
attributes

Additional layouts  
: ScrollView and  
Cardview

Execute Cardview  
project on Tablet  
emulator

# Attributes quick summary

## Sizing using dp

- **density-independent pixels**, or **dp**, : unit of measurement.
- Works by calculating the density of the pixels on the device an app is running on.
- Used for setting size of various attributes of widgets.
- To design layouts that scale to create uniform layouts on different screen size devices

## Sizing fonts using sp

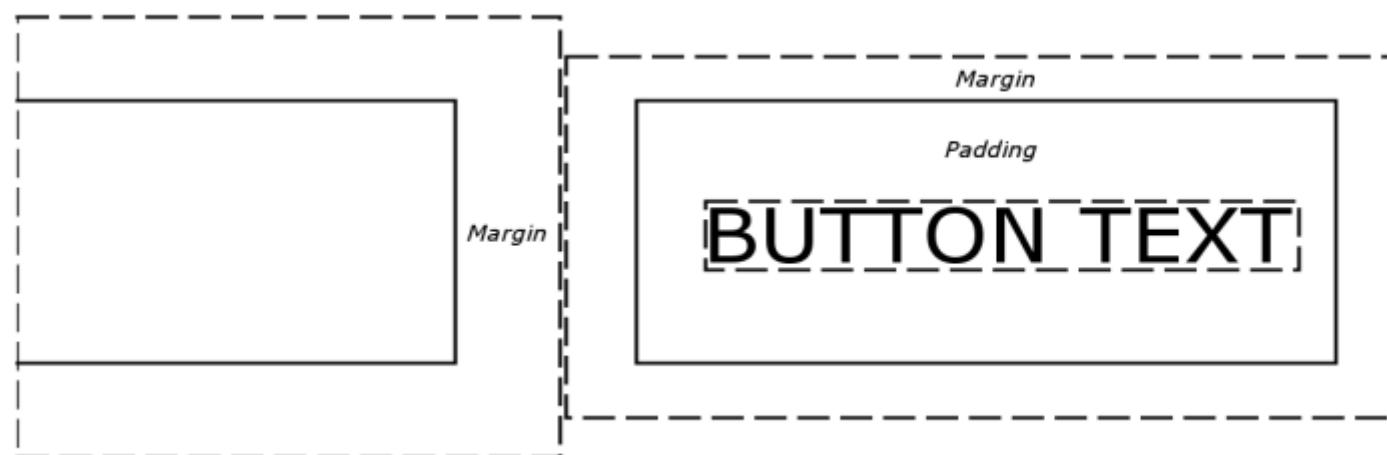
- **scalable pixels**, or **sp** → device-dependent unit of measurement used for sizing

## Determining size with wrap or match

- To size of UI elements relative to other UI elements containing parent element
- Done by setting the layoutWidth and layoutHeight attributes to either wrap\_content or match\_parent.

## Using padding and margin

- **Padding** is the space from the edge of the widget to the start of the content in the widget.
- **Margin** is the space outside of the widget that is left between other widgets – including the margins of other widgets, as shown in figure:



# Using the layout\_weight property

- Divide the screen space among the parts of the UI elements.
- To maintain uniformity in the relative space occupied by the UI elements on the screen independent of the device.
- layout\_weight is used in conjunction with sp and dp units for a simple and flexible layout.

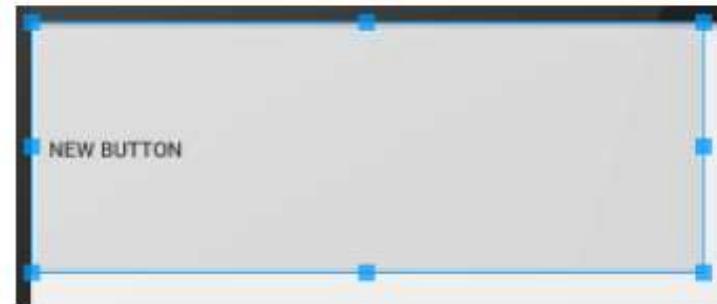
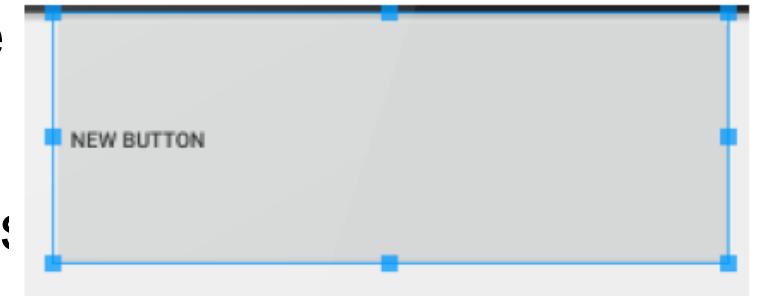
```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    android:layout_weight=".1"  
    android:text="one tenth" />  
  
<Button  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    android:layout_weight=".2"  
    android:text="two tenths" />
```



The UI with the layout\_weight property in use

# Gravity

- Gravity affects the position of items by moving them in a given direction
- Example:
- `android:gravity="left|center_vertical"`
- The element is left aligned horizontally and in center aligned vertically.
- To position the layout itself gravity will be defined as
- `android:layout_gravity="left"`



# Building a UI with CardView and ScrollView



- Create a new project in the usual way and choose the **Empty Activity** project template. Name the project CardView Layout.
- Designing CardView masterpiece inside a ScrollView layout, → allows the user to scroll through the contents of the layout.
- Expand the folders in the project explorer window to view the “res” folder Expand the res folder to view the layout folder.
- Right-click the **layout** folder and select **New**.
- Select **Layout resource file** the **New Resource File** dialog window will be shown.
- In the **File name** field, enter main\_layout.

- In the **Root element field** change the **ConstraintLayout** value to **ScrollView** .
- allow the user to scroll through the content by swiping with their finger when there is too much content to display onscreen.
- Click the **OK** button and Android Studio will generate a new ScrollView in an XML file called `main_layout` and place it in the layout folder ready to build CardView-based UI.
- Android Studio will open the UI designer ready for action.

### **Setting the view with Java code**

- **Load the main\_layout.xml file. Amend the code on the onCreate method.**

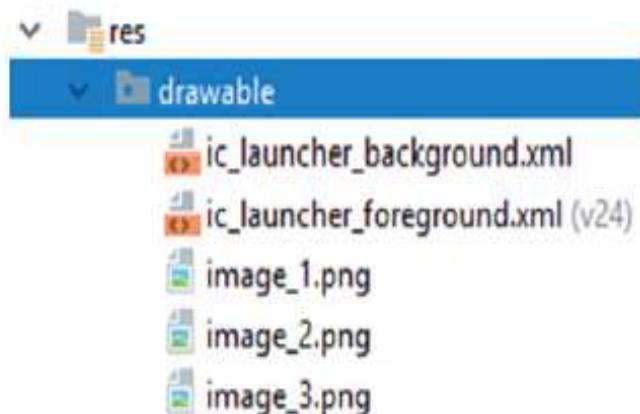
```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main_layout);  
}
```

# Adding image resources

- Images are added and formatted in a cardview widget

There are three images: image\_1.png, image\_2.png, and image\_3.png. To add them to the project, follow these steps:

1. Find the picture files using operating system's file browser and copy them.
2. In the Android Studio project explorer, select the **res/drawable** folder
3. In the pop-up window that asks **Choose Destination Directory**, click **OK** to accept the default destination → the drawable folder.
4. Click **OK** again to **Copy Specified Files**.



## Creating the contents for the cards

- Create three separate layouts, called card\_contents\_1, card\_contents\_2, and card\_contents\_3.

create three more layouts with a `LinearLayout` layout at their root:

1. Right-click the **layout** folder and select **New layout resource file**.
2. Name the file `card_contents_1` and change `...ConstraintLayout` to `LinearLayout` for the root element.
3. Click **OK** to create the new file in the layout folder.
4. Repeat *steps 1 through 3* two more times, changing the filename each time to `card_contents_2` and then `card_contents_3`.

## ➤ Defining dimensions for CardView

- Right-click the **values** folder and select **New | Values resource file**. In the **New Resource**
- **File** pop-up window, name the file **dimens** (short for dimensions) and click **OK**. Android
- Studio will create and open a file called **dimens.xml**.

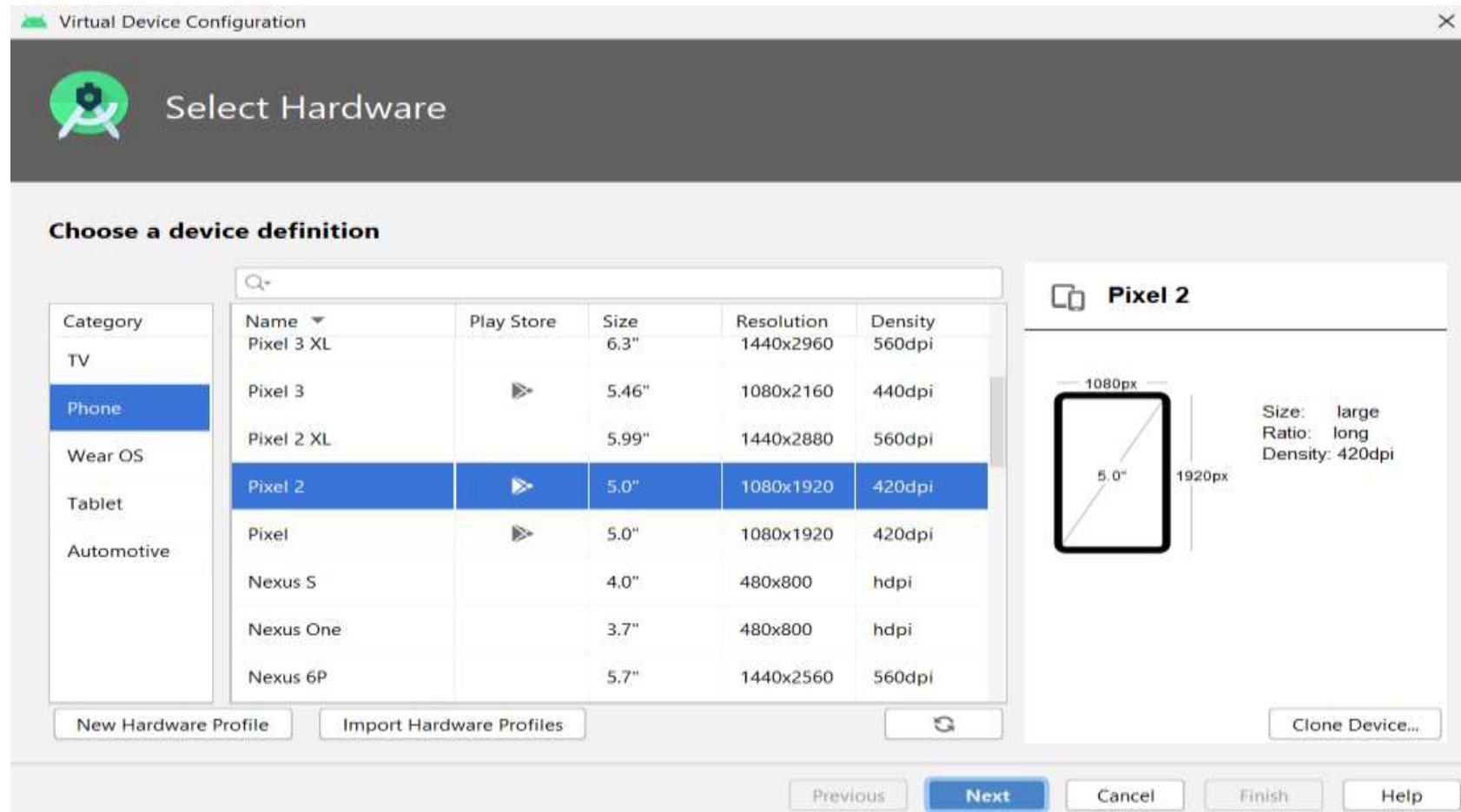
## ➤ Adding CardView to our layout

- Switch to the **main\_layout.xml**

# Creating a tablet emulator



- Select **Tools | AVD Manager** and then click the **Create Virtual Device...** button on the **Your Virtual Devices** window. **Select Hardware** window pictured next:



Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 5  
The Android Lifecycle



# Contents

1. The life and times of an Android app
2. How Android interacts with our apps
3. A simplified explanation of the Android lifecycle
4. How we handle the lifecycle phases
5. Lifecycle demo app
6. Examining the lifecycle demo app output
7. Some other overridden methods
8. The structure of Java code – revisited
9. Introducing fragments and the lifecycle



# Weekly Learning Outcomes

1. Explain the Android Lifecycle phases..
2. Explain how handle the Android Lifecycle phases.



## Required Reading

1. Chapter 6 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



# The life and times of an Android app



# The lifecycle of an Android app

- Lifecycle : the way that all Android apps interact with the Android OS.
- The phases of the lifecycle is that an app goes through from creation to destruction
- Helps to know where to put our Java code, depending on the objective of the app
- Topics covered :
  - ✓ The life and times of an Android app
  - ✓ What is method overriding
  - ✓ The phases of the Android lifecycle
  - ✓ What exactly we need to know and do to code our apps
  - ✓ A lifecycle demonstration mini app.
  - ✓ A quick look at code structure

# Exploring Android UI design How Android interacts with our apps

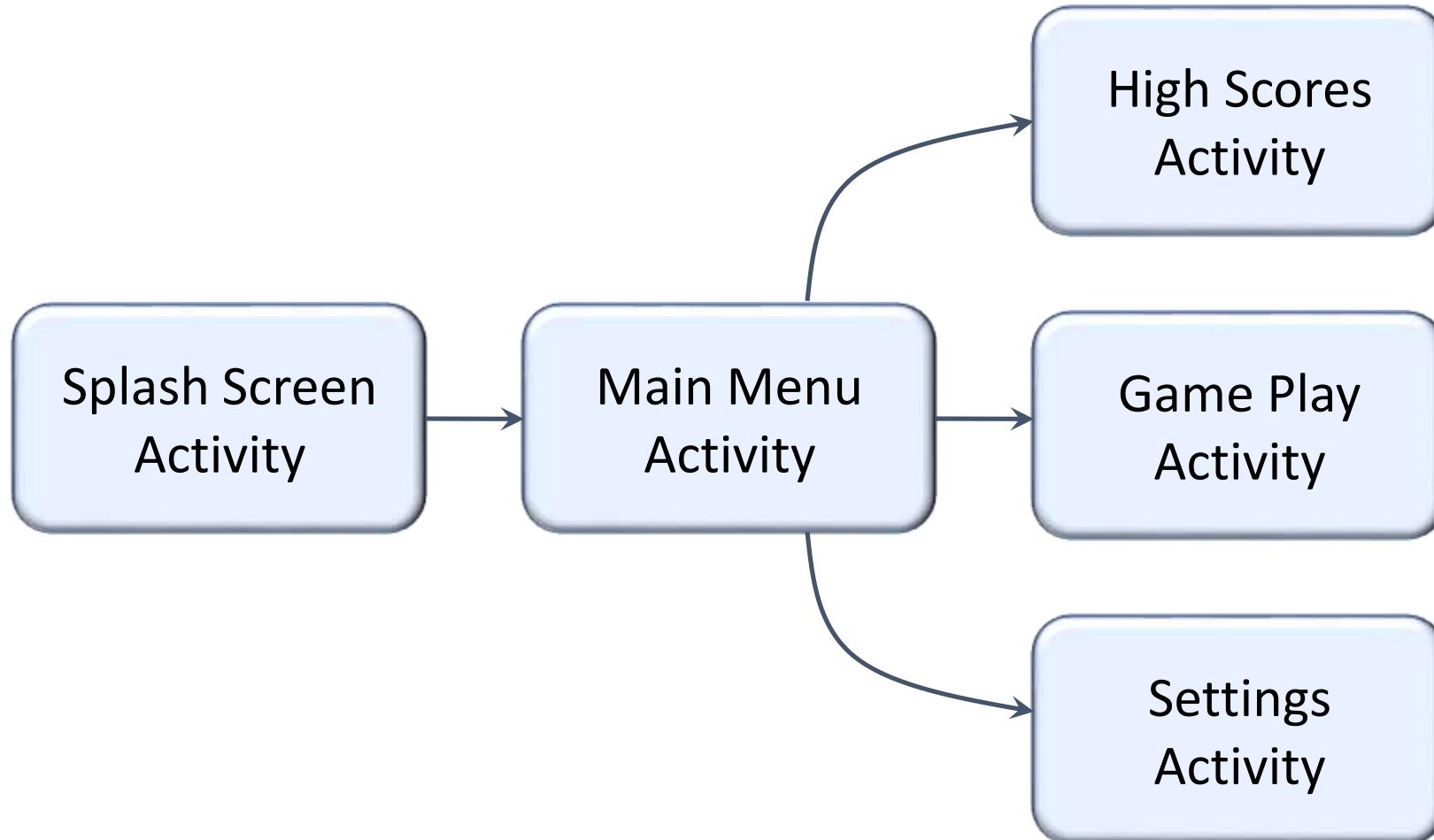


# How Android interacts with apps

- By calling methods that are contained within the Activity class: called by Android when required.
- *onCreate* method starts an activity in android.  

```
super.onCreate(savedInstanceState)
```
- Super is used to call the parent class constructor. *setContentView* is used to set the xml
- Directs Android to invoke the original version of the *onCreate* method.

## Activity component of an App – Example of a game



# A simplified explanation of the Android lifecycle



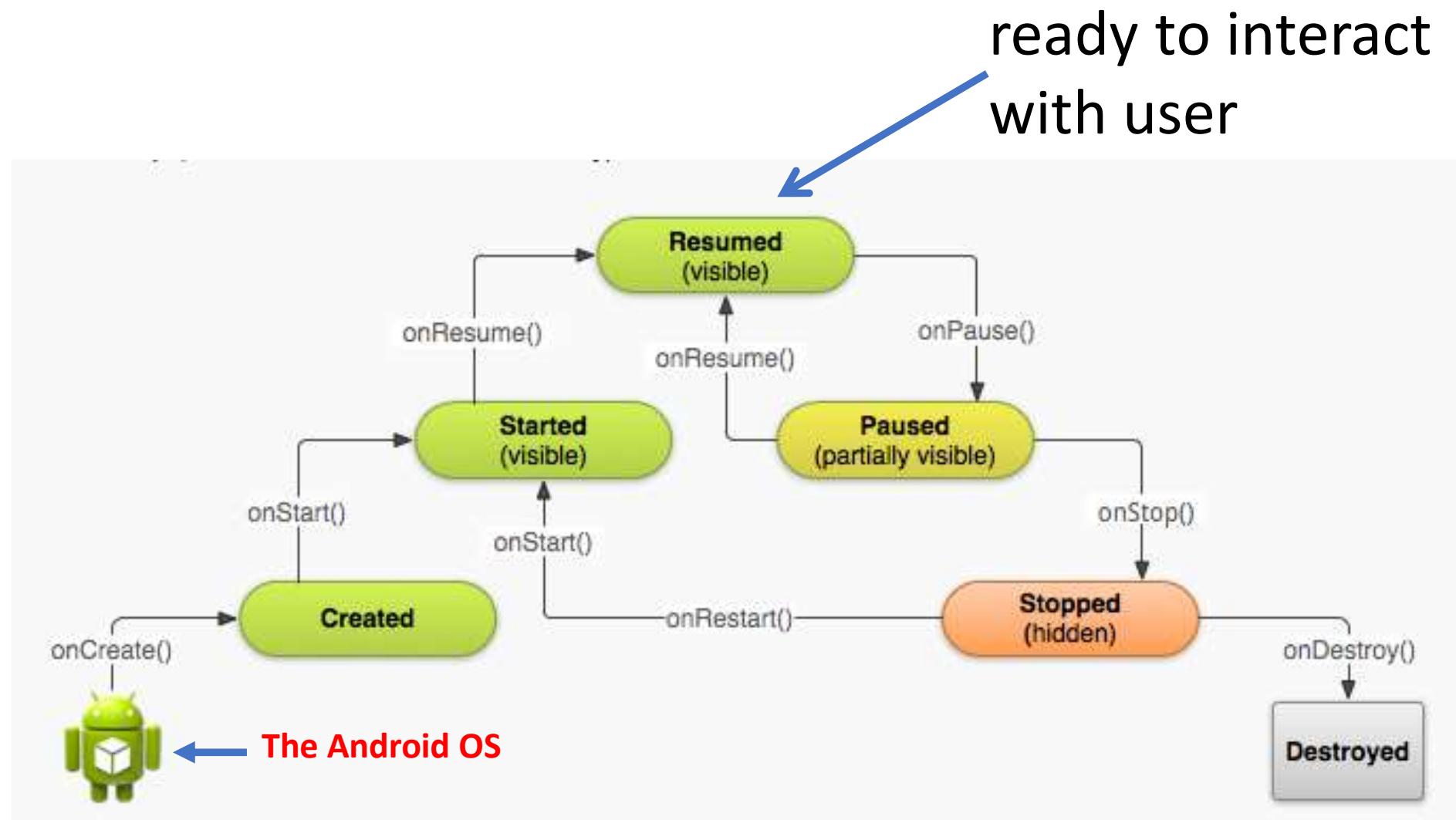
# Android lifecycle phases demystified

- Has multiple distinct phases
- The phase of the Android system decides how the app is viewed by the user – or whether it is viewed at all.
- Resources such as memory and processing power is allocated by Android based on the phase of the android app life cycle.
- Every app on an Android device will in one of the following phases:
  - ✓ Being created
  - ✓ Starting
  - ✓ Resuming
  - ✓ Running
  - ✓ Pausing
  - ✓ Stopping
  - ✓ Being destroyed

## Starting Activities

- Android applications don't start with a call to main(String[])
- Instead, a series of callback methods are invoked by the Android OS
- Each corresponds to specific stage of the Activity / application lifecycle
- Callback methods also used to tear down Activity / application

# Simplified Lifecycle Diagram



## Example – Facebook to email app

- When a user presses facebook app icon , the app is *created* and *started*
- After starting the app it is then in *resuming* stage.
- Once the app resumes, the app will have the full control of the screen and greater share of the memory and processing power – *Running* stage
- When switching over from facebook app to an email app, facebook app would move into *paused* stage, then into *stopping* stage.
- The switched over email app will get into *created* → *started* → *resuming and running stage* .
- When Android stops an app, it can *destroy* it.
- When an app gets into destroy stage, it has to start over again

# Handling the lifecycle phases

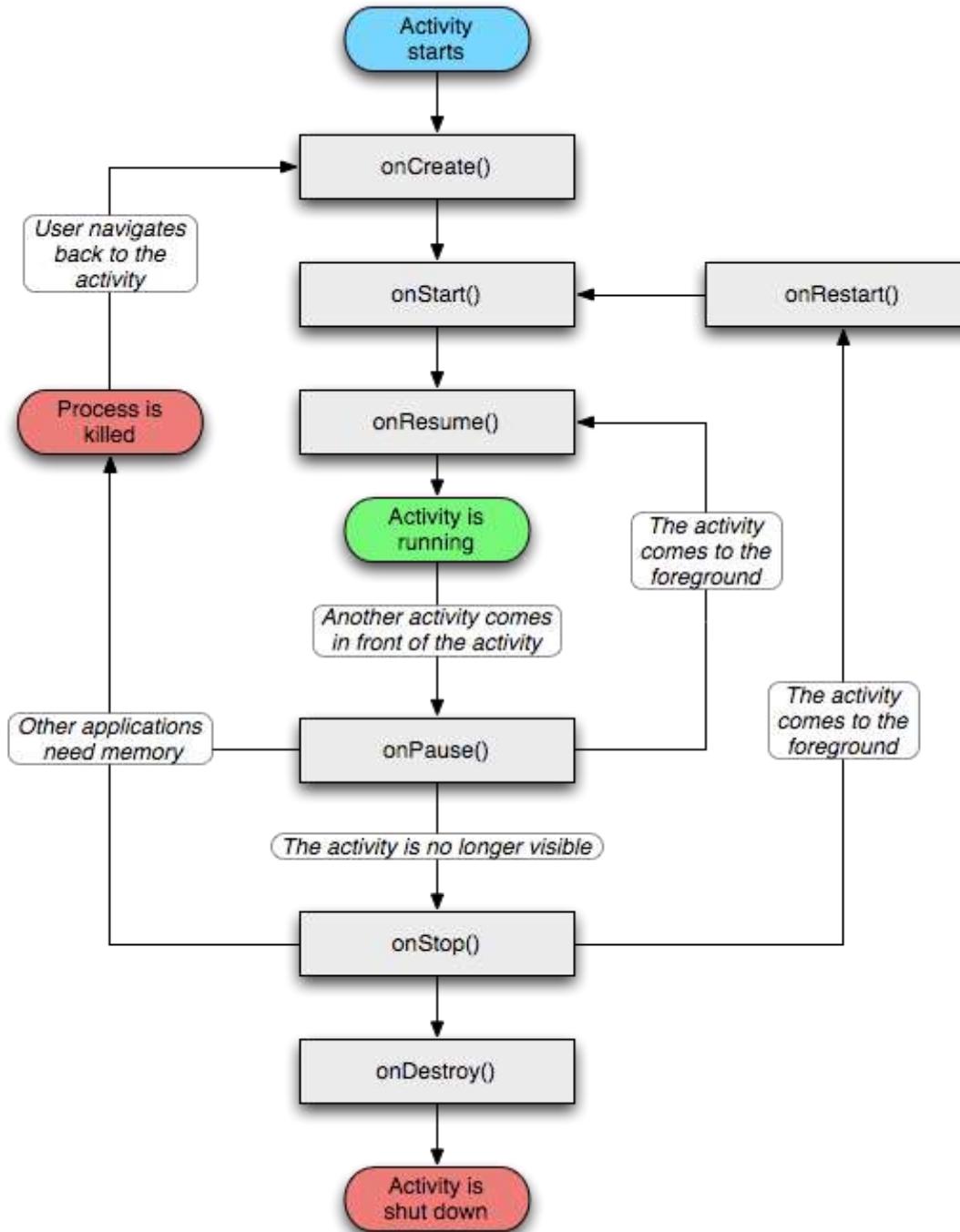
- An Android activity is one screen of the Android app's user interface.
- An Android app may contain one or more activities → one or more screens.
- The Android app starts by showing the main activity, and from there the app may make it possible to open additional activities.
- All activities in android apps are represented by an activity class
- Activity classes are subclasses of *android.app.Activity*
- Activity class contains a set of methods that relates to the lifecycle state of the activity is.
- Android code is *autogenerated*.
- Autogenerated codes can be *overridden* to customize the code according to requirement.

## Methods of Activity class

- ***onCreate***: first method is executed when the activity is being **created**. Declare the UI *setContentView*.
- ***onStart***: This method is executed when the app is in the **starting** phase.
  - ✓ makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive
  - ✓ this method is where the app initializes the code that maintains the UI.
- ***onResume***: This method runs after onStart.
  - ✓ It is also invoked after the app being previously paused.
  - ✓ Reloads the previously saved user data from when the app was interrupted, a phone call or the user running another app.

- ***onPause***: This method is callback when the app is **pausing**.
  - The app would pause when the device goes to sleep or when the user switches to another app.
  - The unsaved data will be saved and will be reloaded in **onResume**
- ***onStop***: This relates to the **stopping** phase. Performs tasks such as releasing system resources or writing information to a database.
- ***onDestroy***: This is when activity is finally being **destroyed**. The application is removed from the memory. Used to kill long running resources.

# Activity Life cycle- Flow of Execution of methods



# Lifecycle demo app



## Lifecycle demo app

- Familiarize with life cycle methods of an app.

Steps to start a new project and adding some code to it:

1. Start a new project.
2. Choose the **Basic Activity** template.
3. Call the project **Lifecycle Demo**.
4. Wait for Android Studio to generate the project files and then open the `MainActivity.java` file in the code editor.

## Coding the lifecycle demo app

## Running the lifecycle demo app

# Coding the lifecycle demo app

- In the MainActivity.java file, find the *onCreate* method and add these two lines of code just before the closing curly } brace, which marks the end of the onCreate method:

```
Toast.makeText(this, "In onCreate", Toast.LENGTH_SHORT).show();
Log.i("info", "In onCreate");
```

- Next to this add five life cycle methods as shown in the code below:
- Android will call these methods irrespective of the order in the code.
- @Override keyword* → methods replace/override the original version of the method that is provided as part of the Android API.
- The *super.on....* : the first line of code within each of the overridden methods, then calls these original versions
- Output of the added code :** output one Toast message and one Log message.

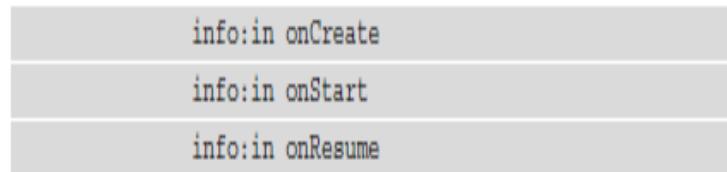
# Life cycle methods code listing

```
@Override  
public void onStart() {  
    // First call the "official" version of this method  
    super.onStart();  
  
    Toast.makeText(this, "In onStart",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "In onStart");  
}  
  
@Override  
public void onResume() {  
    // First call the "official" version of this method  
    super.onResume();  
  
    Toast.makeText(this, "In onResume",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "In onResume");  
}  
  
@Override  
public void onPause() {  
    // First call the "official" version of this method  
    super.onPause();  
  
    Toast.makeText(this, "In onPause",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "In onPause");  
}
```

```
@Override  
public void onStop() {  
    // First call the "official" version of this method  
    super.onStop();  
  
    Toast.makeText(this, "In onStop",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "In onStop");  
}  
  
@Override  
public void onDestroy() {  
    // First call the "official" version of this method  
    super.onDestroy();  
  
    Toast.makeText(this, "In onDestroy",  
        Toast.LENGTH_SHORT).show();  
  
    Log.i("info", "In onDestroy");  
}
```

# Running the lifecycle demo app

1. Run the emulator
2. “Toast” messages will appear on the screen for every life cycle methods in the order *In onCreate → In onStart → In onResume.*
3. *LogCat* → view the system log
4. Following messages obtained in LogCat window



5. On tapping the back button on the emulator, “Toast” message will appear in the order *In onPause → In onStop → In onDestroy*

## Examining the lifecycle demo app output



- When the lifecycle demo app started for the first time: the onCreate, onStart, and onResume methods were called.
- When the app is closed using the **back** button, the onPause, onStop, and onDestroy methods were called.
- When *switching* away and switching to the lifecycle demo → was not necessary to run onCreate .
- Calling of lifecycle methods vary in different device according to the users preferences, and hence the methods calling sequence cannot be predicted.
- The solution to all this complexity is to follow a few simple rules:
  1. Set up app ready to run in the onCreate method.
  2. Load user's data in the onResume method.
  3. Save user's data in the onPause method.
  4. Tidy up the app and make it a good Android citizen in the onDestroy method.

