Threads and services
How apps work

• The system creates a thread for the application, called “main” or “UI thread”
  — It dispatches events to the user interface

• Everything happens in the UI thread
  — Long operations block the whole UI
  — No events can be dispatched

• Being blocked for more than 5 secs means “application not responding”
Threads

- Android natively supports multi-threading
- An application can comprise concurrent threads
- Threads are managed like in Java by
  - Extending class Thread
  - Implementing interface Runnable
    - Method run() is executed when Thread.start() is launched
Some simple rules

• Do not block the UI thread
• Do not access the Android UI toolkit from outside the UI thread
  — Do not manipulate your UI in a worker thread
  — All manipulations to the user interface should be done within the UI thread
Bad examples

```java
public void onClick(View v) {
    Bitmap b = loadImageFromNetwork("http://example.com/image.png");
    myImageView.setImageBitmap(b);
}

public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            myImageView.setImageBitmap(b);
        }
    }).start();
}

public void onClick(View v) {
    Bitmap b = loadImageFromNetwork("http://example.com/image.png");
    myImageView.setImageBitmap(b);
}
```
How can we fix the problem?

- Different ways to access the UI thread from other threads
  - Activity.runOnUiThread(Runnable)
  - View.post(Runnable)
  - View.postDelayed(Runnable, long)
- The Runnable is sent to the UI thread and runs within it
  - It is invoked on a View from outside the UI thread
Example

public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap = loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
Asynchronous tasks

• Enable proper and easy use of the UI thread
  — Allow one to perform background operations and publish results on the UI thread without having to manipulate threads

• One must subclass AsyncTask and implement the doInBackground() method that runs in a pool of background threads

• To run the task call execute() from the UI thread

• We can cancel the task at any time from any thread (through method cancel())
Key elements

• `doInBackground()` executes automatically on a worker thread
  — This step is used to perform long-running computations in background
  — The result of the computation must be passed back

• `onPreExecute()`, `onPostExecute()` and `onProgressUpdate()` are all invoked on the UI thread
  — The value returned by `doInBackground()` is sent to `onPostExecute()`

• We can call `publishProgress()` at anytime in `doInBackground()` to execute `onProgressUpdate()` on the UI thread
public void onClick(View v) {
    new DownloadTask().execute("http://example.com/image.png");
}

private class DownloadTask extends AsyncTask<String, Void, Bitmap> {
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
Broadcast receiver

- A component that allows us to register for system or application intents sent by `sendBroadcast()`
  - All registered receivers for an intent are notified by the Android runtime once the event happens
- A broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work
  - A receiver must be registered
    - Dynamically through `registerReceiver()`
    - Statically in the manifest through tag `<receiver>`
- The implementing class extends class `BroadcastReceiver`
  - Method `onReceive()` is called by the Android system
    - Once the code returns, the system considers the object to be finished and no longer active
Events

- Several system events are defined as final static fields in class Intent
- Other Android system classes also define events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intent.ACTION_BOOT_COMPLETED</td>
<td>Boot completed. Requires the android.permission.RECEIVE_BOOT_COMPLETED permission.</td>
</tr>
<tr>
<td>Intent.ACTION_POWER_CONNECTED</td>
<td>Power got connected to the device.</td>
</tr>
<tr>
<td>Intent.ACTION_POWER_DISCONNECTED</td>
<td>Power got disconnected to the device.</td>
</tr>
<tr>
<td>Intent.ACTION_BATTERY_LOW</td>
<td>Triggered on low battery. Typically used to reduce activities in your app which consume power.</td>
</tr>
<tr>
<td>Intent.ACTION_BATTERY_OKAY</td>
<td>Battery status good again.</td>
</tr>
</tbody>
</table>
Intent broadcast mechanism

• Intent objects are delivered to all interested parties
• Android finds the appropriate activity, service, or broadcast receiver to respond to the intent
  ─ Instantiates them if necessary
• Broadcast intents are delivered only to broadcast receivers, never to activities or services
  ─ An intent passed to startActivity() is delivered only to an activity, never to a service or broadcast receiver, etc.
sendBroadcast

• sendBroadcast
  — Broadcasts the given intent to all interested BroadcastReceivers
    • An optional required permission could be enforced
  — This call is asynchronous; it returns immediately

• sendOrderedBroadcast
  — Broadcasts the given intent to all interested BroadcastReceivers
    • Delivering them one at a time to allow preferred receivers to consume the broadcast before it is delivered to the others
  — This call is asynchronous; it returns immediately
Local broadcasting

