CS 193A

Services

Services

- **service**: A background task used by an app.
  - Example: Google Play Music plays the music using a service.
  - Example: Web browser runs a downloader service to retrieve a file.
  - Useful for long-running tasks, and/or providing functionality that can be used by other applications.

- Android has two kinds of services:
  - **standard services**: For longer jobs; remains running after app closes.
  - **intent services**: For shorter jobs; app launches them via intents.

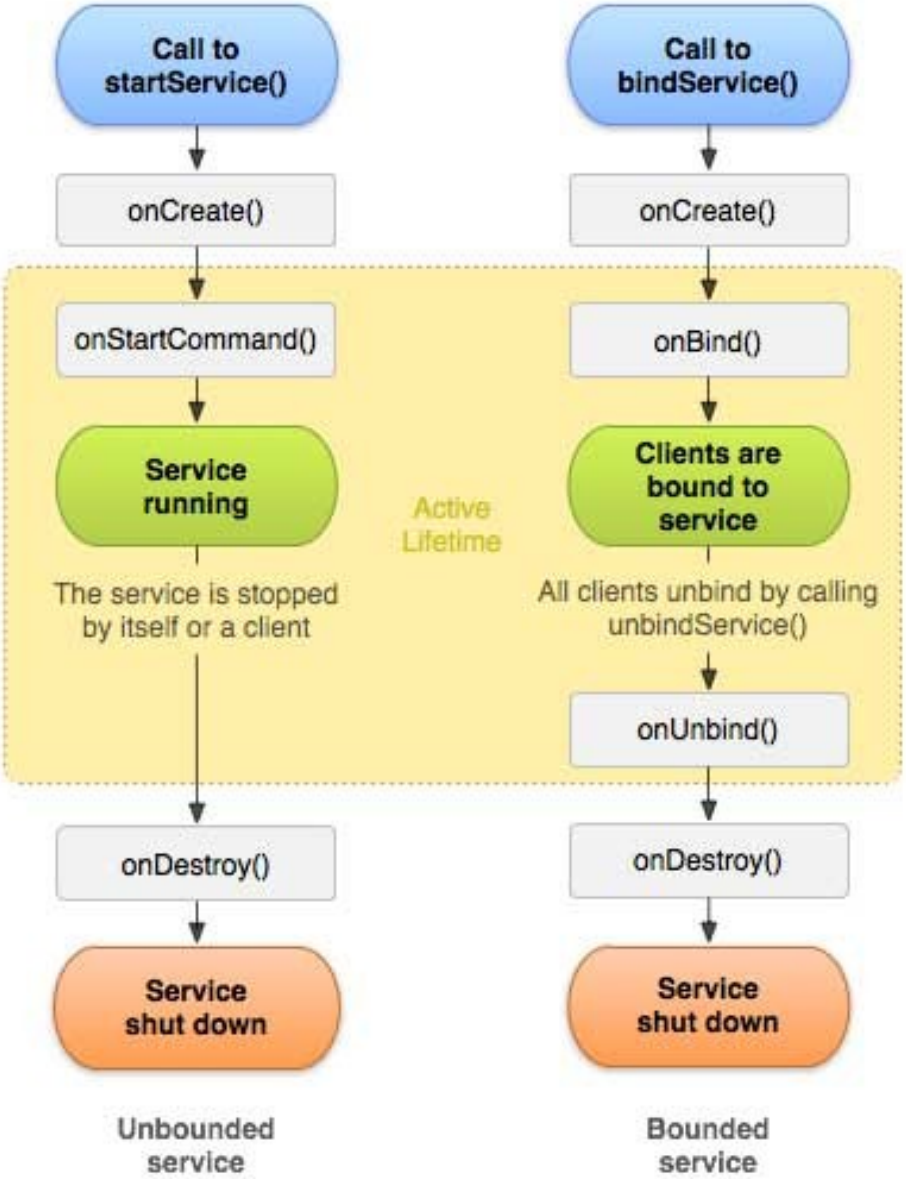
- When/if the service is done doing work, it can **broadcast** this information to any **receivers** who are listening.

