Android: Building Your First App

http://developer.android.com/training/basics/firstapp/

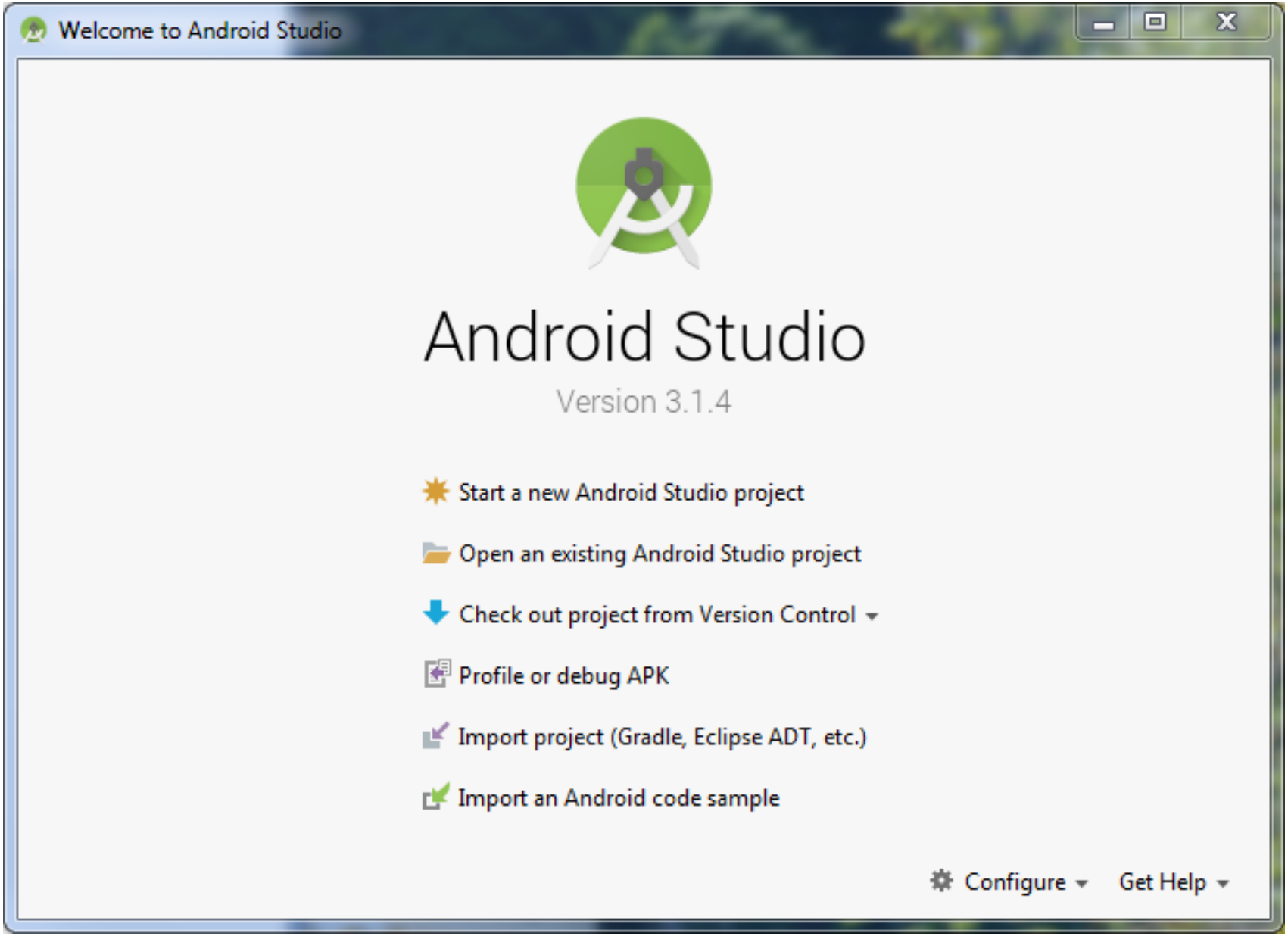
Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2018-2019)

Outline

Outline


Welcome to Android Studio
Android Studio
Version 3.1.4
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Configure
Get Help

Create New Project

Create Android Project

Application name

PDM18kotlin0

Company domain

educ.di.unito.it

Project location

C:\AndroidProjects\ProgMob2019\PDM18kotlin0 ...

Package name

it.unito.di.educ.pdm18kotlin0 Edit

☐ **Include C++ support**

☑ **Include Kotlin support**

Previous Next Cancel Finish

Create New Project

Target Android Devices

Select the form factors and minimum SDK

Some devices require additional SDKs. Low API levels target more devices, but offer fewer API features.

- [x] **Phone and Tablet**

  API 21: Android 5.0 (Lollipop)

  By targeting **API 21 and later**, your app will run on approximately **85,0%** of devices. Help me choose

  - [ ] Include Android Instant App support

- [ ] **Wear**

  API 21: Android 5.0 (Lollipop)

