BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 5

Karim Karimov
Lecturer

Touch gestures

Android provides a variety of APIs to help you create and detect gestures.
Although your app should not depend on touch gestures for basic behaviors (since the gestures may not be available to all users in all contexts), adding touch-based interaction to your app can greatly increase its usefulness and appeal.
To provide users with a consistent, intuitive experience, your app should follow the accepted Android conventions for touch gestures.

Input events

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The **View** class provides the means to do so.

Within the various View classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a **View** (such as a **Button**) is touched, the **onTouchEvent()** method is called on that object. However, in order to intercept this, you must extend the class and override the method. However, extending every View object in order to handle such an event would not be practical. This is why the View class also contains a collection of **nested interfaces** with callbacks that you can much more easily define. These interfaces, called **event listeners**, are your ticket to capturing the user interaction with your UI.

Input events

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the **Button** class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class **event handlers**.

Event listeners

An event listener is an interface in the View class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI. Included in the event listener interfaces are the following callback methods:

onClick()
From **View.OnClickListener**. This is called when the user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.

Event listeners

onLongClick()
From **View.OnLongClickListener**. This is called when the user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).

onTouch()
From **View.OnTouchListener**. This is called when the user performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen (within the bounds of the item).

Event listeners

These methods are the sole inhabitants of their respective interface. To define one of these methods and handle your events, implement the nested interface in your Activity or define it as an anonymous class. Then, pass an instance of your implementation to the respective **View.set...Listener()** method. (E.g., call **setOnClickListener()** and pass it your implementation of the **OnClickListener**.)
The example below shows how to register an on-click listener for a Button.

```
protected void onCreate(savedValues: Bundle) {
    ...
    val button: Button = findViewById(R.id.corky)
    // Register the onClick listener with the implementation above
    button.setOnClickListener { view ->
        // do something when the button is clicked
    }
    ...
}
```

Event listeners

You may also find it more convenient to implement OnClickListener as a part of your Activity. This will avoid the extra class load and object allocation. For example:

```
class ExampleActivity : Activity(), OnClickListener {

    protected fun onCreate(savedValues: Bundle) {
        val button: Button = findViewById(R.id.corky)
        button.setOnClickListener(this)
    }

    // Implement the OnClickListener callback
    fun onClick(v: View) {
        // do something when the button is clicked
    }
}
```

Event listeners

Notice that the **onClick()** callback in the above example has no return value, but some other event listener methods must return a boolean. The reason depends on the event. For the few that do, here's why:

- **onLongClick()** - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return true to indicate that you have handled the event and it should stop here; return false if you have not handled it and/or the event should continue to any other on-click listeners.
- **onKey()** - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return true to indicate that you have handled the event and it should stop here; return false if you have not handled it and/or the event should continue to any other on-key listeners.
- **onTouch()** - This returns a boolean to indicate whether your listener consumes this event. The important thing is that this event can have multiple actions that follow each other. So, if you return false when the **down action** event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event. Thus, you will not be called for any other actions within the event, such as a **finger gesture**, or the eventual **up action event**.

Event handlers

If you're building a custom component from View, then you'll be able to define several callback methods used as default event handlers. In the document about Custom View Components, you'll learn some of the common callbacks used for event handling, including:

- **onKeyDown(int, KeyEvent)** - Called when a new key event occurs.
- **onKeyUp(int, KeyEvent)** - Called when a key up event occurs.
- **onTouchEvent(MotionEvent)** - Called when a touch screen motion event occurs.
- **onFocusChanged(boolean, int, Rect)** - Called when the view gains or loses focus.