• LocalBroadcastManager can be used to register for and broadcast intents to local objects within your application

• Broadcasted data do not leave your app
  — No need to worry about leaking private data

• Other apps cannot communicate with these broadcasts
  — No need to worry about having security holes

• More efficient than sending a global broadcast through the system
Services

- Application components that perform long-running operations in background without a user interface
  - Must be declared in the manifest
  - Any app component can use a service in the same way as any component can use an activity
    - We can declare the service as private, in the manifest file, and block access from other applications
- Continue to run in background even if the user switches to another application
- A component can also bind to a service to interact with it and perform inter-process communication (IPC)
Three types

- **Scheduled**
  - A JobScheduler (API level 21) launches the service
  - The system schedules the jobs for execution at the appropriate times
  - Google recommends that we use JobScheduler to execute background services
- **Started**
  - An application component (such as an activity) calls startService()
  - It runs in the background indefinitely, even if the component that started it is destroyed
  - Usually, it performs a single operation and does not return a result to the caller. When the operation is complete, it stops itself
- **Bound**
  - an application component binds to it by calling bindService()
  - Offers a client-server interface that allows components to interact with the service, even across processes with interprocess communication (IPC)
  - It runs only as long as an application component is bound to it
- **Services can be started and bound at the same time**
Runtime

• A service runs in the main thread of its hosting process
  – The service does not create its own thread and does not run in a separate process

• If your service is going to do any CPU intensive work you should create a new thread within the service to do that work
Callback methods (I)

• **onStartCommand()**
  - The system calls this method when another component requests that the service be started (through `startService(Intent)`)

• **onBind()**
  - The system invokes this method by calling `bindService()` when another component wants to bind with the service
  - You must provide an interface that clients use to communicate with the service by returning an `IBinder`
  - You must always implement this method; however, if you don't want to allow binding, you should return `null`
Callback methods (II)

• stopSelf() or stopService(Intent)
  — For the self-termination of the service or for asking for the termination of a service from the outside
  — No need to implement these methods if we only want to provide binding
  — System-decided termination (i.e., memory shortage)

• onCreate()
  — The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either onStartCommand() or onBind())
    • If the service is already running, this method is not called

• onDestroy()
  — The system calls this method when the service is no longer used and is being destroyed
startService() or bindService()

• If a service is started by invoking startService()
  — It keeps running until it stops itself or another component stops it

• If a service is created by invoking bindService() (and onStartCommand() is not called)
  — It only runs as long as a component is bound to it
  — When the service is unbound from all clients, it is destroyed

• Beginning with API level 21, the system throws an exception if we call bindService() with an implicit intent
IntentService

- Provides a straightforward solution for handling asynchronous requests (expressed through Intents)
  - onHandleIntent(Intent) must be properly redefined
- Clients send requests through startService(Intent)
  - The service is started as needed and handles Intents using a worker thread automatically
  - The service stops itself as soon it runs out of work
- All requests are handled by a single worker thread
  - Implementation of pattern “Work queue processor”
  - This pattern is commonly used to offload tasks from an application's main thread
  - Only one request will be processed at a time
Service or Thread

- A service can run in the background even when the user is not interacting with the application.
- A thread can perform work outside the main thread, but only while the user is interacting with the application.
Service Notifications

• Once running, a service can notify the user of events using
  – Toast notifications are messages that appear on the surface of the current window for a moment then disappear
  – Status bar notifications provide an icon in the status bar with a message, the user can select it to take an action
    • This is the best technique when some background work has completed
System services
Many different services

- Power Service
- KeyGuard Service
- Vibrator Service
- Alarm Service
- Sensor Service
- Audio Service
- Telephony Service
- Connectivity Service
- Wi-Fi Service
Power Service

- The battery of our phones must be used wisely

```java
PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl = pm.newWakeLock(
    PowerManager.PARTIAL_WAKE_LOCK, "My Tag")
wl.acquire();
// screen will stay off during this section
wl.release();
```

- Proper permissions in application's manifest
- Methods: isScreenOn, isPowerSaveMode, ...
- Device battery life will be significantly affected by the use of this API
  - Do not acquire locks unless you really need them
  - Be sure to release them as soon as possible.