- [ ] **TV**

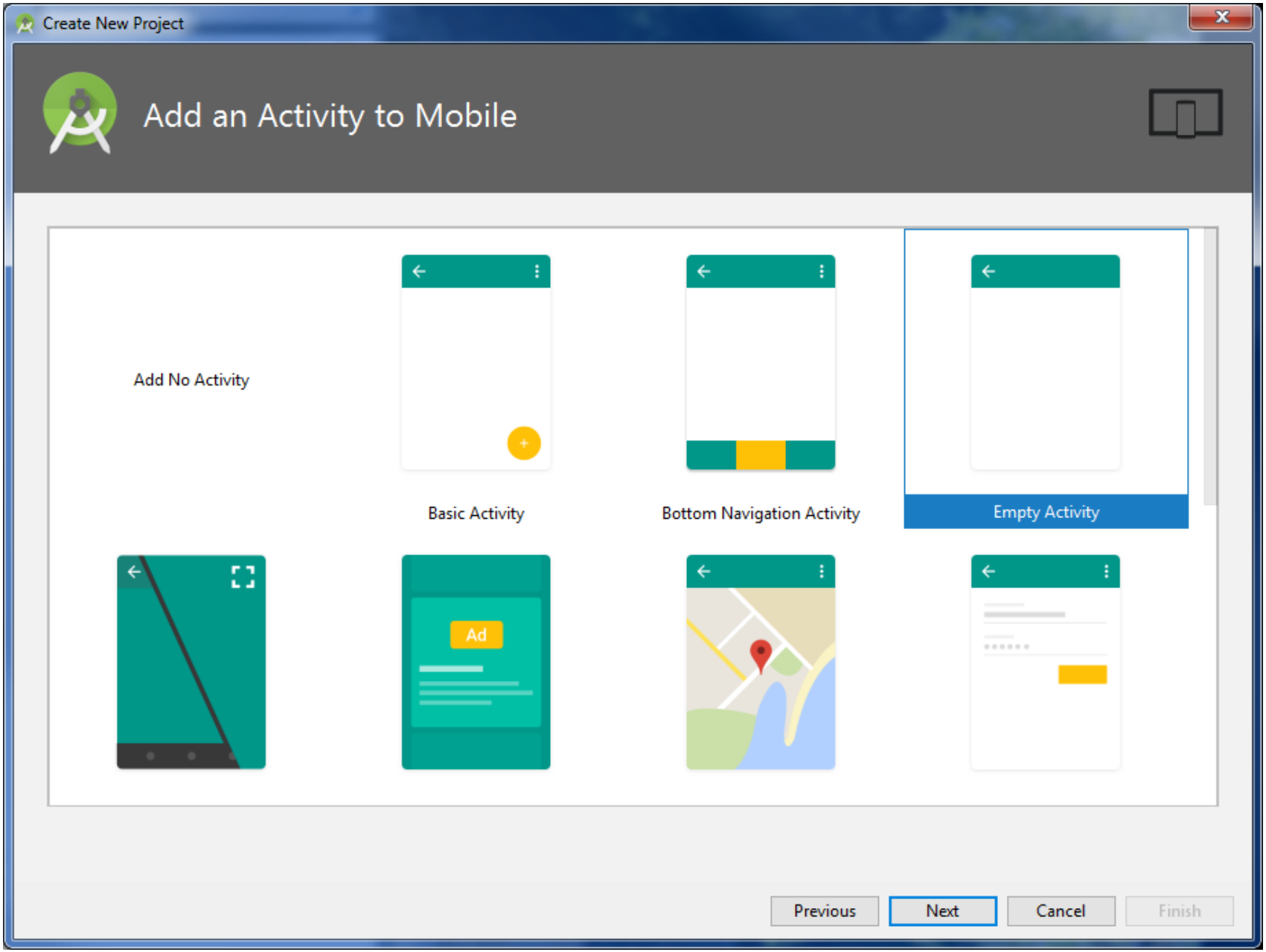
  API 21: Android 5.0 (Lollipop)

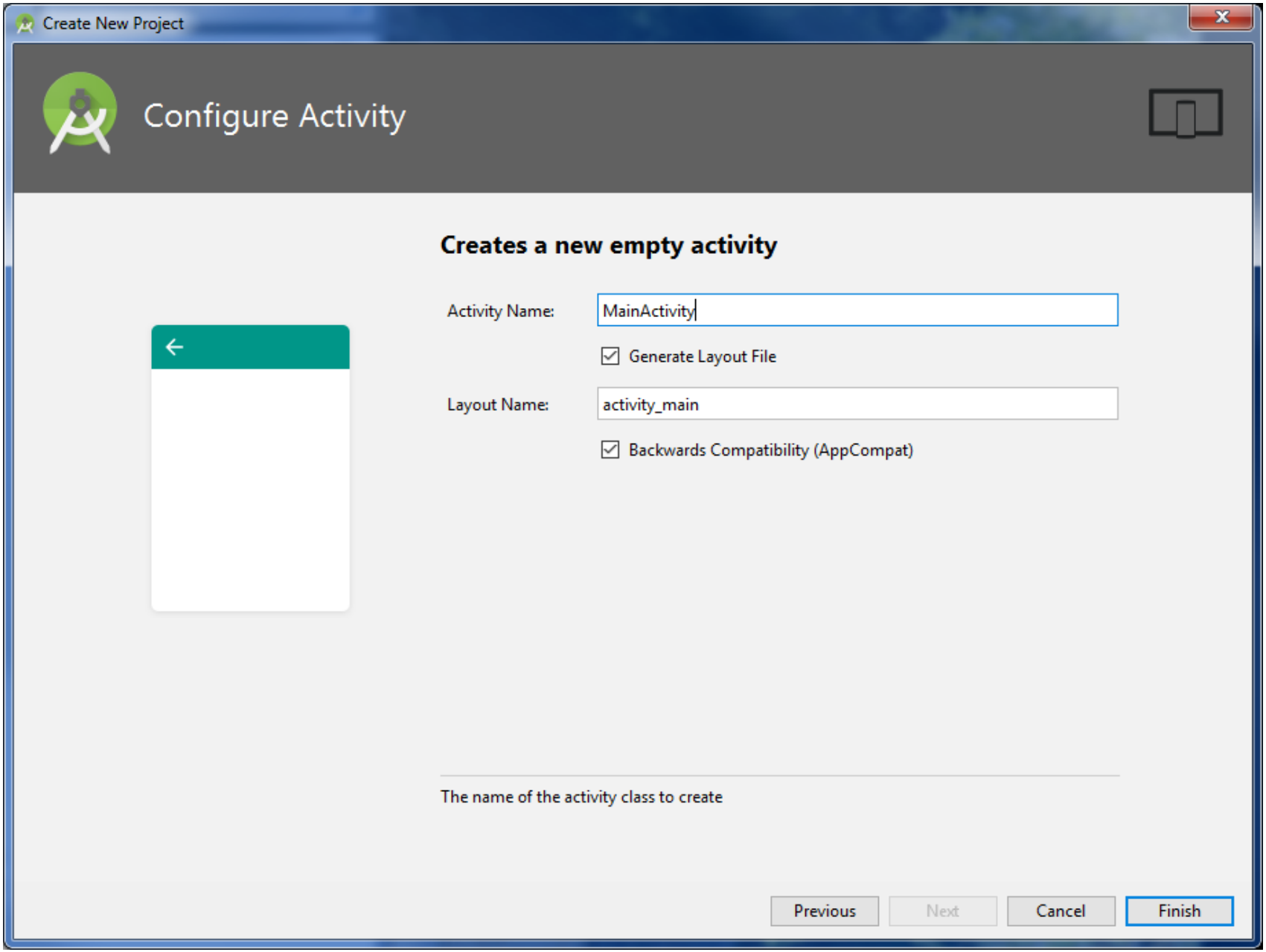
- [ ] **Android Auto**

- [ ] **Android Things**

  API 24: Android 7.0 (Nougat)

