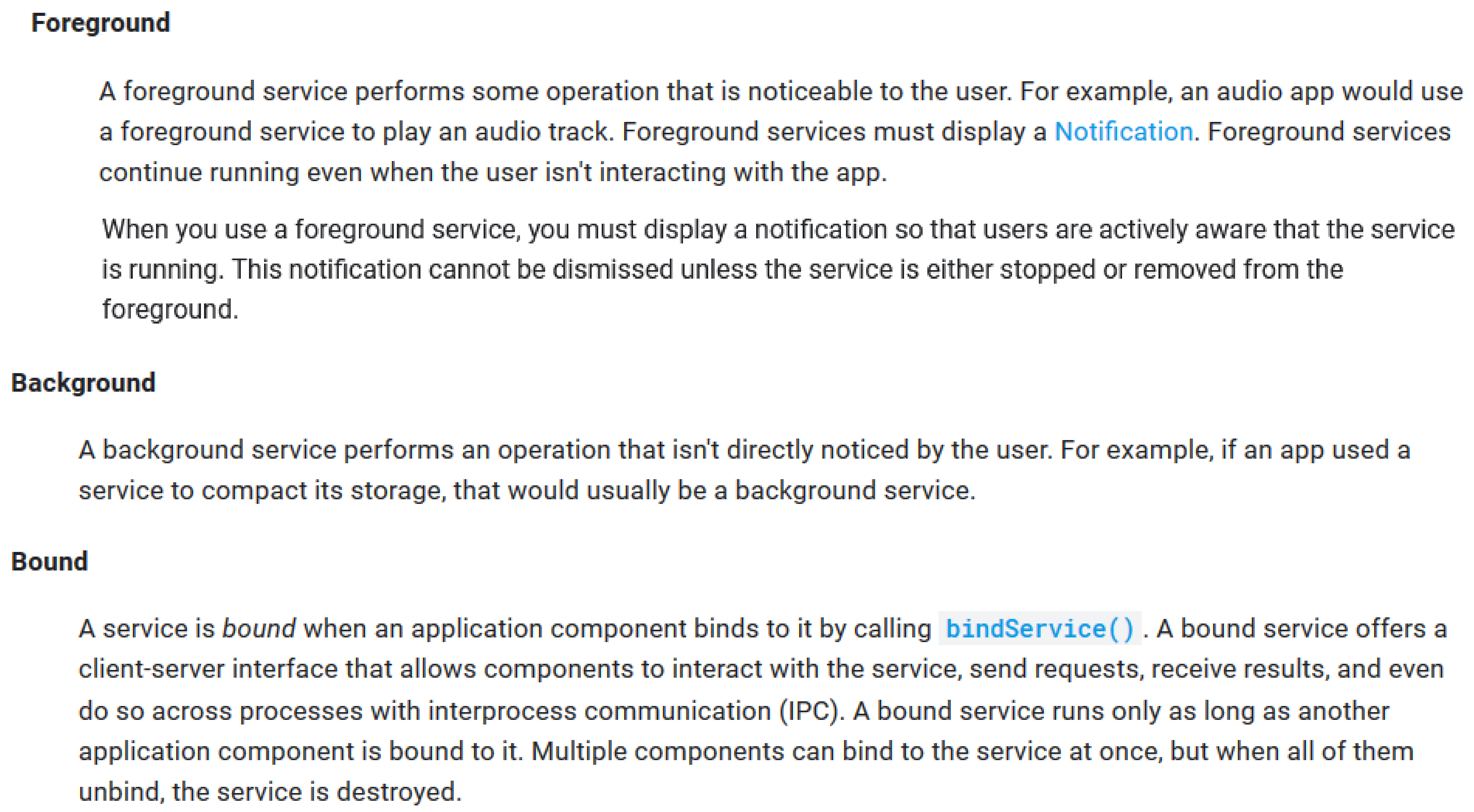
Mobile Computing Course
4th Session

- Services
- Layouts

Services overview

- A Service is an application component that can perform long-running operations in the background.
- It does not provide a user interface.
- Once started, a service might continue running for some time, even after the user switches to another application.
- Additionally, a component can bind to a service to interact with it.
- For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services



Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

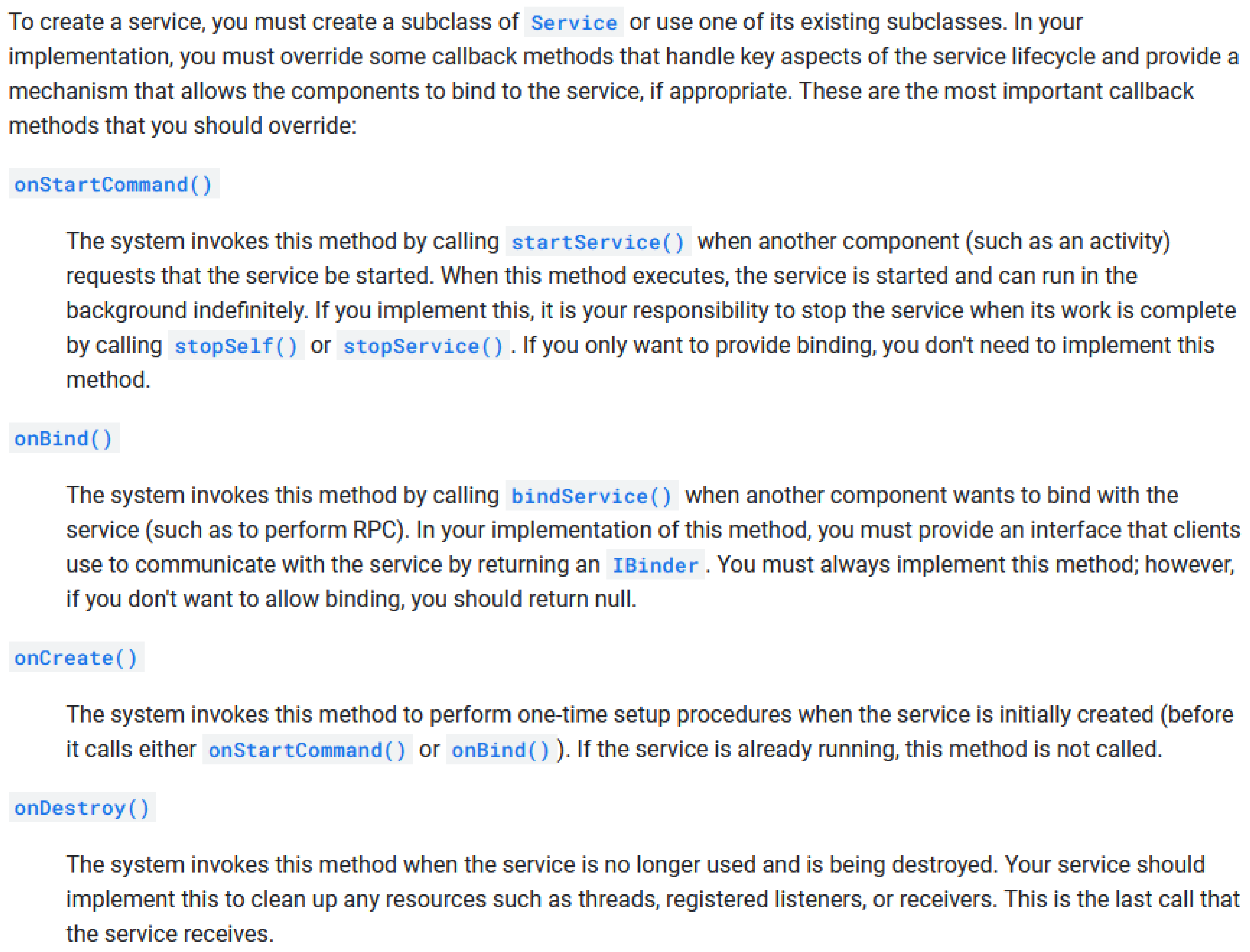
Bound

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, send requests, receive results, and even do so across processes with interprocess communication (IPC). 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.

Choosing between a service and a thread

- A service is simply a component that can run in the background, even when the user is not interacting with your application, so you should create a service only if that is what you need.
- If you must perform work outside of your main thread, but only while the user is interacting with your application, you should instead create a new thread in the context of another application component. For example, if you want to play some music, but only while your activity is running, you might create a thread in onCreate(), start running it in onStart(), and stop it in onStop().
- Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

The basics



To create a service, you must create a subclass of `Service` or use one of its existing subclasses. In your implementation, you must override some callback methods that handle key aspects of the service lifecycle and provide a mechanism that allows the components to bind to the service, if appropriate. These are the most important callback methods that you should override:

`onStartCommand()`

> The system invokes this method by calling `startService()` when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling `stopSelf()` or `stopService()`. If you only want to provide binding, you don't need to implement this method.

`onBind()`

> The system invokes this method by calling `bindService()` when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, 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.

`onCreate()`

> The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either `onStartCommand()` or `onBind()`). If the service is already running, this method is not called.

`onDestroy()`

> The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

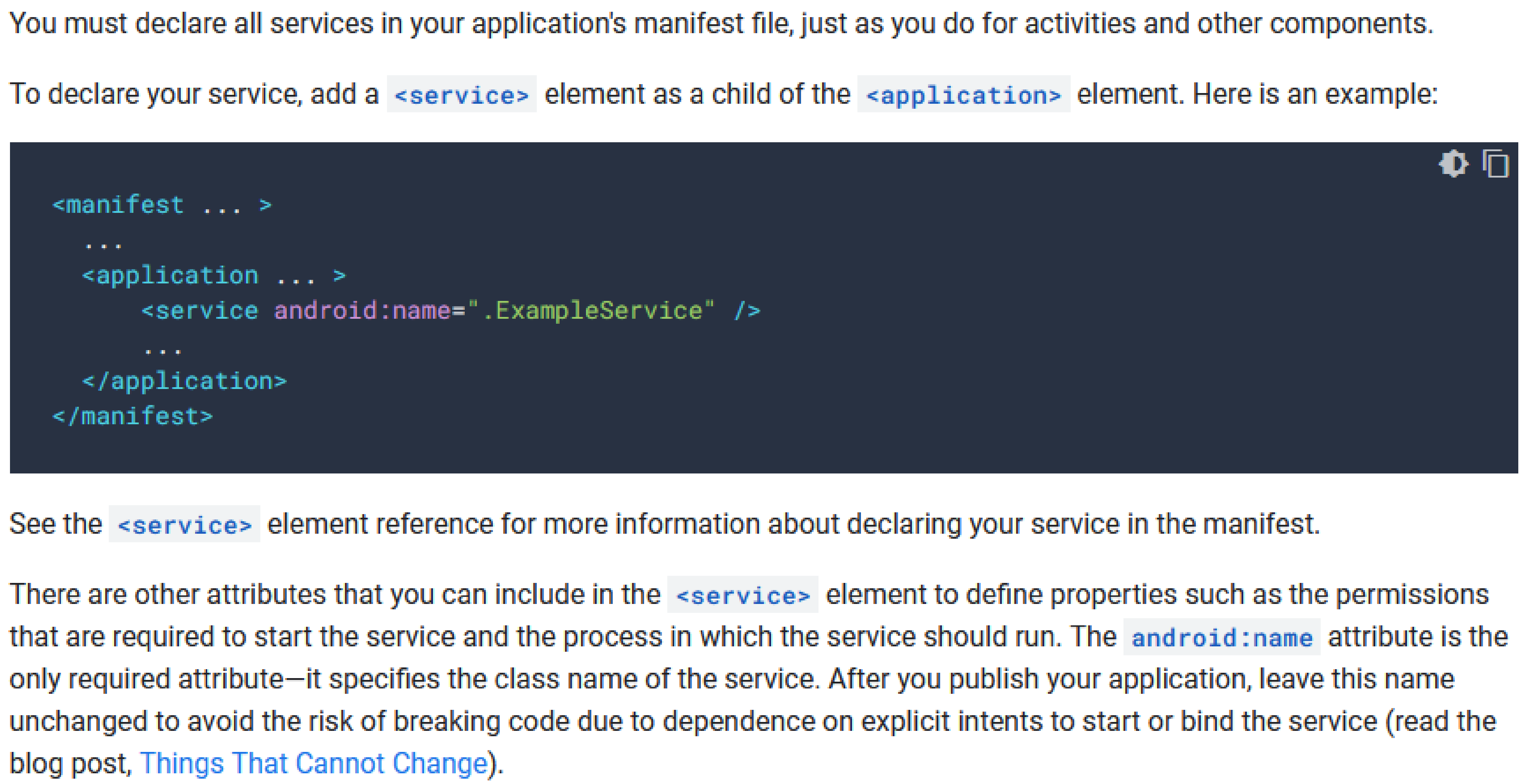
Although this documentation generally discusses started and bound services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple of callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding.

