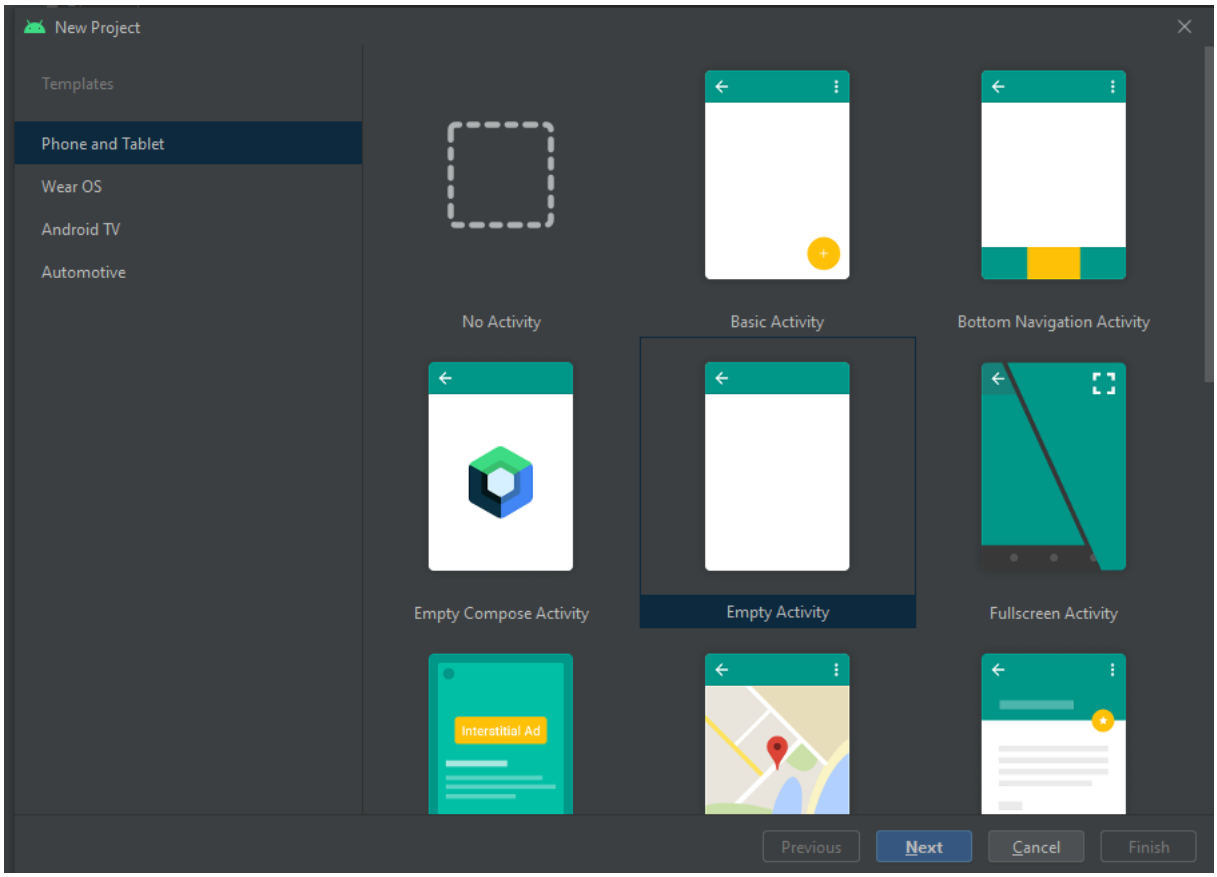
Sensors

Most mobile devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, 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. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

During these labs, a sensor reading application will be created and then rebuilt to use the MVVM design pattern.

