Android Application Components

Lecture 3

Goal of this lecture

- Understand the main components which come into play when designing an app
- Lifecycle and app management
- Differences from desktop apps

Content

- Overview of the four components:

1. **activities**,
   - intents,
2. **services**,
3. **content providers**,
4. **broadcast receivers**

- Manifest definition
- Activity lifecycle
- Task stack management

Activities

- Entry point for any Android app
- Provides the user interface to draw the app
- Consists of Java/Kotlin code (control) and XML layout (view)
- Extend base class Activity (AppCompatActivity)

Example of a chat application with one activity each for:

- List of all contacts
- List of started message threads
- List of messages inside thread
- Preferences, Settings, edit profile etc.

Activities

An app usually has multiple activities, one is the *main activity*, the entry point for the application

- Activities can start each other, can pass data (intents)
- Loosely coupled (no direct references)
- Can start activities from another app

  Start default browser to handle URL, or start sharing app to send multimedia content

  Switching between different activities ~ switching between tabs in a browser

Manifest file

All activities need to be defined in the manifest.xml file

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

Optional tags for icon, theme, configuration, orientation, intent filters etc.

Manifest file

Intent filters are a powerful tool to launch activities implicitly or explicitly.

```
<activity android:name=".FirstActivity"
android:icon="@drawable/app_icon">
    <intent-filter>
      <action android:name="android.intent.action.SEND" />
      <category android:name="android.intent.category.DEFAULT" />
      <data android:mimeType="text/plain" />
    </intent-filter>
<activity>
```

Explicit call to an activity *within* the app to send email

Implicit call to an *external* app (Yahoo) to send mail

Manifest file

If no intent filter is defined, then the app cannot be called by other apps to handle requests.

```
// Create a text message …
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");
// Start the activity
startActivity(sendIntent);
```

Manifest file

The <activity> field may be used to control other apps accessing an activity.

```
<manifest> // of app to be called
   <activity android:name="...."
       android:permission =
       ”com.apps.messengerapp.permission.SHARE_POST”
   />

<manifest> // of calling app
   <uses-permission
       android:name="com.apps.messengerapp.permission.SHARE
       _POST"/>
```

Activity lifecycle

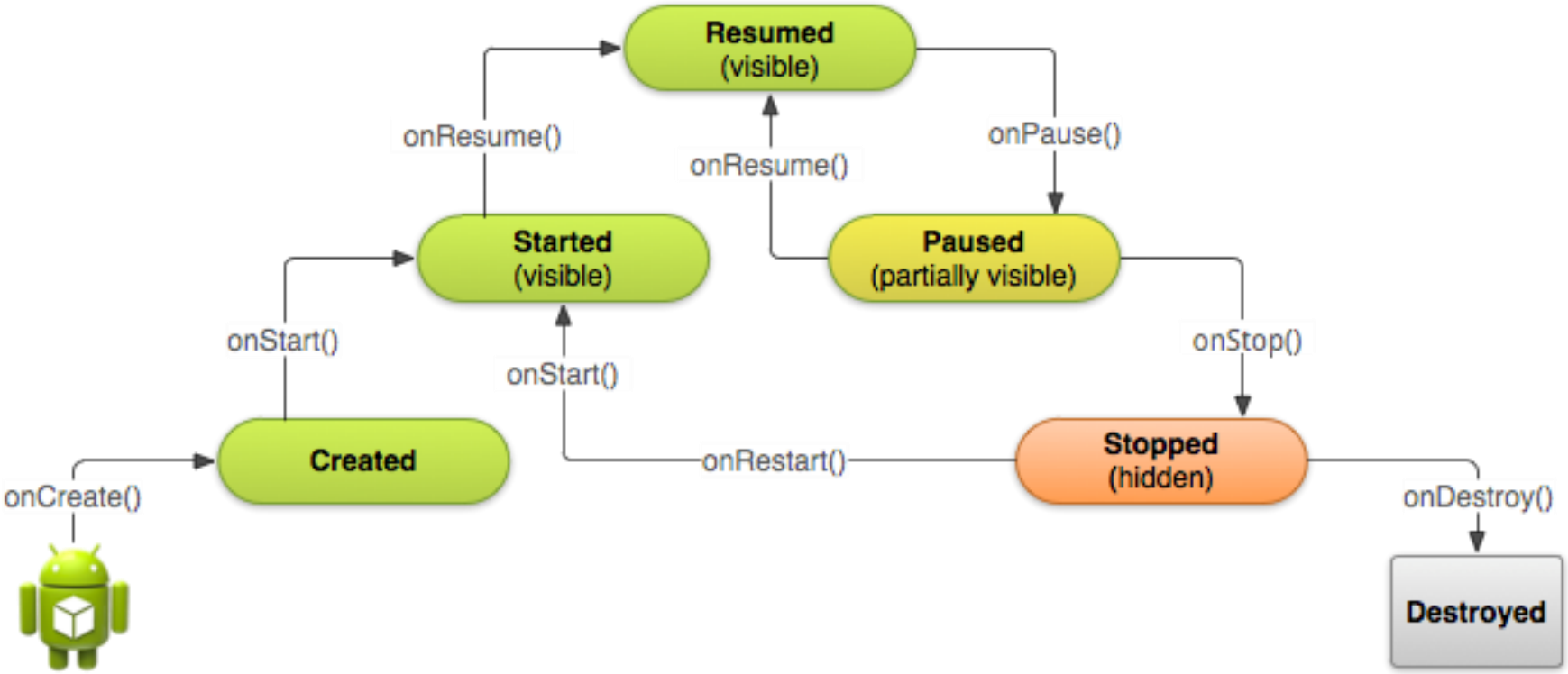
Launching an activity can be expensive, i.e.:

- New Linux process
- Memory allocation for UI elements
- Loading all XML layout elements
- Screen configuration

Activity manager stores copy of activity in background *for a time.*

Old activities are automatically destroyed to free up memory.

Activity lifecycle



Activity lifecycle

A developer cannot impose the state of an activity, only define what happens between transitions.

Callback methods for handling user-triggered transitions:

- **onCreate** – activity is created for the first time.

  Initialize UI,

  Populate with data,

  setContentView to link xml to Java code.

- **onStart** – activity is started and becomes visible for the user.

  Final preparations before becoming interactable for the user.

Activity lifecycle

- **onResume** – activity becomes interactable with the user.
  - Activity is on top of the activity stack,
  - Receives user input,
  - Basic functionality is implemented here.
- **onPause** – activity looses user focus becoming interrupted
  - When user presses: Back, View recent apps, dialog (pop-up)
  - Activity is still visible, but the user may be leaving the screen
  - Not to be used for storing data on network, DB transactions.
  - Very likely to also become *stopped* or *resumed*,
  - Followed by *onStop* or *onResume*.

