CS 528 Mobile and Ubiquitous Computing
Lecture 5: Widget Catalog, SQLite Databases and Sensors

Emmanuel Agu
Paper sign up

- Students present papers in weeks 7-8, 10-13
- Previously 1 student per paper
- Too many students (42)
- So, 2 students per paper this time
- Everyone should sign up
- First presentations in 2 weeks
Skipped Android Nerd Ranch CriminalIntent Chapters
Chapter 9: Displaying Lists with RecyclerView

- RecyclerView allows view of large dataset
- Allows crimes in CriminalIntent to be listed
- Users can check box to indicate if crime has been solved/not solved
Chapter 11: Using View Pager

- View Pager allows users to swipe between screens (e.g. Tinder?)
- Allows users to swipe between Crimes in CriminalIntent
Chapter 12: Dialogs

- Dialogs present users with a choice or important information
- E.g. DatePicker allows users pick date
- Allows users to pick a date on which a crime occurred in CriminalIntent
Chapter 13: The Toolbar

- Many Android apps include a toolbar
- Toolbar includes actions user can take
- In CriminalIntent, menu items for adding crime, navigate up the screen hierarchy
Widget Catalog
What Widget Catalog?

- Several larger widgets are available
- Can use easily just like smaller widgets, to make your apps look nice and professional
- Examples:
  - CalendarView
  - DatePicker
  - TimePicker
  - SeekBar
- Will not explain coding here. Check books, Android documentation
CalendarView

- Allows user pick a date from a displayed calendar
**DatePicker**

- Allows user to pick a date
- Uses date wheel
- Can display a CalendarView as well
DatePicker

DatePicker with CalendarView Android 5.0, landscape
SeekBar

- Allows user to choose a value on a continuous range by sliding a “thumb” along a horizontal line.
**TimePicker**

- Allows user pick a time
Android Nerd Ranch Ch 14
SQLLite Databases
Background on Databases

- **RDBMS**
  - relational data base management system

- **Relational databases introduced by E. F. Codd**
  - Turing Award Winner

- **Relational Database**
  - data stored in tables
  - relationships among data stored in tables
  - data can be accessed and viewed in different ways
Example Database

- Wines

Ref: Web Database Applications with PHP and MySQL, 2nd Edition, by Hugh E. Williams, David Lane
**Relational Data**

- Data in different tables can be related

<table>
<thead>
<tr>
<th>Winery Table</th>
<th>Region Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Winery ID</strong></td>
<td><strong>Region ID</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Keys

- Each table has a key
- Column used to uniquely identify each row
SQL and Databases

- SQL is the language used to manipulate and manage information in a relational database management system (RDBMS)

- SQL Commands:
  - **CREATE TABLE** - creates new database table
  - **ALTER TABLE** - alters a database table
  - **DROP TABLE** - deletes a database table
  - **CREATE INDEX** - creates an index (search key)
  - **DROP INDEX** - deletes an index
SQL Commands

- **SELECT** - get data from a database table
- **UPDATE** - change data in a database table
- **DELETE** - remove data from a database table
- **INSERT INTO** - insert new data in a database table

- SQLite implements most, but not all of SQL
  - [http://www.sqlite.org/](http://www.sqlite.org/)
CriminalIntent Database

- **SQLite** is open source relational database
- Android includes SQLite database in its standard library
- **Goal:** Store crimes in CriminalIntent in a database
- First step, define database table of **crimes**

<table>
<thead>
<tr>
<th>_id</th>
<th>uuid</th>
<th>title</th>
<th>date</th>
<th>solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13090636733242</td>
<td>Stolen yogurt</td>
<td>13090636733242</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>13090732131909</td>
<td>Dirty sink</td>
<td>13090732131909</td>
<td>1</td>
</tr>
</tbody>
</table>

- Create **CrimeDbSchema** class to put crime database in

```
public class CrimeDbSchema {
    public static final class CrimeTable {
        public static final String NAME = "crimes";
    }
}
```
CriminalIntent Database

Next, define the columns of the Crimes database table

```java
public class CrimeDbSchema {
    public static final class CrimeTable {
        public static final String NAME = "crimes";

        public static final class Cols {
            public static final String UUID = "uuid";
            public static final String TITLE = "title";
            public static final String DATE = "date";
            public static final String SOLVED = "solved";
        }
    }
}
```