1. Create a New project -> Empty Activity



2. Define App Name (***Lab5***) and Package (***edu.zut.erasmus_plus.sensors***), choose minimum API (***API28***)

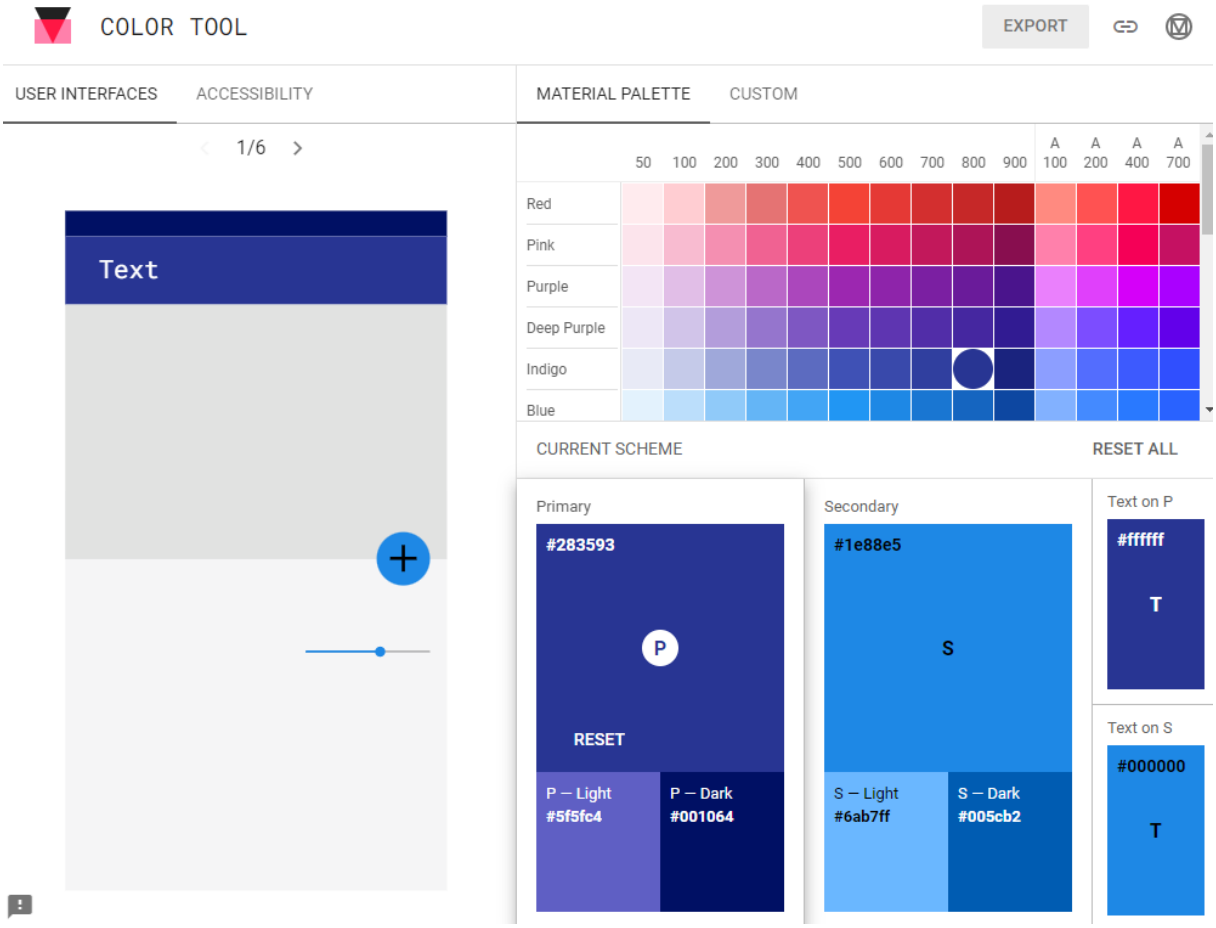
Funded by
the European Union

3. Explore the code
    1. Locate MainActivity.kt, activity_main.xml, AndroidManifest.xml
4. Go to build.gradle -> Upgrade all dependencies and libraries for Project and Module (we can skip)