The service lifecycle

- A service is started by an app's activity using an intent.

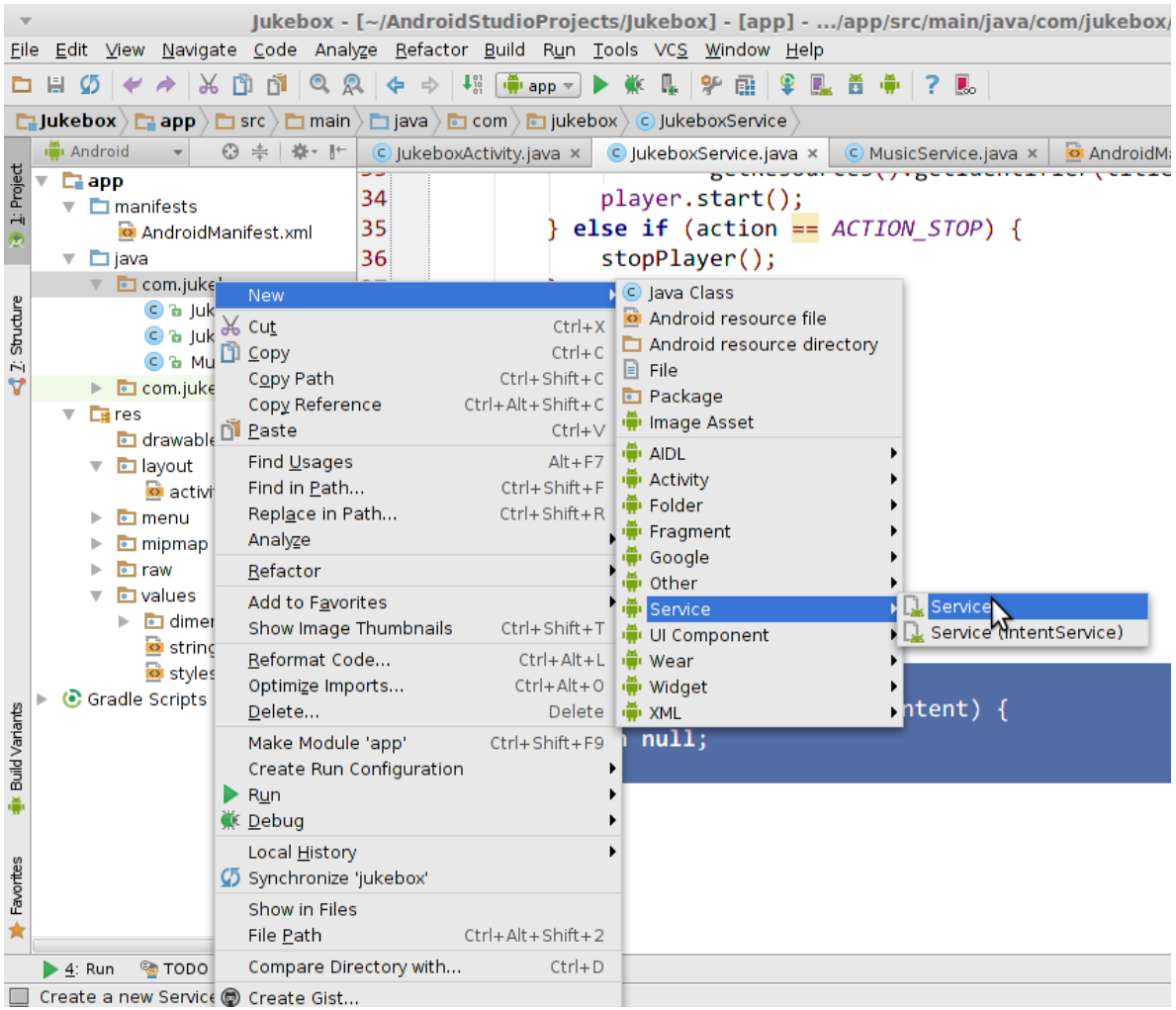
- Service operation modes:
  - **start**: The service keeps running until it is manually stopped.
    - *we'll use this one*
  - **bind**: The service keeps running until no "bound" apps are left.