<table>
<thead>
<tr>
<th>_id</th>
<th>uuid</th>
<th>title</th>
<th>date</th>
<th>solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13090636733242</td>
<td>Stolen yogurt</td>
<td>13090636733242</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>13090732131909</td>
<td>Dirty sink</td>
<td>13090732131909</td>
<td>1</td>
</tr>
</tbody>
</table>
SQLiteOpenHelper

- SQLiteOpenHelper encapsulates database creation, opening and updating
- In CriminalIntent, create subclass of SQLiteOpenHelper called CrimeBaseHelper

```java
public class CrimeBaseHelper extends SQLiteOpenHelper {
    private static final int VERSION = 1;
    private static final String DATABASE_NAME = "crimeBase.db";

    public CrimeBaseHelper(Context context) {
        super(context, DATABASE_NAME, null, VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```
Use CrimeBaseHelper to open SQLite Database

```java
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    private List<Crime> mCrimes;
    private Context mContext;
    private SQLiteDatabase mDatabase;

    ...

    private CrimeLab(Context context) {
        mContext = context.getApplicationContext();
        mDatabase = new CrimeBaseHelper(mContext)
            .getWritableDatabase();
        mCrimes = new ArrayList<>();
    }

    ...
```
Create CrimeTable in onCreate()

Create CrimeTable in onCreate()

```java
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("create table " + CrimeTable.NAME + "(
        _id integer primary key autoincrement, " +
        CrimeTable.Cols.UUID + ", " +
        CrimeTable.Cols.TITLE + ", " +
        CrimeTable.Cols.DATE + ", " +
        CrimeTable.Cols.SOLVED +
    ")
```
```
Use Database

- **CriminalIntent**, previously used ArrayLists
- Modify to use SQLiteDatabase

```java
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    private List<Crime> mCrimes;
    private Context mContext;
    private SQLiteDatabase mDatabase;

    public static CrimeLab get(Context context) {
        ...
    }

    private CrimeLab(Context context) {
        mContext = context.getApplicationContext();
        mDatabase = new CrimeBaseHelper(mContext)
                      .getWritableDatabase();
        mCrimes = new ArrayList<>();
    }

    public void addCrime(Crime c) {
        mCrimes.add(c);
    }

    public List<Crime> getCrimes() {
        return mCrimes;
        return new ArrayList<>();
    }

    public Crime getCrime(UUID id) {
        for (Crime crime : mCrimes) {
            if (crime.getId().equals(id)) {
                return crime;
            }
        }
        return null;
    }
}
```
Writing to the Database using ContentValues

- In Android, writing to databases is done using class `ContentValues`
- `ContentValues` is key-value pair (like Bundle)
- Create method to create `ContentValues` instance from a Crime

```java
public getCrime(UUID id) {
    return null;
}

private static ContentValues getContentValues(Crime crime) {
    ContentValues values = new ContentValues();
    values.put(CrimeTable.Cols.UUID, crime.getId().toString());
    values.put(CrimeTable.Cols.TITLE, crime.getTitle());
    values.put(CrimeTable.Cols.DATE, crime.getDate().getTime());
    values.put(CrimeTable.Cols.SOLVED, crime.isSolved() ? 1 : 0);
    return values;
}
```
Inserting and Updating Rows

- Modify **addCrime** to insert Crime into database

```java
public void addCrime(Crime c) {
    ContentValues values = getContentValues(c);

    mDatabase.insert(CrimeTable.NAME, null, values);
}
```

- Table you want to Insert Crime into
- ContentValue data to insert into database
Inserting and Updating Rows

- Update rows by using ContentValues

```java
public Crime getCrime(UUID id) {
    return null;
}

public void updateCrime(Crime crime) {
    String uuidString = crime.getId().toString();
    ContentValues values = getContentValues(crime);

    mDatabase.update(CrimeType.NAME, values,
                        CrimeTable.Cols.UUID + " = ?",
                      new String[] { uuidString });
}

private static ContentValues getContentValues(Crime crime) {
    ContentValues values = new ContentValues();
    values.put(CrimeType.Cols.UUID, crime.getId().toString());
    ...
}
Pushing Updates

Push updates in `onPause()` method of CrimeFragment

```java
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
    mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);
}

