Android

Desenvolvimento de Software e Sistemas Móveis (DSSMV)
Licenciatura em Engenharia de Telecomunicações e Informática
LETI/ISEP

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

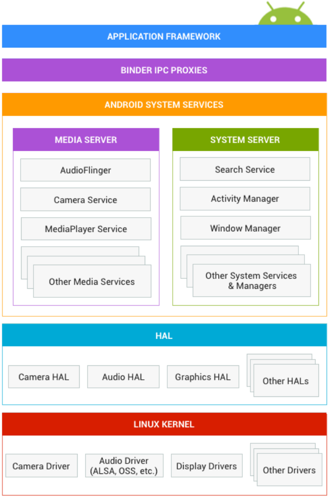
Outline

1. `getSystemService`
2. Internet Connectivity
3. Location
4. Sensors
5. Service
6. Result Receiver
7. BroadcastReceiver
8. Bibliography

getSystemService

getSystemService (I)

- Context.getSystemService method is used to access one of Android system-level services:
  - LocationManager – access GPS device information
  - NotificationManager – manage (create, cancel) notifications
  - ConnectivityManager – access network connection details and manage those connections
  - LayoutInflater – inflate and create views from xml layout resource files
  - ...



getSystemService (II)

- By name
  - `Object getSystemService (String name)`

```
Context context = getApplicationContext();
AudioManager audioManager = (AudioManager) context.getSystemService(
      Context.AUDIO_SERVICE);
Vibrator vibrator = (Vibrator) context.getSystemService(Context.
      VIBRATOR_SERVICE);
```

- By class.
  - `T getSystemService (Class<T> serviceClass)`

```
Context context = getApplicationContext();
AudioManager audioManager = (AudioManager) context.getSystemService(
      AudioManager.class);
Vibrator vibrator = (Vibrator) context.getSystemService(Vibrator.class);
```

Internet Connectivity

Connection State (I)

- A device can have various types of network connections.
- If your app needs internet connection to make HTTP Requests and or you need internet connection in your whole app then it is better to check internet connectivity status before making any HTTP Requests to avoid http exceptions.
- For this android provides `ConnectivityManager` class.
- You need to instantiate an object of this class by calling `getSystemService()` method.

```
ConnectivityManager connectivityManager = (ConnectivityManager) context.
     getSystemService(Context.CONNECTIVITY_SERVICE);
```

Connection State (III)

- Create a `NetworkRequest` object, using `builder` method.
- Add network transports for checking.

```
NetworkRequest networkRequest = new NetworkRequest.Builder()
  .addTransportType(NetworkCapabilities.TRANSPORT_CELLULAR)
  .addTransportType(NetworkCapabilities.TRANSPORT_WIFI)
  .build();
```

- Create a `NetworkCallback` object overridden `onAvailable` and `onLost` event handlers.

```
ConnectivityManager.NetworkCallback networkCallback = new ConnectivityManager.
      NetworkCallback() {
  @Override
  public void onAvailable(Network network) {
    super.onAvailable(network);
    ...
    }
  @Override
  public void onLost(Network network) {
    super.onLost(network);
    ...
  }
};
```

Connection State (IV)

- Register the network calback.

```
connectivityManager.registerNetworkCallback(networkRequest, networkCallback);
```

- Unregister the network calback.

```
connectivityManager.unregisterNetworkCallback(networkCallback);
```

Location

Location Services

- Android gives your applications access to the location services supported by the device through classes in the `android.location` package.
- The central component of the location framework is the `LocationManager` system service, which provides APIs to determine location and bearing of the underlying device (if available).
- `LocationManager` **cannot be instantiate directly**.
    - Request an instance from the system by calling `getSystemService(Context.LOCATION_SERVICE)`.