Activity lifecycle

- **onStop** – activity ceases to be visible.

  Activity is being destroyed, or a new one is launched, or a stopped activity becomes resumed and covers the existing one.

- **onRestart** – activity is restored to the state before being stopped.

  Always followed by onStart.

- **onDestroy** – activity is destroyed.

  Last call an activity may receive.

  Releases all held resources.

  When pressing Back, calling finish(), or system handled.

Saving and restoring an activity

An activity may be destroyed by:

- the normal behavior of the app (pressing *Back* or *finishing* an activity) → removed permanently from system
- the system, for resource allocation → leaves a trace, under the form of *bundled* data useful for restoring

Instance state is stored in a key-value object called a Bundle

E.g. A bundle implicitly stores all text input from edit text views so they can be refilled with the same data

Saving and restoring an activity

When stopping an activity, the system calls *onSaveInstanceState*, implicitly saving view text data, list positions etc.

An activity may need more than implicit info for restoring.

Explicit call between onPause and onStop:

```
static final String STATE_COST = "paymentCost";
static final String STATE_MONTH = "userMonth";

public void onSaveInstanceState(Bundle savedInstanceState) {
    // save user data
    savedInstanceState.putInt(STATE_COST, mCurrentCost);
    savedInstanceState.putInt(STATE_MONTH, mCurrentMonth);

    // called to save view hierarchy
    super.onSaveInstanceState(savedInstanceState);
}
```

Saving and restoring an activity

When an activity is recreated the same Bundle object is used in *onCreate* (==null) and *onRestoreInstanceState* (!=null).

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // always call superclass
    // are we restoring an activity?
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentCost = savedInstanceState.getInt(STATE_COST);
        mCurrentMonth = savedInstanceState.getInt(STATE_MONTH);
    } else {
        // initialize members with implicit values
    }
}
```

Saving and restoring an activity

Alternatively, restore state directly in *onRestoreInstanceState*, called after onStart, only if a saved instance exists (no need to check bundle for being non-null)

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // call superclass first to restore view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // get values from saved state
    mCurrentCost = savedInstanceState.getInt(STATE_COST);
    mCurrentMonth = savedInstanceState.getInt(STATE_MONTH);
}
```

The Task / Back / Activity Stack

Context: an activity may start other activities from other apps. Based on intent filters we can use a third-party email client to send data from our app. If there are multiple apps that can handle the send mail event, the system asks the user to start one. When the email is sent, our app is restored on the screen giving the impression that everything ran within our initial app.

Android maintains this usage experience by holding both apps in the same task.

Task: collection of apps to accomplish one certain user activity.

The Activity Stack

Activities are placed on the activity stack or background stack (FILO – first in last out / last in first out).

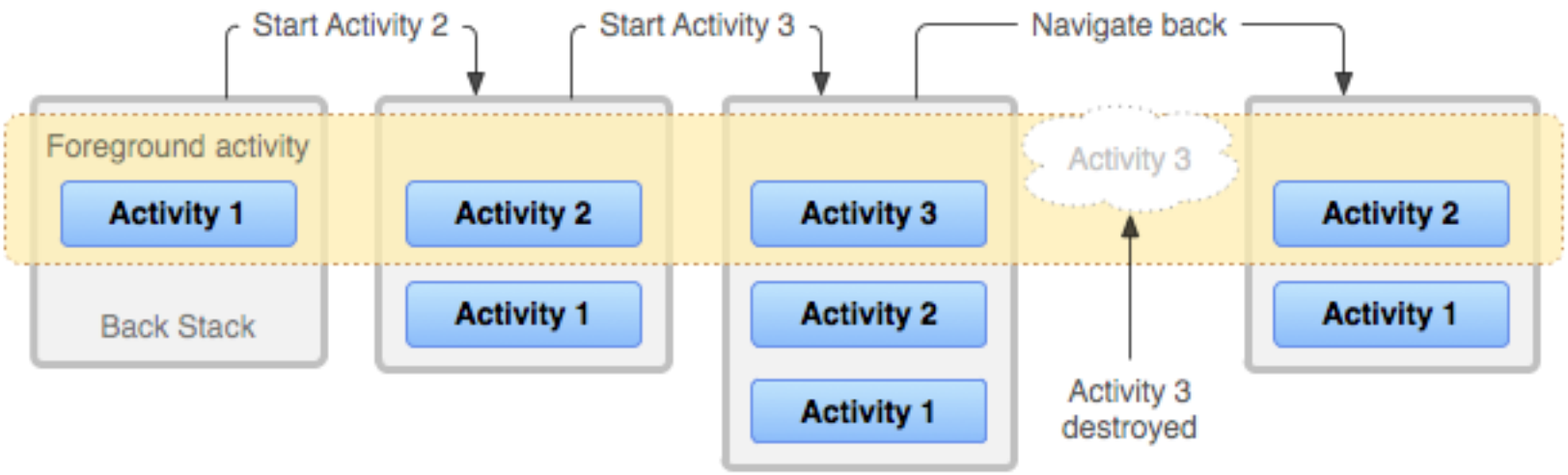
The home screen is the first activity in the stack for most apps.

When new activities are started (by user events) each is placed on top of the stack, in the foreground. All other activities become stopped, in the background.

When an icon is tapped on the screen the associated task is retrieved or a new one is created.

When the stack reaches the original activity (base), the task is destroyed.