Regardless of whether your service is started, bound, or both, any application component can use the service (even from a separate application) in the same way that any component can use an activity—by starting it with an `Intent`. However, you can declare the service as *private* in the manifest file and block access from other applications. This is discussed more in the section about Declaring the service in the manifest.

If a component starts the service by calling `startService()` (which results in a call to `onStartCommand()`), the service continues to run until it stops itself with `stopSelf()` or another component stops it by calling `stopService()`.

If a component calls `bindService()` to create the service and `onStartCommand()` is *not* called, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys it.

Declaring a service in the manifest



You must declare all services in your application's manifest file, just as you do for activities and other components.

To declare your service, add a `<service>` element as a child of the `<application>` element. Here is an example:

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

See the `<service>` element reference for more information about declaring your service in the manifest.

There are other attributes that you can include in the `<service>` element to define properties such as the permissions that are required to start the service and the process in which the service should run. The `android:name` attribute is the only required attribute—it specifies the class name of the service. After you publish your application, leave this name unchanged to avoid the risk of breaking code due to dependence on explicit intents to start or bind the service (read the blog post, Things That Cannot Change).

Creating a started service

Extending the Service class: You can extend the Service class to handle each incoming intent.

Notice that the `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`

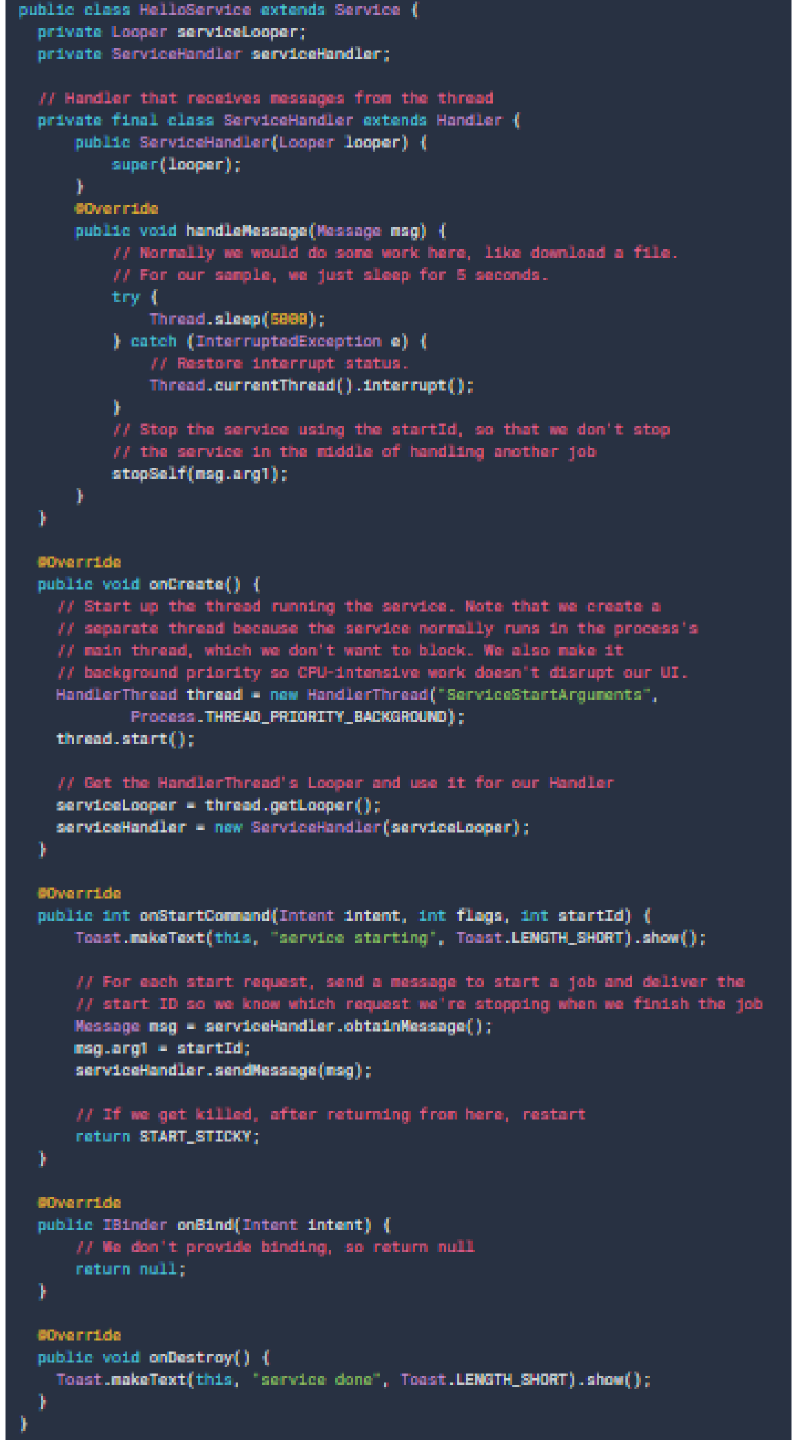
If the system kills the service after `onStartCommand()` returns, *do not* recreate the service unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

`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 unless there are pending intents to start the service. In that case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands but are running indefinitely and waiting for a job.