5. Run App

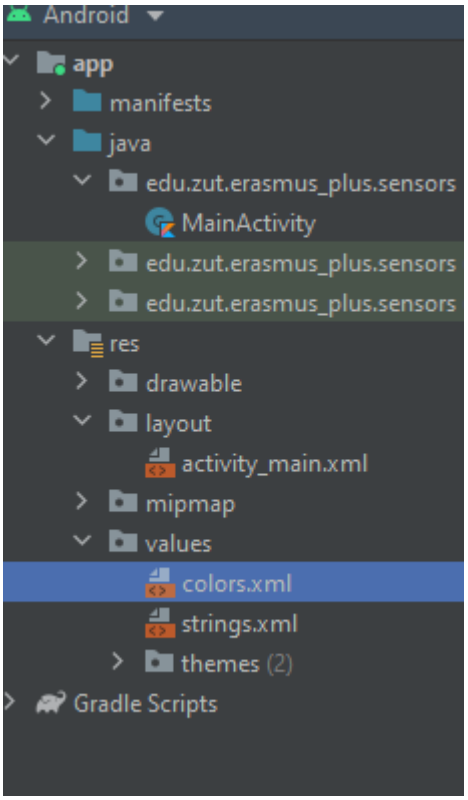
6. Create a colours schema (not obligatory)

   Before defining layout, please use [Color Tool - Material Design] and define your app colours; then, use this colour when defining elements.
   Example colour:
   https://material.io/resources/color/#!/?view.left=0&view.right=0&primary.color=283593&secondary.color=1E88E5
   More information about colour: https://material.io/design/color/the-color-system.html#color-theme-creation



   Once you have selected the correct colour scheme, go to the ACCESSIBILITY tab and check the warnings.
   The next step is to export the configuration. At the top of the screen, select the button EXPORT and save colors.xml.
   Under the project, open colors.xml and insert the value from the downloaded file.