<table>
<thead>
<tr>
<th>Flag Value</th>
<th>CPU</th>
<th>Screen</th>
<th>Keyboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIAL_WAKE_LOCK</td>
<td>On*</td>
<td>Off</td>
<td>Off</td>
</tr>
</tbody>
</table>
Vibrator Service

Vibrator vibrator = (Vibrator) getSystemService(Context.VIBRATOR_SERVICE);

• Some methods:
  — hasVibrator()
  — vibrate(long time);
  — cancel();
  — vibrate(long[] pattern, int repeat);

• Needs android.permission.VIBRATE
Alarm Service

- Allows one to schedule an application to be run in the future
  - When an alarm goes off, the Intent that had been registered for it is broadcast by the system and the target application is run automatically
  - Registered alarms are retained while the device is asleep (and can optionally wake it up)
    - They are cleared if it is turned off and rebooted
- Beginning with API 19, the OS shifts alarms to minimize wakeups and battery use
  - There are new APIs to support applications that need strict delivery guarantees
Alarm Service

```java
AlarmManager as = (AlarmManager) getSystemService(Context.ALARM_SERVICE);

• set(Exact)(int type, long triggerAtTime, PendingIntent operation)
  — Schedule an alarm
• set(Inexact)Repeating(int type, long triggerAtTime, long interval, PendingIntent operation)
  — Schedule a repeating alarm
• cancel(PendingIntent operation)
  — Remove all alarms that match
• setWindow(int type, long windowStartMillis, long windowLengthMillis, PendingIntent operation)
  — Schedule an alarm to be delivered within a given window of time
```
Sensors

• Android supports three broad categories of sensors
  — Motion sensors: accelerometers, gravity sensors, gyroscopes, and rotational vector sensors
  — Environmental sensors: barometers, photometers, and thermometers
  — Position sensors: orientation sensors and magnetometers

• Sensor framework helps
  — Determine which sensors are available on a device
  — Determine an individual sensor's capabilities
  — Acquire raw sensor data
  — Register and unregister sensor event listeners that monitor sensor changes

• Key elements
  — SensorManager, Sensor, SensorEvent, and SensorEventListener
<table>
<thead>
<tr>
<th>Sensor</th>
<th>Type</th>
<th>Description</th>
<th>Common Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_ACCELEROMETER</td>
<td>Hardware</td>
<td>Measures the acceleration force in m/s² that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.</td>
<td>Motion detection (shake, tilt, etc.).</td>
</tr>
<tr>
<td>TYPE_AMBIENT_TEMPERATURE</td>
<td>Hardware</td>
<td>Measures the ambient room temperature in degrees Celsius (°C). See note below.</td>
<td>Monitoring air temperatures.</td>
</tr>
<tr>
<td>TYPE_GRAVITY</td>
<td>Software or Hardware</td>
<td>Measures the force of gravity in m/s² that is applied to a device on all three physical axes (x, y, z).</td>
<td>Motion detection (shake, tilt, etc.).</td>
</tr>
<tr>
<td>TYPE_GYROSCOPE</td>
<td>Hardware</td>
<td>Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).</td>
<td>Rotation detection (spin, turn, etc.).</td>
</tr>
<tr>
<td>TYPE_LIGHT</td>
<td>Hardware</td>
<td>Measures the ambient light level (illumination) in lx.</td>
<td>Controlling screen brightness.</td>
</tr>
<tr>
<td>TYPE_LINEAR_ACCELERATION</td>
<td>Software or Hardware</td>
<td>Measures the acceleration force in m/s² that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.</td>
<td>Monitoring acceleration along a single axis.</td>
</tr>
<tr>
<td>TYPE_MAGNETIC_FIELD</td>
<td>Hardware</td>
<td>Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.</td>
<td>Creating a compass.</td>
</tr>
<tr>
<td>TYPE_ORIENTATION</td>
<td>Software</td>
<td>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.</td>
<td>Determining device position.</td>
</tr>
</tbody>
</table>
Sensor Service

SensorManager sm = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
List<Sensor> sensors = sm.getSensorList(Sensor.TYPE_ALL);

- getDefaultSensor() returns default sensor of a given type
  - If a default sensor does not exist, the method call returns null, which means the device does not have that type of sensor

```java
if (sm.getDefaultSensor(Sensor.TYPE_GRAVITY) != null) {
  List<Sensor> gravSensors = sm.getSensorList(Sensor.TYPE_GRAVITY);
  for (int i=0; i<gravSensors.size(); i++) {
    if (((gravSensors.get(i).getVendor().contains("Google Inc.")) &&
         (gravSensors.get(i).getVersion() == 3)) {
      // Use version 3 gravity sensor.
      s = gravSensors.get(i);
    }
  }
}
```
How to monitor sensor events

- Implement two callback methods that are exposed through the SensorEventListener interface
  - onAccuracyChanged()
    - 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
  - onSensorChanged()
    - A SensorEvent object contains information about the new sensor data
      - 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
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager mSenMan;
    private Sensor mLight;

