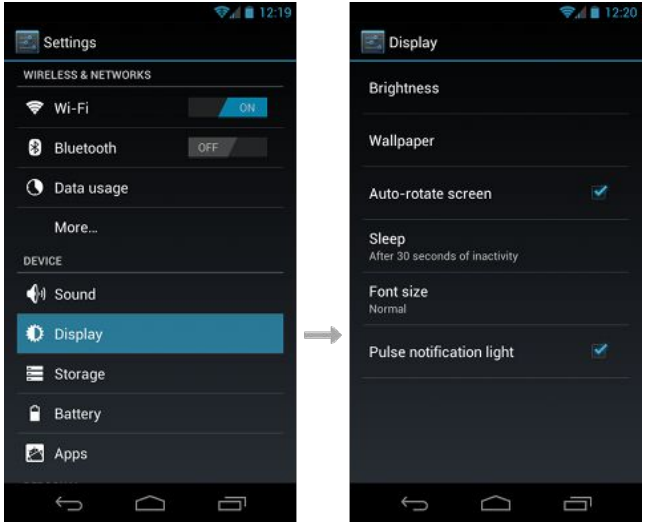
# Application Development with Android Sensor Frameworks

COMPSCI 528
Spring 2022

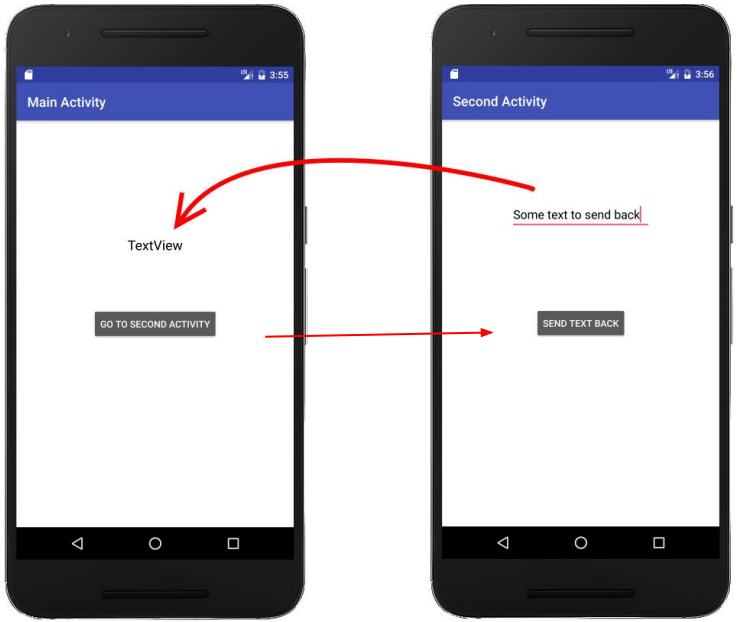
# Activity

- User interaction with a mobile-app is often non-deterministic. The app does not always begin in the same place.
  - For example, you can open an email app from your home screen and it will show a list of emails.
  - A social media app can also launch your email app which might directly bring you to the email app screen that let you compose an email.
- The Activity serves as the entry point for an app's interaction with the user.
- An activity provides the window in which the app draws its User Interface (UI). The window typically fills the entire screen of the mobile device. Generally one activity implements one screen in an app.
  - For example, one of the app's activities may implement a Preference screen, while another activity implements a Select Photo Screen.



# Activity

- Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the main activity, which is the first screen to appear when the user launches the app.
- One activity can start another activity in order to perform different actions.