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager) this.getSystemService
      (Context.LOCATION_SERVICE);
```

    - The method returns a handle to a new `LocationManager` instance.

LocationManager

- This class **provides access to the system location services**.
- These services allow applications to obtain **periodic updates** of the device's geographical location, or to fire an application-specified `Intent` when the device **enters the proximity** of a given geographical location.
- All `Location` API methods require the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permissions.
    - If your application only has the coarse permission then it will not have access to the GPS or passive location providers.
    - Other providers will still return location results, but the update rate will be throttled and the exact location will be obfuscated to a coarse level of accuracy.

Location Aware

- Knowing where the user is allows your application to be smarter and deliver better information to the user.
- When developing a location-aware application for Android, you can utilize **GPS** and Android's **Network Location Provider** to acquire the user location.
  - **GPS is most accurate**, it only works outdoors, it quickly consumes battery power, and doesn't return the location as quickly as users want.
  - Android's **Network Location Provider determines user location using cell tower and Wi-Fi signals**, providing location information in a way that works indoors and outdoors, responds faster, and uses less battery power.
  - To obtain the user location in your application, you **can use both GPS and the Network Location Provider**, or just one.

Requesting Location Updates (I)

- Getting user location in Android works by means of callback.
- You indicate that you'd like to receive location updates from the `LocationManager` by calling `requestLocationUpdates()`, passing it a `LocationListener`.
- **LocationListener** is an implementation of the Observer pattern
  - **Observer is a behavioral design pattern** that allows one objects to notify other objects about changes in their state.
    - The Observer pattern provides a way to subscribe and unsubscribe to and from these events for any object that implements a subscriber interface.
  - The object which is being watched is called the **subject**. The objects which are watching the state changes are called **observers or listeners**.
- Your `LocationListener` must implement several callback methods that the `LocationManager` calls when the user location changes or when the status of the service changes.

Requesting Location Updates (II)

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager) this.getSystemService(Context.
      LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {
public void onLocationChanged(Location location) {
// Called when a new location is found by the network location provider.
makeUseOfNewLocation(location);
}

public void onStatusChanged(String provider, int status, Bundle extras) {}

public void onProviderEnabled(String provider) {}

public void onProviderDisabled(String provider) {}
};

// Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0,
      locationListener);
...
// Stops receiving location updates
locationManager.removeUpdates(locationListener)
```

Requesting Location Updates (II)

- `requestLocationUpdates(String provider, long , float minDistance, LocationListener listener)`
  - The first parameter is the type of location `provider` to use.
  - The frequency at which your listener receives updates can be controlled with the second and third parameter–the second is the minimum time (`minTime`) interval between notifications and the third is the minimum change in distance (`minDistance`) between notifications.
    - Setting both to zero requests location notifications as frequently as possible.
  - The last parameter (`listener`) is the `LocationListener`, which receives callbacks for location updates.

Sensors

Basics

- 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(depends on the device) 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.
- To identify the device sensors you must first access the sensor manager, an Android system service.
- Create an instance of the `SensorManager` class by calling the `getSystemService()` method and passing in the `SENSOR_SERVICE` argument.

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

Identifying

- To get a listing of all the device sensors, use the `getSensorList()` method and the sensor type `TYPE_ALL`.

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

- `Sensor` class represents a specific sensor.

Monitor Events (I)

- To monitor sensor events in your app, you must:
- Implement the `SensorEventListener` interface, which includes the `onSensorChanged()` and `onAccuracyChanged()` callback methods.

```
SensorEventListener sensorEventListener = new SensorEventListener() {
   @Override
   public void onAccuracyChanged(Sensor arg0, int arg1) {...}
   @Override
   public void onSensorChanged(SensorEvent arg0) {
     synchronized (sensorEventListener) {
       switch (arg0.sensor.getType()){
         case Sensor.TYPE_ACCELEROMETER:
           Float X=arg0.values[0];
           ...
         break;
       }
     }
   }
 };
```

Monitor Events (II)

- Get sensor types and values from the `SensorEvent` object, and update your app

```
Sensor sensorAccelerometer = sensorManager.getDefaultSensor(Sensor.
     TYPE_ACCELEROMETER);
```

- Register sensor event listeners for the specific sensor you want to monitor. accordingly.

```
sensorManager.registerListener(sensorEventListener,sensorAccelerometer,
     SensorManager.SENSOR_DELAY_NORMAL);
```

- Unregister sensor event listeners.

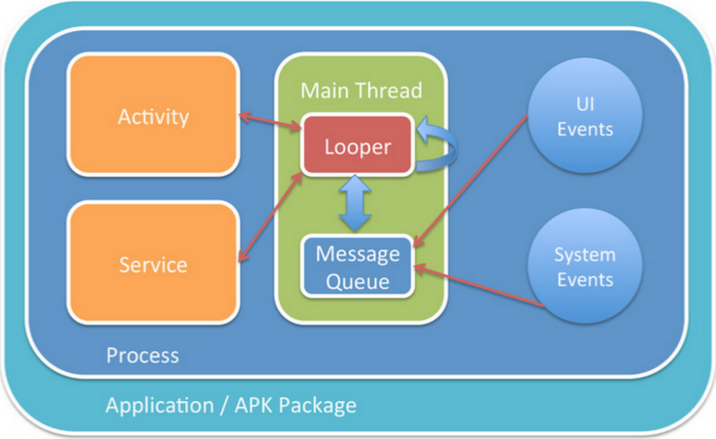
```
sensorManager.unregisterListener(sensorEventListener, sensorAccelerometer);
```