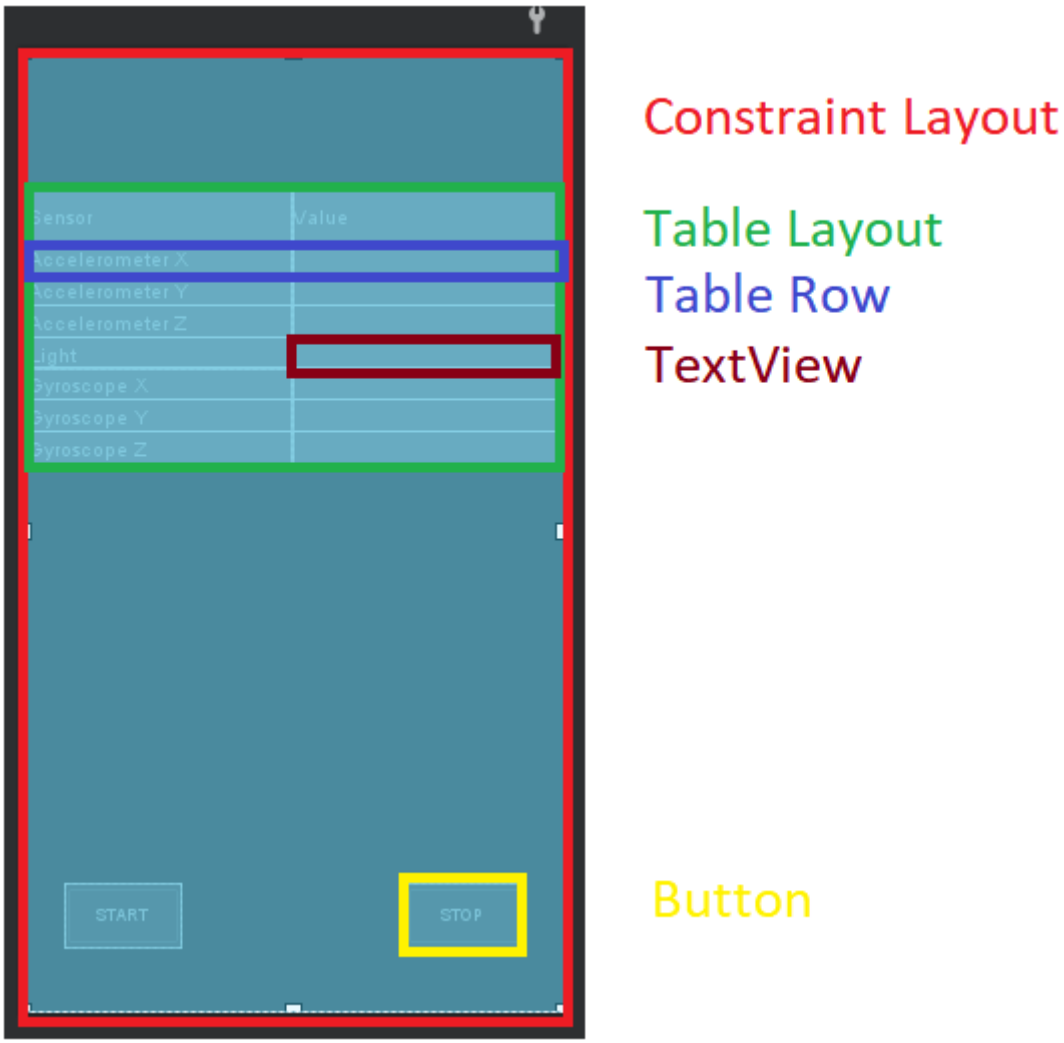
Funded by the European Union



7. Design Layout

Please design layout like this:

Funded by
the European Union



This layout was built using two types of layouts, ConstraintLayout and TableLayout

Skeleton

```
<?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="wrap_content"
    android:layout_height="match_parent"
    android:background="@color/design_default_color_background"
    tools:context=".MainActivity">

    <TableLayout
        android:id="@+id/tableLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="100dp"
        android:padding="5dp"
        android:stretchColumns="2"
        app:layout_constraintTop_toTopOf="parent"
        tools:context=".MainActivity"
        tools:layout_editor_absoluteX="16dp">
```

Funded by
the European Union

```
        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <TextView
                android:layout_width="187dp"
                android:layout_height="match_parent"
                android:layout_column="1"
                android:background="@color/primaryLightColor"
                android:gravity="center"
                android:text="@string/sensor_name"
                android:textColor="@color/primaryTextColor"
                android:textSize="18sp"
                android:textStyle="bold" />

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="35dp"
                android:layout_column="2"
                android:background="@color/primaryDarkColor"
                android:gravity="center"
                android:text="@string/value_sensor"
                android:textColor="@color/primaryTextColor"
                android:textSize="18sp"
                android:textStyle="bold" />
        </TableRow>
```

Please finish, this is only the beginning

Funded by
the European Union

8. Go to the MainActivity.kt and inside onCreate() add SensorManager and Sensor Object

```
mSensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
mGyroscope = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
```

Previously define objects as global for the class e.g.

```
private lateinit var mSensorManager : SensorManager
private var mAccelerometer : Sensor ?= null
...
```

9. Add interface to MainActivity class

```
class MainActivity : AppCompatActivity(), SensorEventListener {
```

10. Add callback methods ***onAccuracyChanged, onSensorChanged***

```
11. override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {
        print("accuracy changed")
    }

    override fun onSensorChanged(event: SensorEvent?) {
        if (event != null && resume) {
            if (event.sensor.type == Sensor.TYPE_ACCELEROMETER) {
                findViewById<TextView>(R.id.acc_X).text =
    event.values[0].toString()
                findViewById<TextView>(R.id.acc_Y).text =
    event.values[1].toString()
                findViewById<TextView>(R.id.acc_Z).text =
    event.values[2].toString()
            }

            if (event.sensor.type == Sensor.TYPE_LIGHT) {
                findViewById<TextView>(R.id.light).text =
    event.values[0].toString()
            }

            if (event.sensor.type == Sensor.TYPE_GYROSCOPE) {
                findViewById<TextView>(R.id.gyro_x).text =
    event.values[0].toString()
                findViewById<TextView>(R.id.gyro_y).text =
    event.values[1].toString()
                findViewById<TextView>(R.id.gyro_z).text =
    event.values[2].toString()
            }
        }
    }
```

Make sure that the object names in the code (R.id.XXXX) were consistent with the ID names in the layout.

12. Define auxiliary methods

```
private fun registerListener()
{
    this.resume = true

    mSensorManager.registerListener(this, mAccelerometer,
SensorManager.SENSOR_DELAY_NORMAL)
    mSensorManager.registerListener(this, mLight,
SensorManager.SENSOR_DELAY_NORMAL)
    mSensorManager.registerListener(this, mGyroscope,
SensorManager.SENSOR_DELAY_NORMAL)

    changeButtonStatus()
}
private fun unRegisterListener()
{
    this.resume = false

    mSensorManager.unregisterListener(this)
    changeButtonStatus()
}
```

Funded by
the European Union

```
private fun changeButtonStatus()
{
    findViewById<Button>(R.id.start_button).isEnabled = !resume
    findViewById<Button>(R.id.stop_button).isEnabled = resume
}
```

and also add variable **resume**

```
private var resume = false
```

13. Use previous methods to register and unregister SensorsListener.

```
override fun onResume() {
    super.onResume()

    registerListener()
}

override fun onPause() {
    super.onPause()

    unRegisterListener()
}
```

Why do we register and unregister with these methods?