`START_REDELIVER_INTENT`

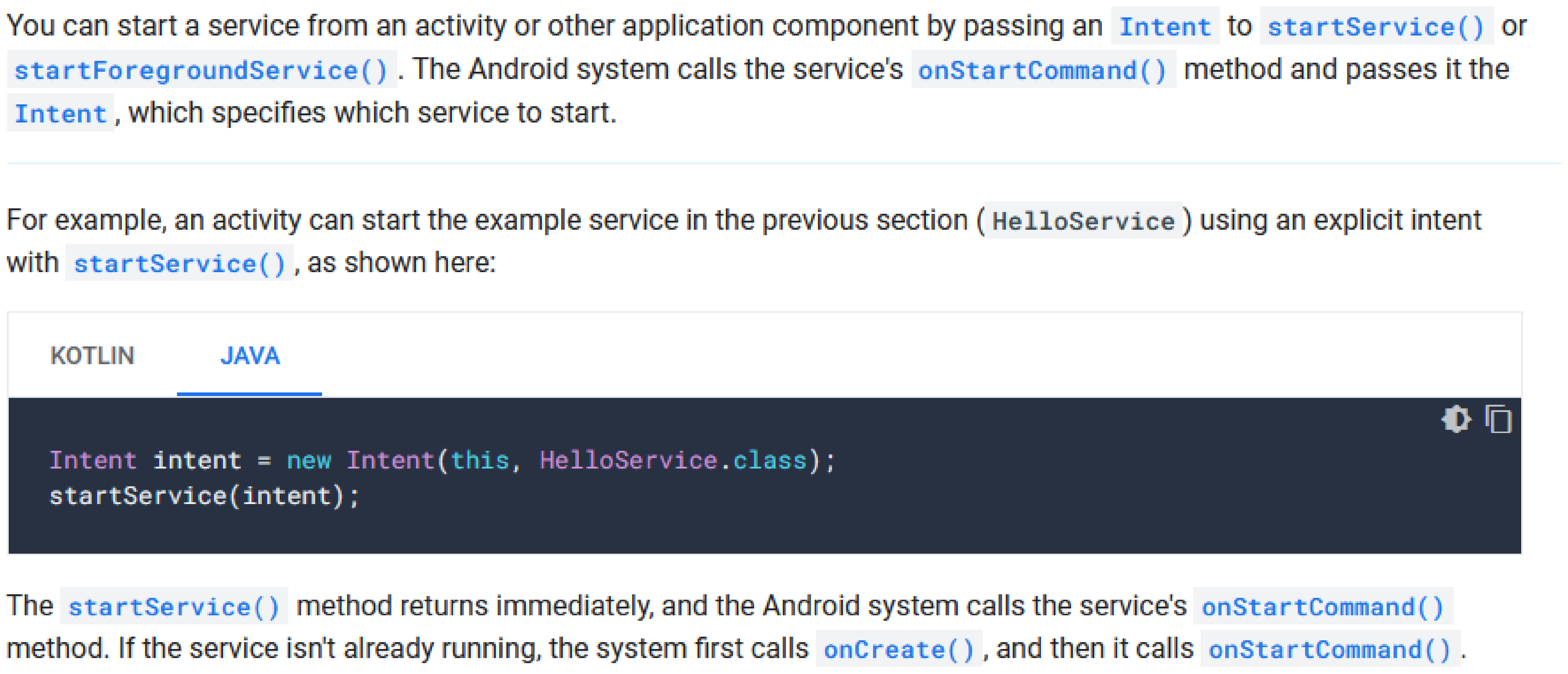
If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()` with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.



```java
public class HelloService extends Service {
  private Looper serviceLooper;
  private ServiceHandler serviceHandler;

  // Handler that receives messages from the thread
  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
          // Normally we would do some work here, like download a file.
          // For our sample, we just sleep for 5 seconds.
          try {
              Thread.sleep(5000);
          } catch (InterruptedException e) {
              // Restore interrupt status.
              Thread.currentThread().interrupt();
          }
          // Stop the service using the startId, so that we don't stop
          // the service in the middle of handling another job
          stopSelf(msg.arg1);
      }
  }

  @Override
  public void onCreate() {
    // Start up the thread running the service. Note that we create a
    // separate thread because the service normally runs in the process's
    // main thread, which we don't want to block. We also make it
    // background priority so CPU-intensive work doesn't disrupt our UI.
    HandlerThread thread = new HandlerThread("ServiceStartArguments",
            Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();

    // Get the HandlerThread's Looper and use it for our Handler
    serviceLooper = thread.getLooper();
    serviceHandler = new ServiceHandler(serviceLooper);
  }

  @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 = serviceHandler.obtainMessage();
      msg.arg1 = startId;
      serviceHandler.sendMessage(msg);

      // 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() {
    Toast.makeText(this, "service done", Toast.LENGTH_SHORT).show();
  }
}
```

Creating a started service

Starting a service:



You can start a service from an activity or other application component by passing an `Intent` to `startService()` or `startForegroundService()`. The Android system calls the service's `onStartCommand()` method and passes it the `Intent`, which specifies which service to start.