Previous | Next | Cancel | Finish

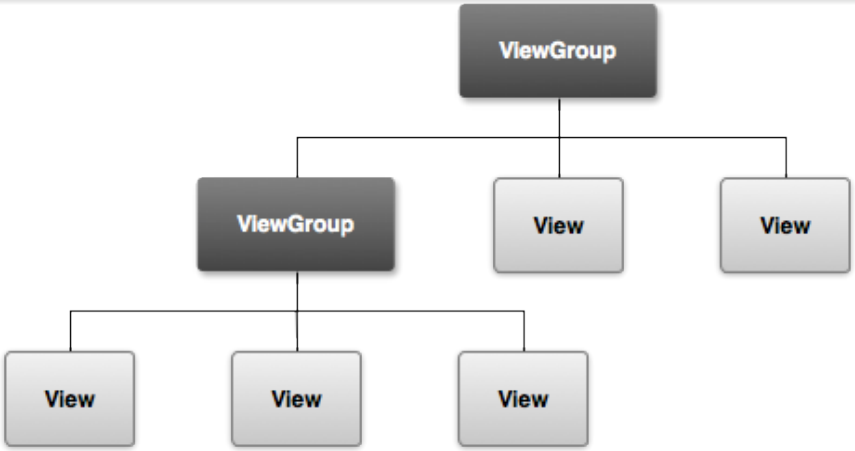

Create New Project
Add an Activity to Mobile
Add No Activity
Basic Activity
Bottom Navigation Activity
Empty Activity
Ad
Previous
Next
Cancel
Finish


Create New Project
Configure Activity
Creates a new empty activity
Activity Name:
MainActivity
Generate Layout File
Layout Name:
activity_main
Backwards Compatibility (AppCompat)
The name of the activity class to create
Previous
Next
Cancel
Finish

The graphical user interface for an Android app is built using a hierarchy of `View` and `ViewGroup` objects.

- `View` objects are usually UI widgets such as **buttons** or **text fields**.
- `ViewGroup` objects are invisible view containers that define how the child views are laid out, such as in a grid or a vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of `View` and `ViewGroup` so you can define your UI in XML using a hierarchy of UI elements.
Layouts are subclasses of the `ViewGroup`.



The default "Hello World" app uses a `ConstraintLayout`.

Some default files

The created Android project is now a basic "Hello World" app that contains some default files. The most important are:

- `app/src/main/res/layout/activity_main.xml`

  This is the XML layout file for the activity you added when you created the project with Android Studio. Following the New Project workflow, Android Studio presents this file with both a text view and a preview of the screen UI. The file includes some default settings and a `TextView` element that displays the message, "Hello world!".

PDM18kotlin0 [C:\AndroidProjects\ProgMob2019\PDM18kotlin0] - ...\app\src\main\res\layout\activity_main.xml [app] - Android Studio

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

app

PDM18kotlin0 > app > src > main > java > it > unito > di > educ > pdm18kotlin0

Android | Project Files

- app
  - manifests
    - AndroidManifest.xml
  - java
    - it.unito.di.educ.pdm18kotlin0
      - MainActivity
    - it.unito.di.educ.pdm18kotlin0 (androidTest)
      - ExampleInstrumentedTest
    - it.unito.di.educ.pdm18kotlin0 (test)
      - ExampleUnitTest
  - res
    - drawable
      - ic_launcher_background.xml
      - ic_launcher_foreground.xml (v24)
    - layout
      - activity_main.xml
    - mipmap
      - ic_launcher (6)
      - ic_launcher_round (6)
    - values
      - colors.xml
      - strings.xml
      - styles.xml
- Gradle Scripts
  - build.gradle (Project: PDM18kotlin0)
  - build.gradle (Module: app)
  - gradle.properties (Global Properties)
  - gradle-wrapper.properties (Gradle Version)
  - proguard-rules.pro (ProGuard Rules for app)
  - gradle.properties (Project Properties)
  - settings.gradle (Project Settings)
  - local.properties (SDK Location)