@Override
public void onPause() {
    super.onPause();
    CrimeLab.get(getActivity()).updateCrime(mCrime);
}
```
More in Text

- See Android Nerd Ranch (2nd edition), chapter 14 for more
  - The rest of the example,
  - How to query the database
  - The rest of the code
Alternatives to sqlite

- SQLite is low level ("Down in the weeds")
- Various alternatives to work higher up the food chain
- Object Relational Mappers - ORM
- Higher level wrappers for dealing with sql commands and sqlite databases
- Many ORMs exist
Android Sensors
What is a Sensor?

- Converts some physical quantity (e.g. light, acceleration, magnetic field) into a signal

**Example:** accelerometer converts acceleration along X, Y, Z axes into signal
So What?

- Raw sensor data can be processed into meaningful info
- Example: Raw accelerometer data can be processed to infer user’s activity (e.g. walking, running, etc)
Android Sensors

- Microphone (sound)
- Camera
- Temperature
- Location (GPS, A-GPS)
- Accelerometer
- Gyroscope (orientation)
- Proximity
- Pressure
- Light

- Different phones do not have all sensor types!!
Android Sensor Framework

- Enables apps to:
  - Access sensors available on device and
  - Acquire raw sensor data

- Specifically, using the Android Sensor Framework, you can:
  - Determine which sensors are available
  - Determine capabilities of individual sensors (e.g. max. range, manufacturer, power requirements, resolution)
  - Acquire raw sensor data and define data rate
  - Register and unregister sensor event listeners

Android Sensor Framework

- Android sensors can be either hardware or software

  - Hardware sensor:
    - physical components built into phone,
    - Measure specific environmental property. E.g. temperature

  - Software sensor (or virtual sensor):
    - Not physical device
    - Derives their data from one or more hardware sensors
    - Example: gravity sensor
Accelerometer Sensor

- Acceleration is **rate of change of velocity**
- Accelerometers
  - Measure **change** of speed in a direction
  - Do not measure velocity
- Phone’s accelerometer measures acceleration along its X,Y,Z axes
Sensor Types Supported by Android

- **TYPE_ACCELEROMETER**
  - **Type:** hardware
  - Measures device acceleration force along X,Y,Z axes including gravity in m/s²
  - **Common uses:** motion detection (shake, tilt, etc)

- **TYPE_LINEAR_ACCELEROMETER**
  - **Type:** software or hardware
  - Measures device acceleration force along X,Y,Z axes excluding gravity in m/s²
  - **Common uses:** monitoring acceleration along single axis
Sensor Types Supported by Android

- **TYPE_GRAVITY**
  - **Type:** Software or hardware
  - Measures **force of gravity along X,Y,Z axes** in m/s$^2$
  - **Common uses:** motion detection (shake, tilt, etc)
Sensor Types Supported by Android

- **TYPE_ROTATION_VECTOR**
  - **Type:** Software or hardware
  - Measures **device’s orientation** by providing 3 rotation vectors
  - **Common uses:** motion detection and rotation

Blue: Fixed reference axes
Red: Rotated axes
Sensor Types Supported by Android

- **TYPE_GYROSCOPE**
  - **Type**: hardware
  - Measures device’s rate of rotation around X,Y,Z axes in rad/s
  - **Common uses**: rotation detection (spin, turn, etc)
Sensor Types Supported by Android

- **TYPE_AMBIENT_TEMPERATURE**
  - **Type:** hardware
  - Measures ambient **room temperature** in degrees Celsius
  - **Common uses:** monitoring room air temperatures

- **TYPE_LIGHT**
  - **Type:** hardware
  - Measures ambient **light level (illumination)** in lux
  - Lux is SI measure of illuminance
    - Measures luminous flux per unit area
  - **Common uses:** controlling screen brightness
Sensor Types Supported by Android

- **TYPE_MAGNETIC_FIELD**
  - **Type:** hardware
  - Measures **magnetic field** for X,Y,Z axes in \( \mu \text{T} \)
  - **Common uses:** Creating a compass

- **TYPE_PRESSURE**
  - **Type:** hardware
  - Measures ambient **air pressure** in hPa or mbar
  - Force per unit area
  - **Common uses:** monitoring air pressure changes
Sensor Types Supported by Android

- **TYPE_ORIENTATION**
  - **Type:** software
  - Measures degrees of rotation about X,Y,Z axes
  - **Common uses:** Determining device position
Sensor Types Supported by Android

- **TYPE_PROXIMITY**
  - **Type:** hardware
  - Measures an **object’s proximity to device’s screen** in cm
  - **Common uses:** to determine whether a handset is being held up to a person’s ear
Sensor Types Supported by Android

- **TYPE_RELATIVE_HUMIDITY**
  - **Type:** hardware
  - Measures relative ambient humidity in percent (%)
  - Expresses \% of max possible humidity currently present in air
  - **Common uses:** monitoring dewpoint, absolute, and relative humidity

- **TYPE_TEMPERATURE**
  - **Type:** hardware
  - Measures **temperature of phone (or device)** in degrees Celsius.
  - Replaced by TYPE_AMBIENT_TEMPERATURE in API 14
  - **Common uses:** monitoring temperatures
2 New Hardware Sensor in Android 4.4

- **TYPE_STEP_DETECTOR**
  - **Type:** hardware
  - Triggers a sensor event each time user takes a step
  - Delivered event has value of 1.0 and timestamp of step

- **TYPE_STEP_COUNTER**
  - **Type:** hardware
  - Also triggers a sensor event each time user takes a step
  - Delivers total *accumulated number of steps since this sensor was first registered by an app*, tries to eliminate false positives

- **Common uses:** Both used in step counting, pedometer apps
- Requires hardware support, available in Nexus 5
Sensor Programming

- Sensor framework is part of `android.hardware`
- Classes and interfaces include:
  - `SensorManager`
  - `Sensor`
  - `SensorEvent`
  - `SensorEventListener`
- These sensor-APIs used for 2 main tasks:
  - Identifying sensors and sensor capabilities
  - Monitoring sensor events
Sensor Events and Callbacks

- App sensors send events asynchronously, when new data arrives

- General approach:
  - App registers callbacks
  - SensorManager notifies app of sensor event whenever new data arrives (or accuracy changes)
Sensor