For example, an activity can start the example service in the previous section (`HelloService`) using an explicit intent with `startService()`, as shown here:

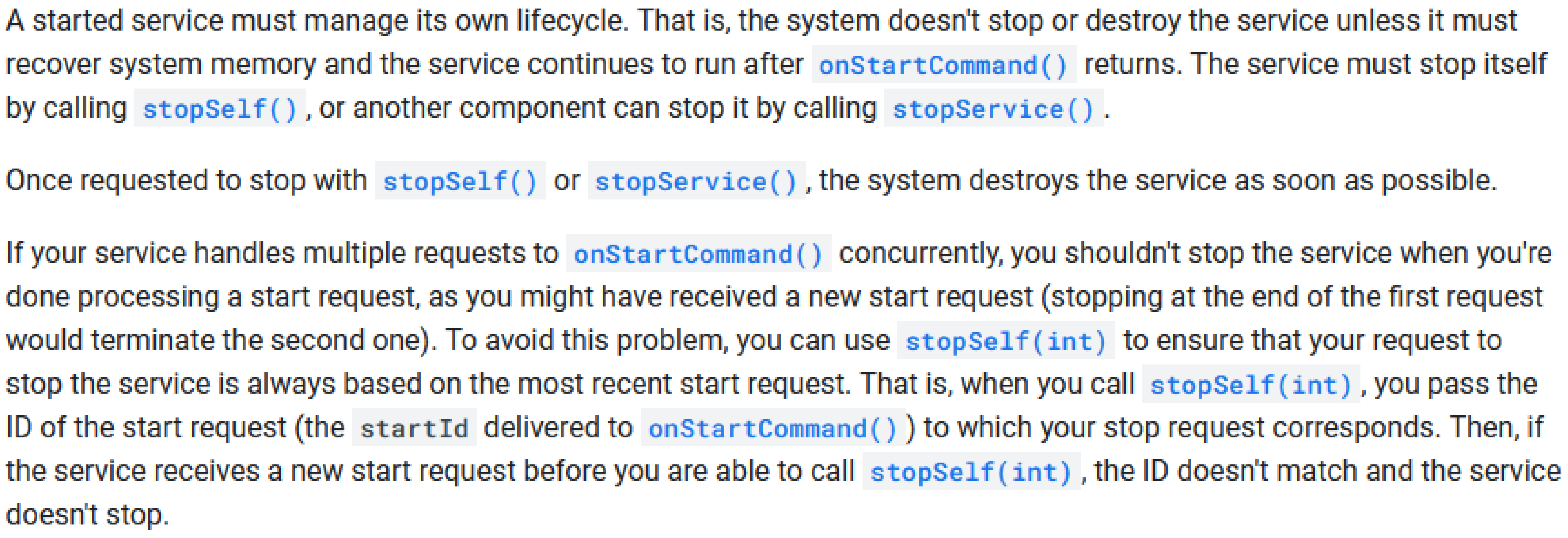
KOTLIN JAVA

```java
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 isn't already running, the system first calls `onCreate()`, and then it calls `onStartCommand()`.

Creating a started service

Stopping a service:



A started service must manage its own lifecycle. That is, the system doesn't stop or destroy the service unless it must recover system memory and the service continues to run after `onStartCommand()` returns. The service must stop itself by calling `stopSelf()`, or another component can stop it by calling `stopService()`.

Once requested to stop with `stopSelf()` or `stopService()`, the system destroys the service as soon as possible.

If your service handles multiple requests to `onStartCommand()` concurrently, you shouldn't stop the service when you're done processing a start request, as you might have received a new start request (stopping at the end of the first request would terminate the second one). To avoid this problem, you can use `stopSelf(int)` to ensure that your request to stop the service is always based on the most recent start request. That is, when