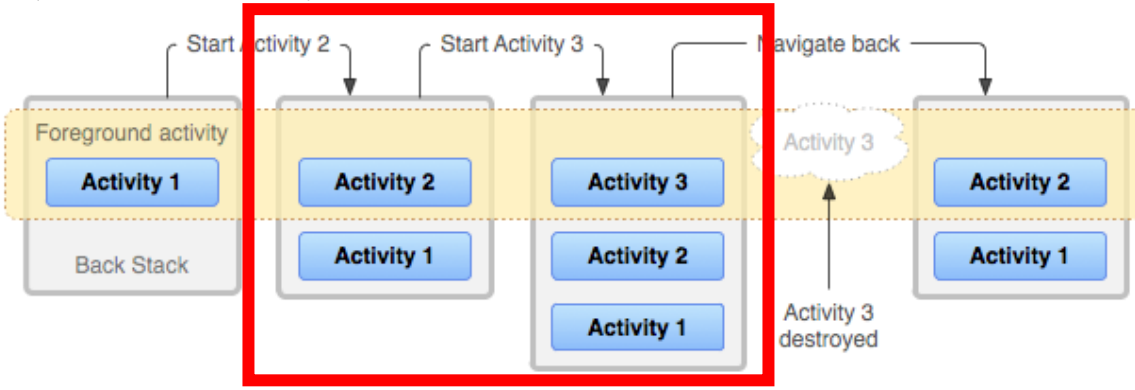
The Activity Stack



The Activity Stack – example of lifecycle methods

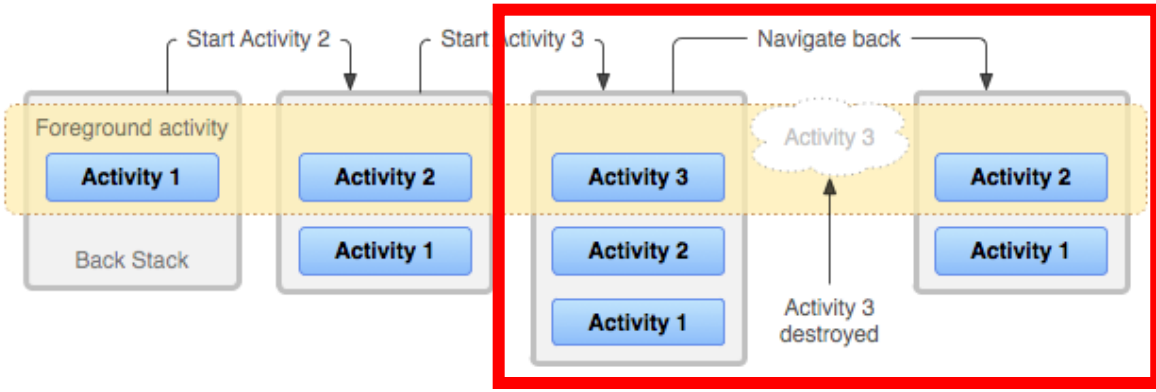
Creating a new activity: Activity2 → Activity3

- Activity2: onPause()
- Activity3: onCreate()
- Activity3: onStart()
- Activity3: onResume()
- Activity2: onStop()



Returning back to an activity: Activity2 ← Activity3

- Activity3: onPause()
- Activity2: onRestart()
- Activity2: onStart()
- Activity2: onResume()
- Activity3: onStop()
- Activity3: onDestroy()



Activities - Summary

- ✓The presentation and user interaction layer.
- ✓Represents a display or a window for user interface.
- ✓Is also an execution unit which contains the activity's functionality.
- ✓An Android application implements one ore more activities.
- ✓Can communicate between them and can launch one another but are still independent.
- ✓Are implemented as subclasses of the Activity class, and are defined in the Manifest file.
- ✓Calling an activity from another activity is done by using an Intent.

Intents

A message object used to request an action from another component of an app (e.g. activity, service)

Used to:

- Launch an activity
- Start/stop a service
- Broadcast information

Asynchronous, so they should not be waited upon to finish.

Intents

Three fundamental usages:

- Launch a new activity with a call to *startActivity*

  Can send data to the activity.

  To retrieve a result at the end, call *startActivityForResult* instead.

  Retrieving using *onActivityResult* in the caller activity.

- Starting a service

  For background tasks, without a UI.

  Using a *JobScheduler* from Android 5.0 (*Service* class before), or WorkManager.

- Handling a broadcast

  A message that any app can intercept, like system events.

  Message other apps with *sendBroadcast*.

Intents

Two types of intents:

- **Explicit** – specify component to start by fully qualified name.

  Usually starts another activity from the same app, e.g. show settings screen or start a background service to download a file.

- **Implicit** – specify a general action to be executed, that should be handled by another app.

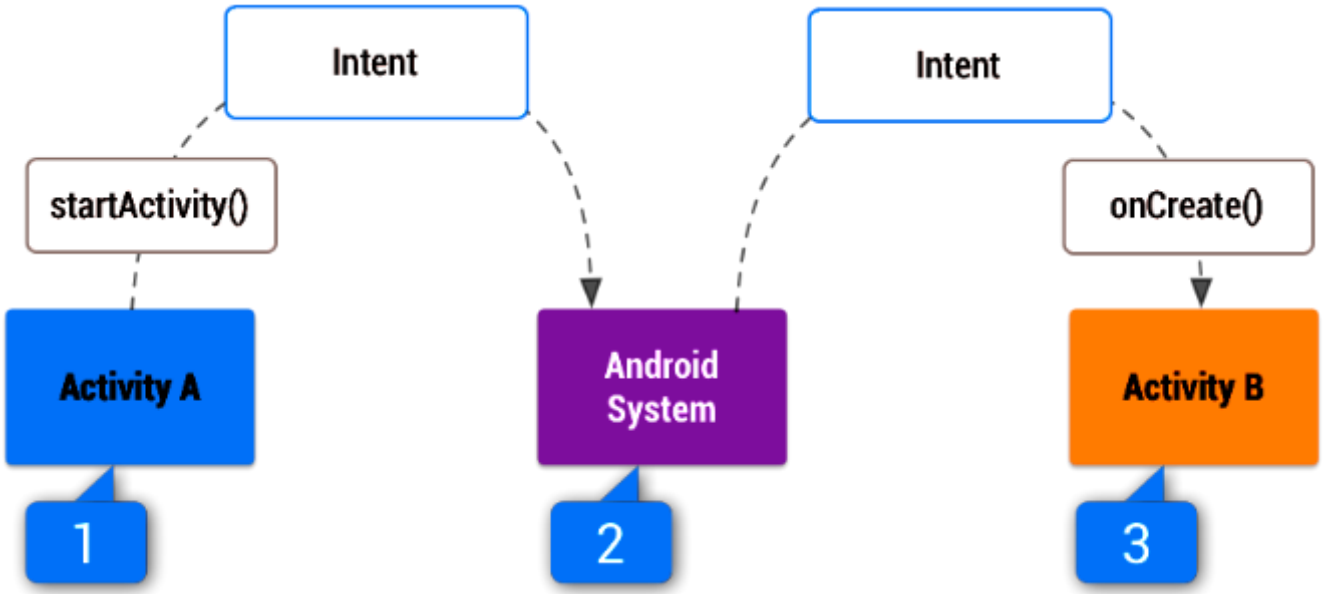
  Simple example: show a location on the map (start Maps).

  Complex example: a restaurant app allows you to take photos of places to eat and make reviews, so you can (1) take a picture (use Camera app), (2) then edit it (use some editing app), (3) then locate the place (use Maps), (4) and finally share the photo with friends (use social media apps).