14. Add onClick() methods (define also at layout)

```
fun resumeReading(view: View) {
    registerListener()
}
fun pauseReading(view: View) {
    unRegisterListener()
}
```

15. Run the app
16. Definition of different colours depending on button status
    Create background_button.xml and add code

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/primaryColor" android:state_enabled="true" />
    <item android:drawable="@color/secondaryColor" android:state_enabled="false"
/>
    <!-- default state-->
    <item android:drawable="@color/primaryColor" />
</selector>
```

17. Change definition background for each button

```
android:background="@drawable/button_background"
```

18. Run app

Funded by
the European Union

MVVM, LiveData

The above application shows the basic approach to programming in Android. It is now recommended that Android applications be developed using the MVVM (Model - View - ViewModel) design pattern.

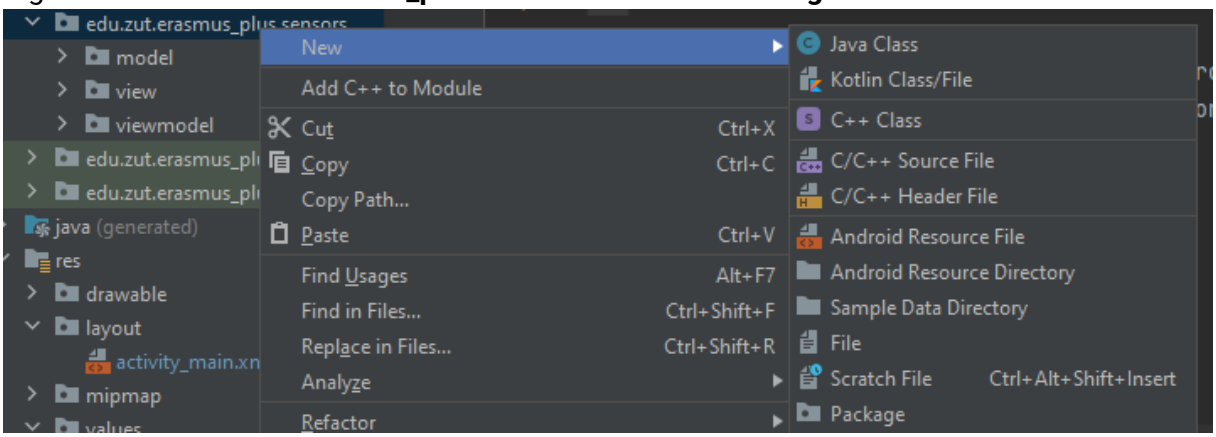
The following is a solution using the Architecture Components introduced in the library AndroidX

We are going to change the application we have just built into an application that follows the MVVM design pattern

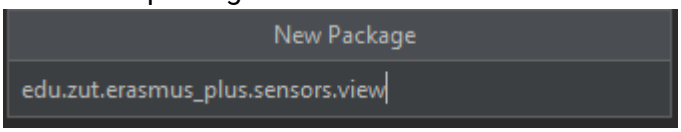
1. Prepare packages
   Create packages

- **edu.zut.erasmus_plus.sensors.viewmodel**
- **edu.zut.erasmus_plus.sensors.view**
- **edu.zut.erasmus_plus.sensors.model**

   Right click on **edu.zut.erasmus_plus.sensors -> new -> Package**



   Then write package name.



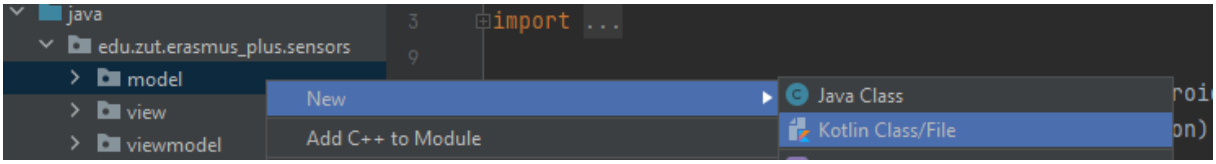
2. Move MainActivity.kt to **edu.zut.erasmus_plus.sensors.view**
   All activities and fragments are classified as a view in the MVVM model. Then drag and drop MainActivity.kt to the proper package name.

3. Create Data Class
   https://kotlinlang.org/docs/data-classes.html

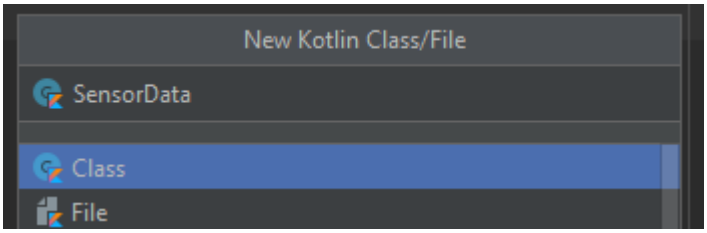
   A data class is a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.
   a) Create file SensorData.kt inside **edu.zut.erasmus_plus.sensors.model**

Funded by the European Union

Right click on model->New-> Kotlin Class/File



Give name



b) Define class

Inside SensorData.kt define all variable