Service

Service (I)

- A Service is an **application component** that can **perform long-running operations** in the background
- A Service **does not provide a user interface**.
- A Service can be started and stoped.
  - An application component (e.g an Activity) can start a Service and it will continue to run in the background even if the user switches to another application.
- A Service is neither a separate process nor a thread.
- A Service **has to be registered into Manifest file**.

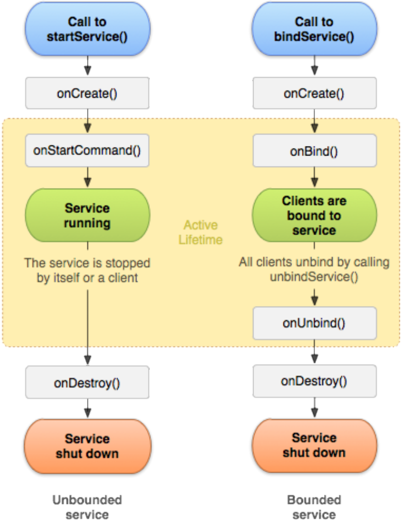
Service (II)


Activity
Main Thread
Looper
UI
Events
Message
Queue
Service
System
Events
Process
Application / APK Package

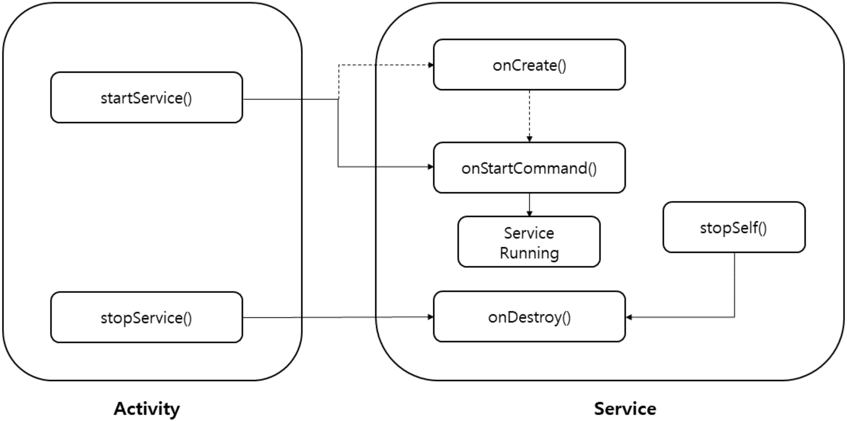
Types of Service

- **Started**
  - A `Service` is **started** when an application component starts it by calling `startService`.
    - A `Service` can run indefinitely, even if the application component that started it is destroyed.
    - A A `Service` **performs a single operation and does not return a result to the caller**.
- **Bound**
  - A `Service` is **bound** when an application component binds to it by calling `bindService`.
    - A bound service offers a client-server interface that allows components to interact with the service.
    - **A bound service runs only as long as another application component is bound to it**.
    - Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.
- The **same service can be both**: started and bound.

Service Lifecycle



Started Service Lifecycle



Started Service (I)

- An implemented service must be a subclass of `Service`.

```java
public class HelloService extends Service {
  @Override
  public void onCreate() {
    ...
  }
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    // If we get killed, after returning from here, restart
    return START_STICKY;
  }
  @Override
  public IBinder onBind(Intent intent) {
    // We don’t provide binding, so return null
    return null;
  }
  @Override
  public void onDestroy() {
    ...
  }
}
```

Started Service (II)

- `onStartCommand` method must return an integer.
  - The integer is a value that describes how the system should continue the service in the event that the system kills it.
- The return value from `onStartCommand` must be one of the following constants:
  - `START_NOT_STICKY`: if the system kills the service after `onStartCommand` returns, do not recreate the service.
  - `START_STICKY`: if the system kills the service after `onStartCommand` returns, recreate the service and call `onStartCommand`, but do not redeliver the last intent. Instead, the system calls `onStartCommand` with a null intent.
  - `START_REDELIVER_INTENT`: if the system kills the service after `onStartCommand` returns, recreate the service and call `onStartCommand` with the last intent that was delivered to the service