## Some other overridden methods

- Basic Activity Template in the Android Project has 2 other auto generated code :  
*onCreateOptionsMenu* and *onOptionsItemSelected*
- Auto generated code for pop-up menu
- *onCreateOptionsMenu* : Load the menu from the *menu\_main.xml*  
`getMenuInflater().inflate(R.menu.menu_main, menu);`
- *onOptionsItemSelected* method is called when the user taps the menu button.
  - ✓ Directs the execution to the item selected
  - ✓ just returns true when nothing happens.

## The structure of Java code – revisited

- Package : Container for the user defined code and libraries
- Classes : the building blocks of code
- Methods: wrap the functional code that does the work.
- Methods can be written within the classes that can be extended.
- The only code, however, that put in these methods was a few calls using Toast and Log.

# Introducing fragments and the lifecycle

- MainActivity.java : Contains the basic activity template along with →  
*Firstfragment.java* and *SecondFragment.java files*
- Contains the code to handle the navigation of the user between the screens of the BasicactivityTemplate
- **Fragment**: Class like an activity in a class
  - ✓ Represent a screen
  - ✓ Controlled by Activity classes
  - ✓ Has its specific lifecycle methods

```
@Override  
public View onCreateView(  
    LayoutInflater inflater, ViewGroup container,  
    Bundle savedInstanceState  
)
```

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 6  
Android Dialog Window & Adapters and Recyclers



# Contents

1. Dialog windows
2. The Note to Self app
3. RecyclerView and RecyclerAdapter
4. Adding RecyclerView, RecyclerAdapter, and ArrayList to the Note to Self project
5. Running the app



## Weekly Learning Outcomes

1. Explain how to use Android Dialog windows in app.
2. Explain how to use RecyclerView and RecyclerViewAdapter in app.



## Required Reading

1. Chapter 14 & Chapter 16 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



# Dialog windows



# Dialog window

pop-up window to display notification or confirmation messages to the user.

- Dialogs in Android : are sophisticated classes that consists of layouts and other specific User Interface (UI) elements.
- Create a dialog window in Android → *FragmentDialog class*.
- Creating the Dialog Demo project : create a new project in Android Studio named “Dialog Demo” with Empty Activity template.
- Coding a DialogFragment class : Create a new JAVA class and refer it as “MyDialog” in same package that has the MainActivity.java file.
- Change the class declaration to extend DialogFragment

```
public class MyDialog extends DialogFragment {}
```

- Import the DialogFragment class: Add the code in the MyDialog.java file

```
package com.gamecodeschool.dialogdemo;  
import androidx.fragment.app.DialogFragment;  
public class MyDialog extends DialogFragment {}
```

- Override the `onCreateDialog` method : declare & Initialize *AlertDialog.Builder*
- *getActivity* Method : part of fragment class → creates a reference that is passed to the MainActivity → Creates a dialog fragment

```
public class MyDialog extends DialogFragment {  
  
    @Override  
    public Dialog onCreateDialog(Bundle savedInstanceState) {  
  
        // Use the Builder class because this dialog  
        // has a simple UI  
        AlertDialog.Builder builder =  
            new AlertDialog.Builder(getActivity());  
        }  
    }
```

- Import the Dialog, Bundle, and AlertDialog classes

```
import android.app.Dialog;  
import android.os.Bundle;  
  
import androidx.appcompat.app.AlertDialog;  
import androidx.fragment.app.DialogFragment;
```

# Configure DialogFragment : Chaining

- **Chaining** : Process of calling more than one method in a sequence on the same object.
- Similar to writing multiple lines of code in a concise manner.
- Example : When a Toast message is created and added a .show() method to the end of it.
- Add the code (which uses chaining) inside the onCreateDialog method

// Dialog will have "Make a selection" as the title

```
builder.setMessage("Make a selection")
```

// An OK button that does nothing

```
.setPositiveButton("OK", new DialogInterface.OnClickListener()
{
    public void onClick(DialogInterface dialog, int id) {
        // Nothing happening here
    }
})
```

- Import the Dialog interface class

```
import android.content.DialogInterface;
```

- Builder.setMessage : sets the main message for the user in the dialog box
- .setPositiveButton : sets the “OK” as the text of the button which is the first argument in this method.
- DialogInterface.OnClickListener : handles any click on the button
- Add the “return” statement : Takes the control to mainactivity

```
return builder.create();
```

- Fully configured dialog window is created

# Using the DialogFragment class

Add a button to our layout. Perform the following steps:

1. Switch to the activity\_main.xml tab and then switch to the **Design** tab.
2. Drag a **Button** onto the layout and set its “*id*” attribute to button.
3. Click on the **Infer Constraints** button to constrain the button in the location to be placed. Note that the position isn't important –it can be used to create an instance of our MyDialog class.
4. Adding the following code to onCreate method

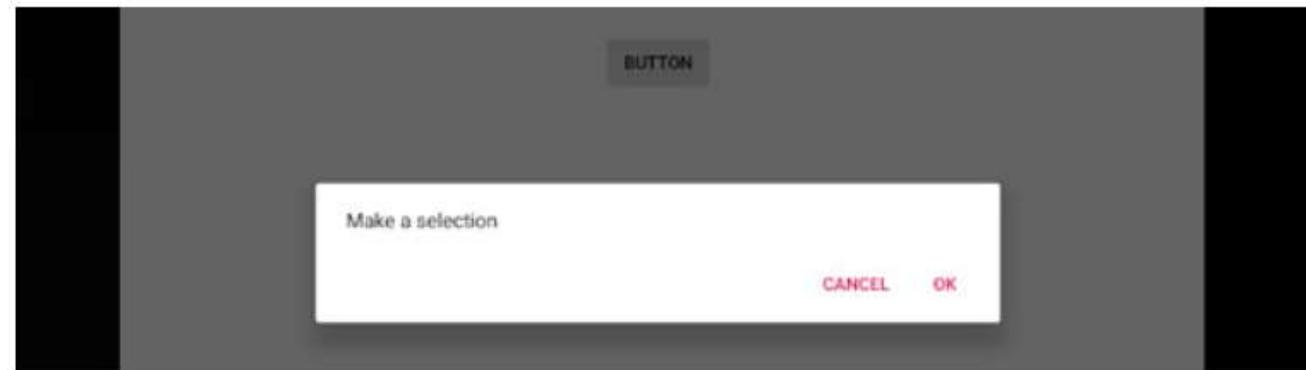
```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            MyDialog myDialog = new MyDialog();
            myDialog.show(getSupportFragmentManager(),"123");
        }
    }
);
```

- Packages to import for this code:

```
import android.view.View;
```

```
import android.widget.Button;
```

- onClick method creates a new instance of MyDialog
- Show method : reference the FragmentManager class with getSupportFragmentManager, passing the identifier “123”
- **FragmentManager**: Class that tracks and controls all Fragment instances for an activity
- **getSupportFragmentManager**: to support older devices by extending the *AppCompatActivity class*
- **Output of the code :**



# The Note to self app



## The Note to Self app

- First phase of three major apps
- Use of Android naming conventions, string resources and proper encapsulation
- Implementing in the real projects
- Improvise the quality of the apps

## Naming Conventions And String Resources

- Naming conventions : are the conventions or rules used for naming the variables, methods, and classes in the code.
- When a variable is a member of a class, name of the variable is prefixed with a lowercase m.
- String Resources : used for hardcoding text in user layouts

## The completed app

- The completed app will allow the user to tap on the floating action button in the lower-right corner of the app to open a dialog window to add a new note.

# Steps to build the app

- Building the project : Create a new project from the basic Activity template and name is as “**Note to Self**”

- Preparing the string resources : Create the string resources from the layout files

Add the code referred in the text (page-366) in the **strings.xml** file from the **res/values** folder

- Coding the Note class : Basic data structure of the app. Holds all the member variables required to represent the user’s notes. \*(refer the code listing from the textbook)
- Implementing the dialog designs : Use the FragmentDialog classes. Create a new layout resource file named as **dialog\_new\_note**
- Drag and drop the “**Plain Text**” widgets from the Text category, and the **CheckBox** widgets from the Button category
- Click on the **Infer Constraints** to position the widgets on the layout.
- Setup the **text**, **id**, and **hint** properties
- Create the layout for **show note** dialog

- Coding the dialog boxes : Implement the class to represent the dialog windows that the user can interact with the help of *FragmentManager* class
  - Coding the DialogNewNote class
  - Coding the DialogShowNote class
- Remove the unwanted autogenerated fragments: Open the *content\_main.xml* layout file. Open the *component tree* window in the *design tab*. Select and delete the *nav\_host\_fragment* element
- Showing our new dialogs : Run the app and click on “*Show Note*” button to view the DialogShowNote
- Coding the floating action button: rounded icon used for core action in the app. Eg: to add a new note in the app.



# Adapters and Recyclers(Chapter 16)



# Objectives

- How to extend the *RecyclerAdapter* class in java code and add a *RecyclerView* instance
- RecyclerAdapter : allows the user to scroll through the contents
- RecyclerView : displays the contents of RecyclerAdapter.
- Theory of adapters and binding them to UI
- Implementing the layout with RecyclerView
- Laying out a list item for use in RecyclerView
- Implementing the adapter with RecyclerAdapter
- Binding the adapter to RecyclerView
- Storing notes in ArrayList and displaying them in RecyclerView
- Discussing how we can improve the Note to Self app further

- Create an array of *TextView* widgets to populate a *ScrollView*,
- Place the title of a note within each *TextView*.
- Solution to display each note so that it is clickable in the Note to Self app.

### Problem with displaying lots of widgets

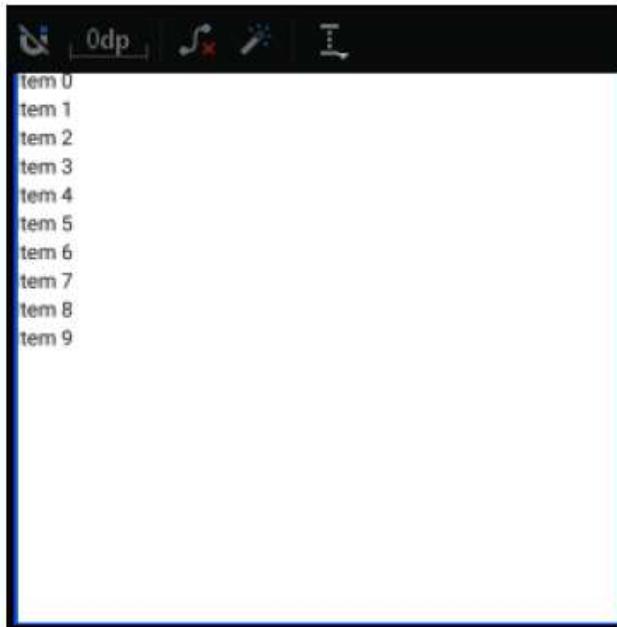
- ✓ Android device might simply run out of memory : handling the scrolling of a vast number of widgets and their data.

### Solution to the problem of displaying lots of widgets

- I. Add a single widget called *RecyclerView* to the UI layout.
- II. Interact with *RecyclerView* with a special type of class : using an adapter-  
*RecyclerAdapter* class
- III. Extend and customize *RecyclerAdapter* class, to control the data from *ArrayList* instance
- IV. and display it in *RecyclerView*.

# RecyclerView and RecyclerAdapter in Use

- Drag and drop the RecyclerView widget to layout, from the palette onto UI



- Use Linear Layout to hold three “TextView” widgets
- Define a list item in its XML file
- RecyclerView : hold multiple instance of the list items
- Customized implementation of RecyclerAdapter class – Handles the data
- Requires some coding for overriding the methods in Recycleadapter

## Set up of RecyclerView with RecyclerAdapter and an ArrayList of notes

- Delete the temporary button and related code, and then add a RecyclerView widget to layout with a specific id property.
- Create an XML layout to represent each item in the list in a LinearLayout that contains three TextView widgets.
- Create a new class that extends RecyclerAdapter and add code to several overridden methods to customize the layout format
- Add code to the MainActivity class to use the RecyclerAdapter class and the RecyclerView widget and bind it to ArrayList instance.
- Add an ArrayList to MainActivity to hold all the notes and update the *createNewNote* method to add any new notes created in the DialogNewNote class to this ArrayList.

Adding RecyclerView, RecyclerAdapter, and ArrayList to  
the Note to Self project



## Removing the temporary "Show Note" button and adding RecyclerView

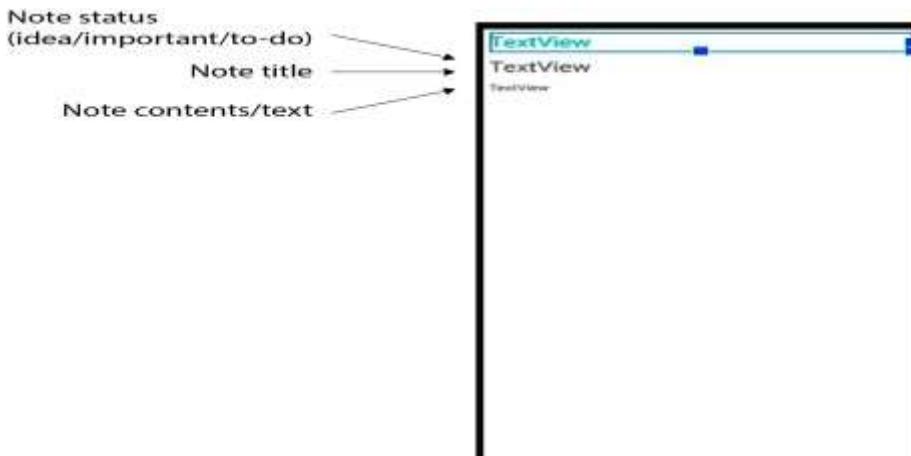
*Android Dialog Windows*, and set up our RecyclerView widget ready for binding to RecyclerAdapter:

1. In the content\_main.xml file, remove the temporary Button with an ID of button.
2. In the onCreate method of the MainActivity.java file, delete the Button instance declaration and initialization along with the anonymous class that handles its clicks.
3. Switch back to the content\_main.xml file in design view and drag a **RecyclerView** widget from the **Containers** category of the palette onto the layout.
4. Set its id property to recyclerView.

# Creating a list item for RecyclerView

A layout is needed to represent each item in our RecyclerView widget. As previously mentioned, use a LinearLayout that holds three TextView widgets.

1. Right-click on the **layout** folder in the project explorer and select **New | Layout resource file**. Enter listitem in the **Name** field and set **Root element** to **LinearLayout**.
2. On the **Design** tab and set the orientation attribute to vertical.

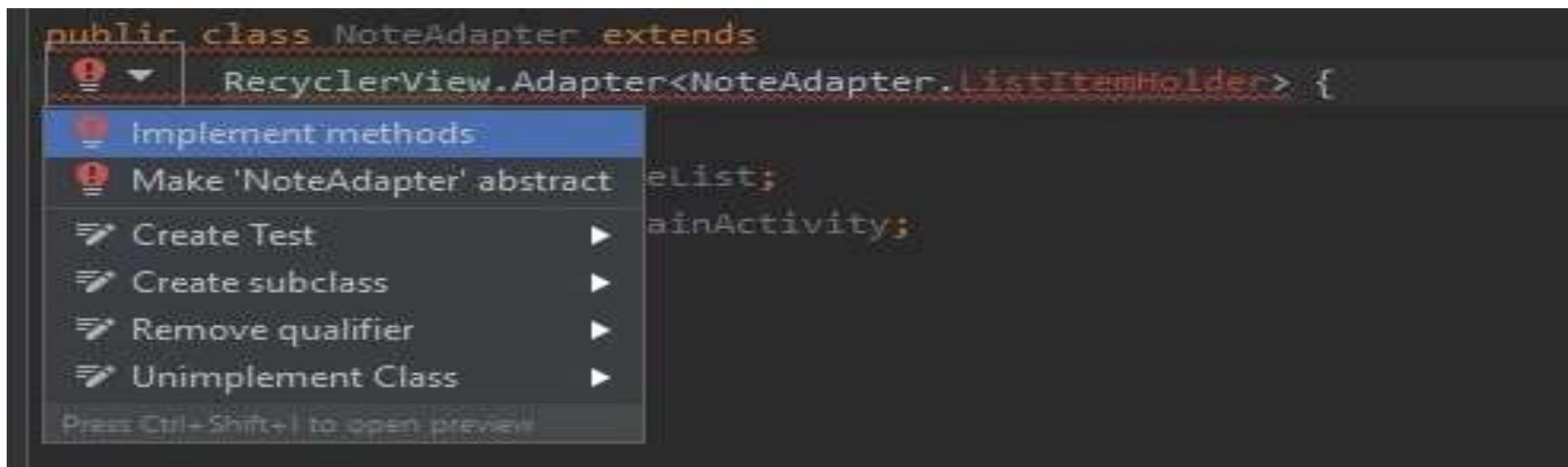


3. Drag three TextView widgets onto the layout. The first (top) will hold the note status/type (idea, important, or to-do). The second (middle) will hold the note title and the third (bottom) the snippet of text from the note itself.
4. Configure the various attributes of the LinearLayout and TextView widgets as shown in the following table:

Widget type	Property	Value to set to
LinearLayout	layout_height	wrap_contents
LinearLayout	Layout_Margin all	5dp
TextView (top)	Id	textViewStatus
TextView (top)	textSize	24sp
TextView (top)	textColor	@color/colorAccent
TextView (middle)	Id	textViewTitle
TextView (middle)	textSize	24sp
TextView (bottom)	Id	textViewDescription

# Coding the RecyclerView class

- Create a new class called NoteAdapter in the same folder as the MainActivity class.
- Edit the code for the NoteAdapter class by adding these import statements and extending it with the RecyclerView.Adapter class
- The quickest way to do this is to click the class declaration, hold the *Alt* key, and then tap the *Enter* key. Choose **Implement methods**, as shown in the following screenshot:



Click **OK** to auto-generate the required methods.

This process adds the following three methods:

- The *onCreateViewHolder* method, which is called when a layout for a list item is required.
- The *onBindViewHolder* method, which is called when RecyclerView is bound to (connected/associated with) RecyclerView in the layout.
- The *getItemCount* method: used to return the number of Note instances in ArrayList. Initially it returns 0.

The screenshot shows the code editor for a Java file named `NoteAdapter.java`. The code defines a class `NoteAdapter` that extends `RecyclerView.Adapter<NoteAdapter.ListItemHolder>`. It contains three methods: `onCreateViewHolder`, `onBindViewHolder`, and `getItemCount`. Three arrows point from the word "Error" to the code in each of these methods, indicating compilation errors. The code is as follows:

```
public class NoteAdapter extends
    RecyclerView.Adapter<NoteAdapter.ListItemHolder> {

    private List<Note> mNoteList;
    private MainActivity mMainActivity;

    @NonNull
    @Override
    public NoteAdapter.ListItemHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        return null;
    }

    @Override
    public void onBindViewHolder(@NonNull NoteAdapter.ListItemHolder holder, int position) {

    }

    @Override
    public int getItemCount() {
        return 0;
    }
}
```

## Coding the NoteAdapter constructor

- Code the **NoteAdapter** constructor method → initialize the members of the NoteAdapter class. Add this constructor to the NoteAdapter class:

```
public NoteAdapter(MainActivity mainActivity,  
                    List<Note> noteList) {  
    mMainActivity = mainActivity;  
    mNoteList = noteList; }
```

## Coding the onCreateViewHolder method

- Adapt the auto-generated onCreateViewHolder method. Add the two highlighted lines of code to the onCreateViewHolder method.

```
View itemView = LayoutInflater.from(parent.getContext())  
    .inflate(R.layout.listitem, parent,  
    false);  
return new ListItemHolder(itemView);
```

## Coding MainActivity to use the RecyclerView and RecyclerAdapter classes

Add three new members to MainActivity class :

- *onBindViewHolder method* :Checks for the type of the note (idea/to-do/important) and assigns appropriate label.
- *getItemCount* : returns the current number of items in the array list.

```
return mNoteList.size();
```

- *ListItemHolder* inner class : gets a reference to each of the TextView widgets in the layout

➤ **Adding code to the onCreate method** : To handle the floating action button clicks

- ✓ initialize the recyclerView reference with theRecyclerView instance in the layout.
- ✓ Initialize NoteAdapter (mAdapter) by calling the constructor and passing the ArrayList instance
- ✓ create a new object, LayoutManager.
- ✓ Invoke setLayoutManager on recyclerView and pass in this new LayoutManager instance.
- ✓ configure some properties of recyclerView.
- ✓ Invoke the setAdapter method → combines adapter with view.

➤ **Modifying the addNote method** : delete the temporary code *Android Dialog Windows*, and add the new code.

```
noteList.add(n);  
mAdapter.notifyDataSetChanged();
```

- ✓ adds a note to ArrayList
- ✓ call the notifyDataSetChanged method → Signals that a new note has been added to the adapter

## Coding the showNote method

- Accessed from the NoteAdapter
- Passed into the NoteAdapter constructor.
- Invoked from the ListerItemHolder inner class when one of the items in RecyclerView is tapped by the user.
- Add the showNote method to the MainActivity class.

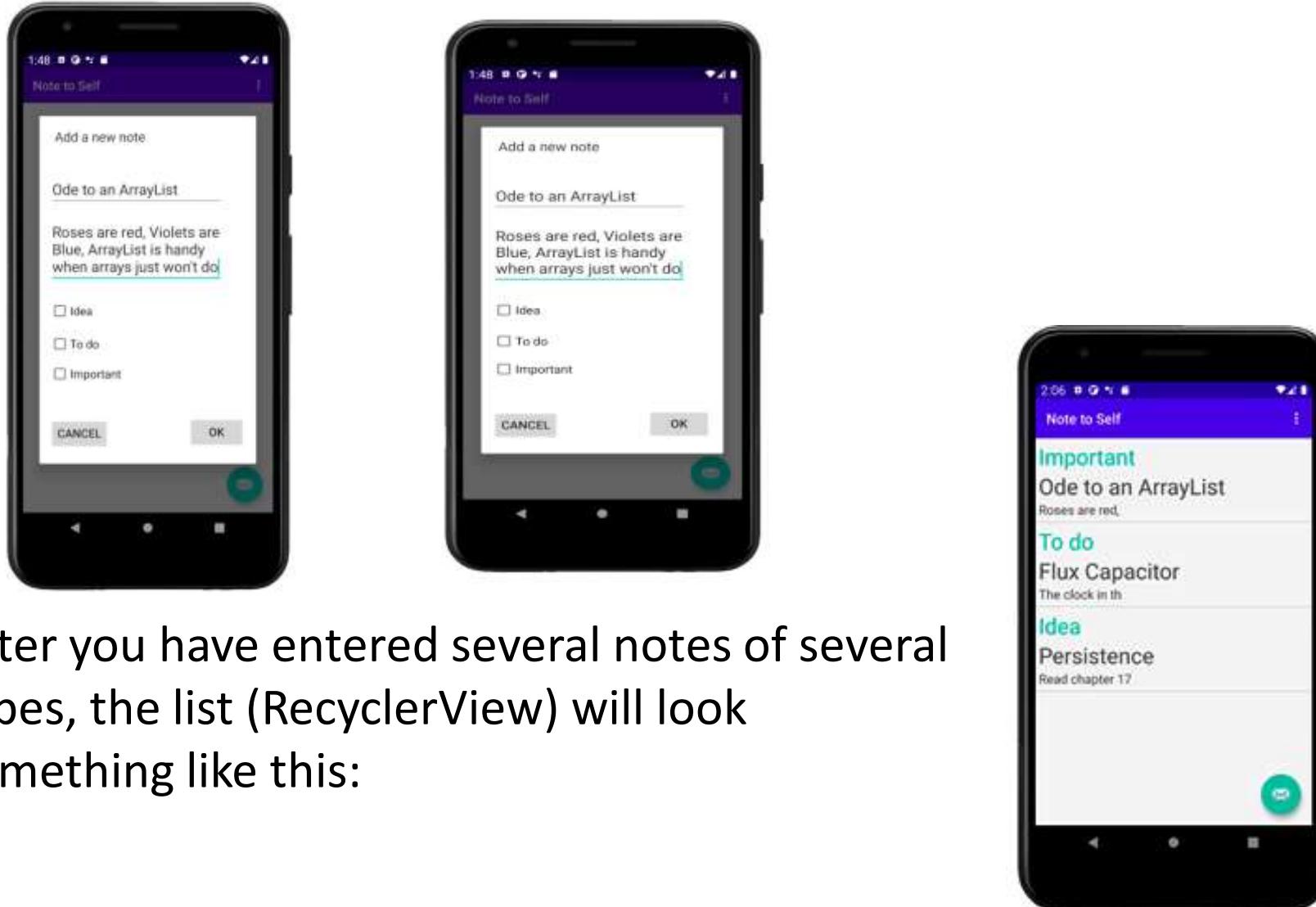
```
public void showNote(int noteToShow){  
    DialogShowNote dialog = new DialogShowNote();  
    dialog.sendNoteSelected(noteList.get(noteToShow));  
    dialog.show(getSupportFragmentManager(), "");  
}
```

- Will launch a new instance of DialogShowNote
- The new instance is passed to the note as pointed to by noteToShow

## Running the app



Now run the app and enter a new note as shown in the following screenshot:



Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 7  
Data Persistence and Sharing & Android Databases



# Contents

1. Android intents
2. Adding a settings page to Note to Self
3. Persisting data with SharedPreferences
4. Reloading data with SharedPreferences
5. Making the Note to Self settings persist
6. More advanced persistence
7. Databases 101& SQL syntax prime
8. Android SQLite API
9. Coding the database class
- 10.Coding the Fragment classes to use the DataManager class
- 11.Running the Age Database app



# Weekly Learning Outcomes

1. Explain about using Android intents to switch between Activity classes and pass data.
2. Design the screen data using the SharedPreferences class & JavaScript Object Notation.
3. Explain the Android SQLite.



## Required Reading

1. Chapter 17 & Chapter 27 (Android Programming for Beginners 3<sup>rd</sup> Edition, 2021 by John Horton , Packt Publishing Ltd.)



# Objectives of the chapter

- Different ways to store data in Android device's permanent storage
- Adding a second Activity class to the app and navigating between the two screens
- Using Android intents to switch between Activity classes and pass data
- Create a simple (very simple) settings screen in a new Activity class for the Note to Self project
- Persist the settings screen data using the SharedPreferences class
- Learn about **JavaScript Object Notation (JSON)** for serialization
- Explore Java's try-catch-finally syntax
- Implement saving data in our Note to Self app

# Android Intents



# Android intents

- ✓ Android apps comprises of multiple Activities
- ✓ Intent class : class that demonstrates the intent of an Activity class.
- ✓ Intent class : enables the user to switch between Activity instances.
- ✓ Activity instances : made from classes with member variables.
- ✓ When switching between the multiple screens in an app the Intents handle variable's data value by passing data between Activity instances.
- ✓ Intents also enables to interact with other apps too.

# Switching Activity

- ✓ Consider an app with two Activity based classes. One of the class is MainActivity class which will be the starting point of the app. Second activity is the SettingsActivity.
- ✓ Switching between the two screens: given in the code.

```
// Declare and initialize a new Intent object called myIntent  
Intent myIntent = new Intent(this, SettingsActivity.class);  
// Switch to the SettingsActivity  
startActivity(myIntent);
```

- ✓ Initializing the Intent object – Using Intent Constructor with two arguments
  1. Current activity referred as “this”
  2. Name of the activity class navigating to → SettingsActivity.class
- ✓ Encapsulation : Hides the information about each other activities → How to share the data?

# Passing data between activities

- ✓ Consider a sign-in screen: user need to pass the user's credentials to each Activity of app → done by using intents.
- ✓ Load the string to pass to the intent →  
`myIntent.putExtra("USER_NAME", username);`
- ✓ Add data using **key-value pairs** : accompanied by an identifier instance → used to identify and retrieve the data from the Activity.
- ✓ Retrieve the data → getExtra method and identifier from key-value pairs
- ✓ The Intent class can help when sending more complex data.

# Adding a settings page to Note to Self



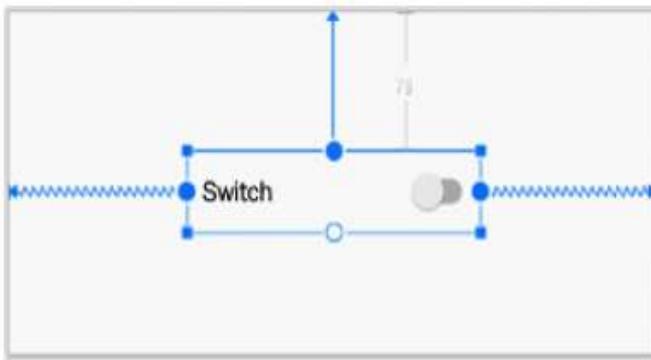
# Adding Another screen (Activity) to Note to Self app

- In the project explorer, right-click the folder that contains all \*.java files and has the same name as the package. From the pop-up context menu, select **New | Activity | Empty Activity**.
- In the **Activity Name** field, enter **SettingsActivity**.
- Leave all the other options at their defaults and left-click **Finish**.
- A new Activity-based class and its associated .java file is created by android Studio
- Register the Activity class with the Android : Open the *AndroidManifest.xml* file from within the manifests folder in the project explorer.  
  
`<activity android:name=".SettingsActivity"></activity>`
- A layout XML file (activity\_settings.xml) is automatically generated

# Designing the settings screen layout

Building a UI for our settings screen :

1. Open the activity\_settings.xml file, switch to the **Design** tab, and layout the screen settings.



1. Drag and drop a **Switch** widget onto the center top of the layout and stretched by dragging the edges to make it larger.
2. Add an id attribute of switch1 to interact with it using Java code from SettingsActivity.java.
3. Use the constraint handles to fix the position of the switch or click the Infer Constraints button to fix it automatically.

# Enabling the user to switch to the settings screen

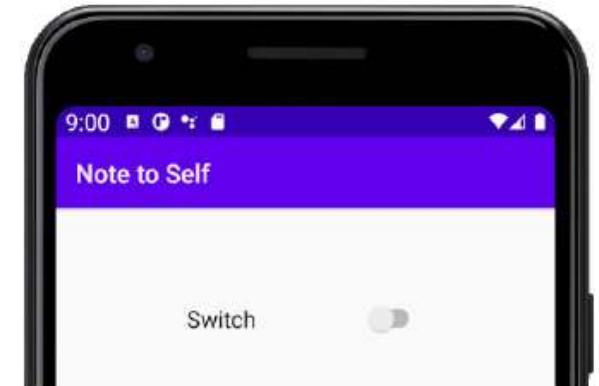
- Place the code to switch to the `SettingsActivity` class within the `nOptionsItemSelected` method in the `MainActivity` class.

```
//noinspection SimplifiableIfStatement  
if (id == R.id.action_settings) {  
    return true;}
```

- The “if” block is auto generated to place the code
- Add the given code inside the ‘if’ block

```
Intent intent = new Intent(this, SettingsActivity.class);  
startActivity(intent);
```

- Import the Intent class
- Run the app and navigate to the new settings screen by clicking on the `setting` menu



# Persisting data with SharedPreferences



- Persist- Data is retained even after the user quits the app and returns back.
- Three ways to make data persist.
- *SharedPreferences* class: provides access to data that can be accessed and edited by all Activity classes of an app.