- Services have similar methods to activities for lifecycle events.
  - `onCreate`, `onDestroy`



Adding a service in Android Studio

- right-click your project's Java package
- click New → Service → **Service**



Service class template

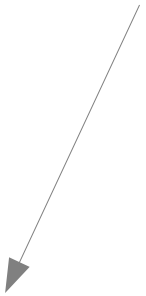
```
public class ServiceClassName extends Service {
    /* this method handles a single incoming request */
    @Override
    public int onStartCommand(Intent intent, int flags, int id) {
        // unpack any parameters that were passed to us
        String value1 = intent.getStringExtra("key1");
        String value2 = intent.getStringExtra("key2");

        // do the work that the service needs to do ...

        return START_STICKY;   // stay running
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;   // disable binding
    }
}
```

AndroidManifest.xml changes

- To allow your app to use the service, add the following to your app's `AndroidManifest.xml` configuration:

  (Android Studio does this for you if you use the New Service option)

  - the `exported` attribute signifies whether other apps are also allowed to use the service (true=yes, false=no)
  - note that you must write a dot (`.`) before the class name below!



```
<application ...>

    <service
        android:name=".ServiceClassName"
        android:enabled="true"
        android:exported="false" />
```

Starting a service

- In your `Activity` class:

```
Intent intent = new Intent(this, ServiceClassName.class);
intent.putExtra("key1", "value1");
intent.putExtra("key2", "value2");
startService(intent);   // not startActivity!
```

- or if the same code is launched from a fragment:

```
Intent intent = new Intent(getActivity(),
                           ServiceClassName.class);
...
```

Intent actions

- Often a service has several "**actions**" or commands it can perform.
  - Example: A music player service can play, stop, pause, ...
  - Example: A chat service can send, receive, ...

- Android implements this with set/getAction methods in `Intent`.
  - In your `Activity` class:

```
Intent intent = new Intent(this, ServiceClassName.class);
intent.setAction("some constant string");
intent.putExtra("key1", "value1");
startService(intent);
```

  - In your `Service` class:

```
String action = intent.getAction();
if (action == "some constant string") { ... } else { ... }
```

Broadcasting a result

- When a service has completed a task, it can notify the app by "sending a broadcast" which the app can listen for:
  - As before, set an **action** in the intent to distinguish different kinds of results.

```
public class ServiceClassName extends Service {
    @Override
    public int onStartCommand(Intent tent, int flags, int id) {
        // do the work that the service needs to do ...
        ...
        // broadcast that the work is done
        Intent done = new Intent();
        done.setAction("action");
        done.putExtra("key1", value1); ...
        sendBroadcast(done);

        return START_STICKY;   // stay running
    }
```

Receiving a broadcast

- Your activity can hear broadcasts using a BroadcastReceiver.
  - Extend BroadcastReceiver with the code to handle the message.
  - Any extra parameters in the message come from the service's intent.

```
public class ActivityClassName extends Activity {
    ...

    private class ReceiverClassName extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            // handle the received broadcast message
            ...
        }
    }
```