activity_main.xml | MainActivity.kt

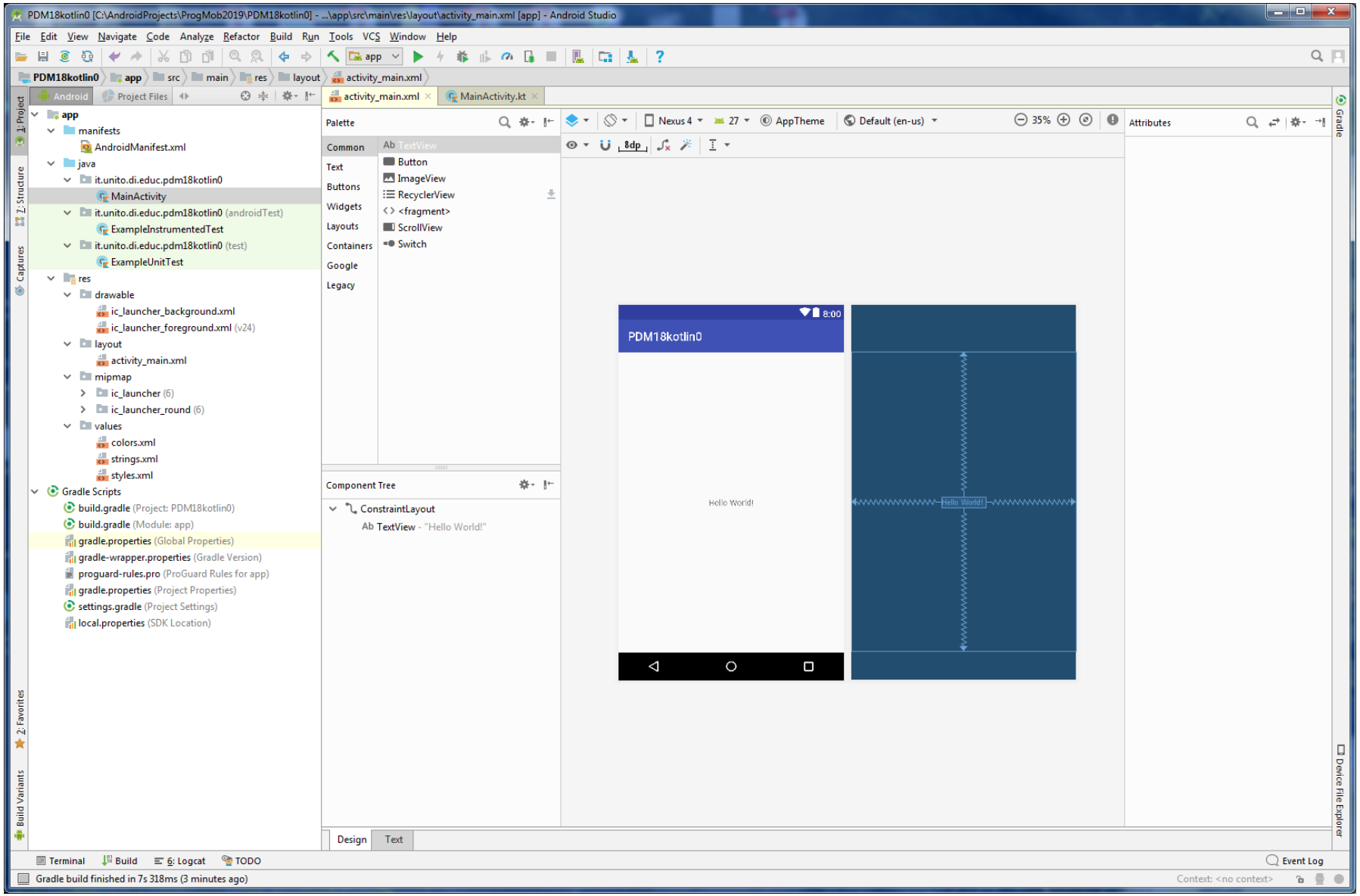
```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

Design | Text

Terminal | Build | 6: Logcat | TODO | Event Log

Gradle build finished in 7s 318ms (a minute ago)

1:1 CRLF UTF-8 Context: <no context>


PDM18kotlin0 [C:\AndroidProjects\ProgMob2019\PDM18kotlin0] - ...\app\src\main\res\layout\activity_main.xml [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
app
PDM18kotlin0 app src main res layout activity_main.xml
Android Project Files
activity_main.xml
MainActivity.kt
app
manifests
AndroidManifest.xml
java
it.unito.di.educ.pdm18kotlin0
MainActivity
it.unito.di.educ.pdm18kotlin0 (androidTest)
ExampleInstrumentedTest
it.unito.di.educ.pdm18kotlin0 (test)
ExampleUnitTest
res
drawable
ic_launcher_background.xml
ic_launcher_foreground.xml (v24)
layout
activity_main.xml
mipmap
ic_launcher (6)
ic_launcher_round (6)
values
colors.xml
strings.xml
styles.xml
Gradle Scripts
build.gradle (Project: PDM18kotlin0)
build.gradle (Module: app)
gradle.properties (Global Properties)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
Palette
Common
Text
Buttons
Widgets
Layouts
Containers
Google
Legacy
TextView
Button
ImageView
RecyclerView
<fragment>
ScrollView
Switch
Component Tree
ConstraintLayout
TextView - "Hello World!"
Nexus 4
27
AppTheme
Default (en-us)
35%
8dp
8:00
PDM18kotlin0
Hello World!
Attributes
Design
Text
Terminal
Build
6: Logcat
TODO
Event Log
Gradle build finished in 7s 318ms (3 minutes ago)
Context: <no context>

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

- `app/src/main/java/it/unito/di/educ/pdm18kotlin0/MainActivity.kt`

  A tab for this file appears in Android Studio when the New Project workflow finishes. When you select the file you see the class definition for the activity you created. When you build and run the app, the Activity class starts the activity and loads the layout file that says "Hello World!'

PDM18kotlin0 [C:\AndroidProjects\ProgMob2019\PDM18kotlin0] - ...\app\src\main\java\it\unito\di\educ\pdm18kotlin0\MainActivity.kt [app] - Android Studio

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

app

PDM18kotlin0 > app > src > main > java > it > unito > di > educ > pdm18kotlin0 > MainActivity.kt

Android | Project Files

- app
  - manifests
    - AndroidManifest.xml
  - java
    - it.unito.di.educ.pdm18kotlin0
      - MainActivity
    - it.unito.di.educ.pdm18kotlin0 (androidTest)
      - ExampleInstrumentedTest
    - it.unito.di.educ.pdm18kotlin0 (test)
      - ExampleUnitTest
  - res
    - drawable
      - ic_launcher_background.xml
      - ic_launcher_foreground.xml (v24)
    - layout
      - activity_main.xml
    - mipmap
      - ic_launcher (6)
      - ic_launcher_round (6)
    - values
      - colors.xml
      - strings.xml
      - styles.xml
- Gradle Scripts
  - build.gradle (Project: PDM18kotlin0)
  - build.gradle (Module: app)
  - gradle.properties (Global Properties)
  - gradle-wrapper.properties (Gradle Version)
  - proguard-rules.pro (ProGuard Rules for app)
  - gradle.properties (Project Properties)
  - settings.gradle (Project Settings)
  - local.properties (SDK Location)

activity_main.xml | MainActivity.kt

```
1  package it.unito.di.educ.pdm18kotlin0
2
3  import android.support.v7.app.AppCompatActivity
4  import android.os.Bundle
5
6  class MainActivity : AppCompatActivity() {
7
8      override fun onCreate(savedInstanceState: Bundle?) {
9          super.onCreate(savedInstanceState)
10         setContentView(R.layout.activity_main)
11     }
12 }
13
```

Terminal | Build | 6: Logcat | TODO | Event Log

Gradle build finished in 7s 318ms (3 minutes ago) 13:1 CRLF UTF-8 Context: <no context>

```kotlin
package it.unito.di.educ.pdm18kotlin0

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

- app/src/androidTest/java/it/unito/di/educ/pdm18kotlin0/ExampleInstrumentedTest.kt

  It is not necessary to have this class in your project to run the app. This class is used to test the application in a controlled environment.

```kotlin
package it.unito.di.educ.pdm18kotlin0

import android.support.test.InstrumentationRegistry
import android.support.test.runner.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getTargetContext()
        assertEquals("it.unito.di.educ.pdm18kotlin0", appContext.packageName)
    }
}
```