*SharedPreferences prefs;*

- Initialize *prefs* object using the *getSharedPreferences* method with a *string* that will be used to refer to all the data read and written using this object.

*prefs = getSharedPreferences("My App", MODE\_PRIVATE);*

- *prefs* object is used to initialize the *editor* by invoking *edit* method

*SharedPreferences.Editor editor;*

*editor = prefs.edit();*

*editor.putString("username", username);*

*editor.commit();*

- *putString* method : save the *username* data, and the variable to store in, initiating the saving process.

# Reloading data with Shared Preferences

- Saved the data of application when closed and brought back after the app is opened again
- Code will reload the three values that the previous code saved.

```
String username = prefs.getString("username", "new user");
int age = prefs.getInt("age", -1);
boolean subscribed = prefs.getBoolean("newsletter-subscriber", false)
```

- The second parameter of the methods `getString`, `getInt`, and `getBoolean` has to initialized with default value.
- The default value will be returned, when there is no data stored with that label.

Android Intents Making the Note to Self settings persist



# Coding the SettingsActivity class

1. Add code to `SettingsActivity.java` file → most of the task is executed by this file
2. Add on some member variables for `SharedPreferences` and `Editor` instances.

```
private SharedPreferences mPrefs;  
private SharedPreferences.Editor mEditor;  
private boolean mShowDividers;
```

3. Import the `SharedPreferences` class:

```
import android.content.SharedPreferences;
```

4. initialize `mPrefs` and `mEditor`:

```
mPrefs = getSharedPreferences("Note to self", MODE_PRIVATE);  
mEditor = mPrefs.edit();
```

5. In the onCreate method, get a reference to *Switch* widget
6. Load the saved data that represents user's previous choice.

```
mShowDividers = mPrefs.getBoolean("dividers", true);  
  
Switch switch1 = findViewById(R.id.switch1);  
  
// Set the switch on or off as appropriate  
  
switch1.setChecked(mShowDividers);
```

5. Create and anonymous class to listen and handle the changes switch widget.
6. Set the prefs object and mShowDividers to ‘true’ → isChecked is true, false otherwise. (Refer to code listing from page 449 of the textbook)
7. When the user navigate to MainActivity or quits the app from the SettingsActivity screen, save the user’s setting and invoke the *onPause* method

### Coding the MainActivity class

Note: Please refer page no 449 from the textbook

More advanced persistence



## More advanced persistence

- As the app features advances with more data → required to save and load java objects including internal data
- The data objects are converted into bits and bytes to store on the disk → *Serialization*
- The stored bits and bytes are converted back to data objects when reloading the app → *De-Serialization*
- **JSON** class in Android hide the complexity Serialization process
- Frequently used to transmit data between web application and servers

# What is JSON?

- **JSON : JavaScript Object Notation**
- widely used in Android
- to send data between web applications and servers.
- To implement the code for Effectively handling the occurrences of failure scenario beyond the user control of an app → Eg: failure to load the data when the SD is corrupted
- JSON uses JAVA exceptions efficiently
- Handle Exceptions : notify user, retry etc;

# Backing up user data in Note to Self

- Import *JSONException* and *JSONObject* classes
- Load the number of variables of a Note object into *JSONObject* : ready for the serialization process
- To start the serialization process : invoke the put method with appropriate key and matching member variable.

*jo.put(JSON\_TITLE, mTitle);*

- Create a class *JSONSerializer*
- Declare the member variable to write onto the file
- Initialize the member variable with a *JSONSerializer* constructor.
- Create a *JSONArray object* : specialized *ArrayList* to handle the JSON objects
- *ConvertToJson* method : converts *JSONObject* instances to jarray
- Combined data is written into the file : using *Writer* instance and *OutputStream* instance

Databases 101& SQL syntax prime(Chapter 27)



# Database?

- A **database** : place of storage and a means to retrieve, store, and manipulate data.
- Internal structure of database varies greatly based on the database
- SQLite : DBMS used in Android → stores all its data in a single file.

## SQL

Structured Query Language → manipulate the data in the database

## SQL Lite

Android database system: posses own version of SQL → *SQL Syntax Primer*

# SQLite example code

Keywords are similar to JAVA:

1. INSERT: add data to the database
2. DELETE: remove data from the database
3. SELECT: read data from the database
4. WHERE: to specify matching criteria to access specific parts of the database, to use INSERT, DELETE, or SELECT on
5. FROM: to specify a table or column name in a database to access the data.

SQL has **types**:

- **integer**: for storing whole numbers
- **text**: for storing a simple name or address
- **real**: For large floating-point numbers

## Operations on database are:

- Creating a table : *Create table <table name>*
- Inserting data into the database : *Insert into <table name> values <values>*
- Retrieving data from the database :  
*select \* from <table name> where <condition>*
- Updating the database structure : *Alter table <Table name>*

# Android SQLite API

- Two fundamental classes to access the database API in Android
- *SQLiteDatabase* class : represents the actual database.
- *SQLiteOpenHelper* class: enable to get access to a database and initialize an instance of SQLiteDatabase
- SQLiteOpenHelper will extend in Age Database app: has to override two methods.
  1. onCreate method: called the first time a database is used.  
put SQL to create our table structure in.
  2. onUpgrade: is called when database structure is modified using ALTER .

# Building and executing queries : Database cursors

- Classes : enables to access the database
- Methods : enable the execution of queries.
- Cursor class: Return type of the database queries.
- The methods of the Cursor class : used to selectively access the data returned from the queries:
  - *Log.i(c.getString(1), c.getString(2));*
- Cursor object : determines the currently accessed row of returned data

# Coding the database class



## Coding the database class

- Create a class using the SQLiteOpenHelper class.
- It will also define some final strings to represent the names of the table and its columns.
- Consists of helper methods : to execute the queries.
- helper methods : return a Cursor object to display the retrieved data
- Create a new class called DataManager and add all required methods in it.(refer to code listing in page no: 680)
- Add a constructor to create an instance of custom version of SQLiteOpenHelper and initialize the db member.

```
db = helper.getWritableDatabase();
```

- Add the helper methods to access from Fragment class

- Add insert method defining the table attributes

```
public void insert(String name, String age)
```

- Execute the INSERT query based on the parameters

```
Log.i("insert() = ", query);  
db.execSQL(query);
```

- Add selectAll to the data manager class

```
public Cursor selectAll() {  
    Cursor c = db.rawQuery("SELECT *" + " from " +  
    TABLE_N_AND_A, null);  
    return c;}
```

- add a searchName method that has a String parameter for the name

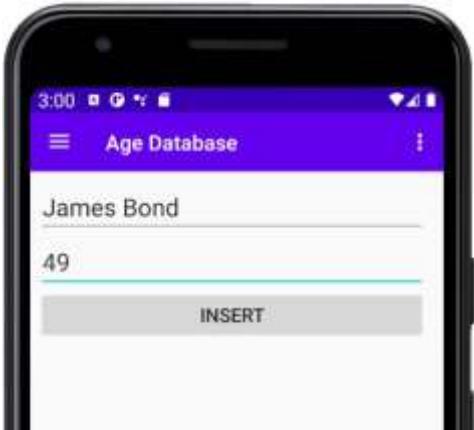
```
public Cursor searchName(String name)  
returns a Cursor instance that will contain all the entries  
Log.i("searchName() = ", query);  
Cursor c = db.rawQuery(query, null);
```

## Coding the Fragment classes to use the DataManager class

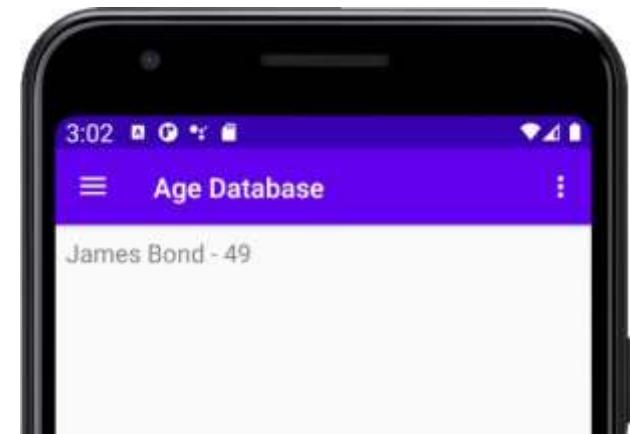
- Add code to the InsertFragment class to update the onCreateView method
- Create instance of DataManager class for ever Fragment classes : get a reference to all the UI widgets in the layout.
- onClick method: use the insert method to add new values to the database
- Values for insert method: passed in the EditText widget.
- Cursor instance : populated with data using SelectAll method
- Output the contents of the Cursor onto the TextView widget.

# Running the Age Database app

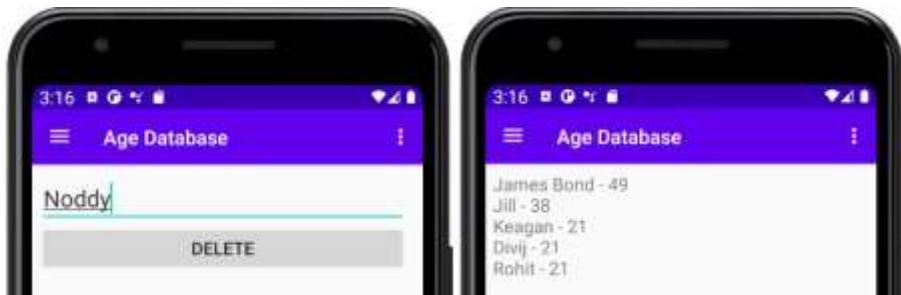
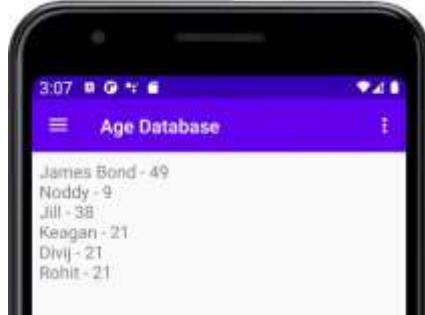
- Add a new name to the database using the **Insert** menu option:



Verify the execution is successful by viewing the **Results** option: →



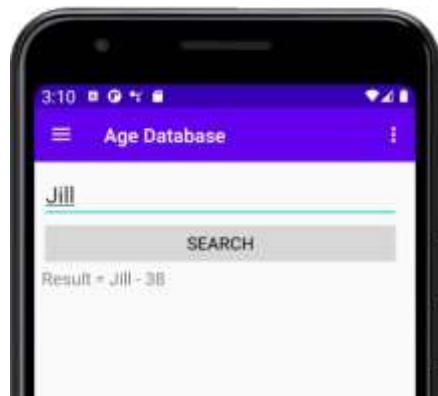
add a few more names and ages →



Use **Delete** menu option and analyse the output at the **Results** option



Search for existing data to test the **Search** menu option: →



Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 9  
**Why React and Why React Native**



# Contents

1. What is React?
2. React Features
3. What's new in React?
4. What is React Native?
5. React and JSX are familiar
6. The mobile browser experience
7. Android and iOS – different yet the same
8. The case for mobile web apps



# Weekly Learning Outcomes

1. Describe the advantages of React
2. Explain how Does React Native Work?



## Required Reading

1. Chapter 1 and Chapter 13 (React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3<sup>rd</sup> Edition, 2020, Adam Boduch, Roy Derks)

## Recommended Reading

React: <https://facebook.github.io/react>

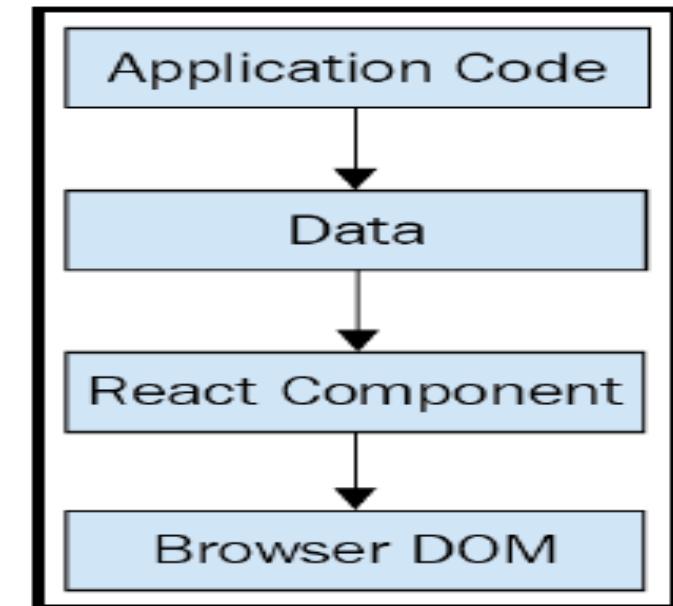


# What is React?



# What is React?

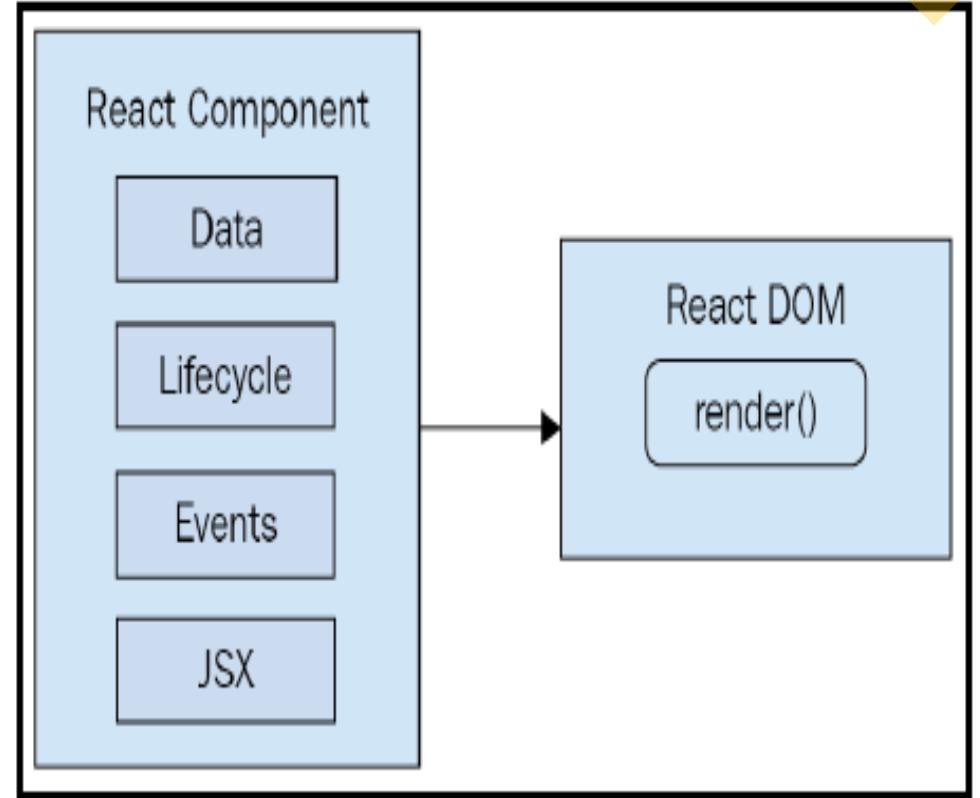
- "A JavaScript library for building user interfaces." It's a library for building user interfaces (UIs).
- React : view layer in an application.
- jQuery manipulates UI elements → Handlebars templates are inserted into the page.
- React components change what the user sees.
- Handles the job of getting HTML into the page to
- render data to UI
- Rendering technology
- Simplifies the application development



Place of React fits frontend code:

# Components of React

- **Two APIs** : React component API & React Dom
- **React component API** : Parts of the page that are rendered by React DOM
- **React DOM** : Performs the actually rendering on a web page.
- **Data** : Renders the data
- **Life Cycle** : Consists of methods and Hooks that respond to components entering and exiting phase
- **Events** : the code that we write for responding to user interactions.
- **JSX** : syntax of React components used to describe UI structures.



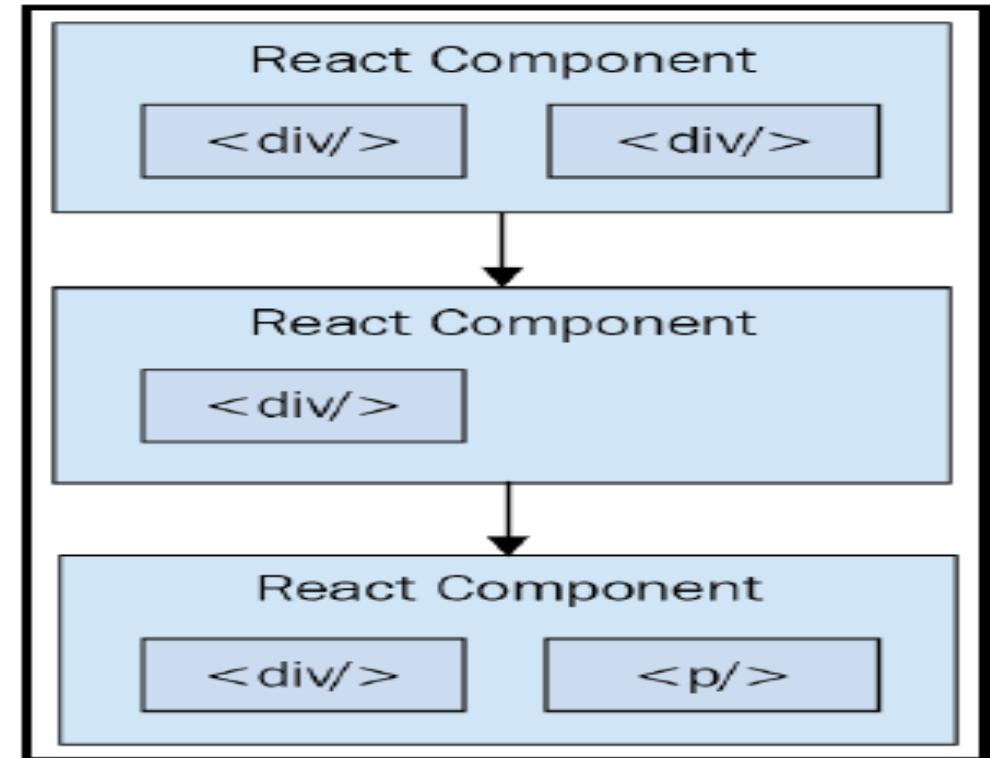
# Declarative UI structures

- This principle states that different concerns, such as logic and presentation, should be separate from one another.
- Syntax used by React components is called **JSX (JavaScript XML)**.
- A component renders content by returning some JSX.
- The JSX itself is usually HTML markup, mixed with custom tags for React components.
- Declarative JSX approach – Need not perform little micro-operations to change the content of a component.
- Imperative programming : Issue in UI development
  
- For example- jQuery to build the application
- to add a class to a paragraph when a button is clicked: called imperative programming

- React components don't require executing steps in an imperative way.
- JSX is central to React components.
- The XML-style syntax makes it easy to describe the structure of the UI.
- Called declarative programming and is very well suited for UI development.
- Once the UI structure is declared – specification of changes has to be given.

# Time and data

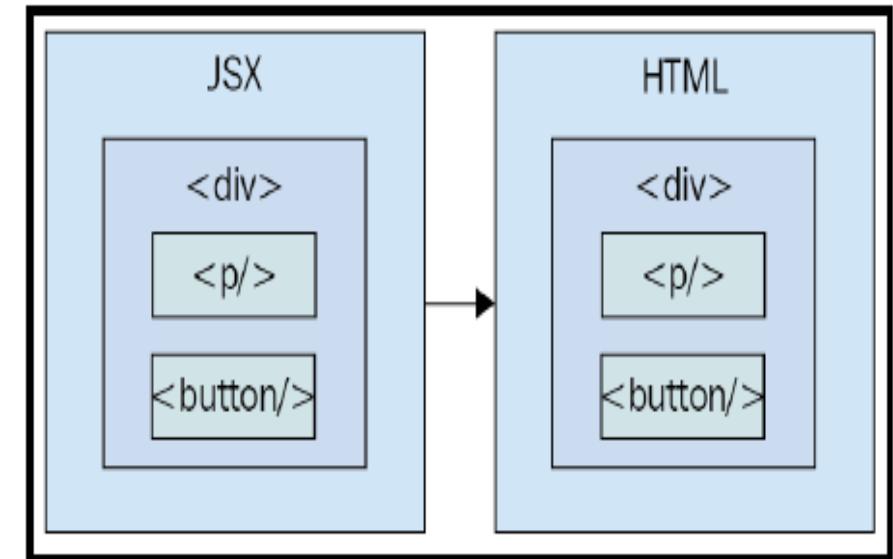
- React components rely on data being passed into them.
- This data represents the dynamic parts of the UI.
- For example, a UI element that's rendered based on a Boolean value could change the next time the component is rendered. Here's a diagram of the idea:



- Each time the React component is rendered, it's like taking a snapshot of the JSX at that exact moment in time.
- As the application moves forward through time, an ordered collection of rendered UI components will be obtained.
- In addition to declaratively describing what a UI should be, re-rendering the same JSX content makes things much easier for developers.
- The challenge is making sure that React can handle the performance demands of this approach.

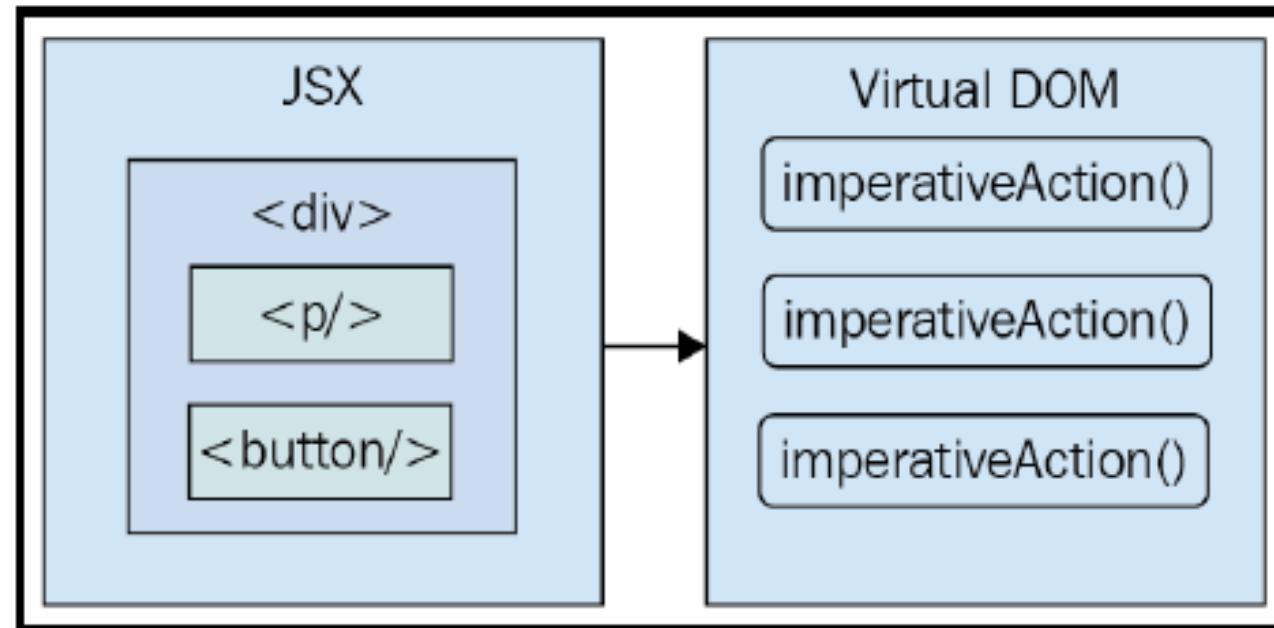
# Performance matters

- Using React to build UIs means that we can declare the structure of the UI with JSX.
- Less error-prone than the imperative approach of assembling the UI piece by piece.
- declarative approach does present a challenge: performance.
- For example, having a declarative UI structure is fine for the initial rendering, because there's nothing on the page yet. So, the React renderer can look at the structure declared in JSX and render it in the DOM browser.
- The **Document Object Model (DOM)** represents HTML in the browser after it has been rendered. The DOM API is how JavaScript is able to change content on the page.



- On the initial render, React components and their JSX are no different from other template libraries.
- For instance, Handlebars will render a template to HTML markup as a string, which is then inserted into the browser DOM.
- Where React is different from libraries such as Handlebars is when data changes, the components has to be re-rendered.
- Handlebars will just rebuild the entire HTML string, the same way it did on the initial render. Since this is problematic for performance, implementing imperative workarounds leads to manually update tiny bits of the DOM.
- Causes a tangled mess of declarative templates and imperative code to handle the dynamic aspects of the UI.
- This does not occur in React which differentiates React from other libraries.
- Components are declarative for the initial render, and they stay this way even as they're re-rendered. It's what React does under the hood that makes re-rendering declarative UI structures possible.

- React has something called the virtual DOM, which is used to keep a representation of the real DOM elements in memory.
- It does this so that each time we re-render a component, it can compare the new content to the content that's already displayed on the page.
- Based on the difference, the virtual DOM can execute the imperative steps necessary to make the changes. So, not only do we get to keep our declarative code when we need to update the UI, but React will also make sure that it's done in a performant way.
- Here's what this process looks like:

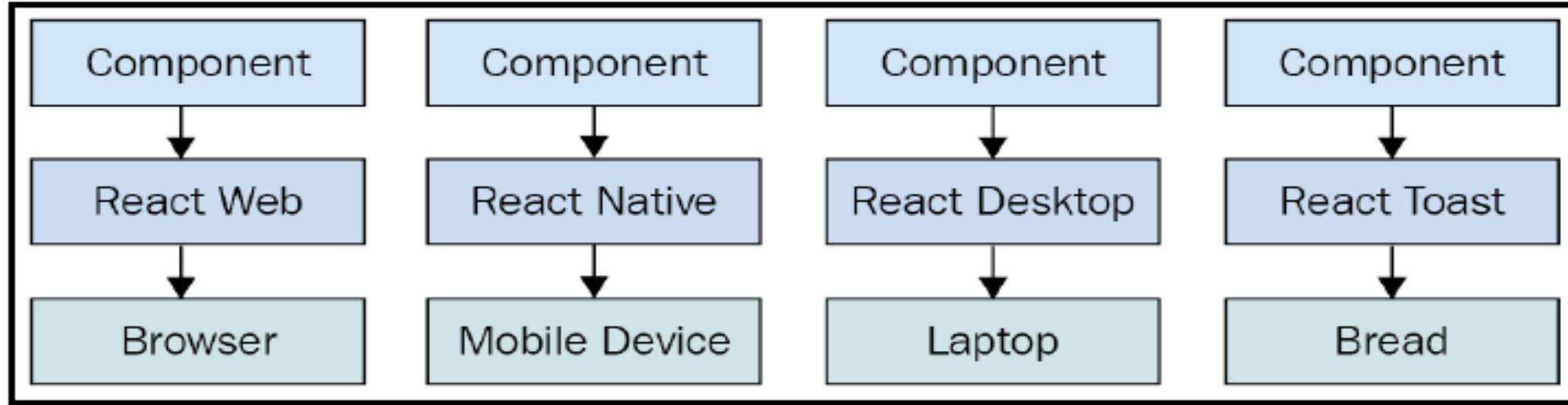


- Like any other JavaScript library, React is constrained by the run-to-completion nature of the main thread.
- For example, if the React internals are busy diffing content and patching the DOM, the browser can't respond to user input
- React should be flexible enough to adapt to different platforms where the app would be deployed in future.

# The Right level of abstraction

- React code is abstraction.
- JSX syntax translates to low-level operations that update the UI.
- The render target happens to be the browser DOM with React, but it isn't restricted to the browser DOM.
- It is not mandatory to know the target for react to translate the declarative UI Components
- React has the potential to be used for any UI to create, on any conceivable device.
- The abstraction level with React is at the right level, and it's in the right place.

# React targeting more than just the browser:



- ✓ From left to right, we have **React Web** (just plain React), **React Native**, **React Desktop**, and **React Toast**. Same pattern is applied to target something new :
  - Implement components specific to the target.
  - Implement a React renderer that can perform the platform-specific operations under the hood.
- ✓ React knowledge can be used to focus on describing the structure of the UI on any platform.

# React Features



# React Features

The features of React 16 include the following:

1. Revamped core architecture
2. Lifecycle methods
3. Context API
4. Rendering fragments
5. Portals
6. Rendering lists and strings
7. Handling errors
8. Server-side rendering

# Revamped core architecture:

- Biggest change in React 16 - change made to the internal reconciliation code.
- These changes don't impact the way that you interact with the React API.
- These changes were made to address some pain points that were preventing React from scaling up in certain situations.
- For example, one of the main concepts of this new architecture is that of fibers. Instead of rendering every component on the page in a run-to compilation way, React renders fibers—smaller chunks of the page that can be prioritized and rendered asynchronously.

# Lifecycle methods:

- React 16 had to revamp some of the lifecycle methods that are available to class components.
- Some lifecycle methods are deprecated and will eventually be removed because they will be problematic for future async rendering functionality in React.
- For example, a common way to initialize state in a React component is to use the `componentWillMount()` lifecycle method. Once this method is removed from React, the initial state directly can be set as an instance value.

# The Context API

React has provided an experimental Context API for developers

- Context is an alternative approach to passing data from one component to the next.
- For example, using properties, data can be passed through a tree of components that is several layers deep. The components in the middle of this tree don't actually use any of these properties—they're just acting as intermediaries. This becomes problematic as your application grows more properties in the sourceadd to the complexity.

# Rendering fragments:

- React component renders several sibling elements,
- Eg: three `<p>` elements: have to wrap them in `<div>`
- React would only allow components to return a single element.
- The only problem with this approach - leads to a lot of unnecessary DOM structure.

# Portals:

- When a React component returns content, it gets rendered into its parent component.
- Then, that parent's content gets rendered into its parent component and so on, all the way to the tree root.
- Render something that specifically targets a DOM element.
- For example, a component that should be rendered as a dialog probably doesn't need to be mounted at the parent.
- Using a portal can control precisely where the component's content is rendered.
- **Rendering lists and strings:** Similar to returning string value. Can just return a list of strings or a list of elements.

# Handling errors:

- Where exactly to handle errors?
- Error boundaries: created by implementing the `componentDidCatch()` lifecycle method in a component.
- Serve as the error boundary by wrapping other components.
- If any of the wrapped components throw an exception, the error boundary component can render alternative content.

# Server-side rendering (SSR):

- Server-side rendering with javascript libraries
- React : the server returns a ready to render HTML page
- JS scripts required to make the page interactive.
- The html is rendered immediately with all the static elements.
- Popular technique for rendering a client-side *single page application* (spa) on the server
- Then send a fully rendered page to the client.
- Allows for dynamic components to be served as static html markup.