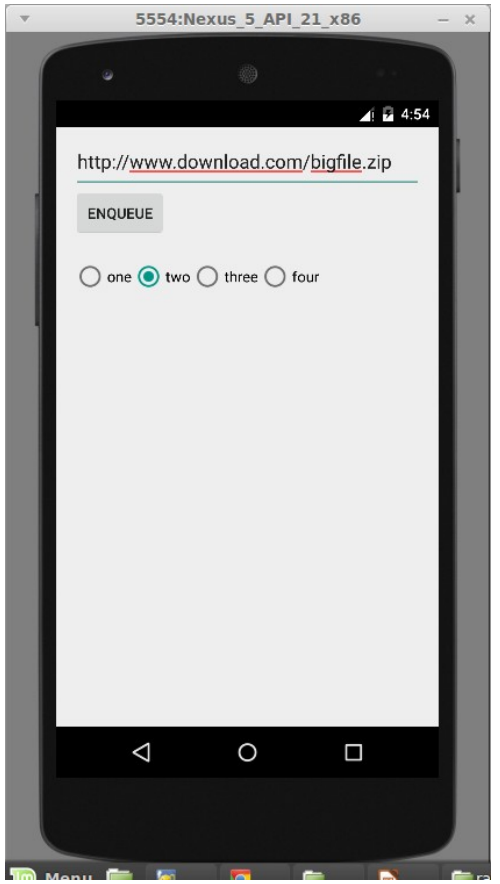
Listening for broadcasts

- Set up your activity to be notified when certain broadcast actions occur.
  - You must pass an **intent filter** specifying the action(s) of interest.

```
public class ActivityClassName extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        IntentFilter filter = new IntentFilter();
        filter.addAction("action");
        registerReceiver(new ReceiverClassName(), filter);
    }
```

Services and threading

- By default, a service lives in the same process and thread as the app that created it.
  - This is not ideal for long-running tasks.
  - If the service is busy, the app's UI will freeze up.
  - Example: If the Downloader app at right tries to download a large/slow file, the radio buttons and other UI elements will not respond during the download.



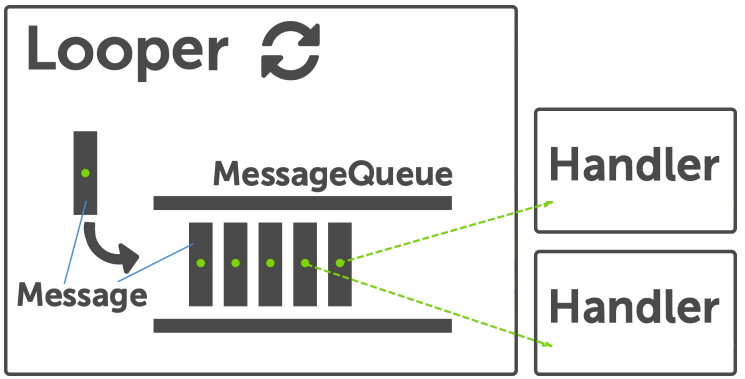
- To make the service and app more independent and responsive, the service should handle tasks in **threads**.

Service with thread

```
public class ServiceClassName extends Service {
    /* this method handles a single incoming request */
    @Override
    public int onStartCommand(Intent intent,
                              int flags, int id) {
        // unpack any parameters that were passed to us
        String value1 = intent.getStringExtra("key1");

        Thread thread = new Thread(new Runnable() {
            public void run() {
                // do the work that the service needs to do
            }
        });
        thread.start();

        return START_STICKY;   // stay running
    }
```

Android thread helper classes

- **job** (or message) **queue**: Common pattern in Android services.
  - New jobs come in to the service via the app's intents.
  - Jobs are "queued up" in some kind of structure to be processed.
  - The thread(s) process jobs in the order they came in.
  - As jobs finish, results are broadcast back to the app.



- Android provides several classes to help implement multi-threaded job/message queues:
  - Looper, Handler, HandlerThread, AsyncTask, Loader, CursorLoader, ...
  - *advantages*: easier to submit/finish jobs; easier synchronization; able to be canceled; support for thread pooling; better handling of Android lifecycle issues; ...

HandlerThread (link)

- `HandlerThread` : just a thread that has some internal data representing a queue of jobs to perform.
  - `Looper` : Lives inside a handler thread and performs a long-running while loop that waits for jobs and processes them. (link)
  - You can give new jobs to the handler thread to process via its looper.

```
HandlerThread hThread = new HandlerThread("name");
hThread.start();

Looper looper = hThread.getLooper();
...
```

Handler (link)

- `Handler` : Represents a single piece of code to handle one job in the job queue.
  - When you construct a handler, pass the **Looper** of the handler thread in which the job should be executed.
  - Submit a job to the handler by calling its `post` method, passing a `Runnable` object indicating the code to run.

```
Handler handler = new Handler(looper);
handler.post(new Runnable() {
    public void run() {
        // the code to process the job
        ...
    }
});
```