Intents

How an intent is delivered to start an activity:

1. Activity A creates an intent with an action and calls it via *startActivity.*
2. Android searches all compatible apps by intent filter for given action.
3. If one single activity is found, the intent is passed via *onCreate.*

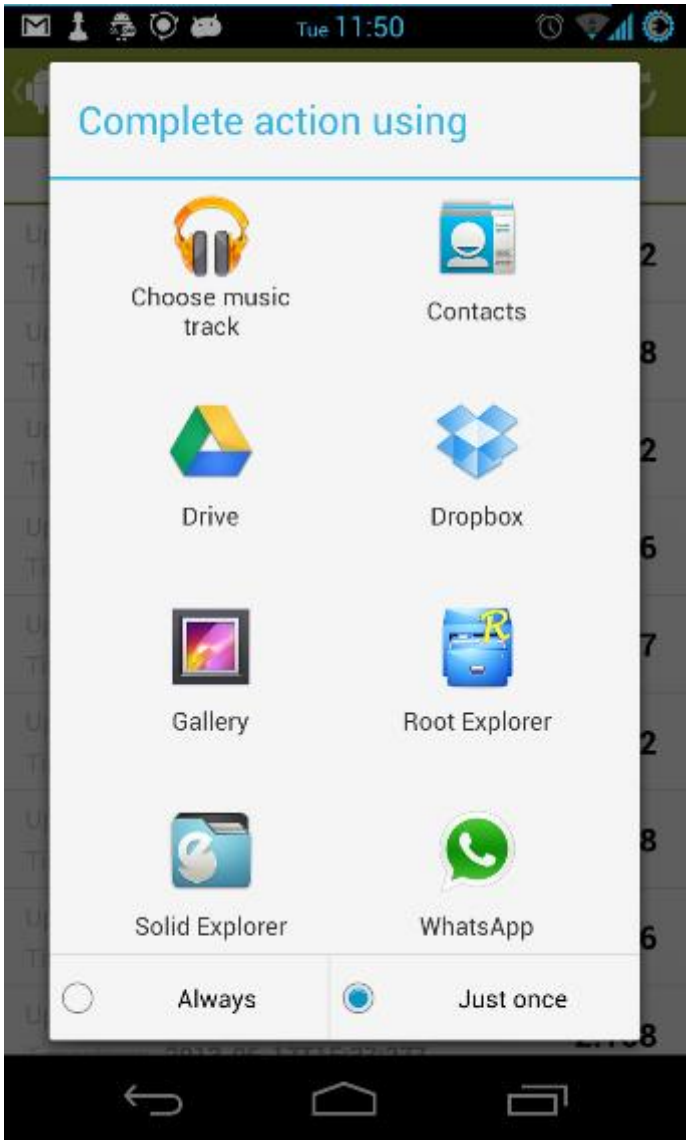


Intents

If multiple activities can handle the intent a dialog chooser is shown.

If an activity declares no intent filter, it can only be started using an explicit intent!

Security: services can only be started explicitly.



Creating an intent

Elements defining an intent:

- Component **name** (explicit vs implicit)
- **Action** – common or custom actions.

  ACTION_VIEW: used with *startActivity*, to open special data types (photo, location, pdf).

  ACTION_SEND: used with *startActivity*, to share data (text, photo).
- **Data** – specifies through an URI the data upon which the action is to be made, and their type (MIME)

Creating an intent

Elements defining an intent:

- **Extras** – key-values pairs carrying additional information

  putExtra (key, value-type), or *putExtras* (Bundle)

  E.g. add EXTRA_EMAIL, EXTRA_SUBJECT for an email intent

- **Category** – type of components that should handle the intent

  addCategory: CATEGORY_BROWSABLE, CATEGORY_LAUNCHER

Favor *Parcelable* instead of *Serializable* in an intent (x10 times faster)

Creating an intent

Explicit intent example:

```
// Simply start another activity
Intent intent = new Intent(this, SecondActivity.class);
Intent.putExtra("key1", "5");
startActivity(intent);


// Executed from an activity, 'this' is the context
// and serverUrl is a path to a database
Intent retrieveIntent = new Intent(this,
RetrieveService.class);
retrieveIntent.setData(Uri.parse(serverUrl));
startService(retrieveIntent);
```

Creating an intent

Implicit intent example:

```
String message = “Mike: 0721123456”;
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, message);
sendIntent.setType("text/plain");
Intent chooser = Intent.createChooser(sendIntent, "Complete
action using");

// Verify that the intent can be handled
if (sendIntent.resolveActivity(getPackageManager()) != null)
{
    startActivity(chooser);
}
```

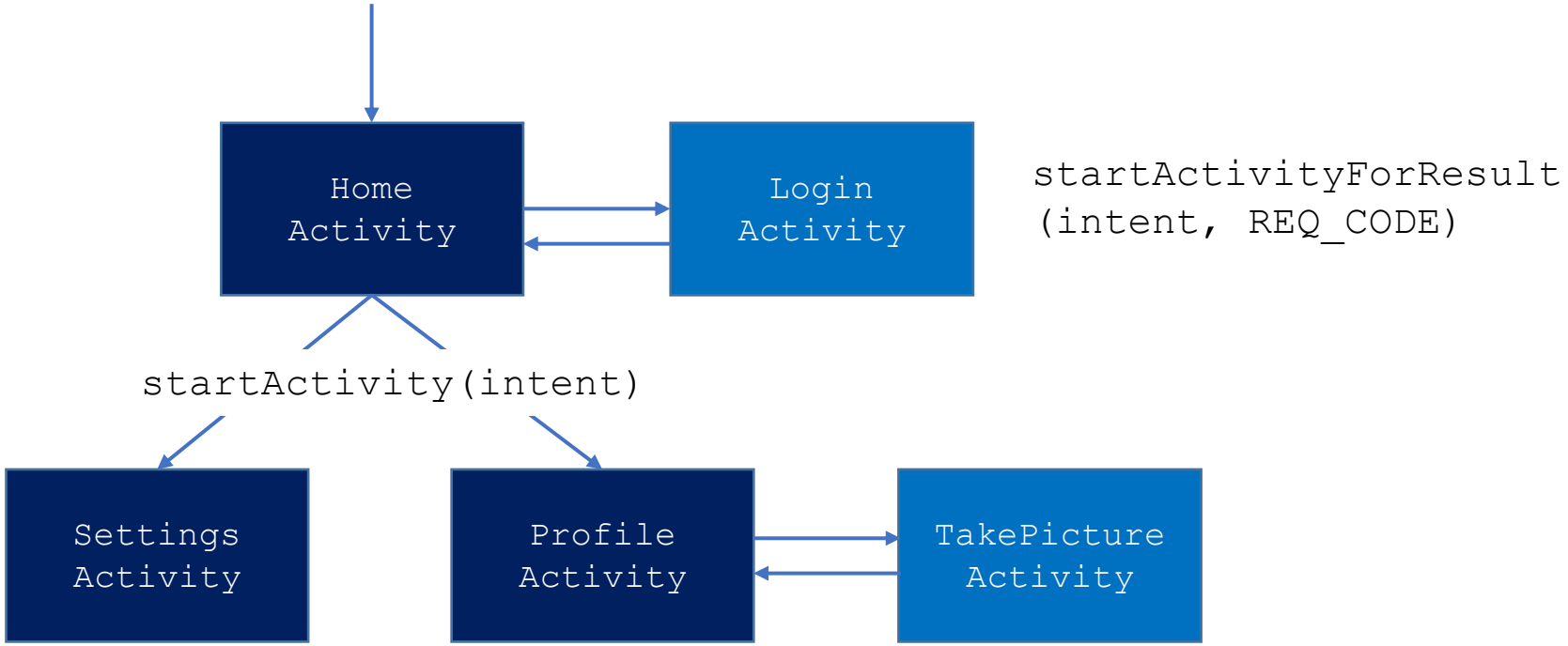
Creating an intent

Accepting the implicit intent:

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name=
              "android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

Starting an activity for a result

The flow of an app may imply forwarding a user to another activity (*dedicated UI*) to retrieve some additional data, e.g., user/password, pinpoint on map, photo, accept permission etc.