# What's new in React?



## What's new in React?

- Memoizing functional components
- Code splitting and loading
- Hooks

# Memoizing functional components:

- The `react.Memo()` function is the modern equivalent of the `purecomponent` class.
- Memoized components avoid re-rendering if the component data hasn't changed.
- Automatically handle checking whether the component data has changed or not and whether or not the component should re-render.
- The challenge with this approach is that it is now common for large react applications to have a lot of functional components.
- Can pass functional components to `react.Memo()` and they'll behave like `purecomponent`.

# Code splitting and loading:

- Done by react.Lazy() function
- Code splitting : reduces the size of the code bundles that are sent to the browser, which can dramatically improve the user experience.
- Provides a huge efficiency gain.
- Code splitting and the user experience of waiting for pieces of the application to load are integral parts of the application.
- By combining react.Lazy() and the suspense component, fine-grained control over the app is split up is achieved.

# Hooks:

- Hooks—functions that extend the behavior of functional React components.
- Hooks are used to "hook into" the React component machinery from your React components.
- React Hooks API : pass functions to build components that have state or that rely on executing side effects when the component is mounted.

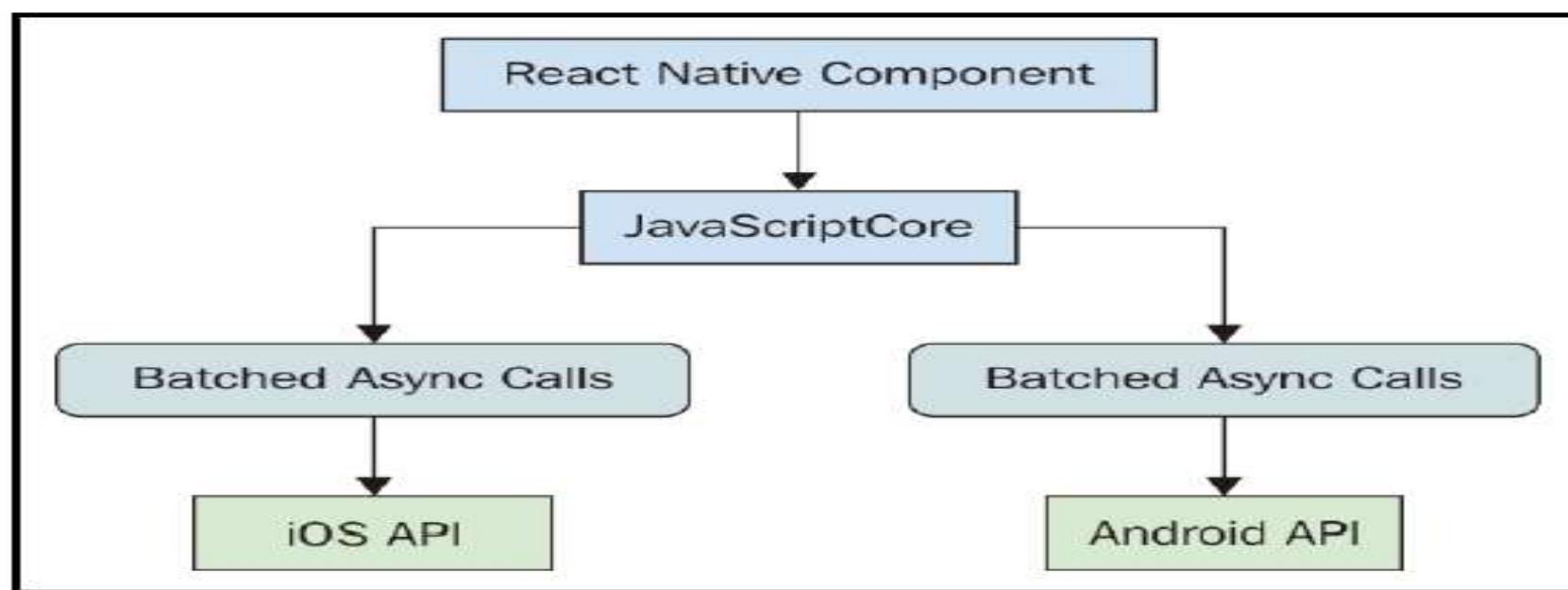
# What is React Native?

Chapter 13



# What is React Native?

- React Native uses a technique that makes asynchronous calls to the underlying mobile OS, which calls the native widget APIs.
- There's a JavaScript engine, and the React API is mostly the same as React for the web.
- The difference is with the target; instead of a DOM, there are asynchronous API calls.
- The concept is visualized here:



- The same React library that's used on the web is used by React Native and runs in **JavaScriptCore**.
- Messages that are sent to native platform APIs are asynchronous and batched for performance purposes.
- React Native ships with components implemented for mobile platforms, instead of components that are HTML elements.

React and JSX are familiar



# React and JSX are familiar

- Huge demand for mobile apps → mobile web browser is not that convenient as the native app experience.
- JSX is a eccentric tool for building uis.
- React is from a development-resource perspective.

# The mobile browser experience



- Mobile browsers lack many capabilities of mobile applications → cannot replicate the same native platform widgets as HTML elements.
- Often better to just use the native widget → requires less maintenance
- Using widgets that are native to the platform → consistent with the rest of the platform.
- For example:
  - ✓ A date picker in an application looks different from all the date pickers the user interacts with on their phone.
- Familiarity is key, and using native platform widgets makes familiarity possible.
- User interactions on mobile devices are fundamentally different from the interactions on the web application.

- Mobile platforms → a gesture system.
- React native is a much better candidate for handling gestures than react for the web
- React native use actual components from the platform → components of the app remain updated when the mobile platform is updated.
- **Consistency** and **familiarity** are key factors for good user experience.

Android and iOS – different yet the same



# Android and iOS – different yet the same

- React Native is not crossplatform solution that will run a single React application natively on any device.
- React Native enables web developers to create robust mobile applications using their existing JavaScript knowledge
- iOS and Android are different on many fundamental levels. Even their user experience philosophies are different, so trying to write a single app that runs on both platforms is categorically misguided.
- Goal of React Native:
  - ✓ Enables the app to take advantage of an iOS-specific widget or an Android-specific widget.
  - ✓ Provide a better user experience for that particular platform
  - ✓ Trump the portability of a component library.

- There are several areas that overlap between iOS and Android where the differences are trivial.
- The two widgets aim to accomplish the same thing for the user, in roughly the same way.
- React Native will handle the difference and provide a unified component.

# The case for mobile web apps



- Not every will be willing to install an app → Due to download count and rating.
- The barrier to entry is much lower with web applications—the user only needs a browser.
- Cannot replicate everything that native platform UIs can offer.
- A good web UI is the first step toward getting download counts and ratings up for the mobile app.
- Goal of a good mobile app:
  - ✓ Standard web (laptop/desktop browsers)
  - ✓ Mobile web (phone/tablet browsers)
  - ✓ Mobile apps (phone-/tablet-native platform)
- high demand for your mobile app compared to the web versions

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 10  
**Rendering with JSX**



# Contents

- 1.Your first JSX content
- 2.Rendering HTML
- 3.Describing UI structures
- 4.Creating your own JSX elements
- 5.Using JavaScript expressions
- 6.Fragments of JSX



# Weekly Learning Outcomes

1. Explain the basics of JSX, including its declarative structure, which leads to more maintainable code
2. Explain the Fragments of JSX



## Required Reading

1. Chapter 2 (React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3<sup>rd</sup> Edition, 2020, Adam Boduch, Roy Derks)

## Recommended Reading

Introducing JSX: <https://reactjs.org/docs/introducing-jsx.html>

Fragments: <https://reactjs.org/docs/fragments.html>



First JSX content



# First JSX content

- Implementing the obligatory "hello world" JSX application.

## Hello JSX

```
import React from 'react';
import { render } from 'react-dom';

render(
  <p>
    Hello, <strong>JSX</strong>
  </p>,
  document.getElementById('root')
);
```

# Basic Steps To Render JSX Onto The Page

- First : import the relevant pieces.
- The render() function takes JSX as the first argument and renders it to the DOM node passed as the second argument.
- The actual JSX content in this example renders a paragraph with some bold text inside.
- Markup is inserted into the DOM directly as a plain string.

# Declarative UI structures

- The JSX content was short and simple.
- JSX content is declarative → describes what to render.
- Declarative : structured and imperative as ordered.
- Easier to get things right with a structure than to perform steps in a specific order.
- The render() function tells React to take your JSX markup and transform it into JavaScript statements that update the UI in the most efficient way possible.
- Enables to declare the structure of UI without carrying out ordered steps to update elements on the screen.
- React supports the standard HTML tags.
- React has unique conventions that should be followed when using HTML tags.

# Rendering HTML



# Rendering HTML

- The job of a React component is to render HTML into the DOM browser.
- JSX has support for HTML tags out of the box.

## Built-in HTML tags

- When rendering JSX, element tags reference React components.
- React comes with HTML components.
- Any HTML tag can be rendered in JSX to get the expected output.

```
import React from 'react';
import { render } from 'react-dom';

render(
  <div>
    <button />
    <code />
    <input />
    <label />

    <p />
    <pre />
    <select />
    <table />
    <ul />
  </div>,
  document.getElementById('root')
);
```

<div> tag is surrounded by a group of all the other tags as its children → React needs a root component to render.

HTML elements rendered using JSX closely follow regular HTML element syntax with a few subtle differences regarding case sensitivity and attributes.

# HTML tag conventions

- Tag names are case sensitive and non-HTML elements are capitalized.
- Easy to scan the markup and spot the built-in HTML elements.
- Can also pass HTML elements any of their standard properties, a warning about the unknown property is logged.
- Here's an example that illustrates these ideas:

```
import React from 'react';
import { render } from 'react-dom';

render(
  <button title="My Button" foo="bar">
    My Button
  </button>,
  document.getElementById('root')
);

render(<Button />, document.getElementById('root'));
```

- This example will fail to compile because React doesn't know about the <Button> element; it only knows about <button>.
- any valid HTML tags can be used as JSX tags.
- HTML tags can be used to describe the structure of your page content.

# Describing UI structures



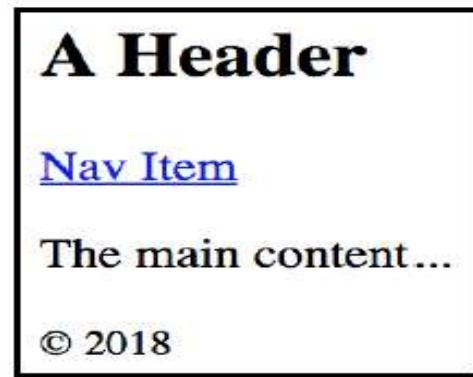
# Describing UI structures

- JSX is capable of describing screen elements in a way that ties them together to form a complete UI structure. Let's look at some JSX markup that declares a more elaborate structure than a single paragraph:

```
import React from 'react';
import { render } from 'react-dom';

render(
  <section>
    <header>
      <h1>A Header</h1>
    </header>
    <nav>
      <a href="item">Nav Item</a>
    </nav>
    <main>
      <p>The main content...</p>
    </main>
    <footer>
      <small>&copy; 2019</small>
    </footer>
  </section>,
  document.getElementById('root')
);
```

- This JSX markup describes some fairly sophisticated UI structure. Yet, it's easier to read than imperative code because it's XML, and XML is good for concisely expressing a hierarchical structure. This is how we want to think of our UI when it needs to change, not as an individual element or property.
- Here is what the rendered content looks like:



- There are a lot of semantic elements in this markup describing the structure of the UI.
- For example, the `<header>` element describes the top part of the page where the title is, and the `<main>` element describes where the main page content goes.
- This type of complex structure makes it clearer for developers to reason about. But before we start implementing dynamic JSX markup, let's create some of our own JSX components.

# Creating your own JSX elements



# Creating your own JSX elements

- Components are the fundamental building blocks of React.
- Components are the vocabulary of JSX markup.
- How to encapsulate HTML markup within a component.
- Build examples that nest custom JSX elements
- How to namespace the components.

# Encapsulating HTML

- Create new JSX elements to encapsulate larger structures.
- Custom tag can be used.
- The React component returns the JSX that goes where the tag is used.
- Let's look at the following example:

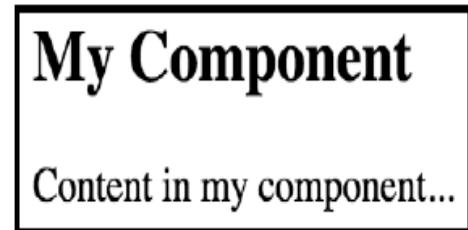
```
import React, { Component } from 'react';
import { render } from 'react-dom';

class MyComponent extends Component {

    render() {
        return (
            <section>
                <h1>My Component</h1>
                <p>Content in my component...</p>
            </section>
        );
    }
}

render(<MyComponent />, document.getElementById('root'));
```

Here's what the rendered output looks like:



- This is the first React component
- Create new JSX element → by creating a class called MyComponent, which extends the Component class from React.
- Rendering a <MyComponent> element.
- The HTML that this component encapsulates is returned by the render() method. In this case, when the JSX <MyComponent> is rendered by react-dom, it's replaced by a <section> element, and everything within it.
- In the preceding example, the MyComponent class was declared in the same scope as the call to render(), so everything worked as expected.
- Import components, adding them to the appropriate scope.
- HTML elements such as <div> often take nested child elements

# Nested elements

- JSX markup is useful for describing UI structures that have parent-child relationships.
- Child elements are created by nesting them within another component: the parent.
- For example, a `<li>` tag is only useful as the child of a `<ul>` tag or a `<ol>` tag
- Can make similar nested structures with your own React components.
- Use the `children` property.
- Here's the JSX markup:

```
import React from 'react';
import { render } from 'react-dom';

import MySection from './MySection';
import MyButton from './MyButton';

render(
  <MySection>
    <MyButton>My Button Text</MyButton>
  </MySection>,
  document.getElementById('root')
);
```

- Importing two of own React components: MySection and MyButton.
- <MyButton> is a child of <MySection>.
- The MyButton component accepts text as its child, instead of more JSX elements.

```
import React, { Component } from 'react';

export default class MySection extends Component {
  render() {
    return (
      <section>
        <h2>My Section</h2>
        {this.props.children}
      </section>
    );
  }
}
```

- This component renders a standard <section> HTML element, a heading, and then {this.props.children}. It's this last piece that allows components to access nested elements or text, and to render them.
- The two braces used in the preceding example are used for JavaScript expressions.
- More details of the JavaScript expression syntax found in JSX markup in the following section.
- Look at the MyButton component:

```
import React, { Component } from 'react';

export default class MyButton extends Component {
  render() {
    return <button>{this.props.children}</button>;
  }
}
```

- This component uses the exact same pattern as MySection; take the {this.props.children} value and surround it with markup. React handles the details.
- In this example, the button text is a child of MyButton, which is, in turn, a child of MySection.
- However, the button text is transparently passed through MySection.
- Not required to write any code in MySection to make sure that MyButton got its text .
- Here's what the rendered output looks like:



- Components can be organized by placing them within a namespace.

# Namespaced components

- A namespace provides an organizational unit for the components so that related components can share the same namespace prefix.
- Instead of writing `<MyComponent>` in JSX markup, it can be written as `<MyNamespace.MyComponent>`. This makes it clear that `MyComponent` is part of `MyNamespace`.
- `MyNamespace` would also be a component. The idea of namespacing is to have a namespace component render its child components using the namespace syntax.
- Let's take a look at an example:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

render(
  <MyComponent>
    <MyComponent.First />
    <MyComponent.Second />
  </MyComponent>,
  document.getElementById('root')
);
```

- This markup renders a <MyComponent> element with two children. Instead of writing <First>, it should be written as <MyComponent.First>, and the same with <MyComponent.Second>.
- First and Second belong to MyComponent within the markup.

```
import React, { Component } from 'react';

class First extends Component {
  render() {
    return <p>First...</p>;
  }
}

class Second extends Component {
  render() {
    return <p>Second...</p>;
  }
}

class MyComponent extends Component {
  render() {

    return <section>{this.props.children}</section>;
  }
}

MyComponent.First = First;
MyComponent.Second = Second;

export default MyComponent;

export { First, Second };
```

- This module declares MyComponent as well as the other components that fall under this namespace (First and Second).
- It assigns the components to the namespace component (MyComponent) as class properties. There are several things that could change in this module.
- For example, not required to directly export First and Second as it is accessible through MyComponent.
- Not required to define everything in the same module;
- Import First and Second as class properties.
- Using namespaces is completely optional consistently.
- Can control the HTML content that a given component renders and provide components with a namespace to avoid confusion.

# Using JavaScript expressions



# Using JavaScript expressions

- JSX has a special syntax that allows you to embed JavaScript expressions.
- Any time React renders JSX content, expressions in the markup are evaluated.
- This is the dynamic aspect of JSX,
- How to use expressions to set property values and element text content.
- How to map collections of data to JSX elements.

## Dynamic property values and text

- Some HTML property or text values are static → don't change as JSX markup is re-rendered.
- Other values, the values of properties or text, are based on data that is found elsewhere in the application. Remember, React is just the view layer.

```
import React from 'react';
import { render } from 'react-dom';

const enabled = false;
const text = 'A Button';
const placeholder = 'input value...';
const size = 50;

render(
  <section>
    <button disabled={!enabled}>{text}</button>
    <input placeholder={placeholder} size={size} />
  </section>,
  document.getElementById('root')
);
```

- Anything that is a valid JavaScript expression, including nested JSX, can go in between the braces: {}. For properties and text, this is often a variable name or object property. In this example, the !enabled expression computes a Boolean value.
- rendered output looks like:



## Mapping collections to elements

- The best way to dynamically control JSX elements is to map them from a collection. Let's look at an example of how this is done:

```
import React from 'react';
import { render } from 'react-dom';

const array = ['First', 'Second', 'Third'];

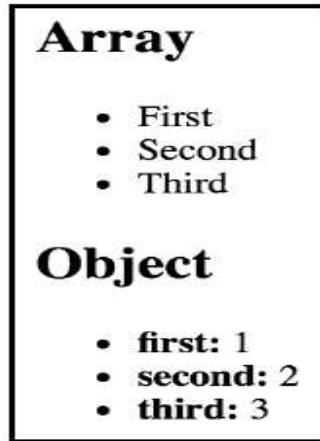
const object = {
  first: 1,
  second: 2,
  third: 3
};

render(
  <section>
    <h1>Array</h1>
    <ul>
      {array.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>

    <h1>Object</h1>
    <ul>
      {Object.keys(object).map(i => (
        <li key={i}>
          <strong>{i}: </strong>
          {object[i]}
        </li>
      ))}
    </ul>
  </section>,
  document.getElementById('root')
);
```

- The first collection is an array called array, populated with string values.
- Moving down to the JSX markup, you can see the call to array.map(), which returns a new array.
- The mapping function is actually returning a JSX element (<li>), meaning that each item in the array is now represented in the markup.
- The result of evaluating this expression is an array
- The object collection uses the same technique → call Object.keys() and then map this array.
- Can control the structure of React components based on the collected data.
- No need to rely on imperative logic to control the UI.

Here's what the rendered output looks like:



- JavaScript expressions bring JSX content to life.
- React evaluates expressions and updates the HTML content based on what has already been rendered and what has changed.
- Understanding how to utilize these expressions is important because they're one of the most common day-to-day activities of any React developer

# Fragments of JSX



# Fragments of JSX

- React 16 introduces the concept of JSX fragments.
- Fragments are a way to group together chunks of markup without having to add unnecessary structure to your page.
- For example, a common approach is to have a react component return content wrapped in a <div> element.
- This element serves no real purpose and adds clutter to the DOM.
- Two versions of a component:
  - ✓ One uses a wrapper element
  - ✓ Uses the new fragment feature:

```
import React from 'react';
import { render } from 'react-dom';

import WithoutFragments from './WithoutFragments';
import WithFragments from './WithFragments';

render(
  <div>
    <WithoutFragments />
    <WithFragments />
  </div>,
  document.getElementById('root')
);
```

- The two elements rendered are <WithoutFragments> and <WithFragments>. Here's what they look like when rendered:



Let's compare the two approaches now.

## Using wrapper elements

The first approach is to wrap sibling elements in <div>. Here's what the source looks like:

```
import React, { Component } from 'react';

class WithoutFragments extends Component {
  render() {
    return (
      <div>
        <h1>Without Fragments</h1>
        <p>
          Adds an extra <code>div</code> element.
        </p>
      </div>
    );
  }
}

export default WithoutFragments;
```

- The essence of this component is the `<h1>` and `<p>` tags.
- to return them from `render()` → wrap them with `<div>`.
- inspecting the DOM using browser dev tools reveals that `<div>` add another level of structure:

```
▼<div>
  <h1>Without Fragments</h1>
  ▼<p>
    "Adds an extra "
    <code>div</code>
    " element."
  </p>
</div>
```

## Using fragments

- Now, let's take a look at the WithFragments component, where we have avoided using unnecessary tags:

```
import React, { Component, Fragment } from 'react';

class WithFragments extends Component {
  render() {
    return (
      <Fragment>
        <h1>With Fragments</h1>
        <p>Doesn't have any unused DOM elements.</p>
      </Fragment>
    );
  }
}

export default WithFragments;
```

- Instead of wrapping the component content in <div>, the <Fragment> element is used.
- This is a special type of element that indicates that only its children need to be rendered.



- With the advent of fragments in JSX markup → less HTML rendered on the page
- No need to use tags such as <div> for the sole purpose of grouping elements together.
- when a component renders a fragment, React knows to render the event's child element wherever the component is used.

Thank You



IT448  
Mobile Application Development  
Week 11  
Component Properties, State, and Context





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

# Contents

- 1.What is component state?
- 2.What are component properties?
- 3.Setting a component state
- 4.Passing property values
- 5.Stateless components
- 6.Container components
- 7.Providing and consuming context



# Weekly Learning Outcomes

1. Explain the state and properties in React components
2. Describe the concept of a container component



## Required Reading

1. Chapter 3 (React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3<sup>rd</sup> Edition, 2020, Adam Boduch, Roy Derks)

## Recommended Reading

Instance Properties: <https://reactjs.org/docs/react-component.html#instance-properties-1>

Setting the Initial State: <https://reactjs.org/docs/react-without-es6.html#setting-the-initial-state>

Context: <https://reactjs.org/docs/context.html>

Spread syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)

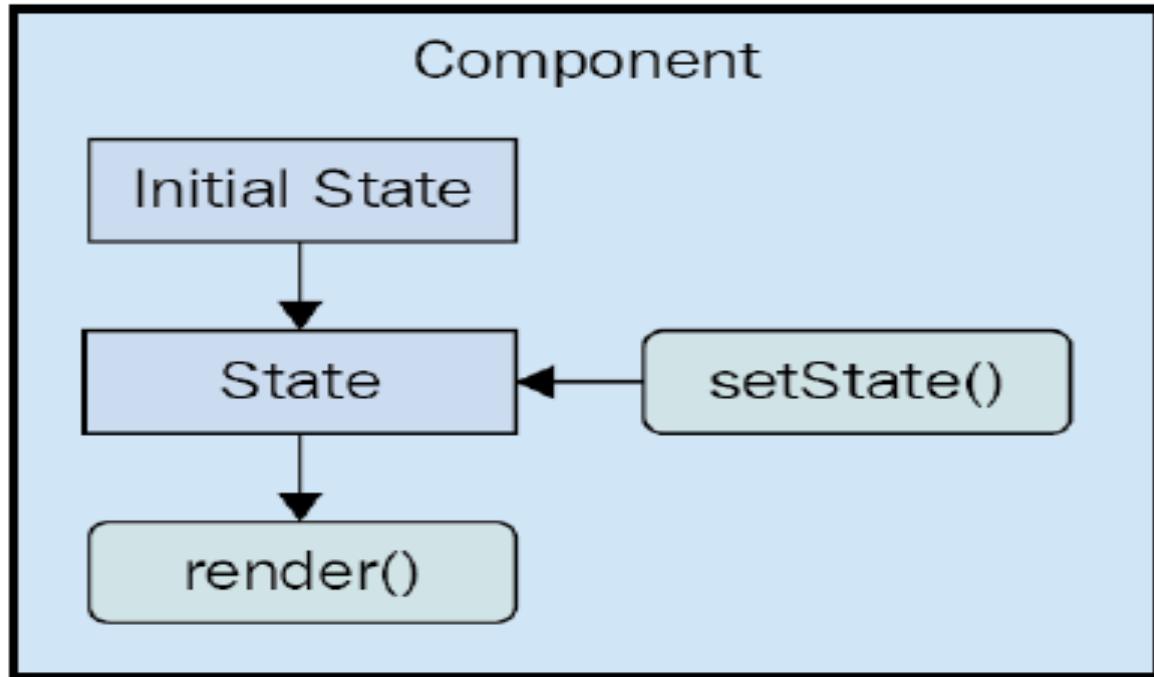


What is component state?



# What is component state?

- React components declare the structure of UI elements using JSX.
- Components need data to be useful.
- For example, a component JSX declare `<ul>` that maps a JavaScript collection to `<li>` elements.
- **State** is the dynamic part of a React component. It declares the initial state of a component, which changes over time.
- Ex: rendering a component where a piece of its state is initialized to an empty array.
- Later on, this array is populated with data using `setState()`. This is called a **change in state**,
- whenever a React component needs to change its state, the component will automatically re-render itself, calling `render()`.
- The process is visualized here:



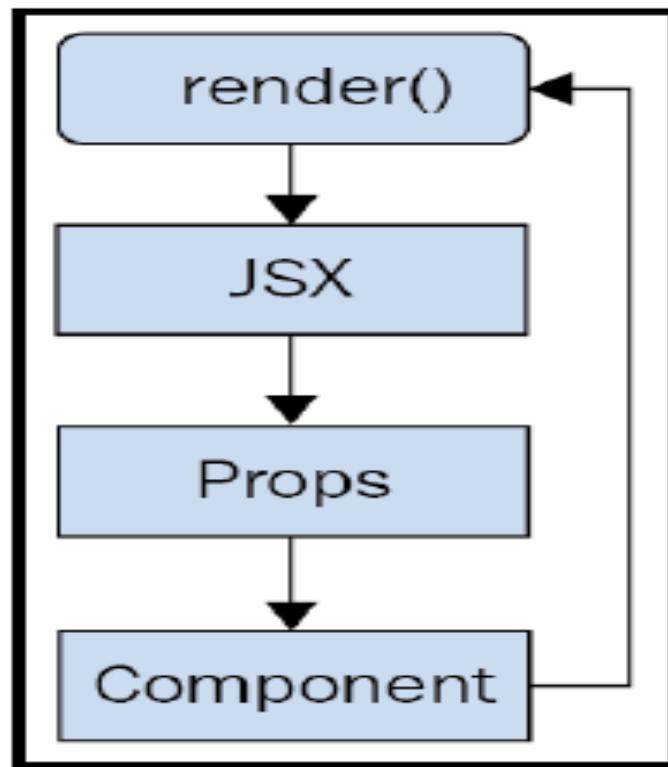
- The state of a component is something that either the component itself can set, or other pieces of code, outside of the component.

What are component properties?



## What are component properties?

- Properties are used to pass data into your React components.
- Instead of calling a method with a new state as the argument, properties are passed only when the component is rendered. That is, you pass property values to JSX elements.
- In the context of JSX, properties are called attributes, probably because that's what they're called in XML parlance.
- Properties are different than state because they don't change after the initial render of the component.
- If a property value has changed, to re-render the initial component, then re-render the JSX that was used to render it in the first place.
- The React internals take care of making sure this is done efficiently.
- Here's a diagram of rendering and re-rendering a component using properties:



- Different than a stateful component. The real difference is that with properties, it's often a parent component that decides when to render the JSX.
- The component doesn't actually re-render itself.

# Setting a component state



## Setting a component state

- React code that sets the state of components.
- The initial state—that is, the default state of a component.
- How to change the state of a component→causing it to re-render itself.
- How to merge a new state with an existing state.?

## Setting an initial component state

- The initial state of a component not explicitly set.
- The initial state of a component will always be an object with one or more properties.

- For example, a component that uses a single array as its state : the initial array is set as a property of the state object.
- Every React component uses a plain object as its state.
- Here's a component that sets an initial state object:

```
import React, { Component } from 'react';

export default class MyComponent extends Component {
  state = {
    first: false,
    second: true
  };

  render() {
    const { first, second } = this.state;

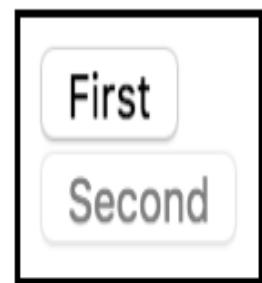
    return (
      <main>
        <section>
          <button disabled={first}>First</button>
        </section>
        <section>
          <button disabled={second}>Second</button>
        </section>
        ...
    );
  }
}
```

- In the JSX that's returned by render(), the state values that this component depends on—first and second.
- These properties are set up in the initial state.
- For example, When this component is rendered only once, and it would render as expected to the initial state set in MyComponent in the preceding code listing:

```
import React from 'react';
import { render } from 'react-dom';
import MyComponent from './MyComponent';

render(<MyComponent />, document.getElementById('root'));
```

Here's what the rendered output looks like:



## Creating a component state

- Render a component with initial state and update its state. This means that the component will be rendered twice.
- Component:

```
import React, { Component } from 'react';

export default class MyComponent extends Component {
  state = {
    heading: 'React Awesomesauce (Busy)',
    content: 'Loading...'
  };

  render() {
    const { heading, content } = this.state;

    return (
      <main>
        <h1>{heading}</h1>
        <p>{content}</p>
      </main>
    );
  }
}
```

- The JSX of this component depends on two state values—heading and content.
- The component also sets the initial values of these two state values, which means that it can be rendered without any unexpected "gotchas."
- Code that renders the component and then re-renders it by changing the state:

```
import React from 'react';
import { render } from 'react-dom';

import MyComponent from './MyComponent';

const myComponent = render(<MyComponent />,
document.getElementById('root'));

setTimeout(() => {
  myComponent.setState({
    heading: 'React Awesomesauce',
    content: 'Done!'
  );
}, 3000);
```

- The component is first rendered with its default state.
- Spot in this code is the `setTimeout()` call. After 3 seconds, it uses `setState()` to change the two state property values. This change will be reflected in the UI.
- The initial state when rendered:



- The rendered output after the change of state:



## Merging the component state

- When the state of a React component is set using `setState()`, the state of the component is merged with the object.
- Can set part of the component state while leaving the rest of the state as it is.
- Implement a component that has some initial state set on it:

```
import React, { Component } from 'react';

export default class MyComponent extends Component {
  state = {
    first: 'loading...',
    second: 'loading...',
    third: 'loading...',
    fourth: 'loading...',
    doneMessage: 'finished!'
  };

  render() {
    const { state } = this;

    return (
      <ul>
        {Object.keys(state)
          .filter(key => key !== 'doneMessage')
          .map(key => (
            <li key={key}>
              <strong>{key}: </strong>
              {state[key]}
            </li>
          )))
      </ul>
    );
  }
}
```

- This component renders the keys and values of its state—except for doneMessage.
- Code that sets the state of each state property individually:

```
import React from 'react';
import { render } from 'react-dom';
import MyComponent from './MyComponent';

const myComponent = render(<MyComponent />,
document.getElementById('root'));

setTimeout(() => {
  myComponent.setState({ first: 'done!' });
}, 1000);

setTimeout(() => {
  myComponent.setState({ second: 'done!' });
}, 2000);

setTimeout(() => {
  myComponent.setState({ third: 'done!' });
}, 3000);

setTimeout(() => {
  myComponent.setState(state => ({
    ...state,
    fourth: state.doneMessage
  }));
}, 4000);
```

- The takeaway from this example is that individual state properties on components can be set. It will efficiently re-render itself.
- Here's what the rendered output looks like for the initial component state:

- **first:** loading...
- **second:** loading...
- **third:** loading...
- **fourth:** loading...

- Here's what the output looks like after three of the setTimeout() callbacks have run:

- **first:** done!
- **second:** done!
- **third:** done!
- **fourth:** finished!

# Passing property values



# Passing property values

- Properties are like state data that gets passed into components.
- Properties are different from state in that they're only set once, which is when the component is rendered.

## Default property values

- Default property values are set as a class attribute called **defaultProps**.
- Let's take a look at a component that declares default property values:

```
import React, { Component } from 'react';

export default class MyButton extends Component {
  static defaultProps = {
    disabled: false,
    text: 'My Button'
  };

  render() {
    const { disabled, text } = this.props;

    return <button disabled={disabled}>{text}</button>;
  }
}
```

- Let's go ahead and render this component without any properties, to make sure that the defaultProps values are used:

```
import React from 'react';
import { render } from 'react-dom';
import MyButton from './MyButton';

render(<MyButton />, document.getElementById('root'));
```

- In this example, the MyButton component renders a <button> element using the default disabled and text property values.

# Stateless components



# Stateless components

- How to set default property values for stateless functional components.

## Pure functional components

- A functional React component—a function.
- This method is the component.
- The job of a functional React component is to return JSX, just like a class based React component.
- It has no state and no lifecycle methods.
- A pure function is a function without side effects.
- This is relevant for React components because, given a set of properties, it's easier to predict what the rendered content will be.
- Functions that always return the same value with a given argument values are easier to test as well.

- Let's look at a functional component now:

```
import React from 'react';

export default ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
);
```

- This function returns a `<button>` element, using the properties passed in as arguments (instead of accessing them through `this.props`).
- This function is pure because the same content is rendered if the same disabled and text property values are passed.

- Now, let's see how to render this component:

```
import React from 'react';
import { render as renderJSX } from 'react-dom';
import MyButton from './MyButton';

function render({ first, second }) {
  renderJSX(
    <main>
      <MyButton text={first.text} disabled={first.disabled} />
      <MyButton text={second.text} disabled={second.disabled} />
    </main>,
    document.getElementById('root')
  );
}

render({
  first: {
    text: 'First Button',
    disabled: false
  },
  second: {
    text: 'Second Button',
    disabled: true
  }
});
```

- There's zero difference between the class-based and function-based React components, from a JSX point of view.
- The JSX looks exactly the same whether the component was declared using the class or function syntax.
- Here's what the rendered HTML looks like:



- Functional components rely on property values being passed to them for anything dynamic.

## Defaults in functional components

- Functional components are lightweight; they don't have any state or lifecycle.
- They support some metadata options.
- For example, the default property values of functional components can be specified in the same way as with a class component.
- Here's an example of what this looks like:

```
import React from 'react';

const MyButton = ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
);

MyButton.defaultProps = {
  text: 'My Button',
  disabled: false
};

export default MyButton;
```

# Container components



## Container components

- This is a common React pattern, and it brings together many of the concepts that you've learned about state and properties.
- The basic premise of container components is simple: don't couple data fetching with the component that renders the data.
- The container is responsible for fetching the data and passing it to its child component.
- It contains the component responsible for rendering the data.
- Some level of substitutability can be achieved with this pattern.
- For example, a container could substitute its child component, or a child component could be used in a different container.
- Let's look at the container pattern in action, starting with the container itself:

```
import React, { Component } from 'react';
import MyList from './MyList';

function fetchData() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(['First', 'Second', 'Third']);
    }, 2000);
  });
}

export default class MyContainer extends Component {
  state = { items: [] };

  componentDidMount() {
    fetchData().then(items => this.setState({ items }));
  }

  render() {
    return <MyList {...this.state} />;
  }
}
```

- The job of this component is to fetch data and to set its state. Any time the state is set, `render()` is called. This is where the child component comes in.
- The state of the container is passed to the `MyList` component as properties.

- Let's take a look at the `MyList` component next:

```
import React from 'react';

export default ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    )))
  </ul>
);
```

- `MyList` is a functional component that expects an `items` property.
- Let's see how the container component is actually used:

```
import React from 'react';
import { render } from 'react-dom';
import MyContainer from './MyContainer';

render(<MyContainer />, document.getElementById('root'));
```

## Providing and consuming context



## Providing and consuming context

- For data that needs to make its way to any component in an app, it can be created and used a context.
- There are two key concepts to remember when using contexts in React—providers and consumers.
- A context provider creates data and makes sure that it's available to any React component.
- A context consumer is a component that uses this data within the context.
- Context similar to global data in a React application.
- React approach to wrap components with a context works better than creating global data
- Have better control of data flows down through the components.
- For example, can have nested contexts and a number of other advanced use cases.

- Consider some application data that determines permissions for given application features.
- Starting at the very top of the component tree, index.js:

```
import React from 'react';
import { render } from 'react-dom';
import { PermissionProvider } from './PermissionContext';
import App from './App';

render(
  <PermissionProvider>
    <App />
  </PermissionProvider>,
  document.getElementById('root')
);
```

- The <App> component is the child of the <PermissionProvider> component. This means that the permission context has been provided to the <App> component and any of its children, all the way down the tree.

- Let's take a look at the `PermissionContext.js` module where the permission context is defined:

```
import React, { Component, createContext } from 'react';

const { Provider, Consumer } = createContext('permissions');

export class PermissionProvider extends Component {
  state = {
    first: true,
    second: false,
    third: true
  };

  render() {
    return (
      <Provider value={this.state}>{this.props.children}</Provider>
    );
  }
}

const PermissionConsumer = ({ name, children }) => (
  <Consumer>{value => value[name] && children}</Consumer>
);

export { PermissionConsumer };
```

- The `createContext()` function is used to create the actual context. The return value is an object containing two components—`Provider` and `Consumer`.

- Now let's look at the App component:

```
import React, { Fragment } from 'react';
import First from './First';
import Second from './Second';
import Third from './Third';

export default () => (
  <Fragment>
    <First />
    <Second />
    <Third />
  </Fragment>
);
```

- This component renders three components that are features and each needs to check for permissions.

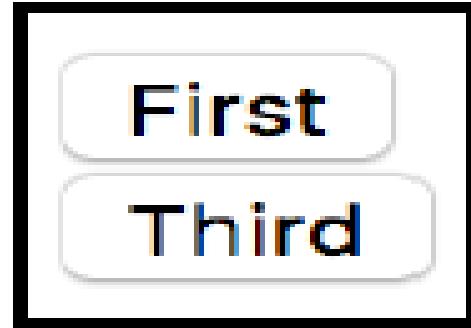
- Now let's take a look at the <First> component (<Second> and <Third> components are almost exactly the same):

```
import React from 'react';
import { PermissionConsumer } from './PermissionContext';

export default () => (
  <PermissionConsumer name="first">
    <div>
      <button>First</button>
    </div>
  </PermissionConsumer>
);
```

- This is where the PermissionConsumer component is put to use.

- Here's what the rendered output of these three components looks like:



- The second component isn't rendered because its permission in the PermissionProvider component is set to false.

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 12  
**The React Component Life Cycle**



# Contents

1. Why components need a life cycle
2. Initializing properties and state
3. Optimizing rendering efficiency
4. Rendering imperative components
5. Cleaning up after components
6. Containing errors with error boundaries



# Weekly Learning Outcomes

1. Explain the life cycle of React components
2. Explain how to write code that responds to life cycle events



## Required Reading

1. Chapter 7 (React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3<sup>rd</sup> Edition, 2020, Adam Boduch, Roy Derks)

## Recommended Reading

React.Component: <https://reactjs.org/docs/react-component.html>

State and Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>



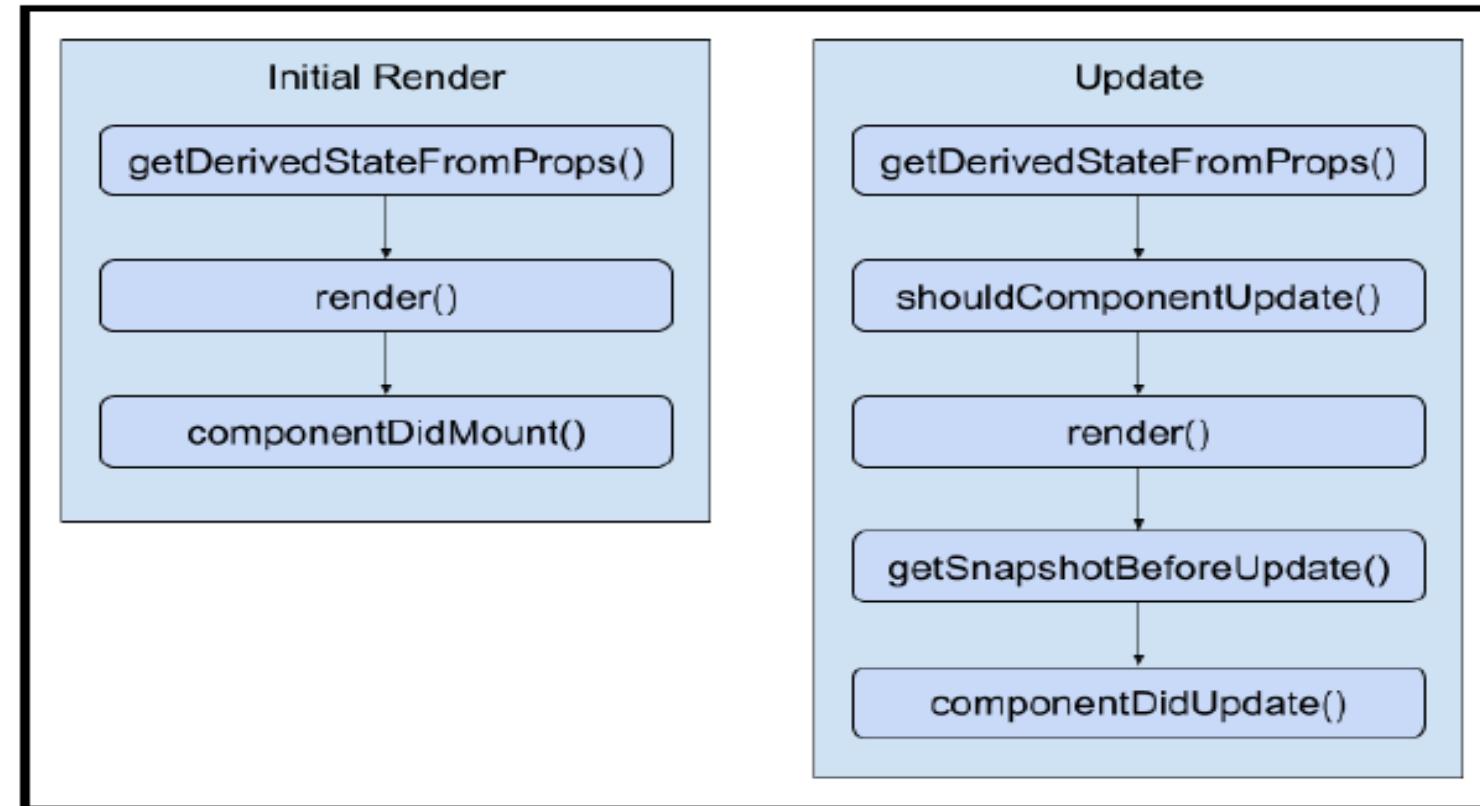
# Why components need a life cycle



## Why components need a life cycle

- React components go through a life cycle.
- render() method that is implemented in components is a life cycle method.
- Rendering is just one life cycle event in a React component.
- For example, there are life cycle events for when the component is mounted to the DOM, when the component is updated.
- Life cycle events are moving part.
- Some components do need to respond to life cycle events such as :
  - perform initialization → render heuristics → unmounted from the DOM → handle errors thrown by the component. → clean up the component

- The following diagram gives an idea of how a component flows through its life cycle, calling the corresponding methods in turn:



These are the two main life cycle flows of a React component. The first happens when the component is initially rendered. The second happens whenever the component is updated.

- Here's a rough overview of each of the methods:
1. `getDerivedStateFromProps()`: This method allows to update the state of the component based on the property values of the component. This method is called when the component is initially rendered and when it receives new property values.
  2. `render()`: Returns the content to be rendered by the component. This is called when the component is first mounted to the DOM, when it receives new property values, and when `setState()` is called.
  3. `componentDidMount()`: This is called after the component is mounted to the DOM. This is where component is initialized such as fetching data.
  4. `shouldComponentUpdate()`: this method is used to compare new states or props with current states or props. returns false if there's no need to re-render the component. This method is used to make the components more efficient.

5. `getSnapshotBeforeUpdate()`: This method is to perform operations directly on the DOM elements of the component before they're actually committed to the DOM. The difference between this method and `render()` is that `getSnapshotBeforeUpdate()` is not asynchronous. With `render()`, there's a good chance that the DOM structure could change between when it's called and when the changes are actually made in the DOM.
6. `componentDidUpdate()`: This is called when the component is updated.
7. `componentWillUnmount()`: This is the only life cycle method that's called when a component is about to be removed.

# Initializing properties and state



## Initializing properties and state

- Implementing the initialization code in React components.
- This involves using life cycle methods that are called when the component is first created.
- Implementing a basic example that sets the component up with data from the API.
- How the state can be initialized from properties.
- How the state can be updated as properties change.

## Fetching component data

- When the components are initialized, it should be populated with their state or properties.
- Otherwise, the component won't have anything to render other than its skeleton markup.
- For instance, to render the following user list component:

```
import React from "react";

const ErrorMessage = ({ error }) => (error ? <strong>{error}</strong> : null);

const LoadingMessage = ({ loading }) => (loading ? <em>{loading}</em> : null);

export default ({ error, loading, users }) => (
  <section>
    <ErrorMessage error={error} />
    <LoadingMessage loading={loading} />
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  </section>
);
```

There are three pieces of data that this JSX relies on, as follows:

- `loading` : This message is displayed while fetching API data.
- `error`: This message is displayed if something goes wrong.
- `users`: Data that's fetched from the API.

- There are two helper components being used here: **ErrorMessage** and **LoadingMessage**. They're used to format the error and the loading states, respectively. If error or loading is null, nothing is rendered. Otherwise, an error or loading message is rendered by the respective component.
- How should we go about making the API call and using the response to populate the users collection? The answer is to use a container component that makes the API call and then renders the **UserList** component:

```
import React, { Component } from "react";
import { users } from "./api";
import UserList from "./UserList";

export default class UserListContainer extends Component {
  state = {
    error: null,
    loading: "loading...",
    users: []
  };

  componentDidMount() {
    users().then(
      result => {
        this.setState({ loading: null, error: null, users: result.users });
      },
    );
  }
}
```

```
        error => {
          this.setState({ loading: null, error });
        }
      );
    }

    render() {
      return <UserList {...this.state} />;
    }
  }
}
```

- The actual API call happens in the `componentDidMount()` method. This method is called after the component is mounted into the DOM.
- Once the API call returns with data, the `users` collection is populated, causing the `UserList` to re-render itself, only this time, it has the data it needs.

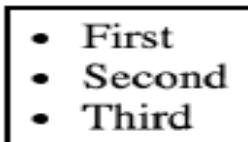
- Let's take a look at the users() mock API function call being used here:

```
export function users(fail) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (fail) {
        reject("epic fail");
      } else {
        resolve({
          users: [
            { id: 0, name: "First" },
            { id: 1, name: "Second" },
            { id: 2, name: "Third" }
          ]
        });
      }
    }, 2000);
  });
}
```

- Here's what the UserList component renders when the loading state is a string, and the users state is an empty array:

*loading...*

- Here's what it renders when loading is null and users is non-empty:



## Initializing state with properties

- The preceding example shows how to initialize the state of a container component by making an API call in the `componentDidMount()` life cycle method.
- However, the only populated part of the component state was the `users` collection.
- For example, the error and loading state messages have default values set when the state is initialized. This is great, but what if the code that is rendering `UserListContainer` wants to use a different loading message? This can be achieved by allowing properties to override the default state.
- Let's build on the `UserListContainer` component:

```
import React, { Component } from "react";
import { users } from "./api";
import UserList from "./UserList";

export default class UserListContainer extends Component {
  state = {
    error: null,
    users: []
  };
}
```

```
componentDidMount () {
  users().then(
    result => {
      this.setState({ error: null, users: result.users });
    },
    error => {
      this.setState({ loading: null, error });
    }
  );
}

render () {
  return <UserList {...this.state} />;
}

static getDerivedStateFromProps (props, state) {
  return {
    ...state,
    loading: state.users.length === 0 ? props.loading : null
  };
}
}

UserListContainer.defaultProps = {
  loading: "loading..."
};
```

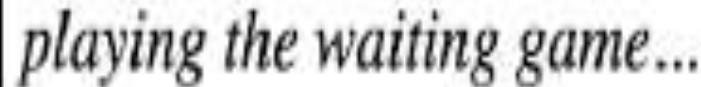
- The loading property no longer has a default string value. Instead, defaultProps provides default values for properties.
- The new life cycle method is `getDerivedStateFromProps()`. It uses the loading property to set the loading state. Since the loading property has a default value, it's safe to just change the state. The method is called before the component mounts and on subsequent re-renders of the component.

- Let's see how we can pass state data to UserListContainer now:

```
import React from "react";
import { render } from "react-dom";
import UserListContainer from "./UserListContainer";

render(
  <UserListContainer loading="playing the waiting game..." />,
  document.getElementById("root")
);
```

- Here's what the initial loading message looks like when UserList is first rendered:



*playing the waiting game...*

# Optimizing rendering efficiency



# Optimizing rendering efficiency

- Heuristics that improve the component rendering performance.
- Implementing a component with specific metadata from the API to determine if the component needs to be re-rendered or not.

## To render or not to render

- The **shouldComponentUpdate()** life cycle method determines if the component needs to be re-rendered.
- If this method returns false as output, the entire life cycle of the component would not be re-rendered.
- Important to check the frequency of rendering the component and the amount of data rendered by the component.
- Helps to identify if there is a change in the state of the component.

- a simple list component:

```
import React, { Component } from "react";

function referenceEquality(arr1, arr2) {
  return arr1 === arr2;
}

function valueEquality(arr1, arr2) {
  for (let i = 0; i < arr1.length; i++) {
    if (arr1[i] !== arr2[i]) {
      return false;
    }
  }
  return true;
}

export default class MyList extends Component {
  state = {
    items: new Array(5000).fill(null).map((v, i) => i)
  };

  shouldComponentUpdate(props, state) {
    if (!referenceEquality(this.state.items, state.items)) {
      return !valueEquality(this.state.items, state.items);
    }

    return false;
  }

  render() {
    return (
      <ul>
        {this.state.items.map(item => (
          <li key={item}>{item}</li>
        )))
      </ul>
    );
  }
}
```

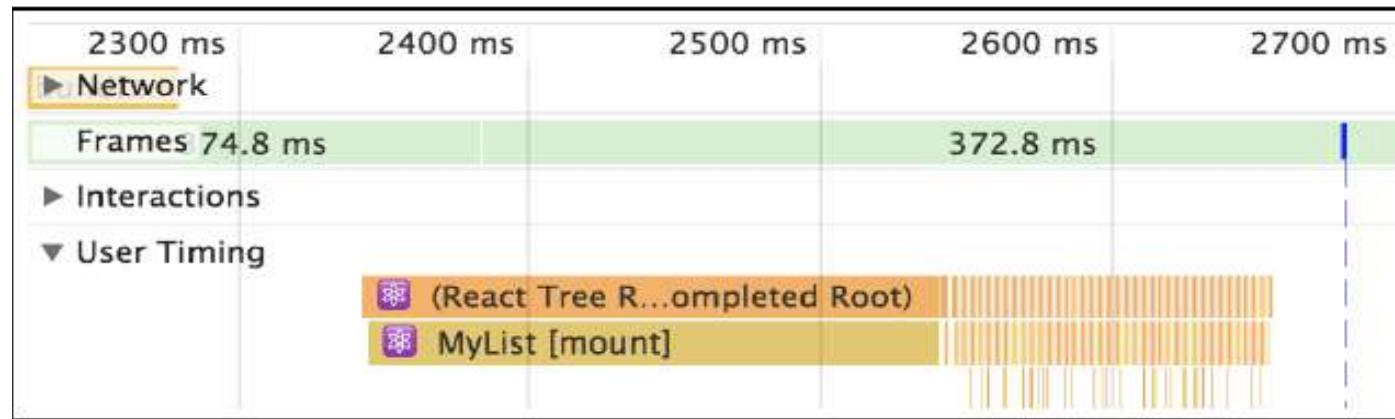
- It compares the new state with the current state with the help of two utility functions:
- **referenceEquality()**: Returns true if two arguments are the same reference. This is an extremely fast check to perform.
- **valueEqulity()**: Returns true if the two array values are the same. This isn't quite as fast because it needs to iterate over the whole array, but it's still faster than the virtual DOM.

```
import React from "react";
import { render as renderJSX } from "react-dom";
import MyList from "./MyList";

function render() {
  const myList = renderJSX(<MyList />, document.getElementById("root"));
  myList.data = myList.setState(state => ({
    items: [0, ...state.items.slice(1)]
  }));
}

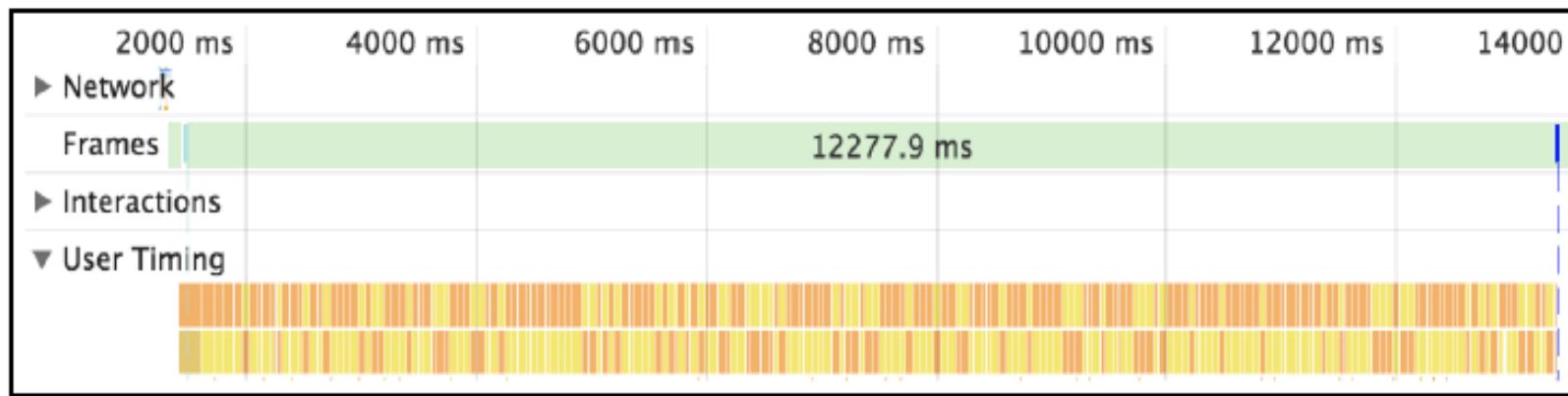
for (let i = 0; i < 100; i++) {
  render();
}
```

- Rendering <MyList>, over and over, in a loop. Each iteration has 5,000 list items to render. As there is no change in the state, the call to `shouldComponentUpdate()` returns false for every iteration.
- performance profile for this component:



- The initial render takes the longest—a few hundred milliseconds. But then you have all of these tiny time slices that are completely imperceptible to the user experience. These are the result of `shouldComponentUpdate()` returning false.

- Profile changes of the component:



- Without `shouldComponentUpdate()`, the end result is much larger time slices with a drastically negative impact on user experience

## Using metadata to optimize rendering

- Use of metadata that's part of the API response to determine whether or not the component should re-render itself. Here's a simple user details component:

```
import React, { Component } from "react";

export default class MyUser extends Component {
  state = {
    modified: new Date(),
    first: "First",
    last: "Last"
  };

  shouldComponentUpdate(props, state) {
    return Number(state.modified) > Number(this.state.modified);
  }

  render() {
    const { modified, first, last } = this.state;

    return (
      <section>
        <p>{modified.toLocaleString()}</p>
        <p>{first}</p>
        <p>{last}</p>
      </section>
    );
  }
}
```

## heuristic in action:

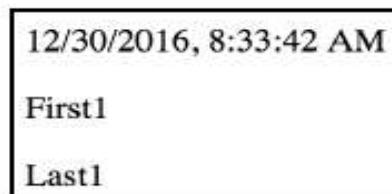
```
import React from "react";
import { render } from "react-dom";
import MyUser from "./MyUser";

const myUser = render(<MyUser />, document.getElementById("root"));

myUser.setState({
  modified: new Date(),
  first: "First1",
  last: "Last1"
});

myUser.setState({
  first: "First2",
  last: "Last2"
});
```

- The MyUser component is now entirely dependent on the modified state. If it's not greater than the previous modified value, no render happens.
- Here's what the component looks like after it's been rendered twice:



# Rendering imperative components



## Rendering imperative components

- At times React components need to implement some imperative code under the covers.
- Hiding the imperative operations : code can render the component without access it.
- Implementing a simple jQuery UI button React component to understand the relevant life cycle methods to encapsulate the imperative code.

## Rendering jQuery UI widgets

- The jQuery UI widget library implements several widgets on top of standard HTML.
- It uses a progressive enhancement technique.
- To enable these widgets: render HTML into the DOM → make imperative function calls to create and interact with the widgets.

- Create a React button component that acts as a wrapper around the jQuery UI widget.

```
import React, { Component } from "react";
import $ from "jquery";
import "jquery-ui/ui/widgets/button";
import "jquery-ui/themes/base/all.css";

export default class MyButton extends Component {
  componentDidMount() {
    $(this.button).button(this.props);
  }

  componentDidUpdate() {
    $(this.button).button("option", this.props);
  }

  render() {
    return (
      <button
        onClick={this.props.onClick}
        ref={button => {
          this.button = button;
        }}
      />
    );
  }
}
```

- In the `componentDidMount()` method, the `button()` function is called and passes its properties from the component.
- The `componentDidUpdate()` method executes similar and is called when property values change.

```
import React, { Component } from "react";
import MyButton from "./MyButton";

export default class MyButtonContainer extends Component {
  componentDidMount() {
    this.setState({ ...this.props, onClick: this.props.onClick.bind(this) });
  }

  render() {
    return <MyButton {...this.state} />;
  }
}

MyButtonContainer.defaultProps = {
  onClick: () => {}
};
```

- Container component controls the state, that passed to `<MyButton>` as properties.

- The component has a default onClick() handler function.

```
import React from "react";
import { render } from "react-dom";
import MyButtonContainer from "./MyButtonContainer";

function onClick() {
  this.setState({ disabled: true });
}

render(
  <section>
    <MyButtonContainer label="Text" />
    <MyButtonContainer
      label="My Button"
      icon="ui-icon-person"
      showLabel={false}
    />
    <MyButtonContainer label="Disable Me" onClick={onClick} />
  </section>,
  document.getElementById("root")
);
```

- There are three jQuery UI button widgets, each controlled by a React component with no imperative code.



# Cleaning up after components



## Cleaning up after components

- how to clean up after components: Done by React.
- `componentWillUnmount()` life cycle method.
- One use case for cleaning up after React components is asynchronous code.
- Component that issues an API call to fetch some data when the component is first mounted.
- This component is removed from the DOM before the API response arrives.

# Cleaning up asynchronous calls

- Asynchronous code cannot set the state of a component that has been unmounted
- Only warning is logged to handle the subtle race condition bugs.
- Create cancellable asynchronous actions as given in the below example.