Started Service (III)

- Starting a service

```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

  - The `startService` method returns immediately, and the Android system calls the service's `onStartCommand` method. If the service is not already running, the system first calls `onCreate`, and then it calls `onStartCommand`.

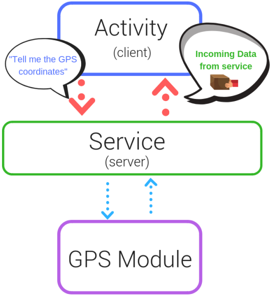
- Stopping a service

```
Intent intent = new Intent(this, HelloService.class);
stopService(intent);
```

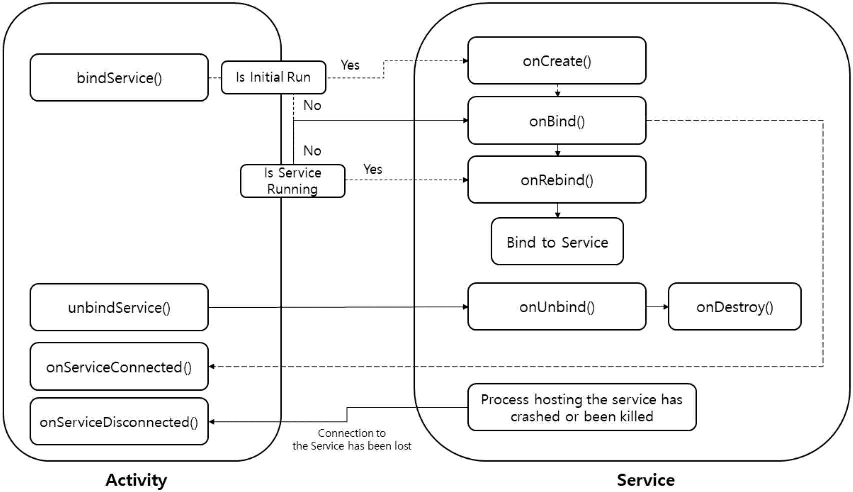
  - The service must stop itself by calling `stopSelf`, or another component can stop it by calling `stopService`.

Bound Service

- A **bound service offers a client-server interface** that allows components to interact with the service, send requests and get results.
- Types of Bound Service
    - Local Bound Service
    - Remote Bound Service
- A local bound service can be created using `Binder` class and for creating remote bound service you need either `Messenger` class or AIDL mechanism.

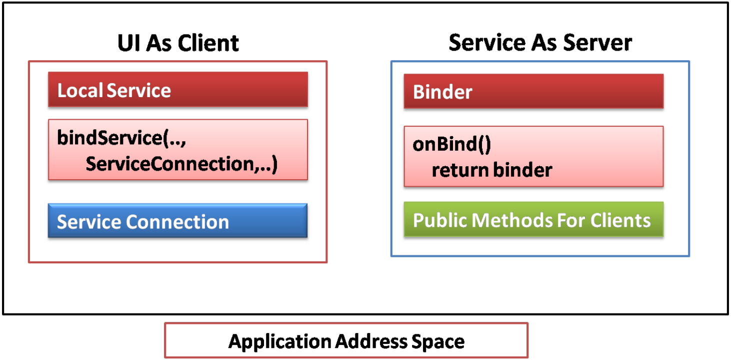


Bound Service Lifecycle



Local Bound Service (I)

- As mentioned before, a `Service` does not create any thread.
- To process requests parallely you have to create threads.



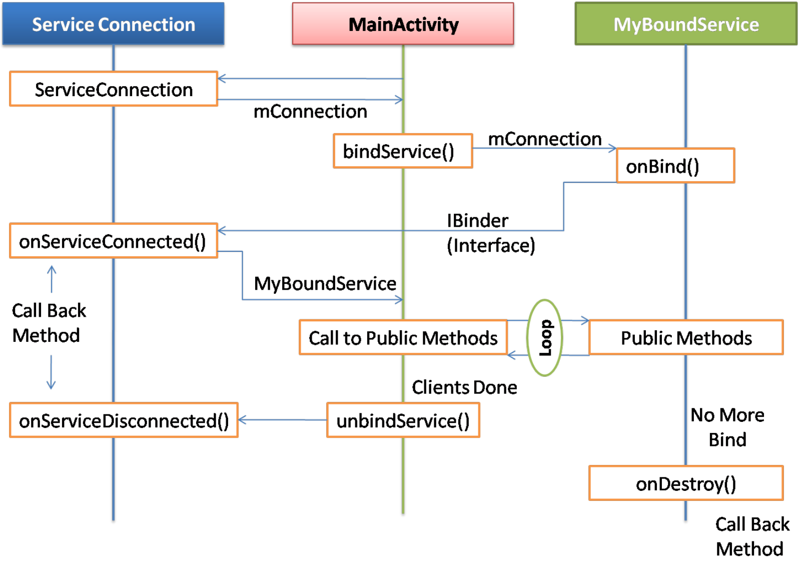
Local Bound Service (II)