- app/src/test/java/it/unito/di/educ/pdm18kotlin0/ExampleUnitTest.kt

  It is not necessary to have this class in your project to run the app. This class is used to test the application in a controlled environment.

```
package it.unito.di.educ.pdm18kotlin0

import org.junit.Test

import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

- app/src/main/AndroidManifest.xml

  The manifest file describes the fundamental characteristics of the app and defines each of its components.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.unito.di.educ.pdm18kotlin0">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

- `app/build.gradle`

  Android Studio uses Gradle to compile and build your app. There is a build.gradle file for each module of your project, as well as a build.gradle file for the entire project. Usually, you're only interested in the build.gradle file for the module, in this case the app or application module. This is where your app's build dependencies are set, including the defaultConfig settings:

  - `applicationId` is the fully qualified package name for your application that you specified during the New Project workflow.
  - `minSdkVersion` is the Minimum SDK version you specified during the New Project workflow. This is the earliest version of the Android SDK that your app supports.
  - `targetSdkVersion` indicates the highest version of Android with which you have tested your application. As new versions of Android become available, you should test your app on the new version and update this value to match the latest API level and thereby take advantage of new platform features.

- The `res/` subdirectories contain the resources for your application:
  - `drawable<density>/` Directories for drawable objects (such as bitmaps) that are designed for various densities, such as medium-density (mdpi) and high-density (hdpi) screens.
  - `mipmap<density>/` Directories suggested for the `ic_launcher.png` and `ic_launcher_round.png` (that appears when you run the default app). The `mipmap<density>/` folders are for placing your app icons in only. Any other drawable assets you use should be placed in the relevant drawable folders. [a]
  - `layout/` Directory for files that define your app's user interface like activity_main.xml, discussed in previous slides, which describes a basic layout for the MainActivity class.
  - `values/` Directory for other XML files that contain a collection of resources, such as string and color definitions. The strings.xml file should define the "Hello world!" string that displays when you run the default app.

[a] It's best practice to place your app icons in mipmap- folders (not the drawable- folders) because they are used at resolutions different from the device's current density. For example, an `tealxxxhdpi` app icon can be used on the launcher for an `xxhdpi` device. Different home screen launcher apps on different devices show app launcher icons at various resolutions. When app resource optimization techniques remove resources for unused screen densities, launcher icons can wind up looking fuzzy because the launcher app has to upscale a lower-resolution icon for display. To avoid these display issues, apps should use the `mipmap/` resource folders for launcher icons. The Android system preserves these resources regardless of density stripping, and ensures that launcher apps can pick icons with the best resolution for display.

Outline

Run on a Real Device

If you have a device running Android, you can install and run your app on it.

Run on a Real Device

If you have a device running Android, you can install and run your app on it.

- Set up your device
  1. Plug in your device to your development machine with a USB cable. If you're developing on Windows, you might need to install the appropriate USB driver for your device. For help installing drivers, see the OEM USB Drivers document [`http://developer.android.com/tools/extras/oem-usb.html`].
  2. Enable USB debugging on your device.
     On most devices running Android 3.2 or older, you can find the option under **Settings** > **Applications** > **Development**. On Android 4.0 and newer, it's in **Settings** > **Developer options**.
     Note: On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings** > **About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.

Run on a Real Device

If you have a device running Android, you can install and run your app on it.

- Set up your device
  1. Plug in your device to your development machine with a USB cable. If you're developing on Windows, you might need to install the appropriate USB driver for your device. For help installing drivers, see the OEM USB Drivers document [`http://developer.android.com/tools/extras/oem-usb.html`].
  2. Enable USB debugging on your device.
     On most devices running Android 3.2 or older, you can find the option under **Settings** > **Applications** > **Development**. On Android 4.0 and newer, it's in **Settings** > **Developer options**.
     Note: On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to **Settings** > **About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.
- Run the app from Android Studio
  1. Select one of your project's files and click **Run** from the toolbar.
  2. In the **Choose Device** window that appears, select the **Choose a running device** radio button, select your device, and click **OK**.

  Android Studio installs the app on your connected device and starts it.

Run on the Emulator

Whether you're using Android Studio or the command line, to run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model a specific device.

Run on the Emulator

Whether you're using Android Studio or the command line, to run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model a specific device.

- Create an AVD
  1. Launch the Android Virtual Device Manager: in Android Studio, select **Tools** > **Android** > **AVD Manager**, or click the AVD Manager icon in the toolbar.
  2. On the AVD Manager main screen, click **Create Virtual Device.**
  3. In the Select Hardware window, select a device configuration, such as Nexus 6, then click **Next**.
  4. Select the desired system version for the AVD and click **Next**.
  5. Verify the configuration settings, then click **Finish**.

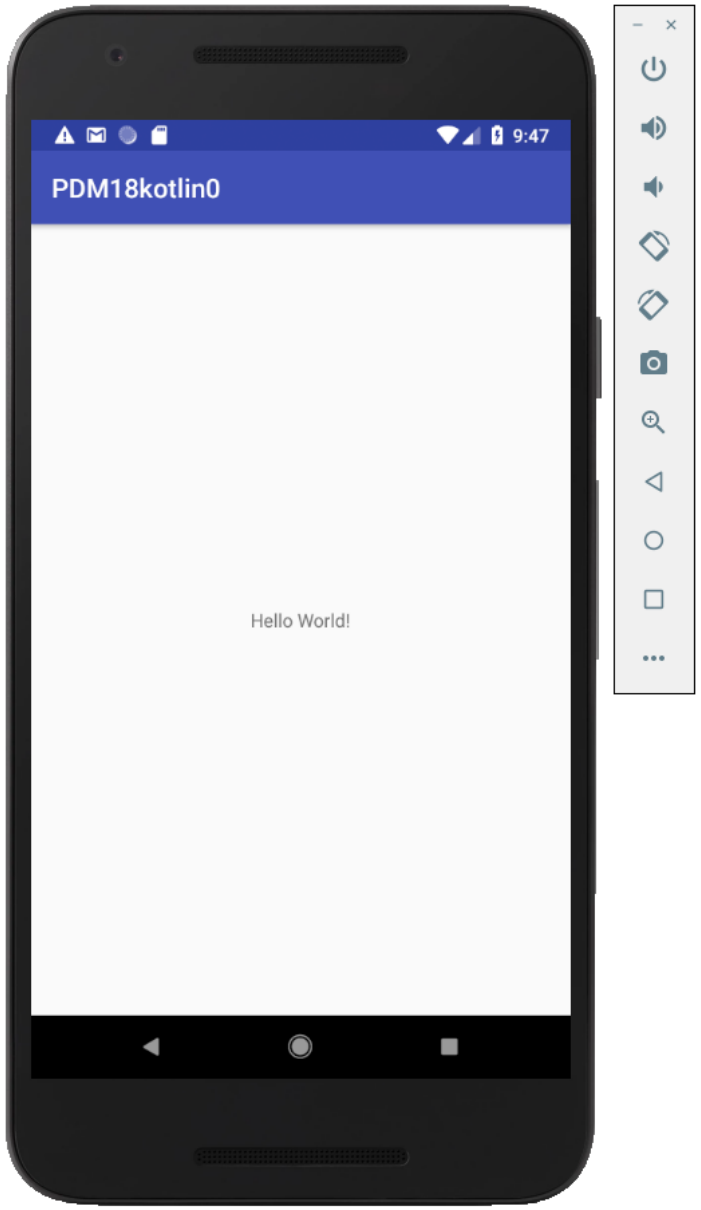
Run on the Emulator

Whether you're using Android Studio or the command line, to run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model a specific device.

- Create an AVD
  1. Launch the Android Virtual Device Manager: in Android Studio, select **Tools** > **Android** > **AVD Manager**, or click the AVD Manager icon in the toolbar.
  2. On the AVD Manager main screen, click **Create Virtual Device.**
  3. In the Select Hardware window, select a device configuration, such as Nexus 6, then click **Next**.
  4. Select the desired system version for the AVD and click **Next**.
  5. Verify the configuration settings, then click **Finish**.
- Run the app from Android Studio
  1. In Android Studio, select your project and click **Run** from the toolbar.
  2. In the Choose Device window, click the **Launch emulator** radio button.
  3. From the Android virtual device pull-down menu, select the emulator you created, and click **OK**.

It can take a few minutes for the emulator to load itself. You may have to unlock the screen. When you do, **PDM18kotlin0** appears on the emulator screen.

Example [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin0.git]



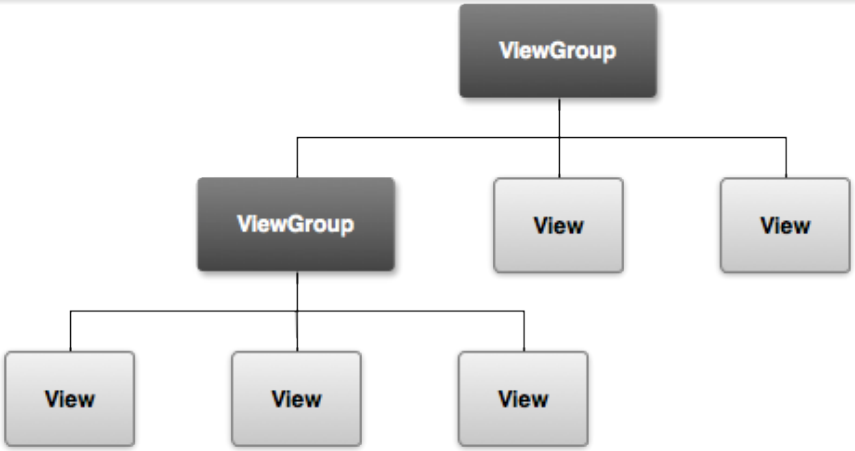
Outline

- In the next slides, we will create a layout in XML that includes a text field and a button.
- Namely, we will see how to:
  - Create a Linear Layout
  - Build an Intent
  - Add a Text Field
  - Add String Resources
  - Add a Button
  - Make the Input Box Fill in the Screen Width

The graphical user interface for an Android app is built using a hierarchy of `View` and `ViewGroup` objects.

- `View` objects are usually UI widgets such as **buttons** or **text fields**.
- `ViewGroup` objects are invisible view containers that define how the child views are laid out, such as in a grid or a vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of `View` and `ViewGroup` so you can define your UI in XML using a hierarchy of UI elements.
Layouts are subclasses of the `ViewGroup`.



In this example, we will work with a `LinearLayout`.

Create a Linear Layout

Create a new application with name "PDM18kotlin1" and an EmptyActivity (of name MainActivity). Then:

1. In Android Studio, from the res/layout directory, open the activity_main.xml file.
   The EmptyActivity template (you chose when you created this project) includes the activity_main.xml file with a ConstraintLayout root view and a TextView child view.
2. Delete the TextView element.
3. Change the <ConstraintLayout> to <LinearLayout>.
4. Add the android:orientation attribute and set it to "horizontal".
5. Remove the xmlns:app attribute and the tools:context attribute.

The result looks like this:

res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
</LinearLayout>
```