```
import { Promise } from "bluebird";

Promise.config({ cancellation: true });

export function users(fail) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (fail) {
        reject(fail);
      } else {
        resolve({
          users: [
            { id: 0, name: "First" },
            { id: 1, name: "Second" },
            { id: 2, name: "Third" }
          ]
        });
      }
    }, 4000);
  });
}
```

- `users()` returns a promise from the Bluebird library that's been configured to have cancellable behaviour.
- Container component: has the ability to cancel asynchronous behaviour:

```
import React, { Component } from "react";
import { render } from "react-dom";
import { users } from "./api";
import UserList from "./UserList";

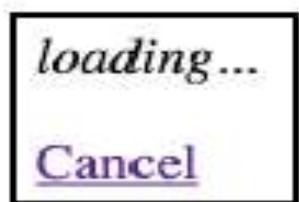
const onClickCancel = e => {
  e.preventDefault();
  render(<p>Cancelled</p>, document.getElementById("root"));
};

export default class UserListContainer extends Component {
  state = {
    error: null,
    loading: "loading...",
    users: []
  };

  componentDidMount() {
    this.job = users();
  }
}
```

```
this.job.then(  
  result => {  
    this.setState({ loading: null, error: null, users: result.users });  
  },  
  
  error => {  
    this.setState({ loading: null, error: error.message });  
  }  
);  
  
componentWillUnmount () {  
  this.job.cancel();  
}  
  
render () {  
  return <UserList onClickCancel={onClickCancel} {...this.state} />;  
}  
}
```

- Output of a component when rendered during a pending API call:



# Containing errors with error boundaries



# Containing errors with error boundaries

- A new feature of React 16 – error boundaries – to handle unexpected component failures.
- Error boundaries: mechanism used to wrap components with error-handling behaviour.
- Error boundaries similar to try/catch syntax for JSX.
  - Revisiting the first example that fetched component data using an API function. Error to be handled: The users() function accepts a Boolean argument, which, when true, causes the promise to reject.
- UserListContainer and UserList components : set up to handle API errors.
- Challenge: with a greater number of components there is an increase in the number of error-handling code.

# Modified Source For Userlistcontainer

```
import React, { Component } from "react";
import { users } from "./api";
import UserList from "./UserList";

export default class UserListContainer extends Component {
  state = {
    error: null,
    loading: "loading...",
    users: []
  };

  componentDidMount() {
    users(false).then(
      result => {
        this.setState({ loading: null, error: null, users: result.users });
      },
      error => {
        this.setState({ loading: null, error });
      }
    );
  }

  render() {
    if (this.state.error !== null) {
      throw new Error(this.state.error);
    }
    return <UserList {...this.state} />;
  }
}
```

- This component is mostly the same as it was in the first example. The first difference is the call to users(), where it's now passing true:

```
componentDidMount () {  
  users(true).then (  
    ...  
  );  
}
```

- This call will fail, resulting in the error state being set. The second difference is in the render() method:

```
if (this.state.error !== null) {  
  throw new Error(this.state.error);  
}
```

- Instead of forwarding the error state onto the UserList component, it's passing the error back to the component tree by throwing an error instead of attempting to render more components.
- The key design change here is that this component is now making the assumption that there is some sort of error boundary in place further up in the component tree that will handle these errors accordingly.

## creating the error boundary itself:

```
import React, { Component } from "react";

export default class ErrorBoundary extends Component {
  state = {
    error: null
  };

  componentDidCatch(error) {
    this.setState({ error });
  }

  render() {
    if (this.state.error === null) {
      return this.props.children;
    } else {
      return <strong>{this.state.error.toString()}</strong>;
    }
  }
}
```

- `componentDidCatch()` life cycle method is utilized by setting the error state of this component when it catches an error. When it's rendered, an error message is rendered if the error state is set. Otherwise, it renders the child components as usual.

- Using ErrorBoundary component:

```
import React from "react";
import { render } from "react-dom";
import ErrorBoundary from "./ErrorBoundary";
import UserListContainer from "./UserListContainer";

render(
  <ErrorBoundary>
    <UserListContainer />
  </ErrorBoundary>,
  document.getElementById("root")
);
```

- Any errors that are thrown by `UserListContainer` or any of its children will be caught and handled by `ErrorBoundary`:

The screenshot shows a browser's developer tools console with the 'Console' tab selected. The title bar says 'Error: epic fail'. The console output is as follows:

```
✖ ► Uncaught Error: epic fail
    at UserListContainer.render (UserListContainer.js:59)
    at finishClassComponent (react-dom.development.js:13085)
    at updateClassComponent (react-dom.development.js:13047)
    at beginWork (react-dom.development.js:13715)
    at performUnitOfWork (react-dom.development.js:15741)
    at workLoop (react-dom.development.js:15780)
    at HTMLUnknownElement.callCallback (react-dom.development.js:100)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:138)
    at invokeGuardedCallback (react-dom.development.js:187)
    at replayUnitOfWork (react-dom.development.js:15194)
    at renderRoot (react-dom.development.js:15840)
    at performWorkOnRoot (react-dom.development.js:16437)
    at performWork (react-dom.development.js:16358)
    at performSyncWork (react-dom.development.js:16330)
    at requestWork (react-dom.development.js:16230)
    at scheduleWork$1 (react-dom.development.js:16096)
    at Object.enqueueSetState (react-dom.development.js:11185)
    at
UserListContainer.../node_modules/react/cjs/react.development.js.Component.setState (react.development.js:273)
    at UserListContainer.set (UserListContainer.js:23)
    at UserListContainer.js:46

✖ ► The above error occurred in the <UserListContainer> component:
    in UserListContainer (at index.js:12)
    in ErrorBoundary (at index.js:11)

React will try to recreate this component tree from scratch using the
error boundary you provided, ErrorBoundary.
```

- The argument that's passed to users() in UserListContainer is removed to stop it from failing.

```
import React from "react";

const LoadingMessage = ({ loading }) => (loading ? <em>{loading}</em> : null);

export default ({ error, loading, users }) => (
  <section>
    <LoadingMessage loading={loading} />
    <ul>
      {users.map(user => (
        <li key={user.id.toUpperCase()}>{user.name}</li>
      ))}
    </ul>
  </section>
);
```

- a different error is thrown, but since it's under the same boundary as the previous error, it'll be handled the same way:

The screenshot shows a browser developer tools window with the "Console" tab selected. A red error message is displayed: "TypeError: i.id.toUpperCase is not a function". Below the message is a stack trace pointing to line 23 of "UserList.js". The stack trace is as follows:

```

Uncaught TypeError: i.id.toUpperCase is not a function
    at UserList.js:23
    at Array.map (<anonymous>)
    at ./src/UserList.js._webpack_exports_.a (UserList.js:23)
    at updateFunctionalComponent (react-dom.development.js:13017)
    at beginWork (react-dom.development.js:13713)
    at performUnitOfWork (react-dom.development.js:15741)
    at workLoop (react-dom.development.js:15780)
    at HTMLUnknownElement.callCallback (react-dom.development.js:100)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:138)
    at invokeGuardedCallback (react-dom.development.js:187)
    at replayUnitOfWork (react-dom.development.js:15194)
    at renderRoot (react-dom.development.js:15840)
    at performWorkOnRoot (react-dom.development.js:16437)
    at performWork (react-dom.development.js:16358)

```

The code snippet from "UserList.js" is shown on the left, with line 23 highlighted:

```

TypeError: i.id.toUpperCase is not a function

(anonymous function)
src/UserList.js:23

20 |     /* Attempts to render the user
list but throws an
21 |         error by attempting to call
toUpperCase() on a number. */
22 |     <ul>
> 23 |         {users.map(i => <li key=
{i.id.toUpperCase()}>{i.name}</li>)}
24 |     </ul>
25 |     </section>
26 | );

```

Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 13

# **Kick-Starting React Native Projects and Building Responsive Layouts with Flexbox**



# Contents

1. Installing and using the Expo command-line tool
2. Viewing your app on your phone
3. Viewing your app on Expo Snack
4. Flexbox is the new layout standard
5. Introducing React Native styles
6. Building Flexbox layouts



# Weekly Learning Outcomes

1. Explain how to kick-start a React Native project using the Expo command-line tool
2. Explain how to use the Flexbox model to lay out our React Native screens



## Required Reading

1. Chapter 14 and Chapter 15 (React and React Native:  
A complete hands-on guide to modern web and  
mobile development with React.js, 3<sup>rd</sup> Edition, 2020,  
Adam Boduch, Roy Derks)

## Recommended Reading

Layout with Flexbox: <https://facebook.github.io/react-native/docs/Flexbox>



# Installing and using the Expo command-line tool



# Installing and using the Expo command-line tool

- The Expo command-line tool is the preferred way start React Native project.
- Handles the creation of the project framework that is required to run a basic React Native application.
- Installing the Expo command-line tool:
  1. type in the following command prompt:  
**npm install -g expo-cli**
  2. Once the installation is complete, new expo command will be available on the system.
  3. To start a new project, run the expo init command, as follows:  
**expo init my-project**

3. The following details appear on the terminal to enter the details of the project:

**? Choose a template: (Use arrow keys)**

**----- Managed workflow -----**

**> blank a minimal app as clean as an empty canvas**

**blank (TypeScript) same as blank but with TypeScript configuration**

**tabs several example screens and tabs using react-navigation**

**----- Bare workflow -----**

**minimal bare and minimal, just the essentials to get you started**

**minimal (TypeScript) same as minimal but with TypeScript configuration**

- The default selection is “**blank Managed workflow**” . This enables to use the Expo tools and services during the development of the application compatible on different mobile devices.

4. The human friendly name, ,icon name of the app has to be entered in the following step.

? Please enter a few initial configuration values.

Read more:

<https://docs.expo.io/versions/latest/workflow/configuration/> » 50%

Completed

```
{  
  "expo": {  
    "name": "<The name of your app visible on the home screen>",  
    "slug": "my-project"  
  }  
}
```

5. The icon name of the app is replaced in the placeholder and the project configuration is updated accordingly completing the Expo project creation process.

**Extracting project files...**

**Customizing project...**

**Installing dependencies...**

**Your project is ready at /path/to/my-project**

- Now, a blank React Native project is created,
- Next → how to launch the Expo development server on the system and view the app on one of the mobile devices.

# Viewing your app on your phone



# Viewing your app on your phone

Step 1: start the Expo development server:

1. In the command-line terminal, enter in the project directory:

**cd path/to/my-project**

2. Run the following command to start the development server:

**npm start**

3. Some information about the developer server is displayed on the terminal:

**Starting project at C:\Users\adamb\React-and-React-Native---Third-Edition\Chapter13\my-project**

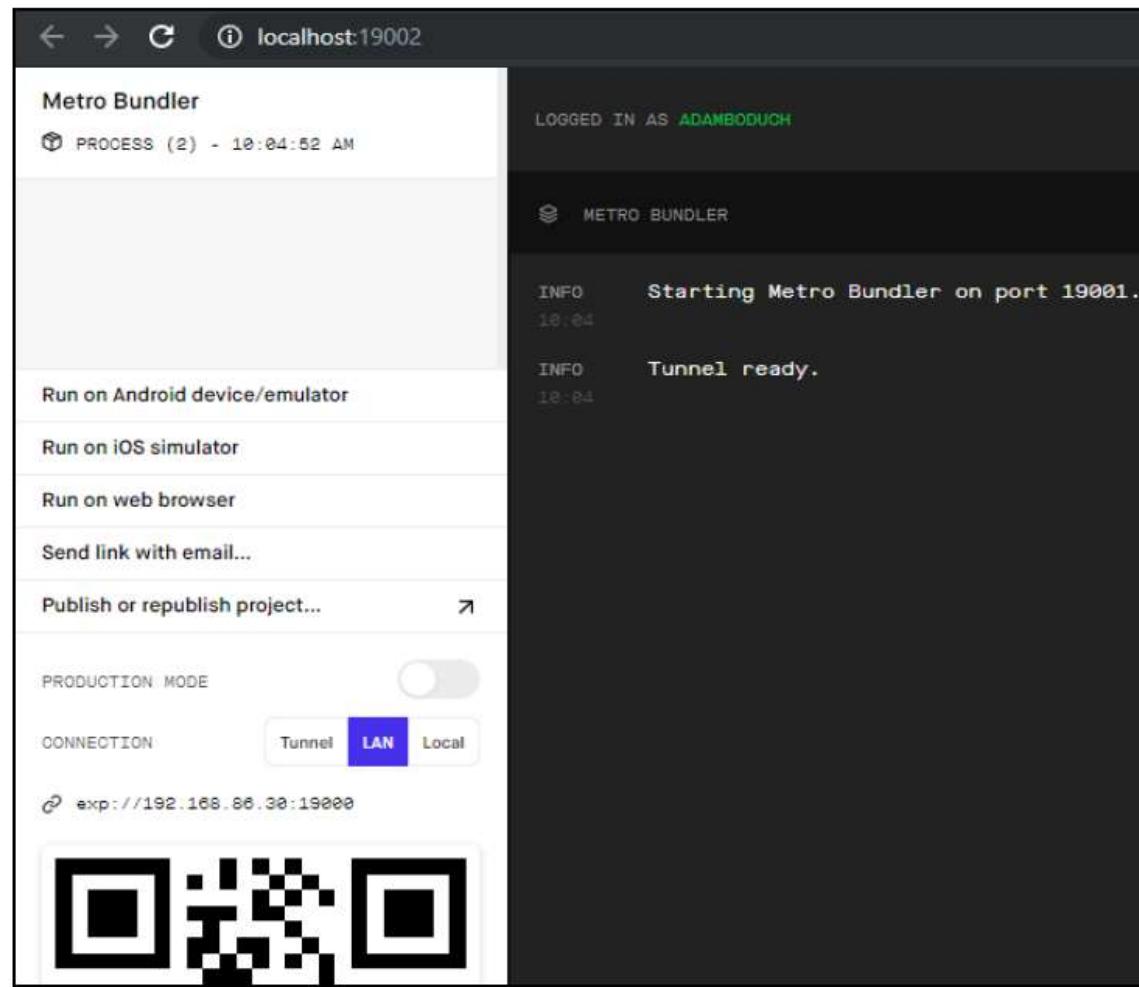
**Expo DevTools is running at http://localhost:19002**

**Opening DevTools in the browser... (press shift-d to disable)**

**Starting Metro Bundler on port 19001.**

**Tunnel ready.**

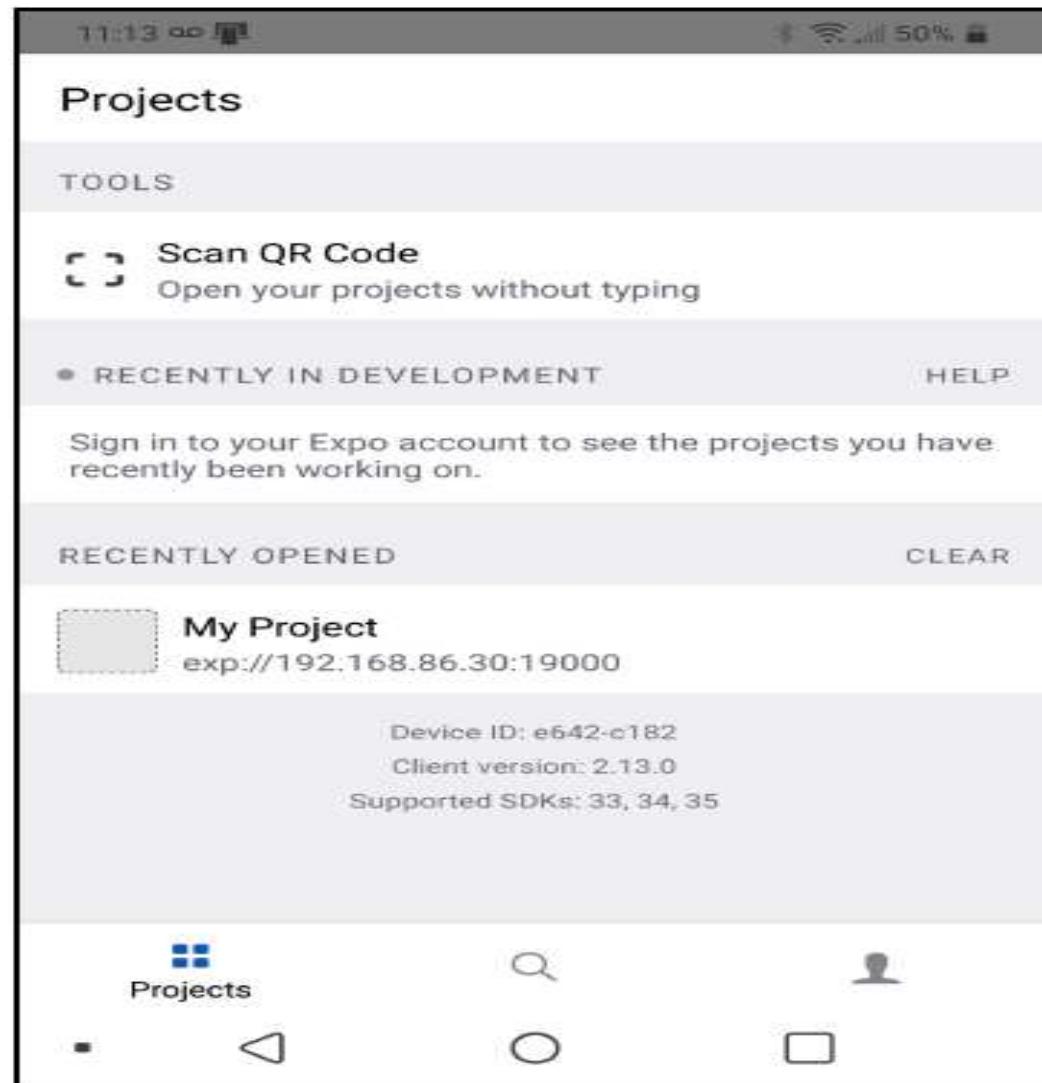
4. It will also open a browser tab with a UI for managing where the application is run, viewing logs, and other miscellaneous activities. Here is what the Expo app looks like:



- The logs of the process that bundles the React Native code and send it to an emulator is displayed on the right side of the screen.
- A QR code that you can scan with the camera on the device appears on the bottom left side.

5. Install the Expo app from play store or app store on the device to view the app on the devices.
6. Then click on the **Scan QR Code** button:

This will open the camera app on the device.

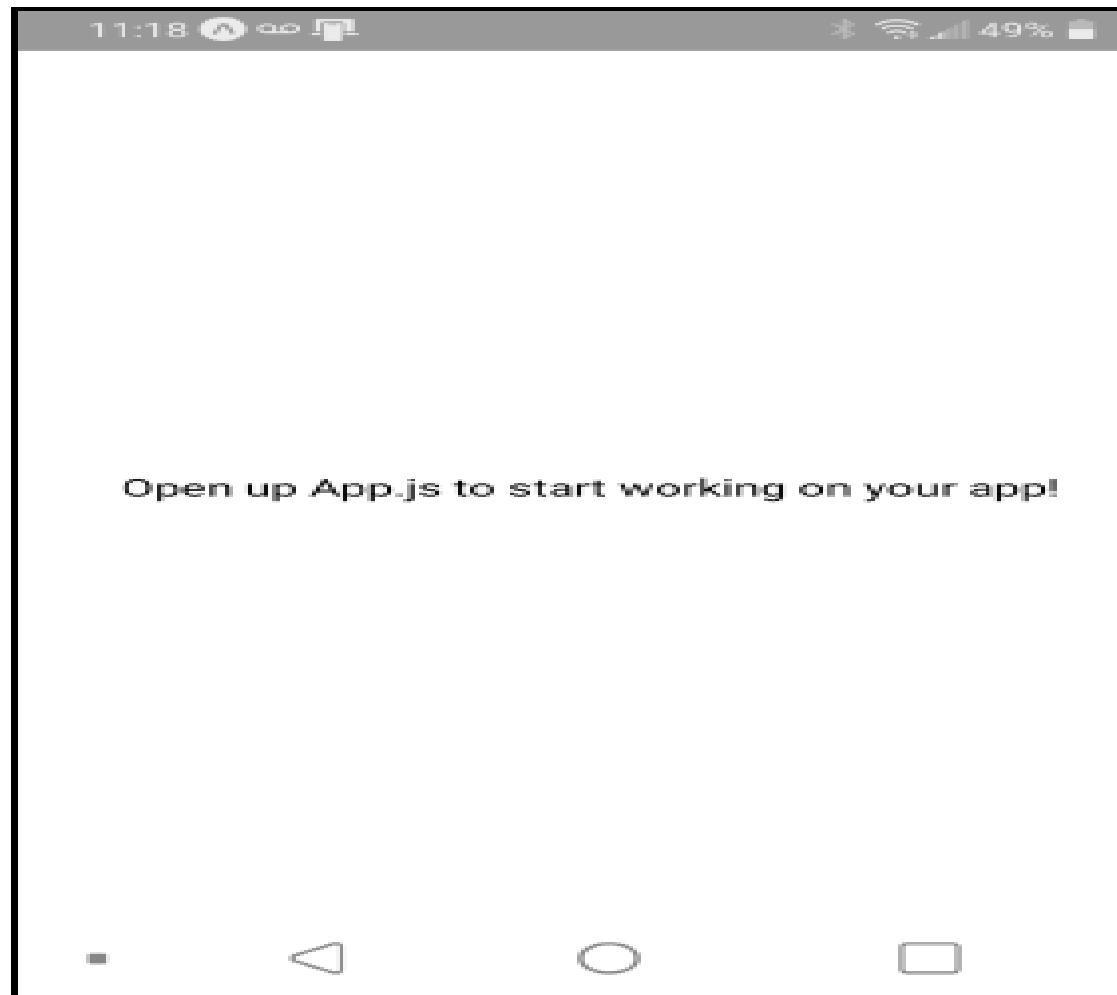


6. Point it at the QR code that is displayed in the Expo UI on the computer. Once the code is scanned, new logs and a new connected device in the Expo UI can be noticed:

The screenshot shows the Expo UI interface. On the left, there's a sidebar with two entries: "Metro Bundler" (with 3 processes) and "LM-G710" (with 1 device). The main area is a log window titled "LOGGED IN AS ADAMBODUCH". It displays the following log entries:

```
INFO Starting Metro Bundler on port 19001.  
10:20  
INFO Tunnel ready.  
10:20  
INFO Building JavaScript bundle: finished in 5548ms.  
11:17
```

7. The device is listed on the left side of this screen and the logs on the right side indicates that a JavaScript bundle has finished building, and the app is running on the device.
8. Device is ready to start developing the app.



Expo setup enables live reloading for free on physical devices synchronized with code updates on the computer.

1. Open up the App.js file inside the my-project folder:

```
import React from "react";
import { StyleSheet, Text, View } from "react-native";

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center"
  }
});
```

## 2. Make a small style change to make the font bold:

```
import React from "react";
import { StyleSheet, Text, View } from "react-native";

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.text}>
        Open up App.js to start working on your app!
      </Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center"
  },
  text: {
    fontWeight: "bold"
  }
});
```

3. A new style called “text” is added and applied to the Text component. The change can be visualized in the device application concurrently...



- Next → Running React Native apps on a variety of virtual device emulators using the Expo Snack service.

# Viewing your app on Expo Snack



# Viewing your app on Expo Snack

- The Snack service provided by Expo is a playground for React Native code.
- It aids to organize the React Native project.
- The Expo account can be created on Snacks and save the work to keep working on them or sharing them with others.
- It is also possible to import code that is stored locally into a Snack or a Git repository.
- When the URL is opened, Snack interface will be loaded and enables to make changes to the code to test things out before running them.
- The pièce de résistance of Snack is the ability to easily run them on virtualized devices.

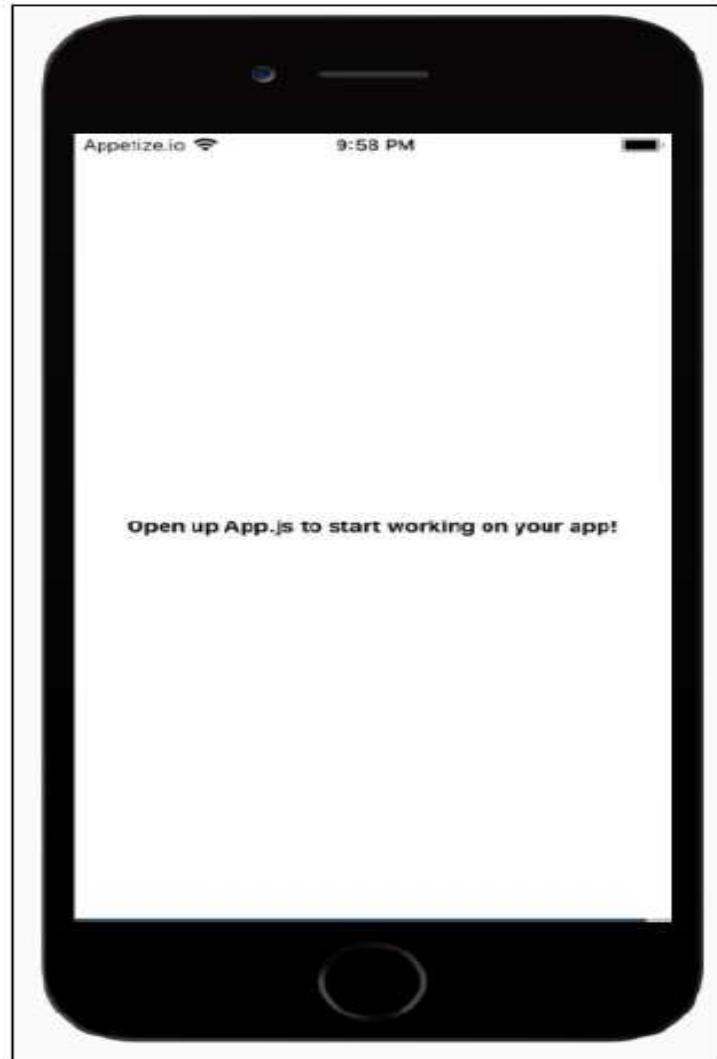
- The controls to run the app on a virtual device can be found on the right side of the UI as shown :



- The top control above the image of the phone controls which device type to emulate: **Android**, **iOS**, or **Web**. The **Tap to play** button will launch the selected virtual device.
- Here's what our app looks like on a virtual iOS device:



- And here's what our app looks like on a virtual Android device:



- This app only displays text and applies some styles to it, so it looks pretty much identical on different platforms.

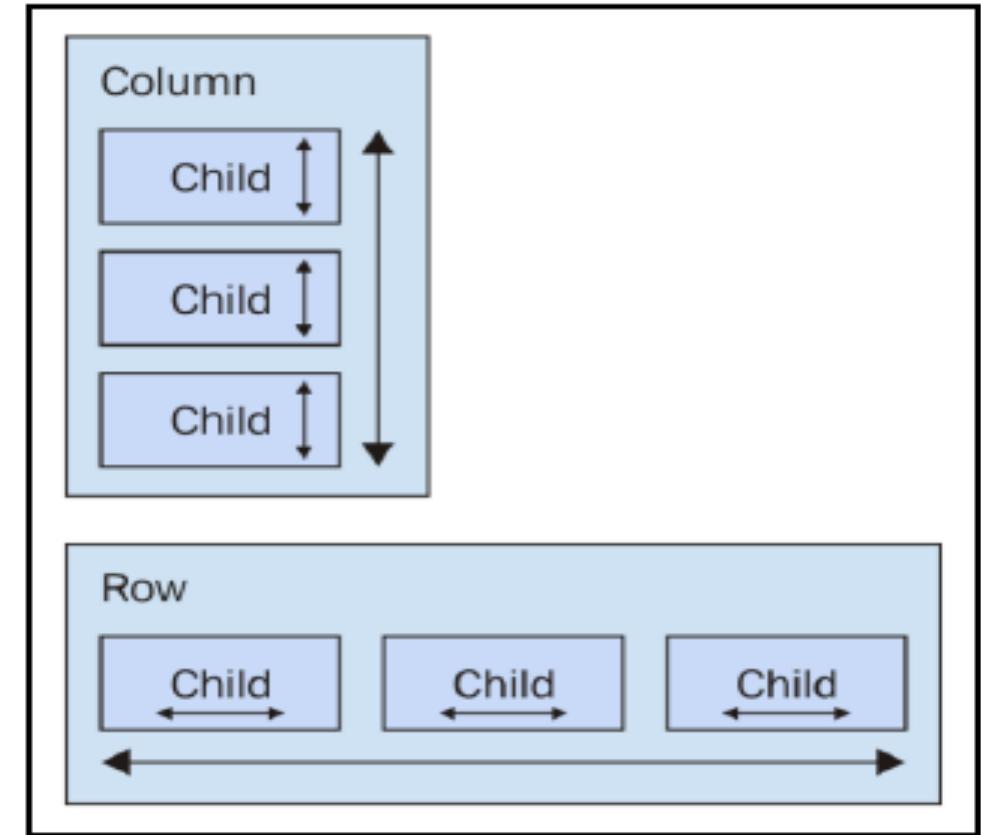
# Flexbox - new layout standard

Chapter-15



# Flexbox - new layout standard

- Flexbox : box model that is flexible to provide a workable layout.
- Box - acts as a container that holds the child elements.
- Child elements are flexible according to the pattern it is to be rendered on the screen.
- Flexbox containers have a direction, either **Column** (up/down) or **Row** (left/right).
- Rows stack on top of one another!
- The key concept - it's the direction that the box flexes, not the direction that boxes are placed on the screen.



# Introducing React Native styles



# Introducing React Native styles

- React Native style sheet :

```
import { Platform, StyleSheet, StatusBar } from "react-native";

export default StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "ghostwhite",
    ...Platform.select({
      ios: { paddingTop: 20 },
      android: { paddingTop: StatusBar.currentHeight }
    })
  },
  box: {
    width: 100,
    height: 100,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "lightgray"
  },
  boxText: {
    color: "darkslategray",
    fontWeight: "bold"
  }
});
```

- This is a JavaScript module
- Use plain objects - to declare React Native styles
- Call `StyleSheet.create()` and export this from the style module.
- Style sheet has three styles: container, box, and boxText.
- Call to `Platform.select()`: inside the container style

```
...Platform.select ({  
  ios: { paddingTop: 20 },  
  android: { paddingTop: StatusBar.currentHeight }  
})
```

- This function will return different styles based on the platform.
- Handling the top padding of the top-level container view.
- This code is used in most of the apps : ensures that React components don't render underneath the status bar of the device.
- The padding vary based on the platform. IOS- paddingTop is 20.
- Android-paddingTop => value of StatusBar.currentHeight

- Let's see how these styles are imported and applied to React Native components:

```
import React from "react";
import { Text, View } from "react-native";
import styles from "./styles";

export default function App() {
  return (
    <View style={styles.container}>
      <View style={styles.box}>
        <Text style={styles.boxText}>I'm in a box</Text>
      </View>
    </View>
  );
}
```

- The styles are assigned to each component via the style property.



# Building Flexbox layouts

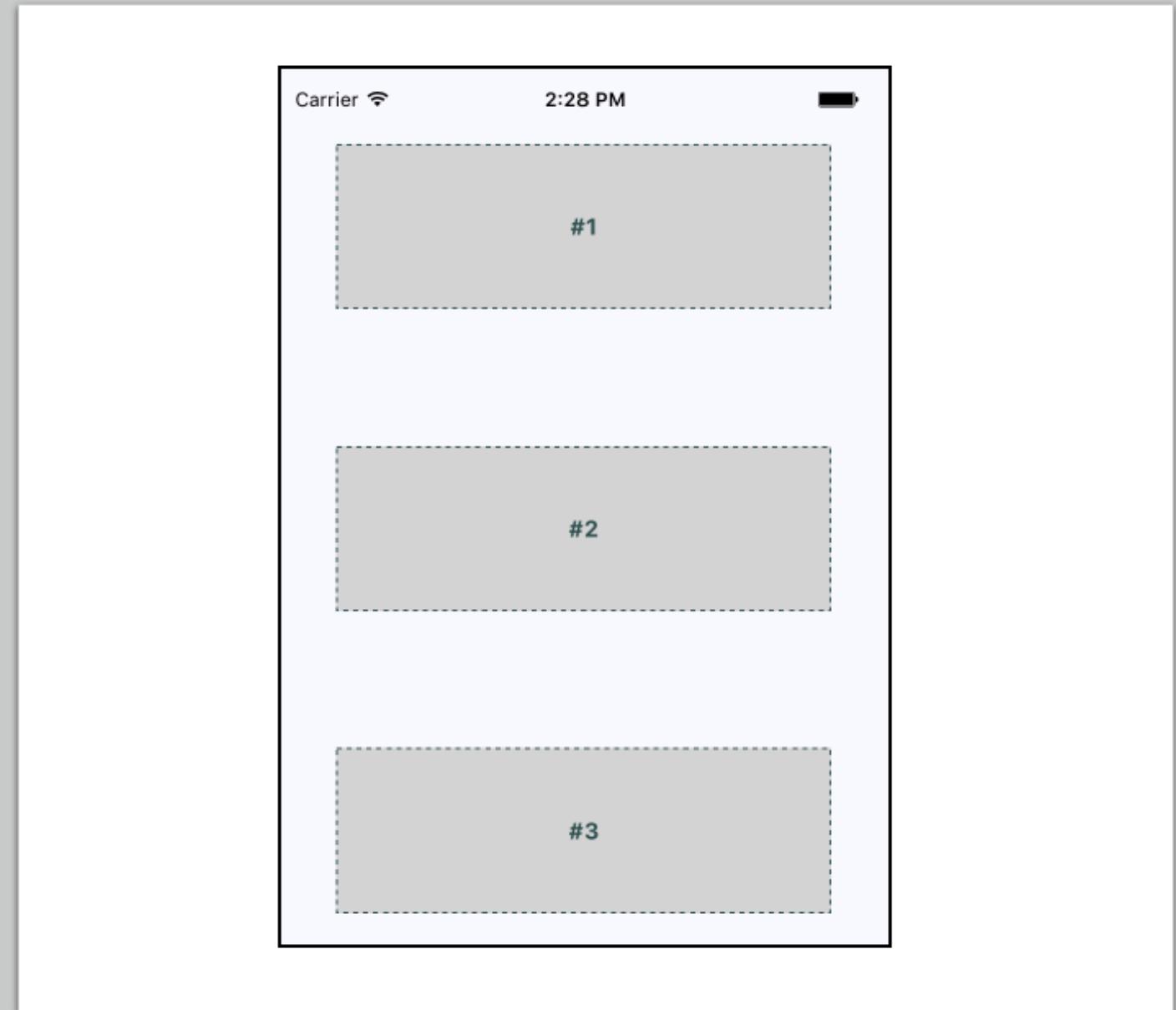


# Building Flexbox layouts

- how powerful the Flexbox layout model is for mobile screens
- Can design the layout that best suits the application.

## Simple three-column layout

- Implement a simple layout with three sections that flex in the direction of the column (top to bottom).



## Components used to create this screen layout:

- The container view- the outermost <View> component : the column
- The child views →rows.
- <Text> component → label each row.
- <View> is similar to a <div> element in HTML
- <Text> is similar to a <p> element in HTML

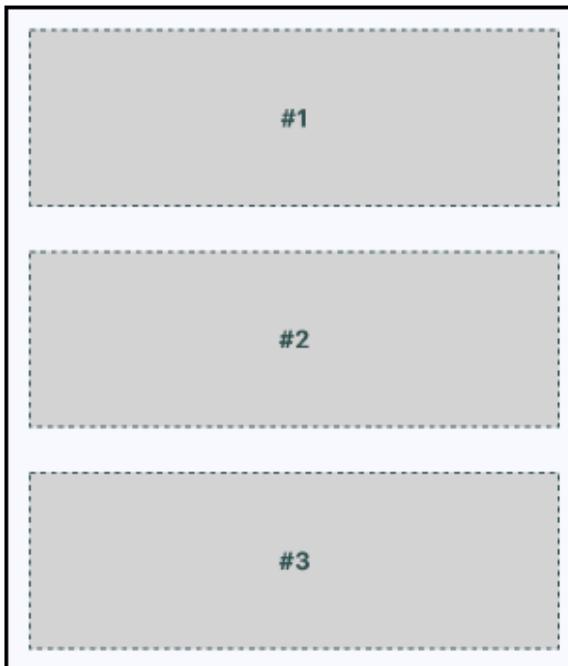
```
import React from "react";
import { Text, View } from "react-native";
import styles from "./styles";

export default function App() {
  return (
    <View style={styles.container}>
      <View style={styles.box}>
        <Text style={styles.boxText}>#1</Text>
      </View>
      <View style={styles.box}>
        <Text style={styles.boxText}>#2</Text>
      </View>
      <View style={styles.box}>
        <Text style={styles.boxText}>#3</Text>
      </View>
    </View>
  );
}
```

# Styles used to create this layout:

```
import { Platform, StyleSheet, StatusBar } from "react-native";

export default StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "column",
    alignItems: "center",
    justifyContent: "space-around",
    backgroundColor: "ghostwhite",
    ...Platform.select({
      ios: { paddingTop: 20 },
      android: { paddingTop: StatusBar.currentHeight }
    })
  },
  box: {
    width: 300,
    height: 100,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "lightgray",
    borderWidth: 1,
    borderStyle: "dashed",
    borderColor: "darkslategray"
  },
  boxText: {
    color: "darkslategray",
    fontWeight: "bold"
  }
});
```

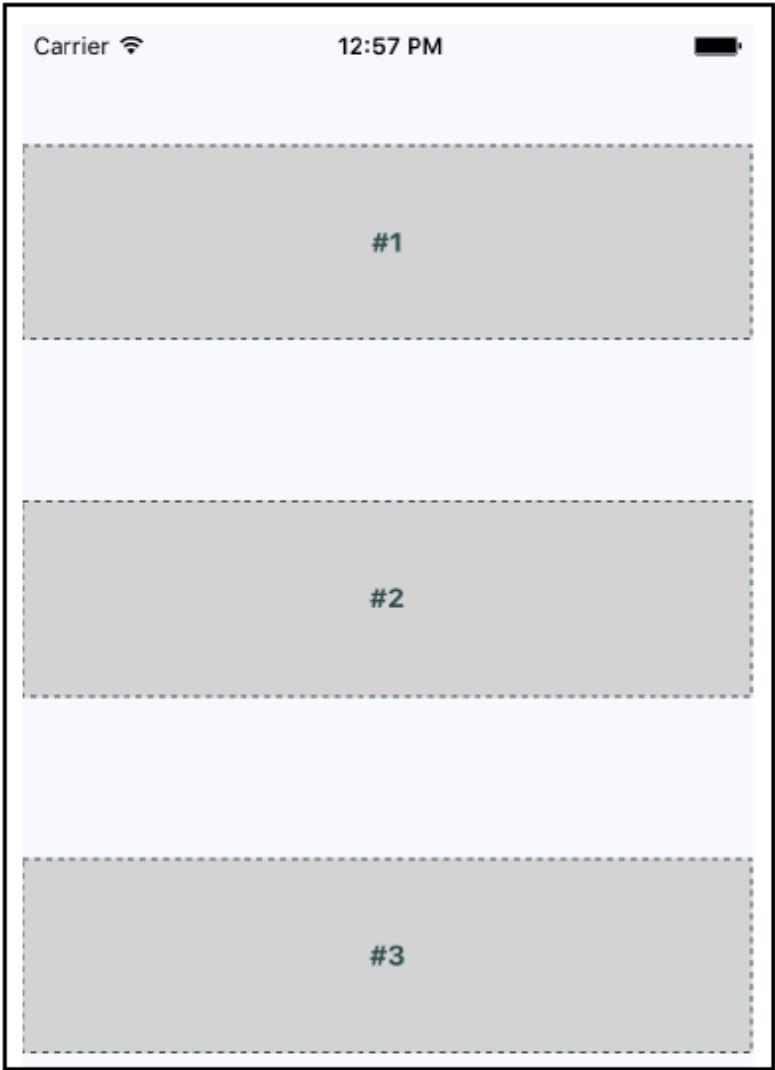


- The flex and flexDirection properties of container enable the layout of the rows to flow from top to bottom.
- The alignItems and justifyContent properties align the child elements to the center of the container and add space around them.
- View of the layout when the device is rotated from a portrait orientation to a landscape orientation:
- The Flexbox automatically figured out how to preserve the layout.

## Improved three-column layout

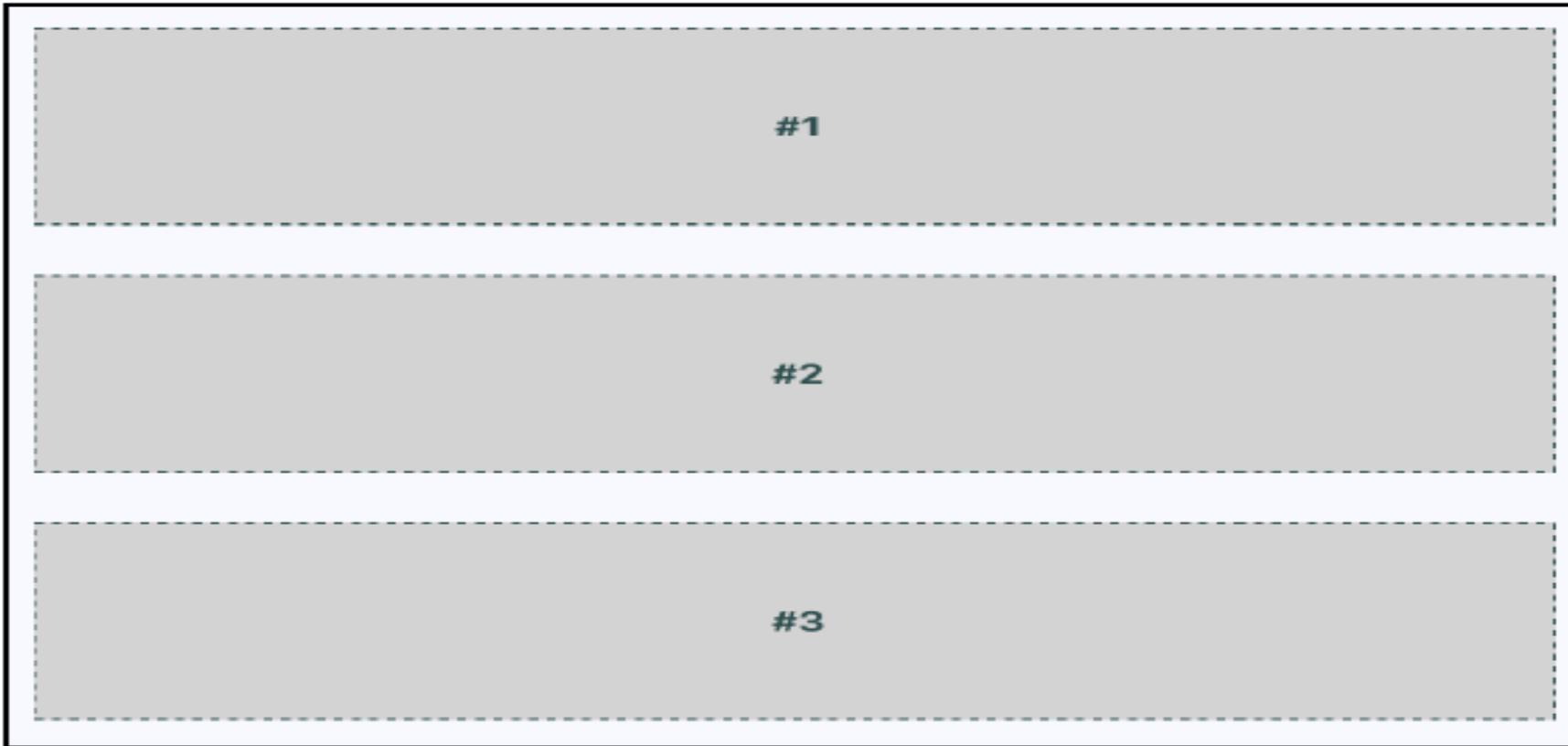
```
import { Platform, StyleSheet, StatusBar } from "react-native";

export default StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "column",
    backgroundColor: "ghostwhite",
    alignItems: "center",
    justifyContent: "space-around",
    ...Platform.select({
      ios: { paddingTop: 20 },
      android: { paddingTop: StatusBar.currentHeight }
    })
  },
  box: {
    height: 100,
    justifyContent: "center",
    alignSelf: "stretch",
    alignItems: "center",
    backgroundColor: "lightgray",
    borderWidth: 1,
    borderStyle: "dashed",
    borderColor: "darkslategray"
  },
  boxText: {
    color: "darkslategray",
    fontWeight: "bold"
  }
});
```



- The key change : alignSelf property.
- Changes the width or height of the elements with the box style based on flexDirection of their container to fill space.

- Every section takes the full width of the screen.
- Entire width of the screen is utilized irrespective of the orientation.



- Implementing a proper Box component that can be used by App.js

```
import React from "react";
import { PropTypes } from "prop-types";
import { View, Text } from "react-native";
import styles from "./styles";

export default function Box({ children }) {
  return (
    <View style={styles.box}>
      <Text style={styles.boxText}>{children}</Text>
    </View>
  );
}

Box.propTypes = {
  children: PropTypes.node.isRequired
};
```

# Flexible rows

flexible row – Enables to create screen layout sections stretch from top to bottom. Here is what the styles for this screen look like:

```
import { Platform, StyleSheet, StatusBar } from "react-native";

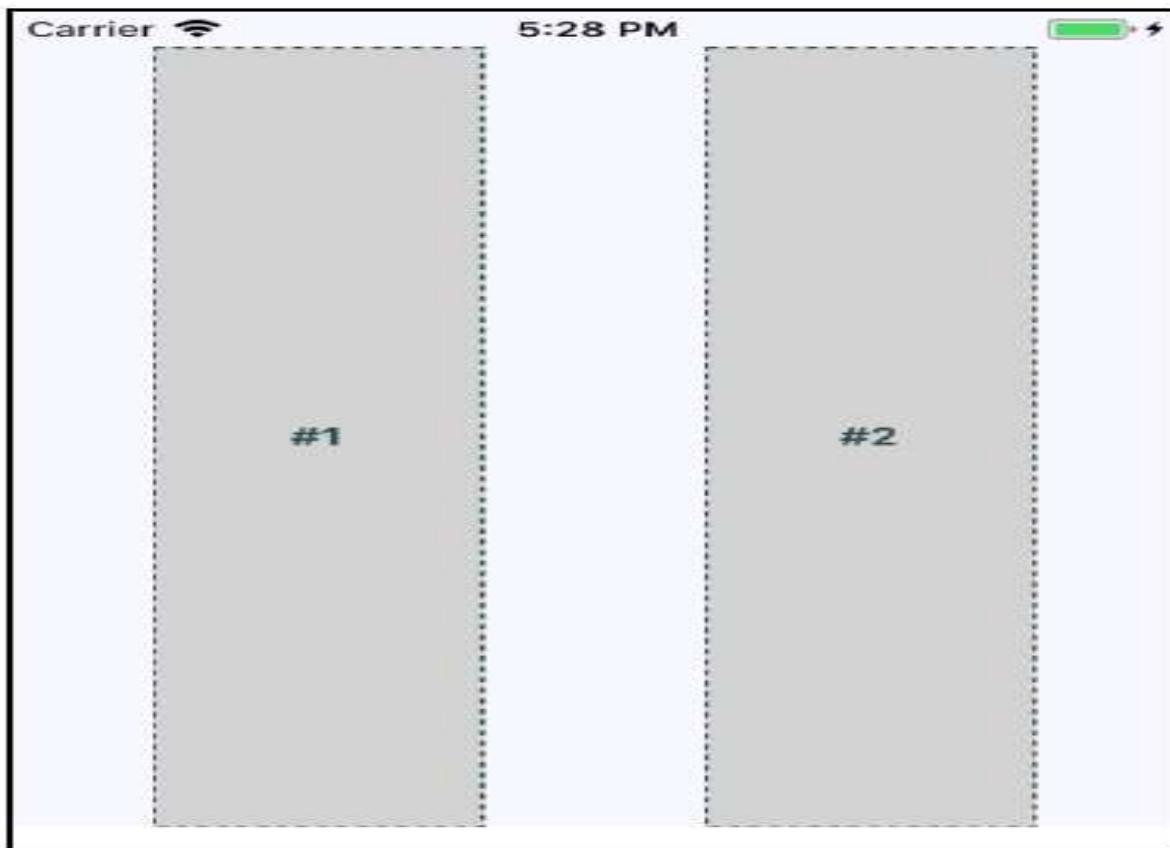
export default StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "row",
    backgroundColor: "ghostwhite",
    alignItems: "center",
    justifyContent: "space-around",
    ...Platform.select({
      ios: { paddingTop: 20 },
      android: { paddingTop: StatusBar.currentHeight }
    })
  },
  box: {
    width: 100,
    justifyContent: "center",
    alignSelf: "stretch",
    alignItems: "center",
    backgroundColor: "lightgray",
    borderWidth: 1,
    borderStyle: "dashed",
    borderColor: "darkslategray"
  },
  boxText: {
    color: "darkslategray",
    fontWeight: "bold"
  }
});
```

- App component, using the Box component :

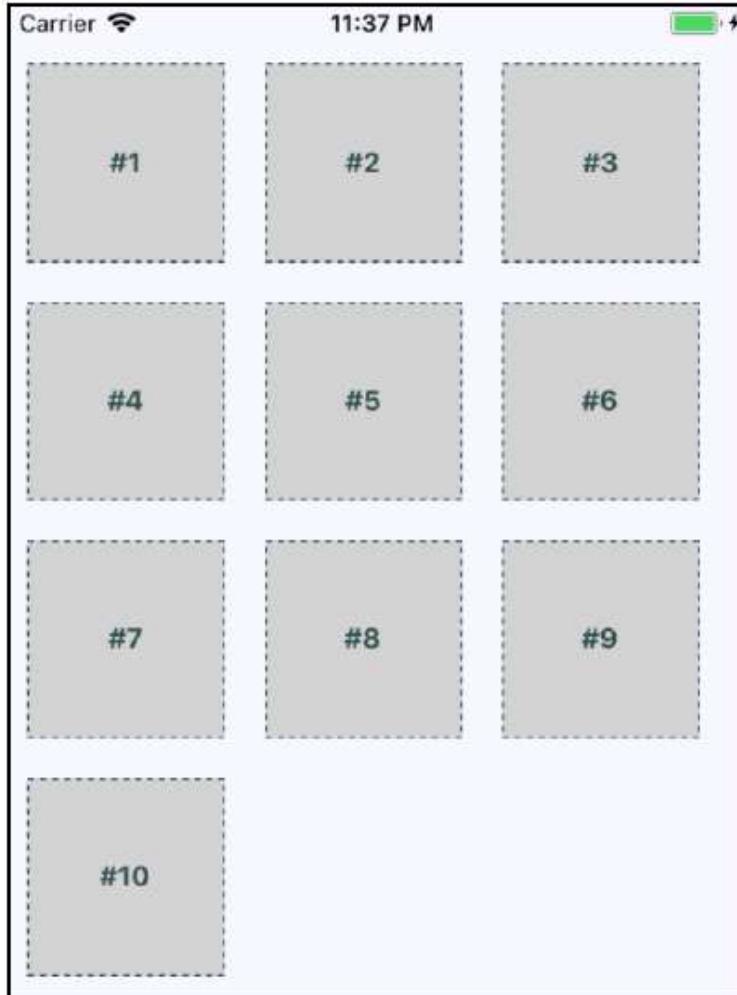
```
import React from "react";
import { Text, View, StatusBar } from "react-native";
import styles from "./styles";
import Box from "./Box";

export default function App() {
  return (
    <View style={styles.container}>
      <Box>#1</Box>
      <Box>#2</Box>
    </View>
  );
}
```

Resulting screen in portrait mode:



# Flexible grids



- When the number of sections to be rendered is not known apriori
- Flexbox → builds a row that flows from left to right until the end of the screen.
- Automatically continues to render the elements from left to right on the next row.
- Visualizing the screen layout as a grid.
- The dimensions of each child determined based on what fits in a given row

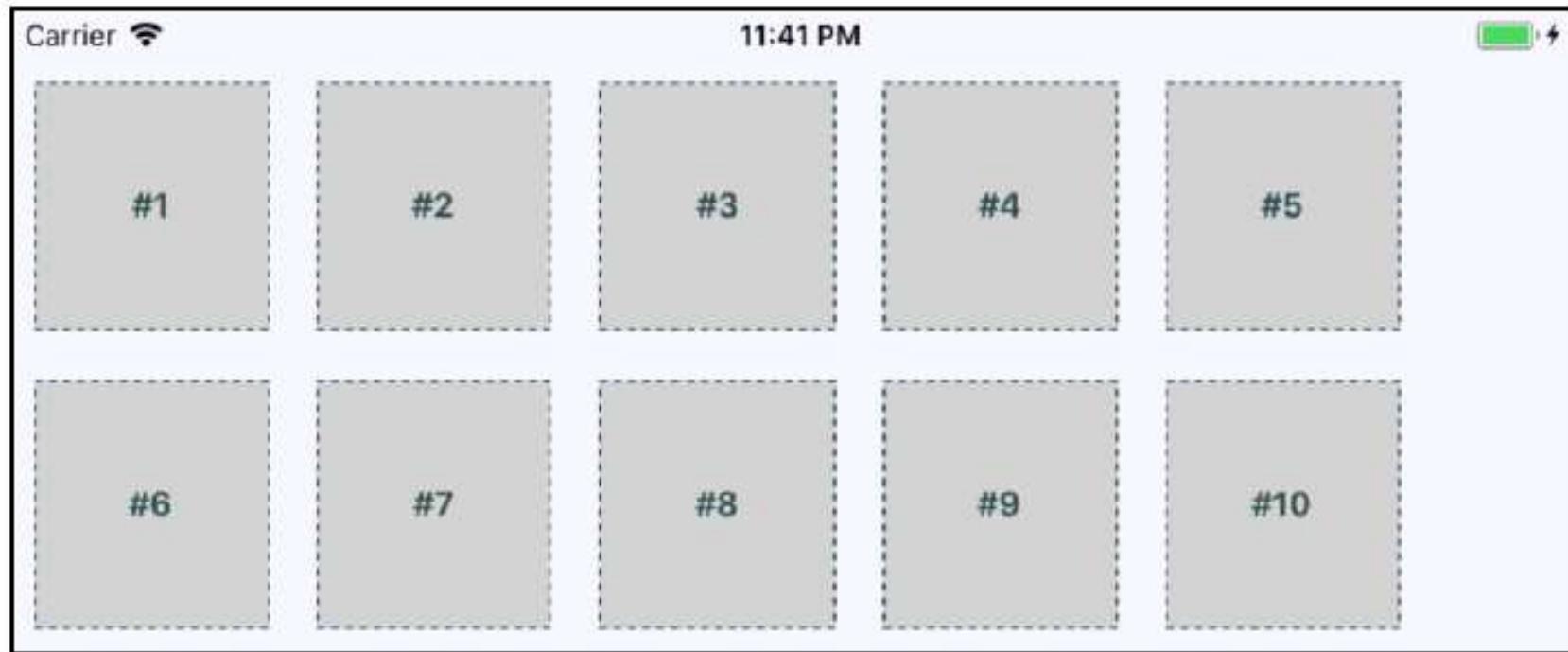
## App component that renders each section:

```
import React from "react";
import { View, StatusBar } from "react-native";
import styles from "./styles";
import Box from "./Box";

const boxes = new Array(10).fill(null).map((v, i) => i + 1);

export default function App() {
  return (
    <View style={styles.container}>
      <StatusBar hidden={false} />
      {boxes.map(i => (
        <Box key={i}>#{i}</Box>
      ))}
    </View>
  );
}
```

landscape orientation that works with flex grids:



# Flexible rows and columns

- Combining rows and columns to create a sophisticated layout for the app.
- Nesting columns within rows or rows within columns.

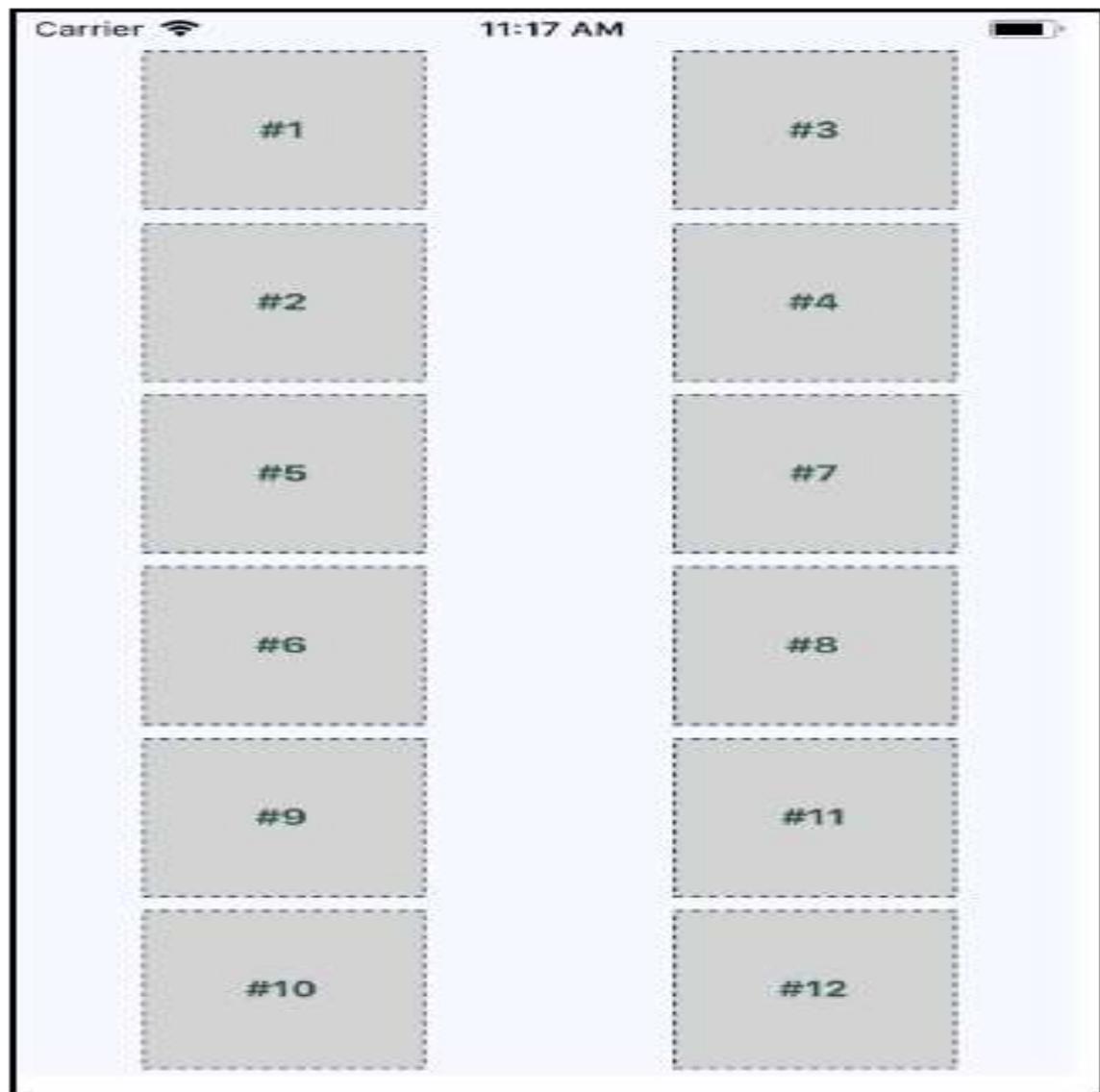
```
import React from "react";
import { View, StatusBar } from "react-native";
import styles from "./styles";
import Row from "./Row";
import Column from "./Column";
import Box from "./Box";

export default function App() {
  return (
    <View>
      <StatusBar style="light-content"></StatusBar>
      <Row>
        <Column>
          <Box>A</Box>
        </Column>
        <Column>
          <Box>B</Box>
        </Column>
      </Row>
    </View>
  );
}
```

```
<View style={styles.container}>
  <StatusBar hidden={false} />
  <Row>
    <Column>
      <Box>#1</Box>
      <Box>#2</Box>
    </Column>
    <Column>
      <Box>#3</Box>
      <Box>#4</Box>
    </Column>
  </Row>
  <Row>
    <Column>
      <Box>#5</Box>
      <Box>#6</Box>
    </Column>
    <Column>
      <Box>#7</Box>
      <Box>#8</Box>
    </Column>
  </Row>
  <Row>
    <Column>
      <Box>#9</Box>
      <Box>#10</Box>
    </Column>
    <Column>
      <Box>#11</Box>
      <Box>#12</Box>
    </Column>
  </Row>
</View>
);
}
```

Abstractions for the layout pieces:  
(`<Row>` and `<Column>`)

The content piece (`<Box>`).



Thank You





الجامعة السعودية الإلكترونية  
SAUDI ELECTRONIC UNIVERSITY  
2011-1432

College of Computing and Informatics  
Bachelor of Science in Computer Science  
IT448  
Mobile Application Development



IT448  
Mobile Application Development  
Week 14  
**Geolocation, Maps and Sensors**



# Contents

1. Where am I?
2. What's around me?
3. Annotating points of interest
4. Sensors overview
5. Motion sensor
6. Position sensor
7. Environment sensor



# Weekly Learning Outcomes

1. Explain about the geolocation and mapping capabilities of React Native
2. Explain the basic configuration MapView components
3. Describe various types of sensors



## Required Reading

1. Chapter 19 (React and React Native: A complete hands-on guide to modern web and mobile development with React.js, 3<sup>rd</sup> Edition, 2020, Adam Boduch, Roy Derks) and Sensors (Android Developer Website)

## Recommended Reading

Geolocation: <https://facebook.github.io/react-native/docs/geolocation>

React Native maps: <https://github.com/react-community/react-native-maps>



Where am I?



# Where am I?

- ✓ The geolocation API : web applications use to figure out the location of the user.  
used by React Native applications because the
- ✓ Same API has been polyfilled in React Native applications.
- ✓ This API is useful for getting precise coordinates from the GPS on mobile devices.
- ✓ information is used to display meaningful location data to the user.
- ✓ For example, latitude and longitude don't mean anything to the user, but you can use this data to look up something that is of use to the user.
- ✓ This might be as simple as displaying where the user is currently located.

- using the geolocation API of React Native to look up coordinates and then use those coordinates to look up human-readable location information from the Google Maps API:

```
import React, { useState, useEffect } from "react";
import { Text, View } from "react-native";
import styles from "./styles";

const API_KEY = "";
const URL = "https://maps.google.com/maps/api/geocode/json?latlng=";

export default function WhereAmI() {
  const [address, setAddress] = useState("loading...");
  const [longitude, setLongitude] = useState();
  const [latitude, setLatitude] = useState();

  useEffect(() => {
    function setPosition({ coords: { latitude, longitude } }) {
      setLongitude(longitude);
      setLatitude(latitude);
      fetch(`${
        URL
      }${latitude},${longitude}`)
        .then(resp => resp.json(), e => console.error(e))
        .then(({ results: [{ formatted_address }] }) => {
          setAddress(formatted_address);
        });
    }
    navigator.geolocation.getCurrentPosition(setPosition);
  });
}
```

```
let watcher = navigator.geolocation.watchPosition(
  setPosition,
  err => console.error(err),
  { enableHighAccuracy: true }
);