# Activity

- Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the main activity, which is the first screen to appear when the user launches the app.
- One activity can start another activity in order to perform different actions.
- For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest.
- You must declare all the activities within an app in the Manifest file.
    - Declare it as a child of <application> element.
    - You can also setup intent filters if you want this activity to be launched by other activities and receive any

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

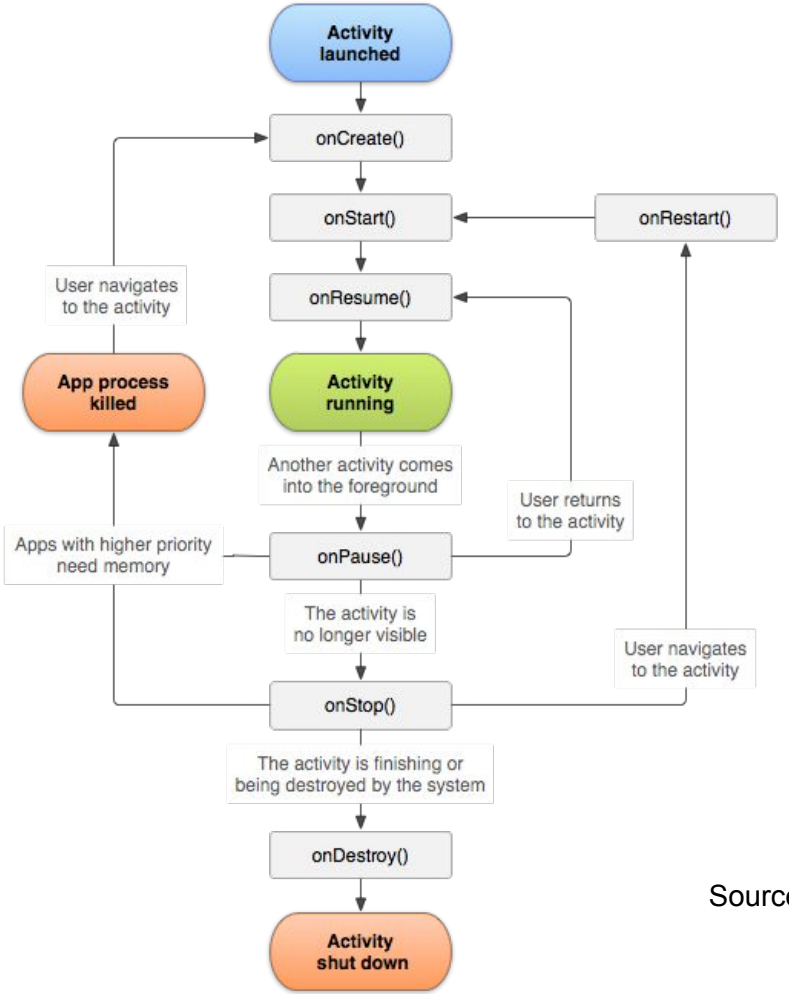
# Activity

As a user navigates through, out of, and back to your app, each activity can transition through different states in their lifecycle. The activity class provides a number of callback functions so that 1) the activity may know a state change, and 2) the activity may know how to behave as the user leaves and re-enter the activity.

For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.

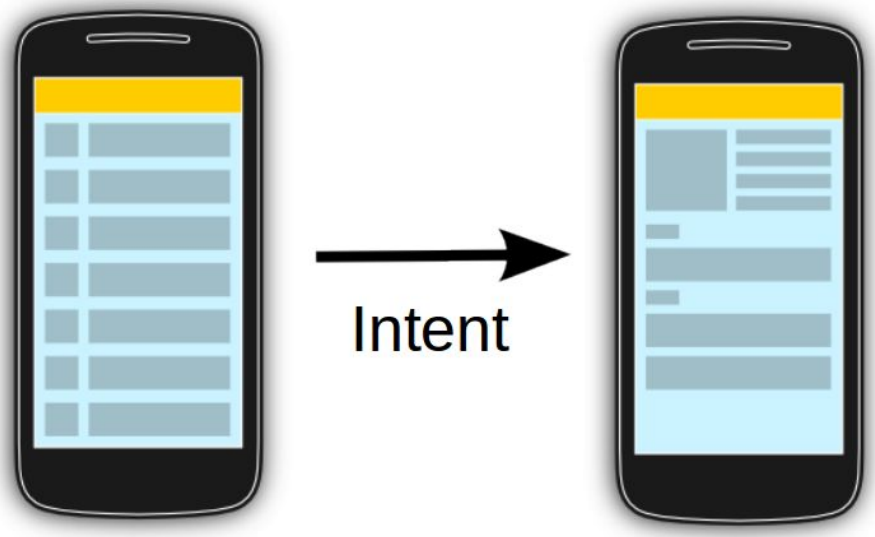
Good implementation of the lifecycle callbacks can help ensure that your app avoids:

- Crashing if the user receives a phone call or switches to another app while using your app.
- Consuming valuable system resources when the user is not actively using it.
- Losing the user's progress if they leave your app and return to it at a later time.




Source: Android Developer Documentation

# Intent




- Intents are used to start activity, pass information between activities ( services, broadcast receiver, etc,.)

# Intent

- The Intent constructor takes two parameters, a Context and a Class.
- Intents are of two types- explicit and implicit
- Explicit intents explicitly define the component which should be called by the Android system.
- Implicit intents specify the action which should be performed and wait for the Android system to find all components registered for this action. Eg:- Share option

# Sensor Overview

- Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions.
- 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.

# Android Sensor Framework Capabilities

- 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.

# Types of Sensor

The Android sensor framework lets you access two types of sensors.

1. **Hardware-based sensors:** Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.
2. **Software-based or virtual sensors:** Software-based sensors are not physical devices, although they mimic hardware-based sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors.

# Types of Sensor

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

| 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. |
| `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. |

Source: Android Developer Documentation

# Android 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.
- **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, the raw sensor data that triggered the event}***

# Android Sensor Framework

Android Sensor Framework allows you to determine which sensors are available on the device at run time.

```
private SensorManager sensorManager;

sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of TYPE_ALL such as TYPE_GYROSCOPE, TYPE_LINEAR_ACCELERATION, or TYPE_GRAVITY.

# Determining if a sensor exists

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

# Determining sensor attributes

In addition to listing the sensors that are on a device, you can use the public methods of the Sensor class to determine the capabilities and attributes of individual sensors. This is useful if you want your application to behave differently based on which sensors or sensor capabilities are available on a device.

- getResolution() methods can obtain information about sensor’s resolution.
- getMaximumRange() methods can obtain a sensor's maximum range of measurement.
- getPower() method can obtain a sensor's power requirements.
- getMinDelay() method returns the minimum time interval (in microseconds) a sensor can use to capture data. If a sensor returns zero, it means the sensor is not a streaming sensor and reports data only when there is a change in the parameters it is sensing.

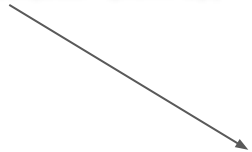
# Listening to sensor data

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

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

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }

    @Override
    public final void onSensorChanged(SensorEvent event) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        float lux = event.values[0];
        // Do something with this sensor value.
    }

    @Override
```

To monitor raw sensor data you need to implement two callback methods that are exposed through the `SensorEventListener` interface: `onAccuracyChanged()` and `onSensorChanged()`.

# Listening to sensor data

```
    @Override
    protected void onResume() {
        super.onResume();
        sensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
    }
}
```



The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the `onSensorChanged()` callback method. The default data delay is delay of 200,000 microseconds. You can specify other data delays, such as `SENSOR_DELAY_GAME`(20,000 microsecond delay), `SENSOR_DELAY_UI` (60,000 microsecond delay), or `SENSOR_DELAY_FASTEST` (0 microsecond delay).

Remember, This is just a suggested or requested value to the Android Sensor Framework. There is no guarantee that you will get your sensor data sampled at your requested rate!

The best practice is that you should request the largest delay your application can tolerate.

# Dealing with Sensors: Best Practices

- Determine Sampling Rate with time stamps.
- Select Sensor Delays carefully.
- Don’t Block onSensorChanged() method.
- Always remember to unregister listener when the sensor is not needed to save battery!

```
private SensorManager sensorManager;
...
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
```