Handling focus

Focus movement is based on an algorithm which finds the nearest neighbor in a given direction. In rare cases, the default algorithm may not match the intended behavior of the developer. In these situations, you can provide explicit overrides with the following XML attributes in the layout file: **nextFocusDown**, **nextFocusLeft**, **nextFocusRight**, and **nextFocusUp**. Add one of these attributes to the View from which the focus is leaving. Define the value of the attribute to be the id of the View to which focus should be given. For example:

```
<LinearLayout
   android:orientation="vertical"
   ... >
  <Button android:id="@+id/top"
          android:nextFocusUp="@+id/bottom"
          ... />
  <Button android:id="@+id/bottom"
          android:nextFocusDown="@+id/top"
          ... />
</LinearLayout>
```

Handling focus

Ordinarily, in this vertical layout, navigating up from the first Button would not go anywhere, nor would navigating down from the second Button. Now that the top Button has defined the bottom one as the nextFocusUp (and vice versa), the navigation focus will cycle from top-to-bottom and bottom-to-top.
If you'd like to declare a View as focusable in your UI (when it is traditionally not), add the **android:focusable** XML attribute to the View, in your layout declaration. Set the value true. You can also declare a View as focusable while in Touch Mode with **android:focusableInTouchMode**.
To request a particular View to take focus, call **requestFocus()**.

Sensors

Most Android-powered 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.

Sensors

The Android platform supports three broad categories of sensors:

- **Motion sensors**
  - These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- **Environmental sensors**
  - These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- **Position sensors**
  - These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

Sensors

You can access sensors available on the device and acquire raw sensor data by using the **Android sensor framework**. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

Supported sensors

Sensor	Type	Description	Common Uses
TYPE_ACCELEROMETER	Hardware	Measures the acceleration force in m/s^2 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.).
TYPE_AMBIENT_TEMPERATURE	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
TYPE_GRAVITY	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
TYPE_GYROSCOPE	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.).
TYPE_LIGHT	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
TYPE_LINEAR_ACCELERATION	Software or Hardware	Measures the acceleration force in m/s^2 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.

Supported sensors

`TYPE_MAGNETIC_FIELD`	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	Creating a compass.
`TYPE_ORIENTATION`	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 `getRotationMatrix()` method.	Determining device position.
`TYPE_PRESSURE`	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
`TYPE_PROXIMITY`	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.
`TYPE_RELATIVE_HUMIDITY`	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
`TYPE_ROTATION_VECTOR`	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.
`TYPE_TEMPERATURE`	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the `TYPE_AMBIENT_TEMPERATURE` sensor in API Level 14	Monitoring temperatures.

Sensor Framework

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the android.hardware package and includes the following classes and interfaces:

SensorManager

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

Sensor

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

Sensor Framework

SensorEvent

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

SensorEventListener

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor **values change** or when sensor **accuracy changes**.

Sensor Framework

In a typical application you use these sensor-related APIs to perform two basic tasks:

Identifying sensors and sensor capabilities

Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

Monitor sensor events

Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the **name** of the sensor that triggered the event, the **timestamp** for the event, the **accuracy** of the event, and the **raw sensor data** that triggered the event.

Identifying Sensors

The Android sensor framework provides several methods that make it easy for you to determine at runtime which sensors are on a device. The API also provides methods that let you determine the capabilities of each sensor, such as its maximum range, its resolution, and its power requirements.
To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the SensorManager class by calling the **getSystemService()** method and passing in the **SENSOR_SERVICE** argument. For example:

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

// get all sensors
val deviceSensors: List<Sensor> = sensorManager.getSensorList(Sensor.TYPE_ALL)
```

Identifying Sensors

You can also determine whether a specific type of sensor exists on a device by using the **getDefaultSensor()** method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as
SensorManager
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null) {
    // Success! There's a magnetometer.
} else {
    // Failure! No magnetometer.
}
```

Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through the SensorEventListener interface: **onAccuracyChanged()** and **onSensorChanged()**. The Android system calls these methods whenever the following occurs:

- **A sensor's accuracy changes.**
  - In this case the system invokes the onAccuracyChanged() method, providing you with a reference to the Sensor object that changed and the new accuracy of the sensor. Accuracy is represented by one of four status constants: **SENSOR_STATUS_ACCURACY_LOW**, **SENSOR_STATUS_ACCURACY_MEDIUM**, **SENSOR_STATUS_ACCURACY_HIGH**, or **SENSOR_STATUS_UNRELIABLE**.
- **A sensor reports a new value**.
  - In this case the system invokes the onSensorChanged() method, providing you with a SensorEvent object. A SensorEvent object contains information about the new sensor data, including: the **accuracy** of the data, the **sensor** that generated the data, the **timestamp** at which the data was generated, and the **new data** that the sensor recorded.