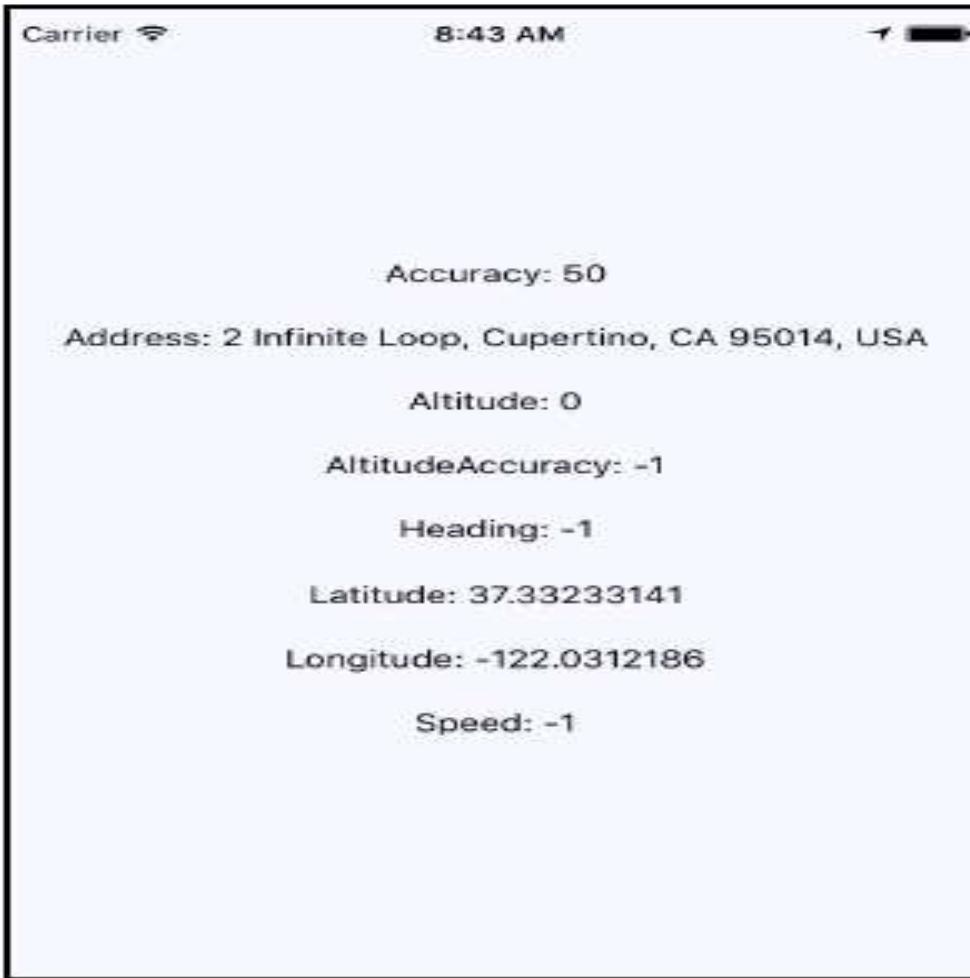
return () => {
  navigator.geolocation.clearWatch(watcher);
};
});

return (
  <View style={styles.container}>
    <Text style={styles.label}>Address: {address}</Text>
    <Text style={styles.label}>Latitude: {latitude}</Text>
    <Text style={styles.label}>Longitude: {longitude}</Text>
  </View>
);
}
```

- The goal of this component is to render the properties returned by the geolocation API on the screen, as well as look up the user's specific location, and display it.
- The setPosition() function is used as a callback in a couple of places. Its job is to set the state of your component.

- First, setPosition() sets the latitude-longitude coordinates. Normally, you wouldn't display this data directly, but this is an example that's showing the data that's available as part of the geolocation API.
- Second, it uses the latitude and longitude values to look up the name of where the user is currently, using the Google Maps API.
- The setPosition() callback is used with getCurrentPosition(), which is only called once when the component is mounted.
- setPosition() with watchPosition() → calls the callback any time the user's position changes.
- The iOS emulator and Android Studio enables location changes via menu options.
- Not required to install the app on a physical device every time when testing by changing locations.

- View of the Screen loaded with the location data:



- The address information that was fetched is probably more useful in an application than latitude and longitude data.

What's around me?



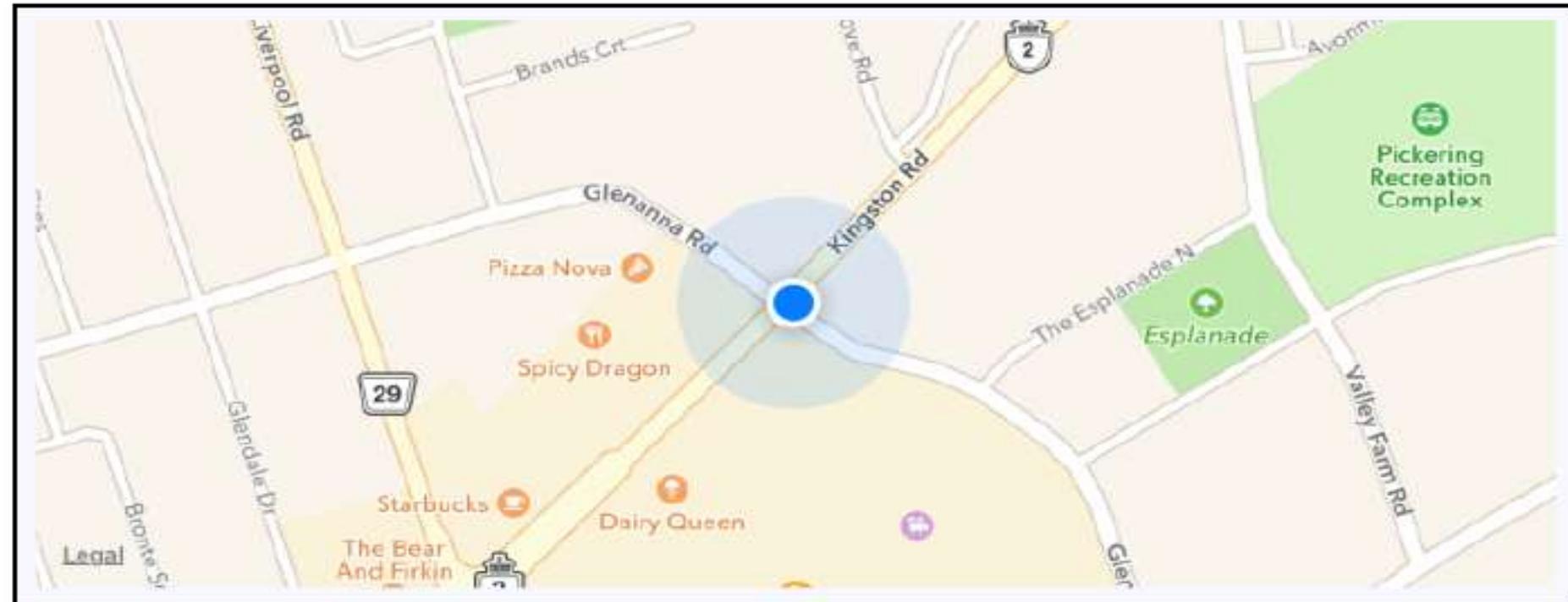
# What's around me?

- Main tool : MapView component from react-native-maps
- used to render maps in the React Native applications.
- Implementation of a basic MapView component

```
import React from "react";
import { View } from "react-native";
import MapView from "react-native-maps";
import styles from "./styles";

export default () => (
  <View style={styles.container}>
    <MapView style={styles.mapView} showsUserLocation followUserLocation />
  </View>
);
```

- Two Boolean properties that is passed to MapView → **showsUserLocation** property and **followUserLocation** property
- showUserLocation Property → will activate the marker on the map, which denotes the physical location of the device running this application.
- followUserLocation Property → The tells the map to update the location marker as the device moves around.
- resulting map:



- The current location of the device is clearly marked on the map.
- By default, points of interest are also rendered on the map.
- These are things in close proximity to the user so that they can see what's around them.
- It's generally a good idea to use the `followUserLocation` property whenever using `showsUserLocation`.
- This makes the map zoom to the region where the user is located.

## Annotating points of interest



# Annotating points of interest

- Annotations → additional information rendered on top of the basic map geography.
- annotations by default when MapView components are rendered.
- The MapView component can render the user's current location and points of interest around the user.
- The challenge here is that you probably want to show points of interest that are relevant to your application, instead of the points of interest that are rendered by default.

# Plotting points

- Pass annotations to the MapView component:

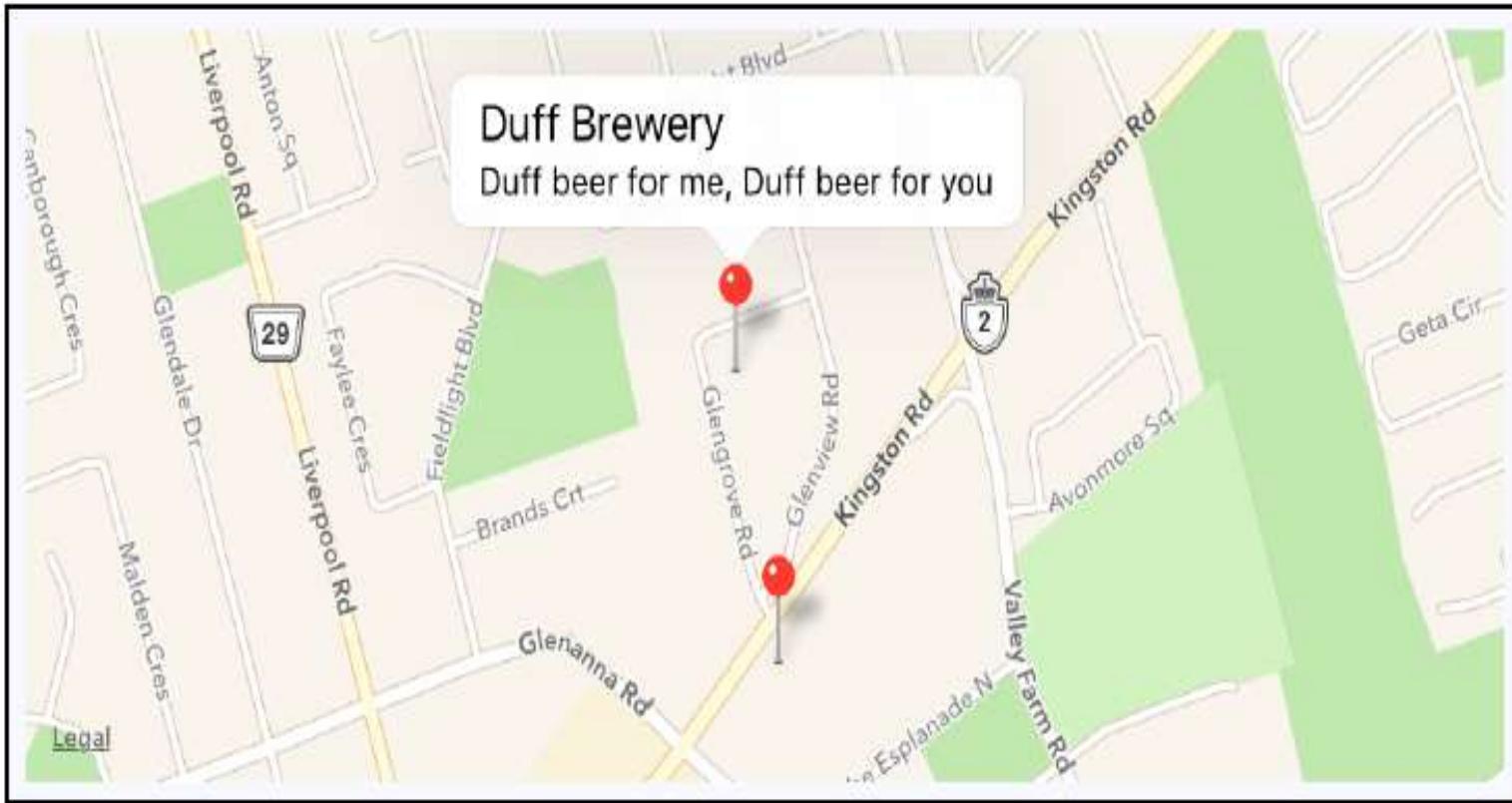
```
import React from "react";
import { View } from "react-native";
importMapView from "react-native-maps";
import styles from "./styles";

export default function App() {
  return (
    <View style={styles.container}>
      <MapView
        style={styles.mapView}
        showsPointsOfInterest={false}
        showsUserLocation
        followUserLocation
      >
        <MapView.Marker
          title="Duff Brewery"
        >
      
```

```
        description="Duff beer for me, Duff beer for you"
        coordinate={{
            latitude: 43.8418728,
            longitude: -79.086082
        }}
    />
<MapView.Marker
    title="Pawtucket Brewery"
    description="New! Patriot Light!"
    coordinate={{
        latitude: 43.8401328,
        longitude: -79.085407
    }}
    />
</MapView>
</View>
);
}
```

➤ setting the showsPointsOfInterest property to false.

Let's see where these breweries are located:



- The callout is displayed when the marker is clicked that shows the location of the brewery on the map.
- The text is render based on the description property values given to <MapView.Marker>

# Plotting overlays

- Render region overlays.
- A point is a single latitude/longitude coordinate.
- Region – the area enclosed by the line connecting the points of several coordinates.
- Here's what the code looks like:

```
import React, { useState } from "react";
import { View, Text } from "react-native";
import MapView from "react-native-maps";
import styles from "./styles";

const ipaRegion = {
  coordinates: [
    { latitude: 43.8486744, longitude: -79.0695283 },
    { latitude: 43.8537168, longitude: -79.0700046 },
    { latitude: 43.8518394, longitude: -79.0725697 },
    { latitude: 43.8481651, longitude: -79.0716377 },
    { latitude: 43.8486744, longitude: -79.0695283 }
  ],
  strokeColor: "coral",
  strokeWidth: 4
};
```

```
const stoutRegion = {
  coordinates: [
    { latitude: 43.8486744, longitude: -79.0693283 },
    ...
  ],
  strokeColor: "firebrick",
  strokeWidth: 4
};

export default function PlottingOverlays() {
  const [ipaStyles, setIpaStyles] = useState([styles.ipaText,
  styles.boldText]);
  const [stoutStyles, setStoutStyles] = useState([styles.stoutText]);
  const [overlays, setOverlays] = useState([ipaRegion]);

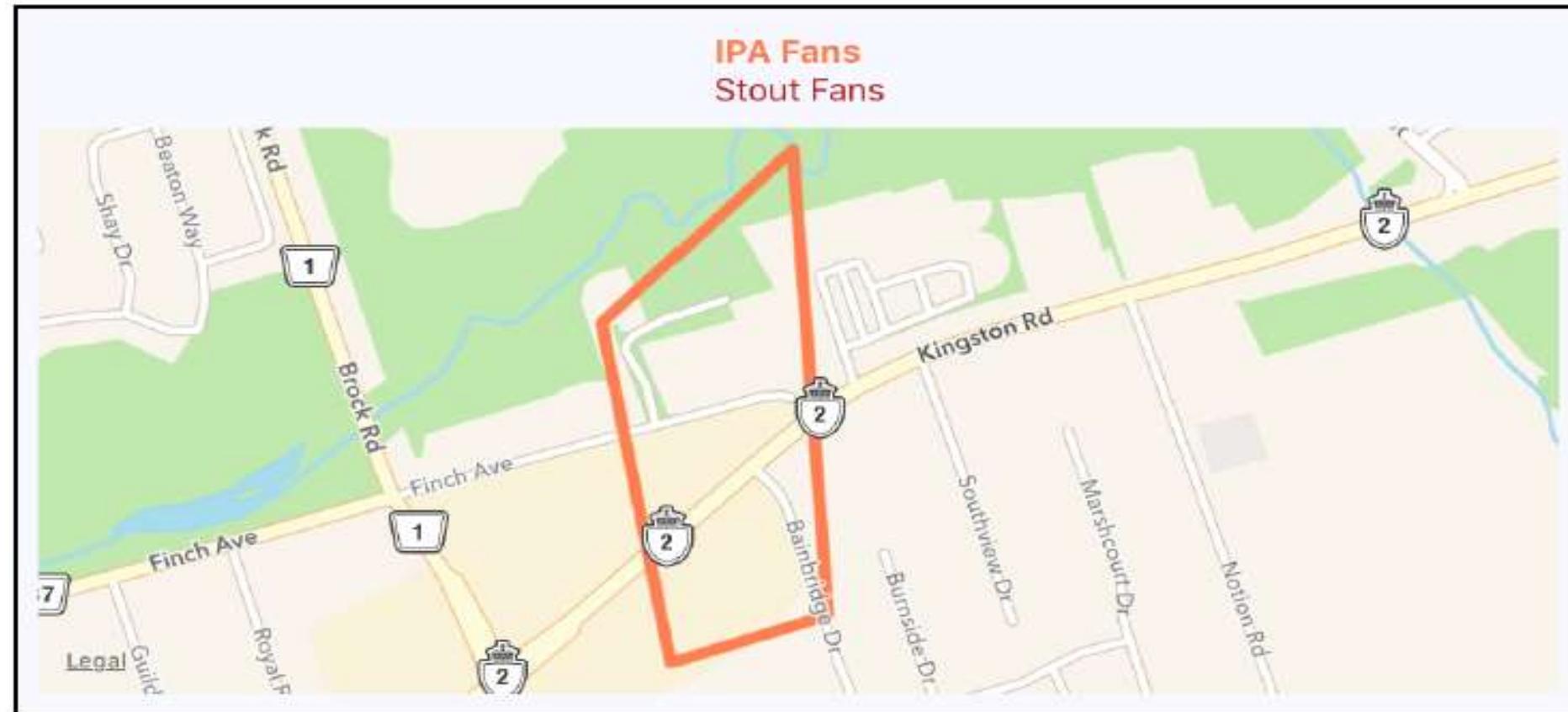
  function onClickIpa() {
    setIpaStyles([...ipaStyles, styles.boldText]);
    setStoutStyles([stoutStyles[0]]);
    setOverlays([ipaRegion]);
  }

  function onClickStout() {
    setStoutStyles([...stoutStyles, styles.boldText]);
  }
}
```

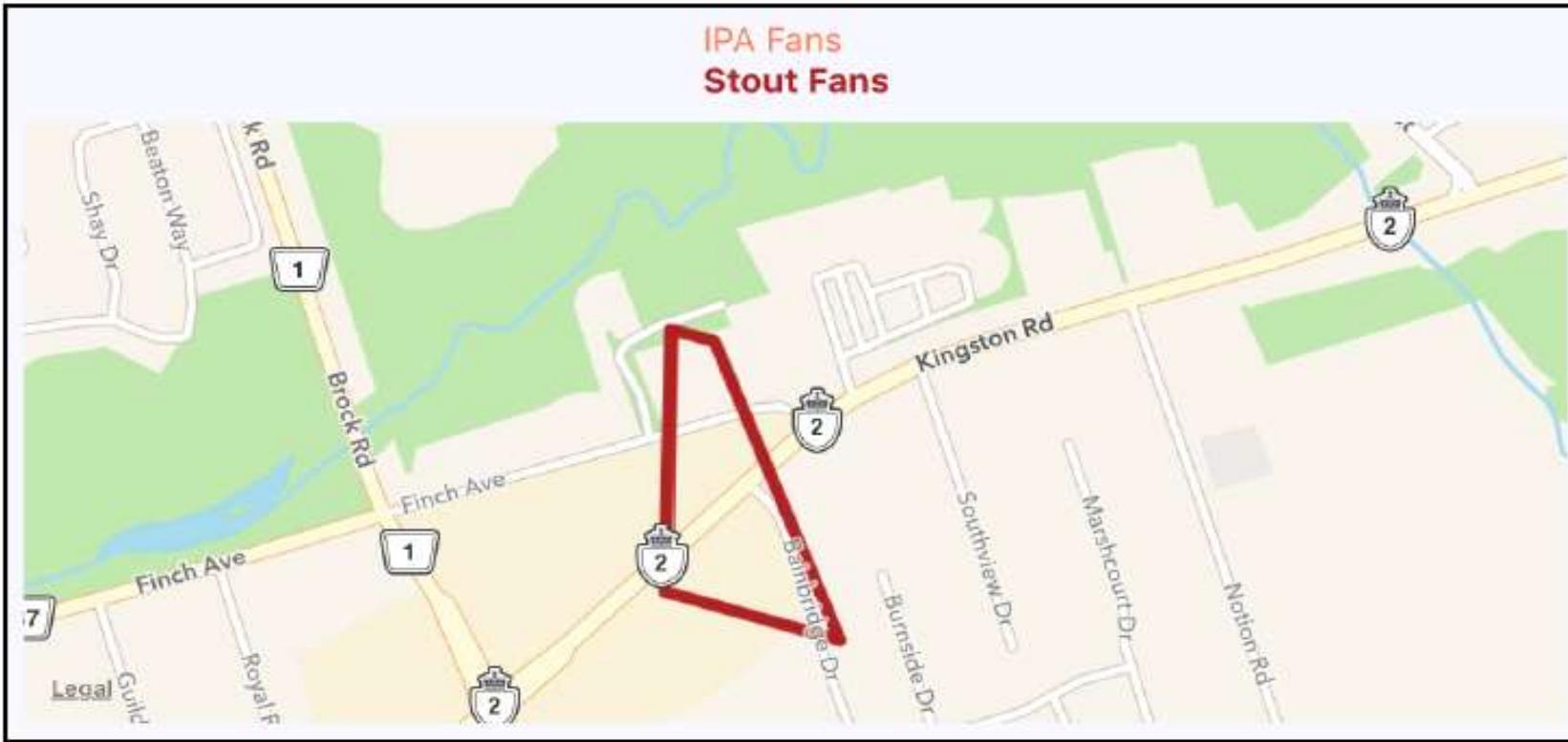
```
    setIpaStyles([ipaStyles[0]]);
    setOverlays([stoutRegion]);
}

return (
  <View style={styles.container}>
    <View>
      <Text style={ipaStyles} onPress={onClickIpa}>
        IPA Fans
      </Text>
      <Text style={stoutStyles} onPress={onClickStout}>
        Stout Fans
      </Text>
    </View>
    <MapView
      style={styles.mapView}
      showsPointsOfInterest={false}
      showsUserLocation
      followUserLocation
    >
      {overlays.map((v, i) => (
        <MapView.Polygon
          key={i}
          coordinates={v.coordinates}
          strokeColor={v.strokeColor}
          strokeWidth={v.strokeWidth}
        />
      ))}
    </MapView>
  </View>
);
```

- The region data consists of several latitude/longitude coordinates that define the shape and location of the region.
- Handling state when the two text links are pressed. By default, the IPA region is rendered, as follows:



- When the stout text is pressed, the IPA overlay is removed from the map and the stout region is added:



- Overlays are useful to highlight an area instead of a latitude/longitude point or an address.

# Sensors overview



# Sensors - overview

- Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions.
- capable of providing raw data with high precision and accuracy,
- useful to monitor three-dimensional device movement or positioning, or changes in the ambient environment near a device.
- Examples:
  - a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing.
  - a weather application use a device's temperature sensor and humidity sensor to calculate and report the dewpoint,
  - travel application use the geomagnetic field sensor and accelerometer to report a compass bearing.

# Introduction to Sensors

- Android sensor framework : access many types of sensors. S
- Sensors : hardware-based and some are software-based.
- Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.
- Software-based sensors are not physical devices. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors.
- The linear acceleration sensor and the gravity sensor are examples of software-based sensors.

**Table 1.** Sensor types supported by the Android platform.

<b>Sensor</b>	<b>Type</b>	<b>Description</b>	<b>Common Uses</b>
<a href="#">TYPE_ACCELEROMETER</a>	Hardware	Measures the acceleration force in m/s <sup>2</sup> that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
<a href="#">TYPE_AMBIENT_TEMPERATURE</a>	Hardware	Measures the ambient room temperature in degrees Celsius (°C).	Monitoring air temperatures.
<a href="#">TYPE_GRAVITY</a>	Software or Hardware	Measures the force of gravity in m/s <sup>2</sup> that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
<a href="#">TYPE_GYROSCOPE</a>	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
<a href="#">TYPE_LIGHT</a>	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
<a href="#">TYPE_LINEAR_ACCELERATION</a>	Software or Hardware	Measures the acceleration force in m/s <sup>2</sup> that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
<a href="#">TYPE_MAGNETIC_FIELD</a>	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in µT.	Creating a compass.

**Table 1.** Sensor types supported by the Android platform.

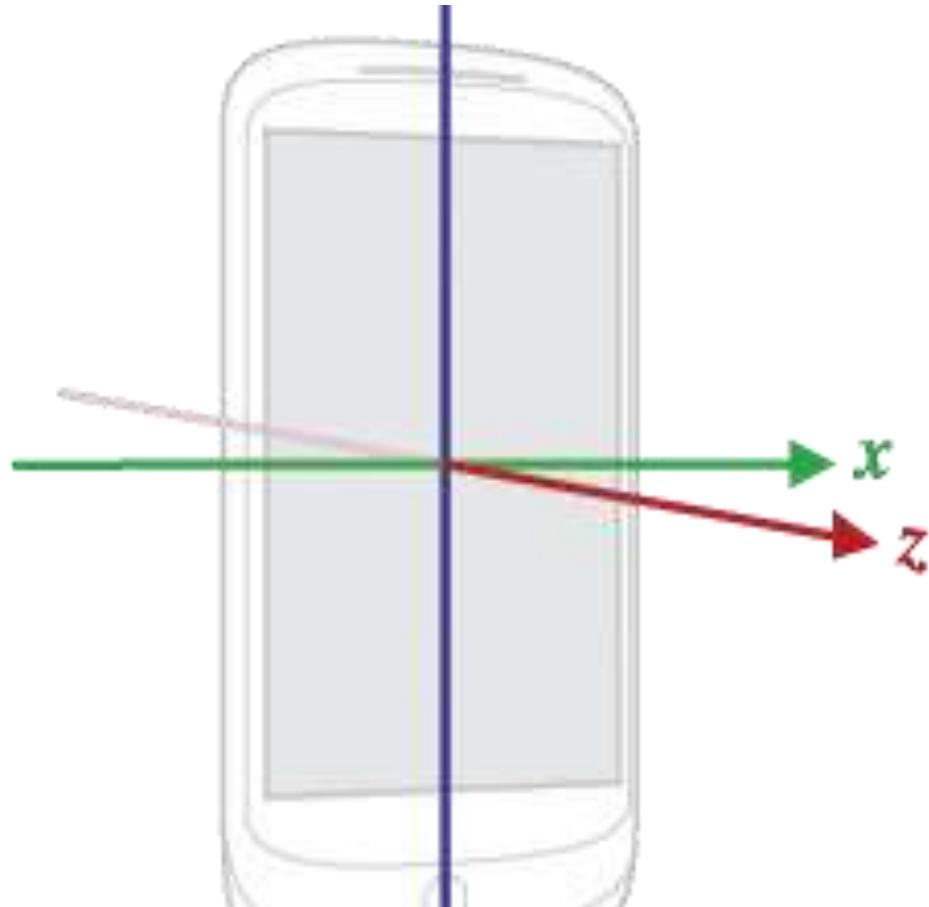
<a href="#">TYPE_ORIENTATION</a>	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the <a href="#">getRotationMatrix()</a> method.	Determining device position.
<a href="#">TYPE_PRESSURE</a>	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
<a href="#">TYPE_PROXIMITY</a>	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
<a href="#">TYPE_RELATIVE_HUMIDITY</a>	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
<a href="#">TYPE_ROTATION_VECTOR</a>	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
<a href="#">TYPE_TEMPERATURE</a>	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the <a href="#">TYPE_AMBIENT_TEMPERATURE</a> sensor in API Level 14	

# Sensors Framework

- SensorManager
  - ✓ Accessing/listing sensors
  - ✓ Registering/unregistering sensor event listeners
  - ✓ Acquiring orientation information
- Sensor
  - ✓ Determining the capability of sensors
- SensorEvent
  - ✓ Providing information about sensor events
  - ✓ Including raw sensor data, sensor, the accuracy, and timestamp.
- SensorEventListener
  - Containing callbacks to receiving notifications of sensor events

## Sensor Coordinate System

- Default orientation
  - X axis – horizontal pointing right
  - Y axis – vertical pointing up
  - Z axis – pointing toward the outside of the screen face.
- Coordinate system does not change when orientation changes
- `getOrientation()` to determine screen rotation
- `remapCoordinateSystem()` to map sensor coordinates to screen coordinates.



# Motion sensor



# Motion sensor

- Measure acceleration forces and rotational forces along three axes.
- Includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- Sensors to monitor the motion of a device.
- The sensors' possible architectures vary by sensor type:
  - The gravity, linear acceleration, rotation vector, significant motion, step counter, and step detector sensors are either hardware-based or software-based.
  - The accelerometer and gyroscope sensors are always hardware-based.
- Most Android-powered devices have an accelerometer, included with a gyroscope.
- Software-based sensors : rely on one or more hardware sensors to derive their data.
- Derive their data either from the accelerometer and magnetometer or from the gyroscope.

- Motion sensors are useful for monitoring device movement, such as tilt, shake, rotation, or swing.
- Movement : a reflection of direct user input (for example, a user steering a car in a game or a user controlling a ball in a game), but it can also be a reflection of the physical environment in which the device is sitting (for example, moving with you while you drive your car).
- Case1 : monitoring motion relative to the device's frame of reference or application's frame of reference;
- Case 2: monitoring motion relative to the world's frame of reference.
- Motion sensors by themselves are not typically used to monitor device position, but they can be used with other sensors.
- All of the motion sensors return multi-dimensional arrays of sensor values for each Sensor Event.
- For example, during a single sensor event the accelerometer returns acceleration force data for the three coordinate axes, and the gyroscope returns rate of rotation data for the three coordinate axes.

# Position sensor



# Position sensor

- Measure the physical position of a device - orientation sensors and magnetometers.
- Two hardware-based sensors to determine the position of a device:
  - ✓ geomagnetic field sensor and the accelerometer.
  - ✓ *proximity sensor*: To determine how close the face of a device is to an object
- Most handset and tablet manufacturers include a geomagnetic field sensor.
- readings from the device's accelerometer and the geomagnetic field sensor - determine a device's orientation
- Position sensors are useful for determining a device's physical position in the world's frame of reference.

- For example, the geomagnetic field sensor in combination with the accelerometer to determine a device's position relative to the magnetic north pole.
- To determine a device's orientation in your application's frame of reference.
- Position sensors do not monitor device movement or motion, such as shake, tilt, or thrust.
- The geomagnetic field sensor and accelerometer return multi-dimensional arrays of sensor values for each SensorEvent
- For example, the geomagnetic field sensor provides geomagnetic field strength values for each of the three coordinate axes during a single sensor event.
- Accelerometer sensor: measures the acceleration applied to the device during a sensor event.

# Environment sensor



# Environment sensor

- Measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity.
- Includes barometers, photometers, and thermometers
- Four sensors to monitor various environmental properties.
- Monitor relative ambient humidity, illuminance, ambient pressure, and ambient temperature near an Android-powered device.
- All four environment sensors are hardware-based and are available only if a device manufacturer has built them into a device.
- Environment sensors are not always available on devices.

- Environment sensors return a single sensor value for each data event.
- For example, the temperature in °C or the pressure in hPa.
- Environment sensors do not typically require any data filtering or data processing.

Thank You