you call `stopSelf(int)`, you pass the ID of the start request (the `startId` delivered to `onStartCommand()`) to which your stop request corresponds. Then, if the service receives a new start request before you are able to call `stopSelf(int)`, the ID doesn't match and the service doesn't stop.

Implementing the lifecycle callbacks

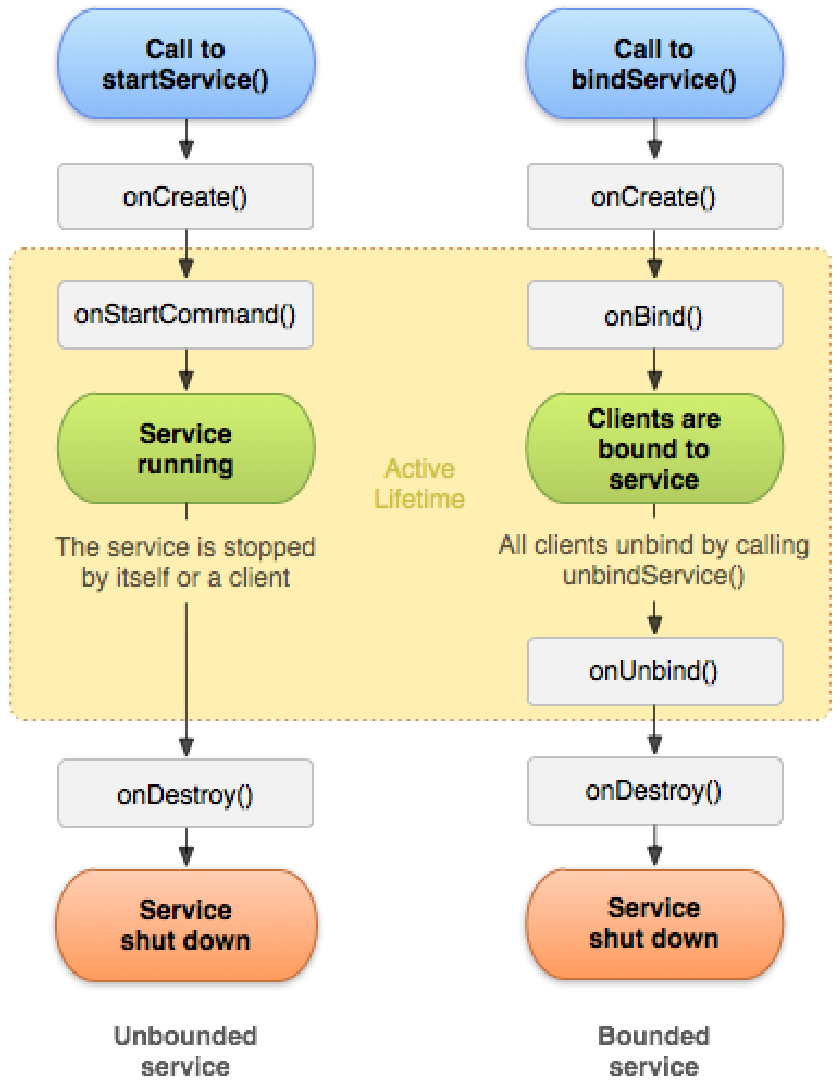
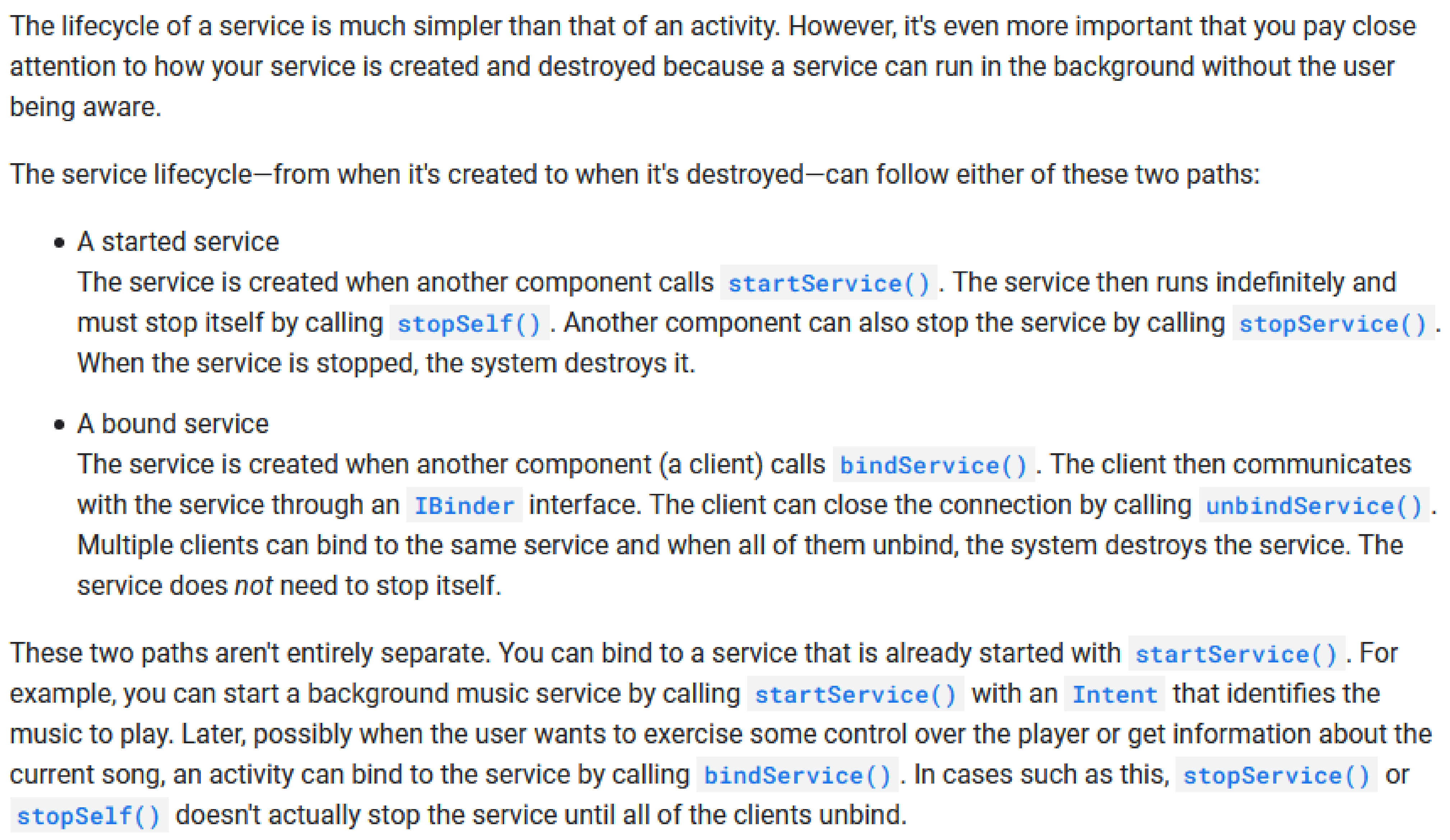


Figure 2 illustrates the typical callback methods for a service. Although the figure separates services that are created by startService() from those created by bindService(), keep in mind that any service, no matter how it's started, can potentially allow clients to bind to it. A service that was initially started with onStartCommand() (by a client calling startService()) can still receive a call to onBind() (when a client calls bindService()).

Figure 2. The service lifecycle. The diagram on the left shows the lifecycle when the service is created with `startService()` and the diagram on the right shows the lifecycle when the service is created with `bindService()`.

Managing the lifecycle of a service



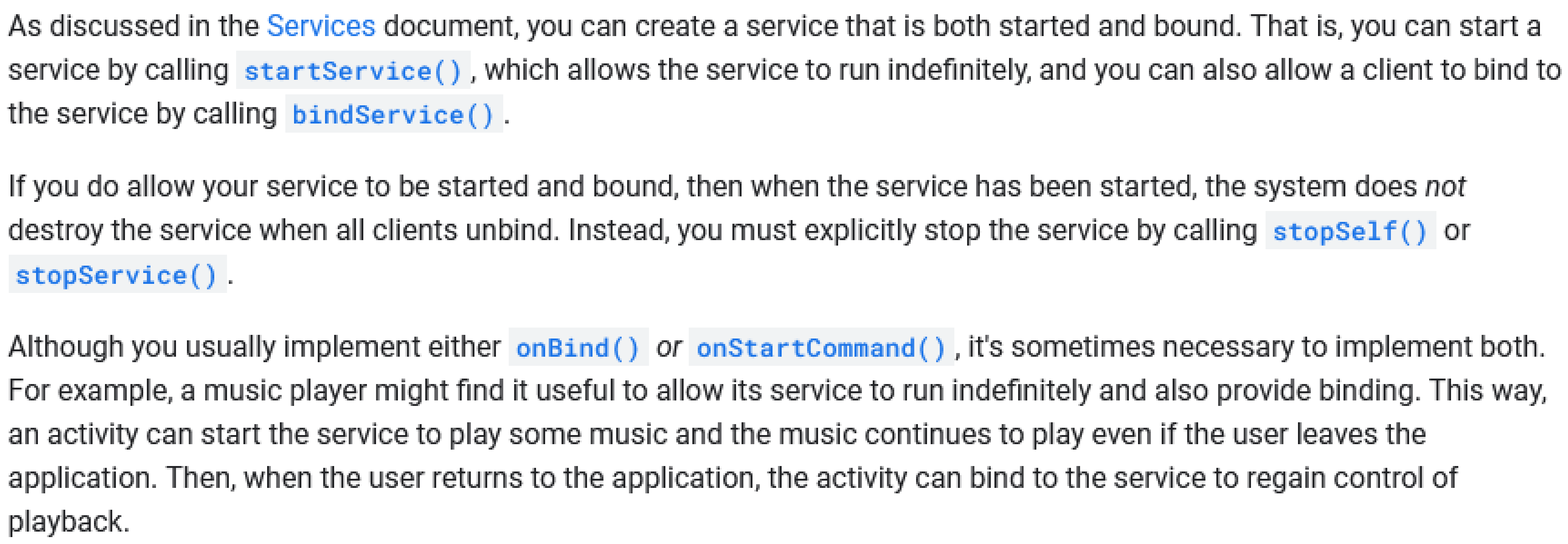
The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed because a service can run in the background without the user being aware.