res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
</LinearLayout>
```

`LinearLayout` is a view group (a subclass of `ViewGroup`).

- It lays out child views in either a vertical or horizontal orientation, as specified by the `android:orientation` attribute.
  - Each child of a `LinearLayout` appears on the screen in the order in which it appears in the XML.
- Two other attributes, `android:layout_width` and `android:layout_height`, are required for all views in order to specify their size.
  - Because (in this example) the `LinearLayout` is the root view in the layout, it should fill the entire screen area that's available to the app by setting the width and height to `"match_parent"`. This value declares that the view should expand its width or height to match the width or height of the parent view.

Add a Text Field

As with every View object, you must define certain XML attributes to specify the EditText object's properties.

1. In the activity_main.xml file, within the <LinearLayout> element, define an <EditText> element with the id attribute set to @+id/edit_message.
2. Define the layout_width and layout_height attributes as wrap_content.
3. Define a hint attribute as a string object named edit_message.

The <EditText> element should read as follows:

res/layout/activity_main.xml

```
<EditText android:id="@+id/edit_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

res/layout/activity_main.xml

```
<EditText android:id="@+id/edit_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="@string/edit_message" />
```

EditText is a view (a subclass of View).

- The android:id attribute provides a unique identifier for the view, which you can use to reference the object from your app code, such as to read and manipulate the object (we will see this in the next slides).
  - The at sign (@) is required when you're referring to any resource object[a] from XML. It is followed by the resource type (id in this case), a slash, then the resource name (edit_message).
    - When you compile the app, the SDK tools use the ID name to create a new resource ID in your project's gen/R.java file that refers to the EditText element.
    - With the resource ID declared once this way, other references to the ID do not need the plus sign.
    - Using the plus sign is necessary only when specifying a new resource ID and not needed for concrete resources such as strings or layouts.

[a] For more information about resource objects see the **Resource Objects** sidebox in the next slide.

- The `android:layout_width` and `android:layout_width` attributes.
  - Instead of using specific sizes for the width and height, the `"wrap_content"` value specifies that the view should be only as big as needed to fit the contents of the view.
  - If you were to instead use `"match_parent"`, then the `EditText` element would fill the screen, because it would match the size of the parent `LinearLayout`.

Resource Objects

A resource object is a unique integer name that's associated with an app resource, such as a bitmap, layout file, or string.

- Every resource has a corresponding resource object defined in your project's `gen/R.java` file. You can use the object names in the `R` class to refer to your resources, such as when you need to specify a string value for the `android:hint` attribute. You can also create arbitrary resource IDs that you associate with a view using the `android:id` attribute, which allows you to reference that view from other code.
- The SDK tools generate the `R.java` file each time you compile your app. You should never modify this file by hand.

- The `android:hint` attribute.
  - This is a default string to display when the text field is empty.
  - Instead of using a hard-coded string as the value, the `"@string/edit_message"` value refers to a string resource defined in a separate file.[a]
    - Because this refers to a concrete resource (not just an identifier), it does not need the plus sign.

[a] This string resource has the same name as the element ID: `edit_message`. However, references to resources are always scoped by the resource type (such as `id` or `string`), so using the same name does not cause collisions.

Add String Resources

By default, your Android project includes a string resource file at `res/values/strings.xml`. Here, you'll add a new string named `"edit_message"` and set the value to "Enter a message."