```
public class LocalBoundService extends Service {
  // IBinder is the Interface between Client and Server which returns this service
  private final IBinder mBinder = new LocalBinder();

   public class LocalBinder extends Binder {
     LocalBoundService getService() {
     // Return this instance of LocalService so clients can call public methods
      return LocalBoundService.this;
     }
   }

  @Override
  public IBinder onBind(Intent intent) {
   return mBinder;
  }

  // Public methods to be called by clients
   ...
  }
}
```

Local Bound Service (III)

```
public class BindingActivity extends Activity {
  // Define a Local Bound Service Reference
  LocalBoundService mService;
  boolean mBound = false;
  @Override
  protected void onStart() {
    super.onStart();
    Intent intent = new Intent(this, LocalService.class);
    // Bind with the bound service with this activity component
    bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
  }
  // Defines callbacks for service binding, passed to bindService()
  private ServiceConnection mConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className,IBinder service){
      LocalBinder binder = (LocalBinder) service;
      mService = binder.getService();
      mBound = true;
    }
    @Override
    public void onServiceDisconnected(ComponentName name) {
      mBound = false;
    }
  };
  ...
}
```

Local Bound Service (IV)



Started Service: IntentService

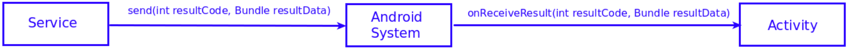
- The `IntentService` class **provides a straightforward structure for running an operation on a single background thread**.
  - It automatically **creates a worker thread**.

DEPRECATED

Result Receiver

ResultReceiver (I)

- ResultReceiver is a generic interface for receiving a callback result from a Service.
- The ResultReceiver class is just a simple wrapper around a Binder that is used to perform the communication.
- An instance of this class can be passed through an intent.
- Use this by creating a subclass and implementing onReceiveResult.
- The sender uses the send method to send the data to the receiver.


Service
send(int resultCode, Bundle resultData)
Android System
onReceiveResult(int resultCode, Bundle resultData)
Activity

ResultReceiver (II)

- Create a ResultReceiver object into the Activity

```
public class MainActivity extends AppCompatActivity {
   ResultReceiver receiver = new ResultReceiver(new Handler(Looper.
        getMainLooper())) {
    @Override
    protected void onReceiveResult(int resultCode, Bundle resultData) {
      super.onReceiveResult(resultCode, resultData);
      if (resultCode == Activity.RESULT_OK) {
       String val = resultData.getString("EXTRA_VALUE");
      }
    }
   };
   ...
}
```

- Put it into the Intent object as an extra

```
   public class MainActivity extends AppCompatActivity {
   ...
    public void onClick(View arg0) {
      Intent intent = new Intent(MainActivity.this, MyService.class);
      intent.putExtra("EXTRA_RECEIVER", receiver);
      startService(intent);
    }
```

ResultReceiver (III)

- Get ResultReceiver object from Service intent object