Starting an activity for a result

1) A->B for result

```
Intent startIntent = new Intent(this, LoginActivity.class);
startActivityForResult(startIntent, REQUEST_LOGIN);
```

2) B stops and passes back result

```
Intent resultIntent = new Intent(this, HomeActivity.class);
resultIntent.putExtra(KEY_DATA, ".. data ..");
setResult(RESULT_OK, resultIntent);
finish();
```

3) A retrieves result

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_LOGIN)
        if (resultCode == RESULT_OK) {
            String someData = data.getStringExtra(KEY_DATA);
            // ...
        } else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "Action was cancelled by user", Toast.LENGTH_SHORT).show();
        } else {
            // ?
        }
}
```

Pending Intents

An intent wrapper which acts like an authorization token.

You pass a pending intent to a foreign application (e.g. NotificationManager, AlarmManager, Home Screen AppWidgetManager), which allows that application to use your application's permissions to execute a predefined piece of code.

Motivation? Say we create the intent:

```
Intent bIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
```

And send it to another app not having the permission:

```
android.permission.BLUETOOTH_ADMIN
```

Pending Intents

The target application (which uses PendingIntent) does not need to have the Bluetooth permission, but the source application which creates PendingIntent must have the required permission.

Pending Intents example

```
Intent myIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

PendingIntent pIntent = PendingIntent.getActivity(this, 0,
myIntent,PendingIntent.FLAG_ONE_SHOT); //FLAG_ONE_SHOT - pIntent can be
used only once

IParcel ip = new IParcel(pIntent);

Intent sndApp = new Intent();

sndApp.setComponent(new ComponentName("com.myapp", "com.myapp.SecondApp"));

sndApp.putExtra("obj",ip);

startActivity(sndApp);
```

Intents - Summary

- Entities through which components of an application are launched.
- Activities, Services or Broadcast Receivers are launched using Intents.
- Can be seen as the links between applications and their components.
- Data structures which encapsulate the abstract description of an action.
- A communication component.
- Several activities can be registered for a certain operation.
- Offers a decoupling between activities.

Services

Tasks which run in the background without a UI.

- Similar to activities, but they may run indefinitely (e.g. music player, file download, client-server task).
- Not to be confused with Linux/Windows OS, server or daemon services.
- Simpler lifecycle than activities: started or stopped.
- Lifecycle controlled mostly by developer (careful!).
- Services do not automatically run on a different thread, they remain on the UI thread (careful!).

Service examples

- Network communication – downloading pics into gallery from cloud storage (e.g., WhatsApp).
- File I/O – some data synch/archiving (e.g., Google Drive).
- Sound playback – active music player controllable through an ongoing notification in the status bar (e.g., Spotify).
- Servers – any type of data synch, weather, games etc.

Types of services

Three use cases:

1. **Foreground** –performs some operation noticeable to the user.

   They must display a notification (status bar) – e.g. play audio track.

   They continue running even when the user isn't interacting with the app.

2. **Background** – performs an operation that isn't directly noticed by the user.

   E.g. Data server synchronization, storage optimization.

   In API>26 use a scheduled job, if app not in foreground.

3. **Bound** – offers a client-server interface which allows interactions with the service, send requests, receive results.

   Runs only as long as another application component is bound to it.

Service basics

→extend *Service* or *IntentService*

→override (some) callback methods to handle lifecycle.

- onStartCommand – (=onCreate) service is started through *startService* and runs indefinitely. You must stop the service if overwriting (*stopSelf*, *stopService*)
- onBind – offers binding for RPC (remote procedure call) by returning an *IBinder*, or *null* in case of no binding allowed.
- onCreate – one-time setup when the service is initially created.
- onDestroy – clean up any resources such as threads, registered listeners, or receivers.

Declaring a service

→in the app manifest

→use the <service> tag

```
<manifest ... >
  <application ... >
        <service
             android:name=".services.SomeService"
             android:enabled="false"
             android:exported="false">
        </service>
    </application>
</manifest>
```

- Name → mandatory and shouldn't change to avoid breaking the code.
- Enabled → whether the service may be started by the system.
- Exported → makes service visible to other apps (security hazard?).
- Services must be called explicitly to avoid running other code (security).

Services by type of lifecycle

From the lifecycle perspective, Android can have two forms of services and they follow two paths, that are:

- Started service

  → Becomes started only when an application component calls startService(). Performs a single operation and doesn't return any result to the caller.

- Bound service

  Is bound only if an application component binds to it using bindService(). Gives a client-server relation that lets the components interact with the service.

Creating a started service

→started by a component with *startService*

→runs until *stopSelf* or *stopService*

→may receive an intent in *onStartCommand*

- May extend **Service** class – manage everything: threads (multi-threading possible), message queue handling, create & destroy.
- Or **IntentService** – uses a worker thread and a queue for each request. No parallel threaded execution, but much simpler coding and management.

Extending an IntentService

```
public class HelloIntentService extends IntentService {
public HelloIntentService() {
      super("HelloIntentService");
  }
   /* The IntentService calls this method from the default worker thread with the
intent that started the service. When this method returns, IntentService stops
the service, as appropriate.*/
@Override
  protected void onHandleIntent(Intent intent) {
/* Normally we would do some work here, like download a file. For our sample, we
just sleep for 5 sec.*/
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          // Restore interrupt status.
          Thread.currentThread().interrupt();
      }
  }
}
```

Extending a Service class

```
@Override
 public int onStartCommand(Intent intent, int flags, int startId) {
     Toast.makeText(this, "service starting", Toast.LENGTH_SHORT).show();

     /* For each start request, send a message to start a job and deliver the
start ID so we know which request we're stopping when we finish the job */
     Message msg = mServiceHandler.obtainMessage(); // init in onCreate
     msg.arg1 = startId;
     mServiceHandler.sendMessage(msg); // main thread queue!