- A class that provides methods used to determine a sensor’s capabilities
- Can be used to create instance of a specific sensor
SensorEvent

- Android system provides information about a sensor event as a **sensor event object**

- **Sensor event object** includes:
  - **Values**: Raw sensor data
  - **Sensor**: Type of sensor that generated the event
  - **Accuracy**: Accuracy of the data
  - **Timestamp**: Event timestamp
<table>
<thead>
<tr>
<th>Sensor</th>
<th>Sensor event data</th>
<th>Description</th>
<th>Units of measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_ACCELEROMETER</td>
<td>SensorEvent.values[0]</td>
<td>Acceleration force along the x axis (including gravity).</td>
<td>m/s²</td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[1]</td>
<td>Acceleration force along the y axis (including gravity).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[2]</td>
<td>Acceleration force along the z axis (including gravity).</td>
<td></td>
</tr>
<tr>
<td>TYPE_GRAVITY</td>
<td>SensorEvent.values[0]</td>
<td>Force of gravity along the x axis.</td>
<td>m/s²</td>
</tr>
<tr>
<td>TYPE_GYROSCOPE</td>
<td>SensorEvent.values[0]</td>
<td>Rate of rotation around the x axis.</td>
<td>rad/s</td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[1]</td>
<td>Rate of rotation around the y axis.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[2]</td>
<td>Rate of rotation around the z axis.</td>
<td></td>
</tr>
<tr>
<td>TYPE_GYROSCOPE_UNCALIBRATED</td>
<td>SensorEvent.values[0]</td>
<td>Rate of rotation (without drift compensation) around the x axis.</td>
<td>rad/s</td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[1]</td>
<td>Rate of rotation (without drift compensation) around the y axis.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[2]</td>
<td>Rate of rotation (without drift compensation) around the z axis.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[3]</td>
<td>Estimated drift around the x axis.</td>
<td></td>
</tr>
</tbody>
</table>
## Sensor Values Depend on Sensor Type

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Sensor event data</th>
<th>Description</th>
<th>Units of measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_LINEAR_ACCELERATION</td>
<td>SensorEvent.values[0]</td>
<td>Acceleration force along the x axis (excluding gravity).</td>
<td>m/s²</td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[1]</td>
<td>Acceleration force along the y axis (excluding gravity).</td>
<td></td>
</tr>
<tr>
<td>TYPE_ROTATION_VECTOR</td>
<td>SensorEvent.values[0]</td>
<td>Rotation vector component along the x axis ($x * \sin(\theta/2)$).</td>
<td>Unitless</td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[1]</td>
<td>Rotation vector component along the y axis ($y * \sin(\theta/2)$).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[2]</td>
<td>Rotation vector component along the z axis ($z * \sin(\theta/2)$).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SensorEvent.values[3]</td>
<td>Scalar component of the rotation vector ($\cos(\theta/2)$).¹</td>
<td></td>
</tr>
<tr>
<td>TYPE_SIGNIFICANT_MOTION</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TYPE_STEP_COUNTER</td>
<td>SensorEvent.values[0]</td>
<td>Number of steps taken by the user since the last reboot while the sensor was activated.</td>
<td>Steps</td>
</tr>
<tr>
<td>TYPE_STEP_DETECTOR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
SensorEventListener

- An interface used to create 2 callbacks that receive notifications (sensor events) when:
  - Sensor values change \texttt{(onSensorChange())} or
  - When sensor accuracy changes \texttt{(onAccuracyChanged())}
SensorManager

- A class that provides methods for:
  - Accessing and listing sensors
  - Registering and unregistering sensor event listeners
  - Acquiring orientation information
- Can be used to create instance of sensor service
- Also provides sensor constants used to:
  - Report sensor accuracy
  - Set data acquisition rates
  - Calibrate sensors
Sensor API Tasks

- **Sensor API Task 1: Identifying sensors and their capabilities**
  - Why identify sensor and their capabilities at runtime?
    - Disable app features using sensors not present, or
    - Choose sensor implementation with best performance

- **Sensor API Task 2: Monitor sensor events**
  - Why monitor sensor events?
    - To acquire raw sensor data
    - Sensor event occurs every time sensor detects change in parameters it is measuring
Sensor Availability

- Different sensors are available on different Android versions

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Android 4.0 (API Level 14)</th>
<th>Android 2.3 (API Level 9)</th>
<th>Android 2.2 (API Level 8)</th>
<th>Android 1.5 (API Level 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_ACCELEROMETER</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TYPE_AMBIENT_TEMPERATURE</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TYPE_GRAVITY</td>
<td>Yes</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TYPE_GYROSCOPE</td>
<td>Yes</td>
<td>Yes</td>
<td>n/a¹</td>
<td>n/a¹</td>
</tr>
<tr>
<td>TYPE_LIGHT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TYPE_LINEAR_ACCELERATION</td>
<td>Yes</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TYPE_MAGNETIC_FIELD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TYPE_ORIENTATION</td>
<td>Yes²</td>
<td>Yes²</td>
<td>Yes²</td>
<td>Yes</td>
</tr>
<tr>
<td>TYPE_PRESSURE</td>
<td>Yes</td>
<td>Yes</td>
<td>n/a¹</td>
<td>n/a¹</td>
</tr>
<tr>
<td>TYPE_PROXIMITY</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TYPE_RELATIVE_HUMIDITY</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TYPE_ROTATION_VECTOR</td>
<td>Yes</td>
<td>Yes</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>TYPE_TEMPERATURE</td>
<td>Yes²</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Identifying Sensors and Sensor Capabilities

- Need a reference to the sensor service.
- How? First create instance of \textbf{SensorManager} by calling \texttt{getSystemService()} and passing in \texttt{SENSOR_SERVICE} argument