```
public class MyService extends Service {
   ...
   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
     super.onStartCommand(intent, flags, startId);
     final ResultReceiver receiver = intent.getParcelableExtra("EXTRA_RECEIVER"
          );
   ...
   }
   ...
}
```

- Create a Bundle with data and send it

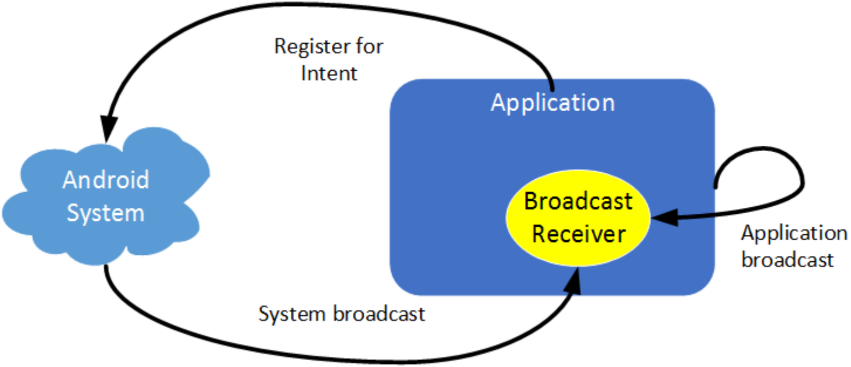
```
public class MyService extends Service {
   ...
   Bundle bundle = new Bundle();
   bundle.putString("EXTRA_VALUE", "Message from MyService");
   receiver.send(Activity.RESULT_OK, bundle);
   ...
}
```

BroadcastReceiver

Broadcast Receiver (I)

- Android apps can send or receive broadcast messages from the Android system and other Android apps.
  - **publish-subscribe** design pattern.
- These broadcasts are sent when an event of interest occurs.
  - **Android system sends broadcasts when various system events occur**, such as when the system boots up or the device starts charging.
  - **Apps can also send custom broadcasts**, for example, to notify other apps of something that they might be interested in (for example, some new data has been downloaded).
- **Apps can register to receive specific broadcasts**.
  - **When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast**.
- They can be used as a messaging system across apps and outside of the normal user flow.

Broadcast Receiver (II)



Receiving broadcasts (I))

- Manifest-declared receivers
  - If you declare a broadcast receiver in your **manifest**, the system launches your app (if the app is not already running) when the broadcast is sent.
  - To declare a broadcast receiver in the manifest, perform the following steps:
    1. Specify the `<receiver>` element in your app's manifest.

```
<receiver android:name=".MyBroadcastReceiver" android:exported="
      true">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
  </intent-filter>
</receiver>
```

    2. Subclass `BroadcastReceiver` and implement `onReceive`.

```
public class MyBroadcastReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    ...
  }
}
```

Receiving broadcasts (II))

- Manifest-declared receivers (cont.)
  - **The system package manager registers the receiver when the app is installed**.
  - The receiver then becomes a separate entry point into your app which means that the system can start the app and deliver the broadcast if the app is not currently running.
  - The system creates a new `BroadcastReceiver` component object to handle each broadcast that it receives.
    - This object is valid only for the duration of the call to `onReceive`.
    - Once your code returns from this method, the system considers the component no longer active.

Receiving broadcasts (III))

- Context-registered receivers
  - To register a receiver with a context, perform the following steps:

1. Create an instance of BroadcastReceiver.

```
public class MainActivity extends AppCompatActivity {
  private BroadcastReceiver broadcastReceiver = new
        BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      ...
    }
  };
}
```

2. Create an `IntentFilter` and register the receiver by calling `registerReceiver`

```
public class MainActivity extends AppCompatActivity {
  private BroadcastReceiver broadcastReceiver = new ...
    protected void onCreate(Bundle savedInstanceState) {
    ...
    IntentFilter intentFilter= new IntentFilter(ConnectivityManager.
           CONNECTIVITY_ACTION);
    intentFilter.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
    registerReceiver(broadcastReceiver,intentFilter);
```

Receiving broadcasts (IV))

- Context-registered receivers (cont)
  - To register a receiver with a context, perform the following steps (cont):
    3. To stop receiving broadcasts, call `unregisterReceiver`

```
public class MainActivity extends AppCompatActivity {
  private BroadcastReceiver broadcastReceiver = new ...
  protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(broadcastReceiver);
  }
}
```

- **Context-registered receivers receive broadcasts as long as their registering context is valid**.

Sending broadcasts (I))

- The sendOrderedBroadcast(Intent, String) method sends broadcasts to one receiver at a time.
- The sendBroadcast(Intent) method sends broadcasts to all receivers in an undefined order. This is called a Normal Broadcast.

```java
Intent intent = new Intent();
intent.setAction("com.example.broadcast.MY_NOTIFICATION");
intent.putExtra("data","Notice me senpai!");
sendBroadcast(intent);
```

- The LocalBroadcastManager.sendBroadcast method sends broadcasts to receivers that are in the same app as the sender. If you don't need to send broadcasts across apps, use local broadcasts.

Bibliography

Resources

- "Mastering Android Application Development", by Antonio Pachon Rui, 2015
- https://developer.android.com/index.html
- http://simple.sourceforge.net/home.php
  http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php