```
package edu.zut.erasmus_plus.sensors.model


data class SensorData(
    var accX: Float,
    var accY: Float,
    var accZ: Float,
    var gyroX: Float,
    var gyroY: Float,
    var gyroZ: Float,
    var light: Float
)
```

4. Data Binding
   Detailed information: https://developer.android.com/topic/libraries/data-binding

   Once the interface has been created, it is necessary to link existing objects to the code. The currently recommended approach is to use Data Binding. In addition to automatically creating the code, it also allows you to update the assembled view when the data changes using LiveData. The Data Binding Library was built with observability in mind, a pattern that has become quite popular in mobile application development. Observability is a complement to data binding, whose basic concept only considers the view and data objects. However, it is through this pattern that data can automatically propagate its changes to the view. This eliminates the need to manually update views every time new data is available, which simplifies the code base and reduces the amount of template code.

   Our application will use the previously created date class to broadcast information about sensor changes

5. Enabling Data Binding
   You'll now enable data binding in the project. Open the app's build.gradle file, and add this line inside android tag.

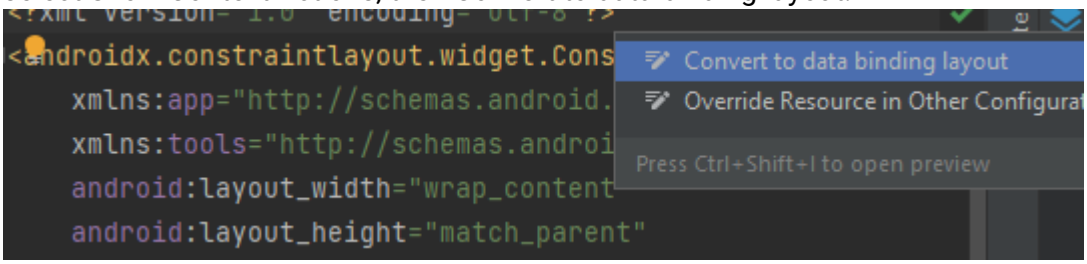
```
android {
    compileSdk 31

    dataBinding {
        enabled true
    }
```

Now convert the layout to a Data Binding layout.

To convert a regular layout to Data Binding layout:

1. Wrap your layout with a <layout> tag
2. Add layout variables (optional)
3. Add layout expressions (optional)

Or
Android Studio offers a handy way to do this automatically: Right-click the root element, select Show Context Actions, then Convert to data binding layout:




```
<?xml version="1.0" encoding="utf-8"?>
<layout 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">

    <data>

    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:background="@color/cardview_shadow_start_color"
        tools:context=".view.MainActivity">
```

The **<data>** tag will contain layout variables. We will add values there later

Layout variables are used to write **layout expressions**. Layout expressions are placed in the value of element attributes, and they use the *@{expression}* format. For example:

Funded by
the European Union