```java
private SensorManager mSensorManager;

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

- Then list sensors available on device by calling \texttt{getSensorList()} 

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

- To list particular type, use \texttt{TYPE_GYROSCOPE, TYPE_GRAVITY, etc}

Determining if Device has at least one of particular Sensor Type

- Device may have multiple sensors of a particular type.
  - E.g. multiple magnetometers
- If multiple sensors of a given type exist, one of them must be designated “the default sensor” of that type
- To determine if specific sensor type exists use `getDefaultSensor()`
- **Example:** To check whether device has a magnetometer

```java
private SensorManager mSensorManager;
...

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

- Some useful methods of **Sensor** class methods:
  - `getResolution()`: get sensor’s resolution
  - `getMaximumRange()`: get maximum measurement range
  - `getPower()`: get sensor’s power requirements
  - `getMinDelay()`: min time interval (in microseconds) sensor can use to sense data. Return values:
    - **0 value**: Non-streaming sensor, reports data only if sensed parameters change
    - **Non-zero value**: streaming sensor
Monitoring Sensor Events

- To monitor raw sensor data, 2 callback methods exposed through `SensorEventListener` interface need to be implemented:
  - `onSensorChanged`:
    - Invoked by Android system to report new sensor value
    - Provides `SensorEvent` object containing information about new sensor data
    - New sensor data includes:
      - **Accuracy**: Accuracy of data
      - **Sensor**: Sensor that generated the data
      - **Timestamp**: Times when data was generated
      - **Data**: New data that sensor recorded
Monitoring Sensor Events

- **onAccuracyChanged:**
  - invoked when accuracy of sensor being monitored changes
  - Provides reference to `sensor object` that changed and the new accuracy of the sensor
  - Accuracy represented as status constants
    SENSOR_STATUS_ACCURACY_LOW,
    SENSOR_STATUS_ACCURACY_MEDIUM,
    SENSOR_STATUS_ACCURACY_HIGH,
    SENSOR_STATUS_UNRELIABLE
Example: Monitoring Light Sensor Data

- **Goal:** Monitor light sensor data using `onSensorChanged()`, display it in a **TextView** defined in `main.xml`.

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

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

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

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }
}
```
Example: Monitoring Light Sensor Data (Contd)