Monitoring Sensor Events

```
class SensorActivity : Activity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var mLight: Sensor? = null

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)

        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
        mLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Do something here if sensor accuracy changes.
    }

    override fun onSensorChanged(event: SensorEvent) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        val lux = event.values[0]
        // Do something with this sensor value.
    }

    override fun onResume() {
        super.onResume()
        mLight?.also { light ->
            sensorManager.registerListener(this, light, SensorManager.SENSOR_DELAY_NORMAL)
        }
    }

    override fun onPause() {
        super.onPause()
        sensorManager.unregisterListener(this)
    }
}
```

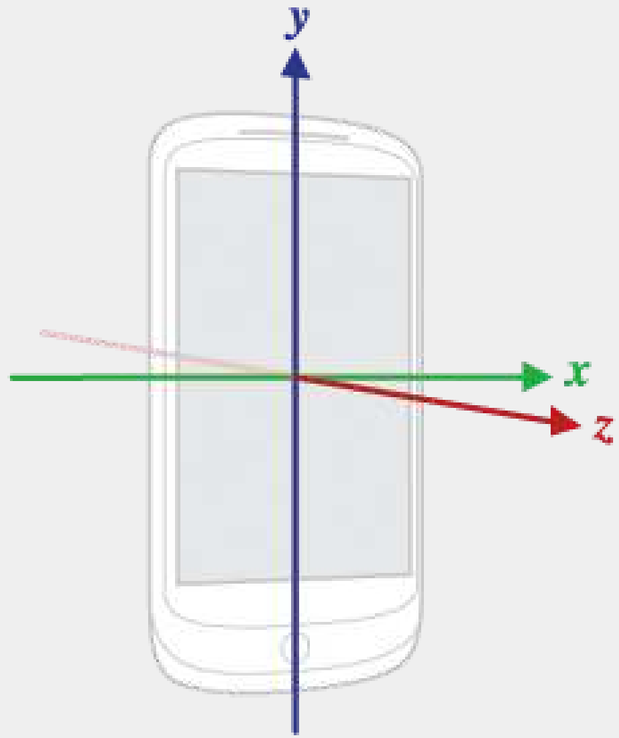
Google Play filters

If you are publishing your application on Google Play you can use the **<uses-feature>** element in your manifest file to filter your application from devices that do not have the appropriate sensor configuration for your application. The <uses-feature> element has several hardware descriptors that let you filter applications based on the presence of specific sensors. The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity. The following is an example manifest entry that filters apps that do not have an accelerometer:

```
<uses-feature
android:name="android.hardware.sensor.accelerometer"
          android:required="true" />
```

Coordinate system

In general, the sensor framework uses a standard 3-axis coordinate system to express data values. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see figure right). When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values. This coordinate system is used by the following sensors:



- **Acceleration sensor**,
- **Gravity sensor**
- **Gyroscope**
- **Linear acceleration sensor**
- **Geomagnetic field sensor**

Best Practices

Only gather sensor data in the foreground

On devices running Android 9 (API level 28) or higher, apps running in the background have the following restrictions:

- Sensors that use the **continuous** reporting mode, such as accelerometers and gyroscopes, don't receive events.
- Sensors that use the **on-change** or **one-shot** reporting modes don't receive events.

Given these restrictions, it's best to detect sensor events either when your app is in the foreground or as part of a foreground service.

Best Practices

Unregister sensor listeners

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. The following code shows how to use the **onPause()** method to unregister a listener:

```
private lateinit var sensorManager: SensorManager
...
override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(this)
}
```

Best Practices

Don't block the onSensorChanged() method

Sensor data can change at a high rate, which means the system may call the onSensorChanged(SensorEvent) method quite often. As a best practice, you should do as little as possible within the onSensorChanged(SensorEvent) method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the onSensorChanged(SensorEvent) method.

Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

References

- **Touch gestures** - https://developer.android.com/develop/ui/views/touch-and-input/gestures
- **Input events** - https://developer.android.com/develop/ui/views/touch-and-input/input-events
- **Interactive view** - https://developer.android.com/develop/ui/views/layout/custom-views/making-interactive
- **Sensors** - https://developer.android.com/guide/topics/sensors/sensors_overview

End of the session