1. In Android Studio, from the `res/values` directory, open `strings.xml`.
2. Add a line for a string named `"edit_message"` with the value, "Enter a message".
3. Add a line for a string named `"button_send"` with the value, "Send".

   You'll create the button that uses this string in the next slide.

The result for `strings.xml` looks like this:

`res/values/strings.xml`

```
<resources>
    <string name="app_name">PDM18kotlin1</string>
    <string name="edit_message">Enter a message</string>
    <string name="button_send">Send</string>
</resources>
```

Best Practice

For text in the user interface, always specify each string as a resource. String resources allow you to manage all UI text in a single location, which makes the text easier to find and update. Externalizing the strings also allows you to localize your app to different languages by providing alternative definitions for each string resource.

Add a Button

1. From the `res/layout` directory, edit the `activity_main.xml` file.
2. Within the `<LinearLayout>` element, define a `<Button>` element immediately following the `<EditText>` element.
3. Set the button's width and height attributes to `"wrap_content"` so the button is only as big as necessary to fit the button's text label.
4. Define the button's text label with the `android:text` attribute; set its value to the `button_send` string resource you defined in the previous slide.

Your `<LinearLayout>` should now look: `res/layout/activity_main.xml`[a]

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
      <EditText android:id="@+id/edit_message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
      <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

[a] This button doesn't need the `android:id` attribute, because it won't be referenced from the activity code.

The layout is currently designed so that both the `EditText` and `Button` widgets are only as big as necessary to fit their content, as shown here: 

(the `EditText` and `Button` widgets have their widths set to `"wrap_content"`). This works fine for the button, but not as well for the text field, because the user might type something longer. It would be nice to fill the unused screen width with the text field. You can do this inside a `LinearLayout` with the weight property, which you can specify using the `android:layout_weight` attribute.

Inside a `LinearLayout` the `android:layout_weight` attribute is such that:

- The weight value is a number that specifies the amount of remaining space each view should consume, relative to the amount consumed by sibling views. This works kind of like the amount of ingredients in a drink recipe: "2 parts soda, 1 part syrup" means two-thirds of the drink is soda.

Example

If you give one view a weight of 2 and another one a weight of 1, the sum is 3, so the first view fills 2/3 of the remaining space and the second view fills the rest. If you add a third view and give it a weight of 1, then the first view (with weight of 2) now gets 1/2 the remaining space, while the remaining two each get 1/4.

- The default weight for all views is 0, so if you specify any weight value greater than 0 to only one view, then that view fills whatever space remains after all views are given the space they require.

Make the Input Box Fill in the Screen Width

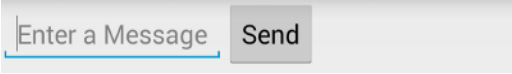
To fill the remaining space in your layout with the EditText element, do the following:

1. In the activity_main.xml file, assign the <EditText> element's layout_weight attribute a value of 1.
2. Also, assign <EditText> element's layout_width attribute a value of 0dp.

res/layout/activity_main.xml

```
<EditText
    android:layout_weight="1"
    android:layout_width="0dp"
    ... />
```

To improve the layout efficiency when you specify the weight, you should change the width of the EditText to be zero (0dp). Setting the width to zero improves layout performance because using "wrap_content" as the width requires the system to calculate a width that is ultimately irrelevant because the weight value requires another width calculation to fill the remaining space. The result is as shown here: 

(the EditText widget is given all the layout weight, so it fills the remaining space in the LinearLayout—compare with: 

).

Run Your App

Your complete activity_main.xml layout file should now look:

res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send" />
</LinearLayout>
```

This layout is applied by the default Activity class that the SDK tools generated when you created the project. In Android Studio, from the toolbar, click **Run** to run the app and see the results.

Outline

- Up to now we have an app that shows an activity (a single screen) with a text field and a button.
- In the next slides, we will add some code to `MainActivity` that starts a new activity when the user clicks the **Send** button.
- Namely, we will see how to:
  - Respond to the Send Button
  - Build an Intent
  - Create the Second Activity
  - Receive the Intent
  - Display the Message

Respond to the Send Button

1. From the `res/layout` directory, edit the `activity_main.xml` file.
2. To the `<Button>` element, add the `android:onClick` attribute.

   `res/layout/activity_main.xml`

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

   The `android:onClick attribute`'s value, `"sendMessage"`, is the name of a method in your activity that the system calls when the user clicks the button.

3. In the `java/it.unito.di.educ.pdm18kotlin1/` directory, open the `MainActivity.kt` file.
4. Within the `MainActivity` class, add the `sendMessage()` method stub shown below.

   `java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
/** Called when the user clicks the Send button */
fun sendMessage(view: View) {
    // Do something in response to button
}
```

java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt

```
/** Called when the user clicks the Send button */
fun sendMessage(view: View) {
    // Do something in response to button
}
```

In order for the system to match this method to the method name given to `android:onClick`, the signature must be exactly as shown. Specifically, the method must:

- Be public
- Have a void return value
- Have a `View` as the only parameter (this will be the `View` that was clicked)

In the next slides, we will fill in this method to read the contents of the text field and deliver that text to another activity.

Build an Intent

1. In `MainActivity.kt`, inside the `sendMessage()` method, create an `Intent` to start an activity called `DisplayMessageActivity` with the following code:[1]

   `java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
/** Called when the user clicks the Send button */
fun sendMessage(view: View) {
  val intent = Intent(this, DisplayMessageActivity::class.java)
}
```

Intents. An `Intent` is an object that provides runtime binding between separate components (such as two activities).

- The `Intent` represents an app's "intent to do something."
- You can use intents for a wide variety of tasks, but most often they're used to start another activity.

The constructor used here takes two parameters:

- A Context as its first parameter (this is used because the Activity class is a subclass of Context)
- The `Class` of the app component to which the system should deliver the `Intent` (in this case, the activity that should be started)

[1] Android Studio indicates that you must import the `Intent` class, and raise an error for the reference to `DisplayMessageActivity` because the class doesn't exist yet (ignore the error for now; we'll create the class soon).

2. At the top of the file, import the `Intent` class:

   `java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
import android.content.Intent
```

Tip

In Android Studio, press Alt + Enter (option + return on Mac) to import missing classes.