     // If we get killed, after returning from here, restart
     return START_STICKY;
 }
```

More about Handlers

Service stickiness

The onStartCommand (*oSC)* can return one of three constants and determines what happens if a service is killed by the system:

- START_NON_STICKY – do not recreate the service unless there are pending intents to deliver.
- START_STICKY – recreate service, call *oSC*, but do not redeliver last intent. Delivers *null* intent or next in queue.
- START_REDELIVER_INTENT – recreate service, call *oSC* with last intent. Then delivers next in queue…

Foreground service

```
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 0, notificationIntent, 0);

/* extra code from next slide goes here */

Notification notification =
          new Notification.Builder(this, CHANNEL_ID)
    .setContentTitle(getText(R.string.notification_title))
    .setContentText(getText(R.string.notification_message))
    .setSmallIcon(R.drawable.icon)
    .setContentIntent(pendingIntent)
    .setTicker(getText(R.string.ticker_text))
    .build();

startForeground(ONGOING_NOTIFICATION_ID, notification);
```

Foreground service (Android 8 / API 26)

- Must explicitly create a (custom) notification channel with CHANNEL_ID if API ≥ 26.
- Example of checking the current build version on the underlying device at runtime:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

    CharSequence name = "My special channel";
    String description = "My custom channel description";
    int importance = NotificationManager.IMPORTANCE_DEFAULT;

    NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name, importance);
    channel.setDescription(description);

    // Register the channel with the system; you can't change the importance
    // or other notification behaviors after this
    NotificationManager notificationManager = getSystemService(NotificationManager.class);
    notificationManager.createNotificationChannel(channel);
}
```

Service starting and stopping

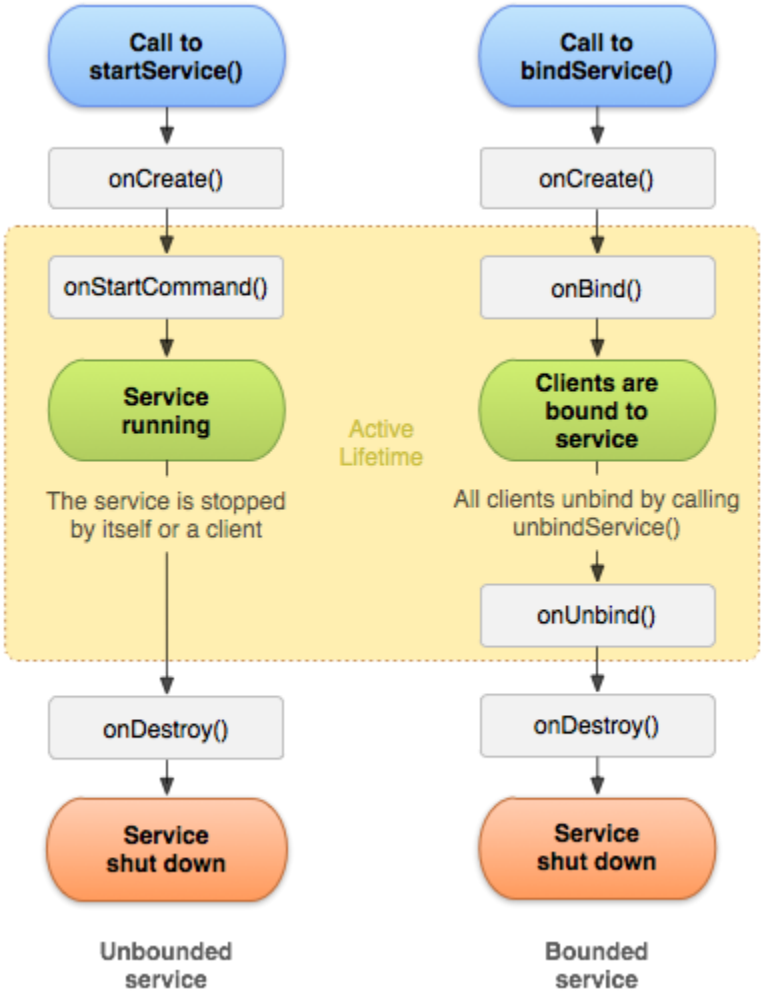
```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

startService → (calls *onCreate* once →) onStartCommand

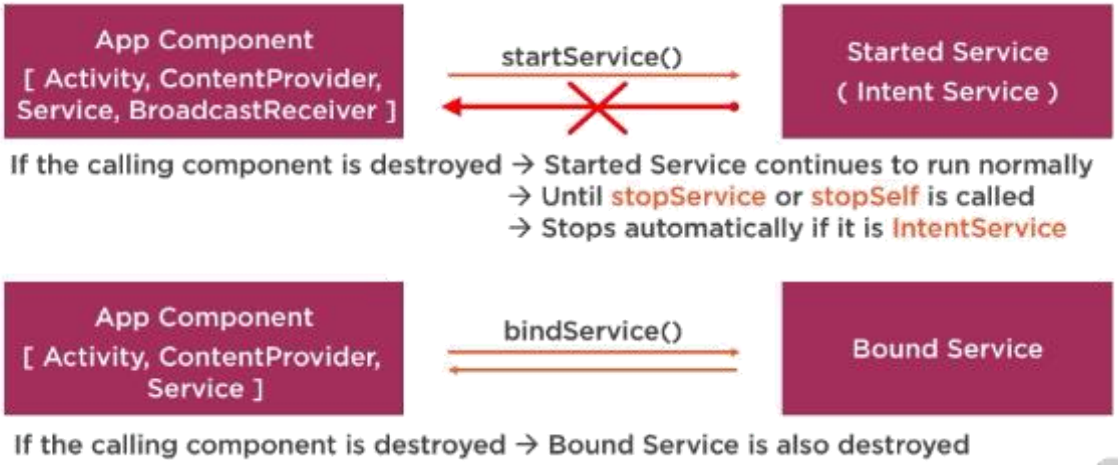
Multiple requests to startService == queuing of jobs

One call to *stopService* stops it entirely.

Services lifecycle



Started versus bound services



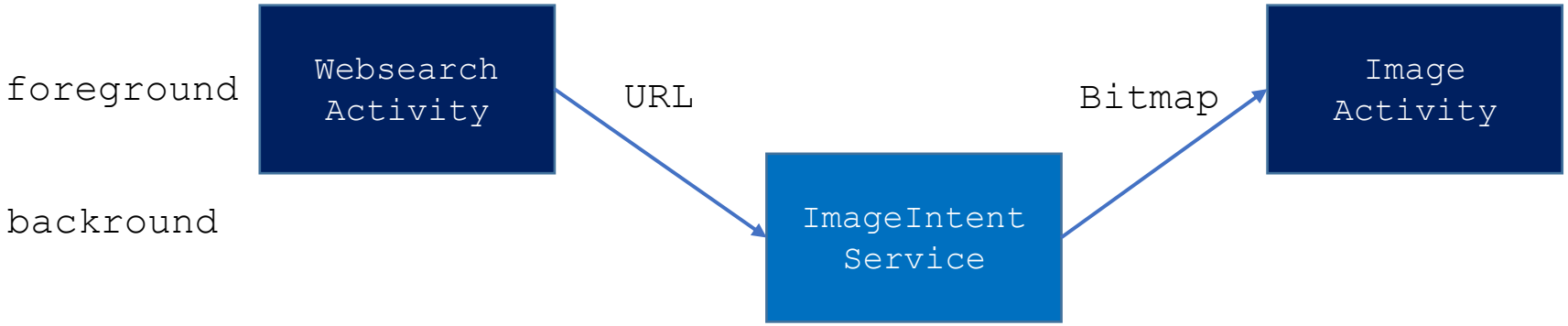
Started Service vs Bound Service

Started Service / IntentService	Bound Service
Just to accomplish task [May be Long Task]	For long-standing connection
Invoked by startService()	Invoked by bindService()
onBind() returns null	onBind() returns IBinder
Continue to run even if the calling component is destroyed	If the calling component is destroyed then Bound Services too gets destroyed

Code example of background service (1/4)

A demo of how we can download an image by URL in the background (service) and display it in a dedicated activity.

- WebsearchActivity – lets user select an image (retrieves URL)
- ImageActivity – displays image to user
- ImageIntentService – background service which takes URL and downloads image to `Bitmap`, then starts ImageActivity.



Code example of background service (2/4)

WebsearchActivity starts service:

```
// say the user has saved the image URL in the clipboard
// simple way to extract clipboard data (string)
ClipboardManager clipboard = (ClipboardManager)
          getSystemService(Context.CLIPBOARD_SERVICE);
ClipData abc = clipboard.getPrimaryClip();
ClipData.Item item = abc.getItemAt(0);
String url = item.getText().toString();


// put URL as extra and start service
Intent intent = new Intent(this, ImageIntentService.class);
intent.putExtra(EXTRA_URL, url);
startService(intent);
```

Code example of background service (3/4)

```
public class ImageIntentService extends IntentService {
    public ImageIntentService() { super("ImageIntentService"); }
    protected void onHandleIntent(Intent intent) {
        if (intent != null)                 {
            final String param = intent.getStringExtra(WebsearchActivity.EXTRA_URL);
            handleDownloadAction(param);  }
    }
private void handleDownloadAction(String url) {
        // starts task on separate thread automatically
        try {
            String longURL = URLTools.getLongUrl(url);
            Bitmap bmp = null;
            try {
                InputStream in = new URL(longURL).openStream();
                bmp = BitmapFactory.decodeStream(in);
            } catch (Exception e) {
                Log.e("Error Message", e.getMessage());
                e.printStackTrace(); }
            ((MyApplication) getApplicationContext()).setBitmap(bmp);
            // start second activity to show result
            Intent intent = new Intent(getApplicationContext(), ImageActivity.class);
            startActivity(intent);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();  }  } }
```

```
intent.putExtra("bitmapKey", bmp);
```

Code example of background service (4/4)

ImageActivity retrieves bitmap:

```
Intent intent = getIntent();
Bitmap bitmap = (Bitmap)
intent.getParcelableExtra("bitmapKey");
```

```
public class ImageActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image);

        // check if application has expected payload
        MyApplication myApplication = (MyApplication) getApplicationContext();
        if (myApplication.getBitmap() == null) {
            Toast.makeText(this, "Error transmitting URL.", Toast.LENGTH_SHORT).show();
            finish();
        } else {
            ImageView imageView = (ImageView) findViewById(R.id.imageView);
            imageView.setImageBitmap(myApplication.getBitmap());
        }
    }
}
```

Content providers

Interfaces for data exchange between applications.

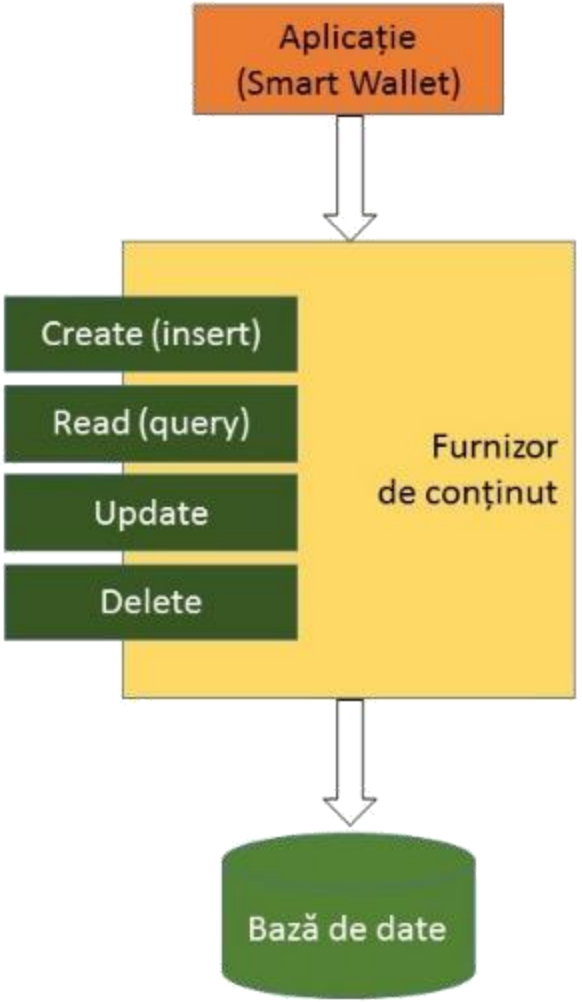
- Android runs every app in a sandbox → all data is isolated from the data of other apps.
- Small amounts of data may be shared through intents, but content providers are more scalable.
- CRUD interface.

Content providers

Contacts provider – exposes all contacts data

Media store – handles all photos and music on the phone

Phone manufacturers use providers to create custom apps on top of Android standard: Samsung, LG, Sony, Xiaomi, One+ have their own look-and-feel.



Content providers

- Allow applications to access data published by other applications.
- Allow an application to publish and share it's own data.
- Offer a mechanism for data transfer between applications.
- Applications must have the necessary permissions set in order to communicate.
- Extend the class *ContentProvider* for data publishing.
- Extend the class *ContentResolver* for data access.
- Data is stored in files or in DB SQLite.
- Examples: accessing or publishing contacts, history.

Broadcast receivers

A publish/subscribe system similar to the observer pattern.

- Receiver = "code" in latent state which runs after an event to which it is subscribed has happened.
- System events are broadcasting all the time: call, text, battery, system-on/off/lock and can trigger any number of receivers.
- May send custom events within app or to other apps.
- No UI, no memory footprint.
- When activated, they will execute code and start an activity or service.

System broadcasts

The Android system sends out broadcasts packaged in an intent with an <action> tag.

Example events:

- ACTION_TIME_TICK – a minute has passed
- ACTION_BOOT_COMPLETED – system has booted & is unlocked
- ACTION_PACKAGE_ADDED – a new app was installed
- ACTION_POWER_CONNECTED – external power supply
- ACTION_SHUTDOWN – the system is shutting down
- ACTION_BATTERY_CHANGED – battery level changes

Receiving broadcasts

By declaring receivers in the manifest or in context.

1) Manifest-declared broadcast receiver (1/2)