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

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

- Get new light sensor value
- Register sensor when app becomes visible
- Unregister sensor if app is no longer visible to reduce battery drain
Handling Different Sensor Configurations

- Different phones have different sensors built in
- E.g. Motorola Xoom has pressure sensor, Samsung Nexus S doesn’t
- If app uses a specific sensor, how to ensure this sensor exists on target device? Two options
  - **Option 1:** Detect device sensors at runtime, enable/disable app features as appropriate
  - **Option 2:** Use Google Play filters so only devices possessing required sensor can download app
Option 1: Detecting Sensors at Runtime

- Following code checks if device has a pressure sensor

```java
private SensorManager mSensorManager;
...

mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null) {
    // Success! There's a pressure sensor.
}
else {
    // Failure! No pressure sensor.
}
```
Option 2: Use Google Play Filters to Target Specific Sensor Configurations

- Can use `<uses-feature>` element in AndroidManifest.xml to filter your app from devices without required sensors

- **Example:** following manifest entry ensures that only devices with accelerometers will see this app on Google Play

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

- **Can list** accelerometers, barometers, compass (geomagnetic field), gyroscope, light and proximity using this approach
Example Step Counter App

- **Goal:** Track user’s steps, display it in TextView
- **Note:** Phone hardware must support step counting

```java
package com.starboardland.pedometer;

import android.app.Activity;
import android.content.Context;
import android.hardware.*;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.Toast;

public class CounterActivity extends Activity implements SensorEventListener {

    private SensorManager sensorManager;
    private TextView count;
    boolean activityRunning;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        count = (TextView) findViewById(R.id.count);

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

    https://theelfismike.wordpress.com/2013/11/10/android-4-4-kitkat-step-detector-code/
```
Example Step Counter App (Contd)

```java
@Override
protected void onResume() {
    super.onResume();
    activityRunning = true;
    Sensor countSensor = sensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER);
    if (countSensor != null) {
        sensorManager.registerListener(this, countSensor, SensorManager.SENSOR_DELAY_UI);
    } else {
        Toast.makeText(this, "Count sensor not available!", Toast.LENGTH_LONG).show();
    }
}

@Override
protected void onPause() {
    super.onPause();
    activityRunning = false;
    // if you unregister the last listener, the hardware will stop detecting step events
    // sensorManager.unregisterListener(this);
}

https://theelfismike.wordpress.com/2013/11/10/android-4-4-kitkat-step-detector-code/
```
Example Step Counter App (Contd)

```java
@Override
public void onSensorChanged(SensorEvent event) {
    if (activityRunning) {
        count.setText(String.valueOf(event.values[0]));
    }
}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
}
```
Best Practices for Sensor Usage

1. **Unregister sensor listeners**: when done using sensor or when app is paused
   - Otherwise sensor continues to acquire data, draining battery

2. **Don’t test sensor code on emulator**
   - Must test sensor code on physical device, emulator doesn’t support sensors
Best Practices for Sensor Usage (Contd)

3. **Don’t block onSensorChange( ) method:**
   - Android system may call onSensorChanged( ) often
   - So... don’t block it
   - Perform any heavy processing (filtering, reduction of sensor data) outside `onSensorChanged( )` method

4. **Avoid using deprecated methods or sensor types:**
   - `TYPE_ORIENTATION` sensor type deprecated, use `getOrientation( )` method instead

   - `TYPE_TEMPERATURE` sensor type deprecated, use `TYPE_AMBIENT_TEMPERATURE` sensor type instead
Best Practices for Sensor Usage (Contd)

5. **Verify sensors before you use them:**
   - Don’t assume sensor exists on device, check first before trying to acquire data from it

6. **Choose sensor delays carefully:**
   - Sensor data rates can be very high
   - Choose delivery rate that is suitable for your app or use case
   - Choosing a rate that is too high sends extra data, wastes system resources and battery power
References

- Busy Coder’s guide to Android version 6.3
- CS 65/165 slides, Dartmouth College, Spring 2014
- CS 371M slides, U of Texas Austin, Spring 2014