3. Inside the `sendMessage()` method, use `findViewById()` to get the `EditText` element.

   `java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
fun sendMessage(view: View) {
  val intent = Intent(this, DisplayMessageActivity::class.java)
  val edit_message = findViewById<EditText>(R.id.edit_message)
}
```

4. At the top of the file, import the `EditText` class.

5. Assign the text to a local message variable, and use the `putExtra()` method to add its text value to the intent.

   java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt

```
fun sendMessage(view: View) {
  val intent = Intent(this, DisplayMessageActivity::class.java)
  val edit_message = findViewById<EditText>(R.id.edit_message)
  val message = edit_message.text.toString()
  intent.putExtra(EXTRA_MESSAGE, message)
}
```

An `Intent` can carry data types as key-value pairs called extras. The `putExtra()` method takes the key name in the first parameter and the value in the second parameter.

6. At the top of the `MainActivity` class, add the `EXTRA_MESSAGE` definition as follows:

   java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt

```
const val EXTRA_MESSAGE = "it.unito.di.educ.pdm18kotlin1.MESSAGE"
class MainActivity : AppCompatActivity() {
    ...
}
```

   For the next activity to query the extra data, you should define the key for your intent's extra using a public constant.

Best Practice

Define keys for intent extras using your app's package name as a prefix. This ensures the keys are unique, in case your app interacts with other apps.

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

With this new code, the complete `sendMessage()` method that's invoked by the **Send** button now looks like this:

`java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
/** Called when the user clicks the Send button */
fun sendMessage(view: View) {
  val intent = Intent(this, DisplayMessageActivity::class.java)
  val edit_message = findViewById<EditText>(R.id.edit_message)
  val message = edit_message.text.toString()
  intent.putExtra(EXTRA_MESSAGE, message)
  startActivity(intent)
}
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`.

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

With this new code, the complete `sendMessage()` method that's invoked by the **Send** button now looks like this:

`java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
/** Called when the user clicks the Send button */
fun sendMessage(view: View) {
  val intent = Intent(this, DisplayMessageActivity::class.java)
  val edit_message = findViewById<EditText>(R.id.edit_message)
  val message = edit_message.text.toString()
  intent.putExtra(EXTRA_MESSAGE, message)
  startActivity(intent)
}
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`.

Now we need to create the `DisplayMessageActivity` class in order for this to work.

BETTER VERSION

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

With this new code, the complete `sendMessage()` method that's invoked by the **Send** button now looks like this:

`java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
    /** Called when the user taps the Send button */
    fun sendMessage(view: View) {
        val edit_message = findViewById<EditText>(R.id.edit_message)
        val message = edit_message.text.toString()
        val intent = Intent(this, DisplayMessageActivity::class.java).apply {
            putExtra(EXTRA_MESSAGE, message)
        }
        startActivity(intent)
    }
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`.

BETTER VERSION

7. In the `sendMessage()` method, to finish the intent, call the `startActivity()` method, passing it the `Intent` object created in step 1.

With this new code, the complete `sendMessage()` method that's invoked by the **Send** button now looks like this:

`java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt`

```
    /** Called when the user taps the Send button */
    fun sendMessage(view: View) {
        val edit_message = findViewById<EditText>(R.id.edit_message)
        val message = edit_message.text.toString()
        val intent = Intent(this, DisplayMessageActivity::class.java).apply {
            putExtra(EXTRA_MESSAGE, message)
        }
        startActivity(intent)
    }
```

The system receives this call and starts an instance of the `Activity` specified by the `Intent`.

Now we need to create the `DisplayMessageActivity` class in order for this to work.

Create the Second Activity

All subclasses of `Activity` must implement the `onCreate()` method.

- This method is where the activity receives the intent with the message, then renders the message.
- Also, the `onCreate()` method must define the activity layout with the `setContentView()` method.
- This is where the activity performs the initial setup of the activity components.

Android Studio includes a stub for the `onCreate()` method when you create a new activity.

1. In Android Studio, in the java directory, select the package, com.mycompany.myfirstapp, right-click, and select **New** > **Activity** > **Empty Activity**.
2. In the **Choose options** window, fill in the activity details:
   - **Activity Name**: DisplayMessageActivity
   - **Layout Name**: activity_display_message
   - **Package name**: it.unito.di.educ.PDM18kotlin1

   Click **Finish**.

3. Open the `DisplayMessageActivity.kt` file.
   The class already includes an implementation of the required `onCreate()` method.
   We will update the implementation of this method later.

With Android Studio, you can run the app now, but not much happens. Clicking the **Send** button starts the second activity, but it uses a default "Hello world" layout provided by the template.

3. Open the `DisplayMessageActivity.kt` file.
   The class already includes an implementation of the required `onCreate()` method.
   We will update the implementation of this method later.

With Android Studio, you can run the app now, but not much happens. Clicking the **Send** button starts the second activity, but it uses a default "Hello world" layout provided by the template.

In the next slides we will update the activity to instead display a custom text view.

Receive the Intent

Every `Activity` is invoked by an `Intent`, regardless of how the user navigated there. You can get the `Intent` that started your activity by calling `getIntent()` and retrieve the data contained within the intent.

1. In the `java/it.unito.di.educ.pdm18kotlin1/` directory, open the `DisplayMesageActivity.kt` file.
2. In the `onCreate()` method, remove the following line:

```
setContentView(R.layout.activity_display_message)
```

3. Get the intent and assign it to a local variable.

```
//Intent intent = getIntent(); // l'oggetto intent è definito nel Companion Object di Kotlin -
      verificare
```

4. At the top of the file, import the `Intent` class.
5. Extract the message delivered by `MainActivity` with the `getStringExtra()` method.

```
val message = intent.getStringExtra(EXTRA_MESSAGE)
```

Display the Message

1. In the `onCreate()` method, create a `TextView` object.

```
var textView = TextView(this)
```

2. Set the text size and message with `setText()`.

```
textView.setTextSize(TypedValue.COMPLEX_UNIT_DIP, 40f)
textView.text = message
```

3. Then add the `TextView` as the root view of the activity's layout by passing it to `setContentView()`.

```
setContentView(textView)
```

4. At the top of the file, import the `TextView` class.

Run Your App

The complete `onCreate()` method for `DisplayMessageActivity` now looks like this:

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Get the message from the intent
        val message = intent.getStringExtra(EXTRA_MESSAGE)

        // Create the text view
        var textView = TextView(this)
        textView.textSize = 40f
        textView.text = message

        // Set the text view as the activity layout
        setContentView(textView)
    }
```

You can now run the app. When it opens, type a message in the text field, click **Send**, and the message appears on the second activity.

Run Your App - BETTER VERSION

The complete `onCreate()` method for `DisplayMessageActivity` now looks like this:

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Create and initialize the text view
        var textView = TextView(this).apply {
            textSize = 40f
            text = intent.getStringExtra(EXTRA_MESSAGE)
        }

        // Set the text view as the activity layout
        setContentView(textView)
    }
```

You can now run the app. When it opens, type a message in the text field, click **Send**, and the message appears on the second activity.

Example

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin1.git]