```
<receiver
    android:name=".receivers.StartupReceiver"
    android:enabled="true"
    android:exported="false">
    <intent-filter>
        <action
android:name="android.intent.action.BOOT_COMPLETED"/>
        <action
android:name="android.intent.action.INPUT_METHOD_CHANGED"/
>
    </intent-filter>
</receiver>
```

Receiving broadcasts

1) Manifest-declared broadcast receiver (2/2)

```
public class StartupReceiver extends BroadcastReceiver {

    private static final String TAG = "StartupReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        String msg = "Action: " + intent.getAction() + "\n";
        msg += "URI: " + intent.toUri(Intent.URI_INTENT_SCHEME);
        Log.d(TAG, msg);
        Toast.makeText(context, msg, Toast.LENGTH_SHORT).show();
    }
}
```

Receiving broadcasts

By declaring receivers in the manifest or in context.

2) Context-declared broadcast receiver (1/1)

```
BroadcastReceiver receiver = new PaymentReceiver();
IntentFilter filter = new IntentFilter();
filter.addAction(Intent.BATTERY_LOW);
filter.addAction(Intent.ACTION_SCREEN_ON);
this.registerReceiver(receiver, filter); // global receiver
LocalBroadcastManager.getInstance(this).registerReceiver(receiver
, filter); // local receiver
```

Runtime impact

During *onReceive*, the running process has a foreground priority (highest).

- If the duration of *onReceive* is >16 ms (~1 frame @60FPS) then the UX might get affected.
- If a new thread is created to run asynchronously after *onRecieve*, it might get killed (runs with background priority).

Solutions:

1. Call goAsync to signal the system you are running the job in another thread.
2. Create a JobService and run it with a JobScheduler.

Runtime impact

```
public class StartupReceiver extends BroadcastReceiver {
@Override
    public void onReceive(final Context context, final Intent intent) {
        final PendingResult pendingResult = goAsync();

        AsyncTask<String, Integer, String> asyncTask = new
        AsyncTask<String, Integer, String>() {
            @Override
            protected String doInBackground(String... params) {
                String msg =  "Action: " + intent.getAction() + "\n";
                msg += "URI: " + intent.toUri(Intent.URI_INTENT_SCHEME);
                Log.d(TAG, msg);
                // must call finish() to recycle receiver
                pendingResult.finish();
                return msg;
            }
        };
        asyncTask.execute();
    }
}
```

With goAsync a broadcast may run for 10sec until ANR. This way you have 10 sec between goAsync and pendingResult.finish() (signals the system to free up the receiver)

Sending a broadcast

Three methods:

- *sendOrderedBroadcast* – chain of receivers (by <priority>) that can pass data to the next, or cancel the broadcast entirely.
- *sendBroadcast* – sends broadcast to all receivers in random order, independently; is more efficient.
- *LocalBroadcastManager.sendBroadcast* - visible only to receivers inside app.

```
Intent bIntent = new Intent();
bIntent.setAction("com.upt.broadcast.MY_NOTIFICATION");
bIntent.putExtra("name", "User123");
bIntent.putExtra("sum", 12345);
bIntent.setPackage("com.upt");
sendBroadcast(bIntent);
```

Application context

All application components reside in the same application context

Context: the virtual medium (process) in which all app components are created and share information.

- Created when the first component of the app is launched.
- Lives as long as an app is alive.
- Independent from activity lifecycle.
- Can be obtained with *Context.getApplicationContext()* or *Activity.getApplication()*.
- All activities and services extend Context (this).

Application context

Required all the time in method calls for:

- Loading (xml) resources. // [context.]findViewById(…)
- Launching a new activity. // startActivity(this,…)
- Obtaining a system service. // getPackageManager(this)
- Retrieving an internal app file path. // [context.]getFilesDir()
- Creating views // new TextView(this)
- Writing to shared preferences // [context.]getSharedPreferences

Application context

Not all context instances are equal:

- Application – a singleton running in the app process.
- Activity/service – extends ContextWrapper (providing the same API, base context) and creates a *ContextImpl* for more complex operations.
- Broadcast receiver.
- Content provider.

Application context pitfalls

What is the nature of the **context**?

```
public class CustomProvider {
    private static CustomProvider sInstance;

    public static CustomProvider getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new CustomProvider(context);
        }

        return sInstance;
    }

    private Context mContext;

    private CustomProvider(Context context) {
        mContext = context;
    }
}
```

Application context pitfalls

What is the nature of the **context**?

- If it is an activity or service, we keep them in memory.
- *mContext* is a singleton handled by a single reference in the wrapping class. As such, *mContext* and all its referred objects will never be garbage collected.
- This means keeping all views, lists associated with the activity in memory, creating a memory leak.

Application context pitfalls

Modified singleton:

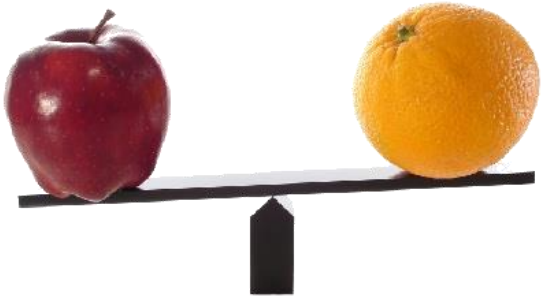
```
public class CustomProvider {
    private static CustomProvider sInstance;

    public static CustomProvider getInstance(Context context) {
        if (sInstance == null) {
            sInstance = new CustomProvider(context.getApplicationContext());
        }

        return sInstance;
    }

    private Context mContext;

    private CustomProvider(Context context) {
        mContext = context;
    }
}
```

Application context pitfalls

The nature of the context does not matter anymore, because the Application is a singleton itself.

Why not always refer to the application context?

Not all contexts are created equal!



Application context differences

	Application	Activity	Service	CProvider	BReceiver
Showing a dialog		X			
Launch an activity		X			
Loading a layout	~	X	~	~	~
Start a service	X	X	X	X	X
Binding to a service	X	X	X	X	
Send a broadcast	X	X	X	X	X
Register a receiver	X	X	X	X	
Loading resources	X	X	X	X	X