```
android:text="@{String.valueOf(index + 1)}"
android:visibility="@{age < 13 ? View.GONE : View.VISIBLE}"
android:transitionName='@{"image_" + id}'

// Bind the name property of the viewmodel to the text attribute
android:text="@{viewmodel.name}"
// Bind the nameVisible property of the viewmodel to the visibility attrib-
ute
android:visibility="@{viewmodel.nameVisible}"
// Call the onLike() method on the viewmodel when the View is clicked.
android:onClick="@{() -> viewmodel.onLike()}">
```

More info about layout expression https://developer.android.com/topic/libraries/data-binding/expressions#expression_language

6. Create class to reading sensors value
   Now we will move all the code responsible for collecting values for the sensors to a separate class
   a) Delete code from Activity_Main.kt

```
class MainActivity : AppCompatActivity(), SensorEventListener {
    private lateinit var mSensorManager : SensorManager
    private var mAccelerometer : Sensor ?= null
    private var mLight : Sensor ?= null
    private var mGyroscope : Sensor ?= null
    private var resume = false
```

   Now we have many errors, please delete unnecessary methods and references to deleted objects.
   After this, MainActivity.kt looks like this:

```
package edu.zut.erasmus_plus.sensors

import ...
class MainActivity : AppCompatActivity(){

    override fun onCreate(savedInstanceState: Bundle?) {
        requestWindowFeature(Window.FEATURE_NO_TITLE)
        requestedOrientation = ActivityInfo.SCREEN_ORIENTATION_PORTRAIT

        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

   Practically all existing code has been removed.

   b) Create file **SensorDataLiveData.kt** inside **edu.zut.erasmus_plus.sensors.model**

Funded by the European Union

```
class SensorDataLiveData(
    context: Context,
    private val sensorDelay: Int = SensorManager.SENSOR_DELAY_UI
) : LiveData<SensorData>(), SensorEventListener {

    private val mSensorManager: SensorManager =
        context.getSystemService(Context.SENSOR_SERVICE) as SensorManager
    private val accelerometer: Sensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    private val gyroscope: Sensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
    private val light: Sensor =
mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)

    private val mAccelerometerReading = FloatArray(3)
    private val mGyroscopeReading = FloatArray(3)
    private val mLightReading = FloatArray(1)

    override fun onActive() {
        super.onActive()
        registerListeners()
    }
    override fun onInactive() {
        super.onInactive()
        unregisterListeners()
    }
    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {}

    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor == accelerometer) {
            System.arraycopy(event.values, 0, mAccelerometerReading, 0,
mAccelerometerReading.size)
        } else if (event.sensor == gyroscope) {
            System.arraycopy(event.values, 0, mGyroscopeReading, 0,
mGyroscopeReading.size)
        } else if (event.sensor == light) {
            System.arraycopy(event.values, 0, mLightReading, 0,
mLightReading.size)
        }

        value = SensorData(
            mAccelerometerReading[0], mAccelerometerReading[1],
mAccelerometerReading[2],
            mGyroscopeReading[0], mGyroscopeReading[1],
mGyroscopeReading[2],
            mLightReading[0]
        )
    }

    fun unregisterListeners() {
        mSensorManager.unregisterListener(this)
    }

    fun registerListeners() {
        mSensorManager.registerListener(
            this,
            accelerometer,
            SensorManager.SENSOR_DELAY_NORMAL,
            sensorDelay
        )
        mSensorManager.registerListener(
```

Funded by
the European Union

Now, this class is responsible for handling the sensors. Our goal is to have every change be communicated using observers. We will use the **LiveData** class for this purpose. Changes will observe changes on the previously created **SensorData** model. The class definition takes the following form.

```
class SensorDataLiveData(context: Context ) : LiveData<SensorData>(),
SensorEventListener
```

With the methods from the LiveData class (**onActive()** and **onInactive()**) it can start or stop collecting events from the sensors. The **onActive()** method is run when the first observer joins our class and the second when the last one disconnects.
The rest of the code is consistent with what was previously implemented in **MainActivity()**, except that in the **onSensorChanged()** method, the **value** object returns the result. In our case, it is an object of **SensorData** class with the last values from sensor readings.