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

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

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

```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();
    mSenMan.registerListener(this, mLight,
                            SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    super.onPause();
    mSenMan.unregisterListener(this);
}
```
Telephony Service

- Provides access to information about the telephony services on the device
  - Applications can use the methods in this class to determine telephony services and states, as well as to access some types of subscriber information
  - Applications can also register a listener to receive notification of telephony state changes
- Some methods
  - `getCallState()`, `getDataState()`, `getDataActivity()`, `getNetworkType()`, `getCellLocation()`, `getPhoneType()`, and `isNetworkRoaming()`
  - Through usual acquisition of a TelephonyManager
Text messages

• android.telephony.SmsManager vs android.telephony.gsm.SmsManager
• Only the default SMS app (selected by the user in system settings) is able to write to interact with it (through Intents)
• SmsManager manages SMS operations
  – getCarrierConfigValues(), divideMessage(), sendDataMessage(), sendTextMessage()
  • sent and delivery: two intents to be fired when the message is sent and/or delivered

http://android-developers.blogspot.it/2013/10/getting-your-sms-apps-ready-for-kitkat.html
Connectivity Service

- Checks the state of network connectivity
  ```java
  ConnectivityManager cm = (ConnectivityManager)
    getSystemService(Context.CONNECTIVITY_SERVICE);
  ```
- Monitors Wi-Fi, GPRS, UMTS, ecc.
- Sends broadcast intents when network connectivity changes
- Attempts to "fail over" to another network when connectivity to a network is lost
- Provides an API that allows applications to query the state of the available networks
- Needs proper permissions
Wi-Fi Service

• Manages all aspects of Wi-Fi connectivity
  – getWifiState()
    • Returns WIFI_STATE_DISABLED, WIFI_STATE_DISABLING, WIFI_STATE_ENABLED, WIFI_STATE_ENABLING, WIFI_STATE_UNKNOWN
  – isWifiEnabled() / setWifiEnabled()
  – getConfiguredNetworks()
  – addNetwork(WifiConfiguration config)
  – updateNetwork(WifiConfiguration config)
  – removeNetwork(int netid)
  – startScan()
  – getScanResults()
Summary

- Android Applications are structured as a single Activity or a group of Activities
  - Intents to call other activities
  - Layout and Views to setup the GUI
  - Events to manage the interactions with the user

- Activities execute only in foreground
  - Updates in background mode
    - A Service provides a robust environment for background tasks
  - Notifications in case of changes
import com.google.android.gms.maps.*;
import com.google.android.gms.maps.model.*;
import android.app.Activity;
import android.os.Bundle;

public class MapPane extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.map_activity);

        // Get a handle to the Map Fragment
        GoogleMap map = ((MapFragment) getFragmentManager()
            .findFragmentById(R.id.map)).getMap();

        LatLng sydney = new LatLng(-33.867, 151.206);
        map.setMyLocationEnabled(true);
        map.moveCamera(CameraUpdateFactory.newLatLngZoom(sydney, 13));

        map.addMarker(new MarkerOptions()
            .title("Sydney")
            .snippet("The most populous city in Australia.")
            .position(sydney));
    }
}

Change the Map Type
Indoor Maps
Custom Markers and Info windows
Flat Markers
Polylines
Many other things ...
Testing
Different types

- Unit testing
- UI testing
- Functional testing
- Integration testing
- Security testing
- Compatibility testing
Three main elements

- **JUnit**
  - The Android JUnit extensions provide component-specific test case classes
- **Monkeyrunner**
  - An API for testing devices with Python programs
- **UI/Application Exerciser Monkey**
  - A command-line tool for stress-testing UIs by sending pseudo-random events to a device
Monkeyrunner

• Provides an API for writing programs that control an Android device or emulator from outside of Android code
• We can write a Python program that installs an Android application or test package, runs it, sends keystrokes to it, takes screenshots of its user interface, and stores screenshots on the workstation.
  — Functional testing
  — Regression testing
• One can develop an entire system of Python-based modules and programs for controlling Android devices
UI/Application Exerciser Monkey

- Is a program that runs on an emulator or device and generates pseudo-random streams of user events
- It includes a number of options:
  - Basic configuration options, such as setting the number of events to attempt
  - Operational constraints, such as restricting the test to a single package
  - Event types and frequencies
  - Debugging options
Other testing tools

- Autoandroid
- Positron (not maintained anymore)
- Selendroid
- Robotium
- …