7. Create ViewModel

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

Create file SensorViewModel.kt inside **edu.zut.erasmus_plus.sensors.viewmodel**

Inside file adds two private variables and their get methods.

```
class SensorViewModel(application: Application): AndroidViewModel(applica-
tion) {
  private val _sensor = SensorDataLiveData(application)
  private var _pauseReading = MutableLiveData<Boolean>()

  val sensor: LiveData<SensorData>
    get() = _sensor

  fun getPauseReading(): MutableLiveData<Boolean> {
    return _pauseReading
  }
```

Pierwsza zmienna służy do odczytywania wartości sensorów, z wcześniej stworzonej klasy SensorDataLiveData, gdzie konieczne jest przekazanie kontekstu aplikacji. Stąd modyfikujemy definicję klasy jak powyżej. Proszę zauważyć iż nasza klasa dziedziczy po AndroidViewModel.

The **_pauseReading** variable determines if we stop/start reading the sensors. It needs to be initialised, so we add the following code to the SensorViewModel:

```
init {
    _pauseReading = MutableLiveData(false)
}
```

Funded by
the European Union

The last method that needs to be added is the correct response to stop/start sensor reading. Add the following function to the SensorViewModel:

```
fun changeButtonStatus()
{
  if(_pauseReading.value==true) _sensor.registerListeners()
  else _sensor.unregisterListeners()
  _pauseReading.value?.let {
    _pauseReading.value = !it
  }
}
```

8. Change layout

After creating VM, we must change layout and add a link to the VM

a) Add information about variable

Open activity_mail.xml

Inside properties **<data> </data>** insert new variable

```
<data>
    <variable
        name="sensorViewModel"
        type="edu.zut.erasmus_plus.sensors.viewmodel.SensorViewModel" />
</data>
```

Adding variable (**sensorViewModel**) will allow you to use objects from the **SensorViewModel** class

b) Change properties inside Open activity_mail.xml

Find the **TextView** object responsible for displaying the **acc_X** value (around line 64) and change the **android:text** property.

```
<TextView
    android:id="@+id/acc_X"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="2"
    android:text="@{String.valueOf(sensorViewModel.sensor.accX)}"
    android:textAlignment="center"
    android:textSize="18sp" />
```

c) Change the **android:text** property for all objects representing the reading value from the sensors (7 times)

d) Change onClick and enable properties for Button

Change as follow.

Funded by
the European Union

```
<Button
    android:id="@+id/start_button"
. . .
    android:enabled="@{sensorViewModel.pauseReading}"
    android:onClick="@{()->sensorViewModel.changeButtonStatus()}"


. . .
 />

<Button
    android:id="@+id/stop_button"
. . .
    android:enabled="@{!sensorViewModel.pauseReading}"
    android:onClick="@{()->sensorViewModel.changeButtonStatus()}"
. . .
 />
```

Only the changed parts are shown above.

9. Change view – MainActivity.kt
   a) Open MainActivity.kt
   b) Inside onCreate, change code like this:

```
class MainActivity : AppCompatActivity() {
    private var resume = false

    private lateinit var binding: ActivityMainBinding
    private val sensorViewModel: SensorViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        requestWindowFeature(Window.FEATURE_NO_TITLE)
        requestedOrientation = ActivityInfo.SCREEN_ORIENTATION_PORTRAIT

        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        binding.sensorViewModel = sensorViewModel
        binding.lifecycleOwner = this
    }
}
```

Notice that we added two variables, binding and **sensorViewModel**. The binding variable uses DataBinding and provides us with automatic access to variables in the layout without using findViewById().

The sensorViewModel variable binds the created VM to the view.

10. Run Application

The above steps allowed us to use the now recommended model of creating applications using the Android Components defined in the androidX library.

Funded by
the European Union

Materials developed as part of the project:
Innovative Open Source courses for Computer Science curriculum

Publikacja sfinansowana z funduszy Komisji Europejskiej w ramach programu Erasmus+ Publikacja została zrealizowana przy wsparciu finansowym Komisji Europejskiej. Publikacja odzwierciedla jedynie stanowisko jej autorów i Komisja Europejska oraz Narodowa Agencja Programu Erasmus+ nie ponoszą odpowiedzialności za jej zawartość merytoryczną. PUBLIKACJA BEZPŁATNA

Funded by
the European Union