The service lifecycle—from when it's created to when it's destroyed—can follow either of these two paths:

- A started service
  The service is created when another component calls `startService()`. The service then runs indefinitely and must stop itself by calling `stopSelf()`. Another component can also stop the service by calling `stopService()`. When the service is stopped, the system destroys it.
- A bound service
  The service is created when another component (a client) calls `bindService()`. The client then communicates with the service through an `IBinder` interface. The client can close the connection by calling `unbindService()`. Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. The service does *not* need to stop itself.

These two paths aren't entirely separate. You can bind to a service that is already started with `startService()`. For example, you can start a background music service by calling `startService()` with an `Intent` that identifies the music to play. Later, possibly when the user wants to exercise some control over the player or get information about the current song, an activity can bind to the service by calling `bindService()`. In cases such as this, `stopService()` or `stopSelf()` doesn't actually stop the service until all of the clients unbind.

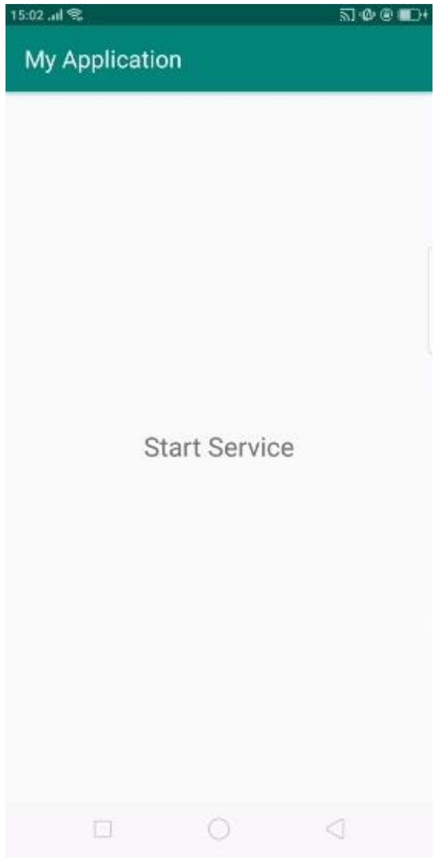
Managing the lifecycle of a service

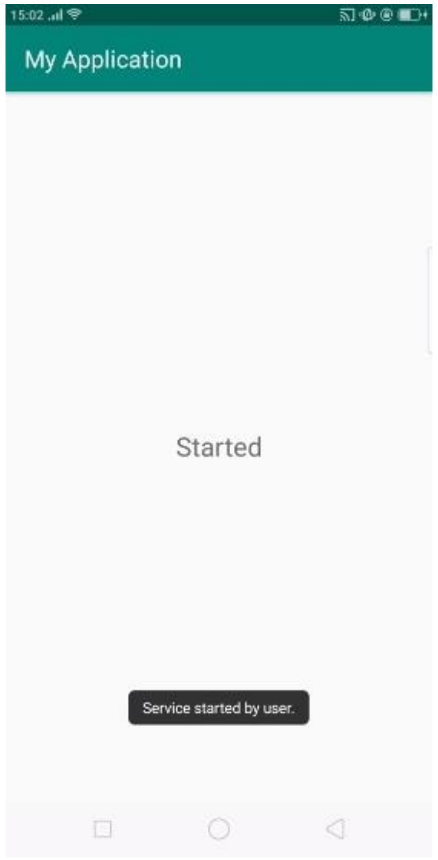


As discussed in the Services document, you can create a service that is both started and bound. That is, you can start a service by calling `startService()`, which allows the service to run indefinitely, and you can also allow a client to bind to the service by calling `bindService()`.

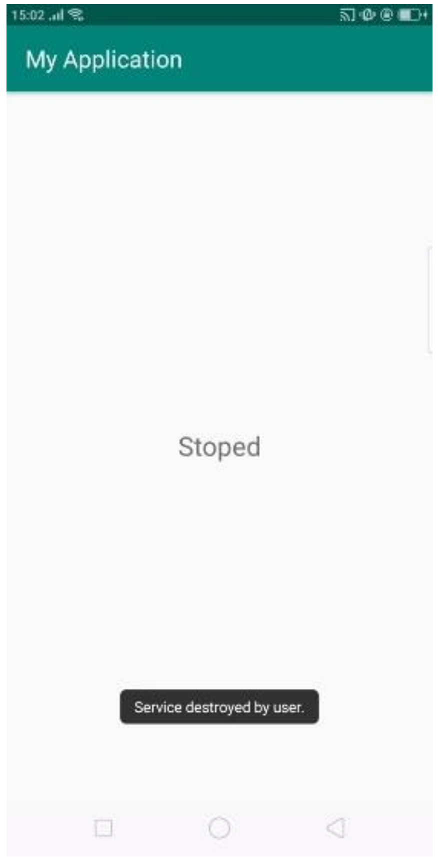
If you do allow your service to be started and bound, then when the service has been started, the system does *not* destroy the service when all clients unbind. Instead, you must explicitly stop the service by calling `stopSelf()` or `stopService()`.

Although you usually implement either `onBind()` *or* `onStartCommand()`, it's sometimes necessary to implement both. For example, a music player might find it useful to allow its service to run indefinitely and also provide binding. This way, an activity can start the service to play some music and the music continues to play even if the user leaves the application. Then, when the user returns to the application, the activity can bind to the service to regain control of playback.





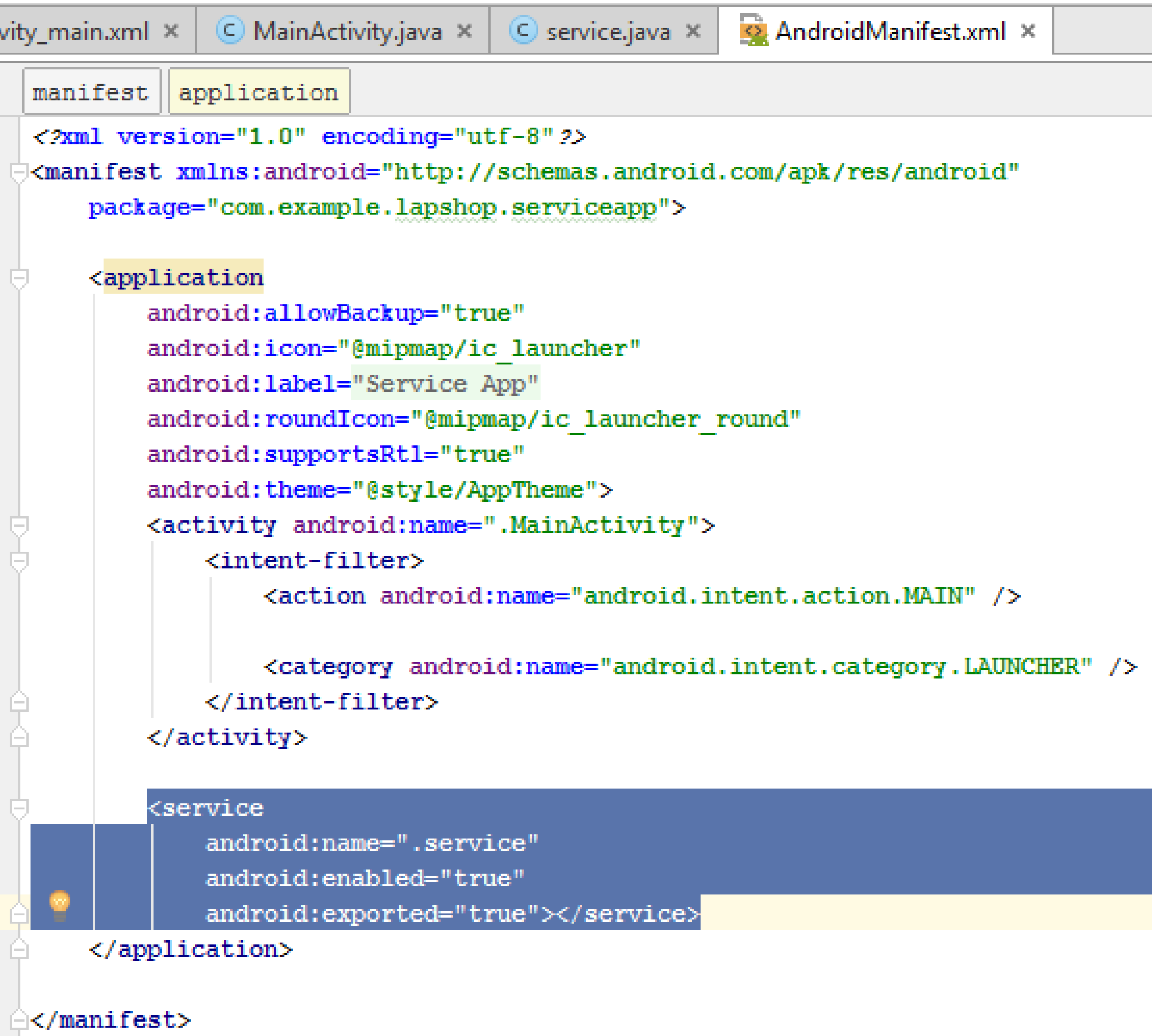
Service started by user.



Service destroyed by user.

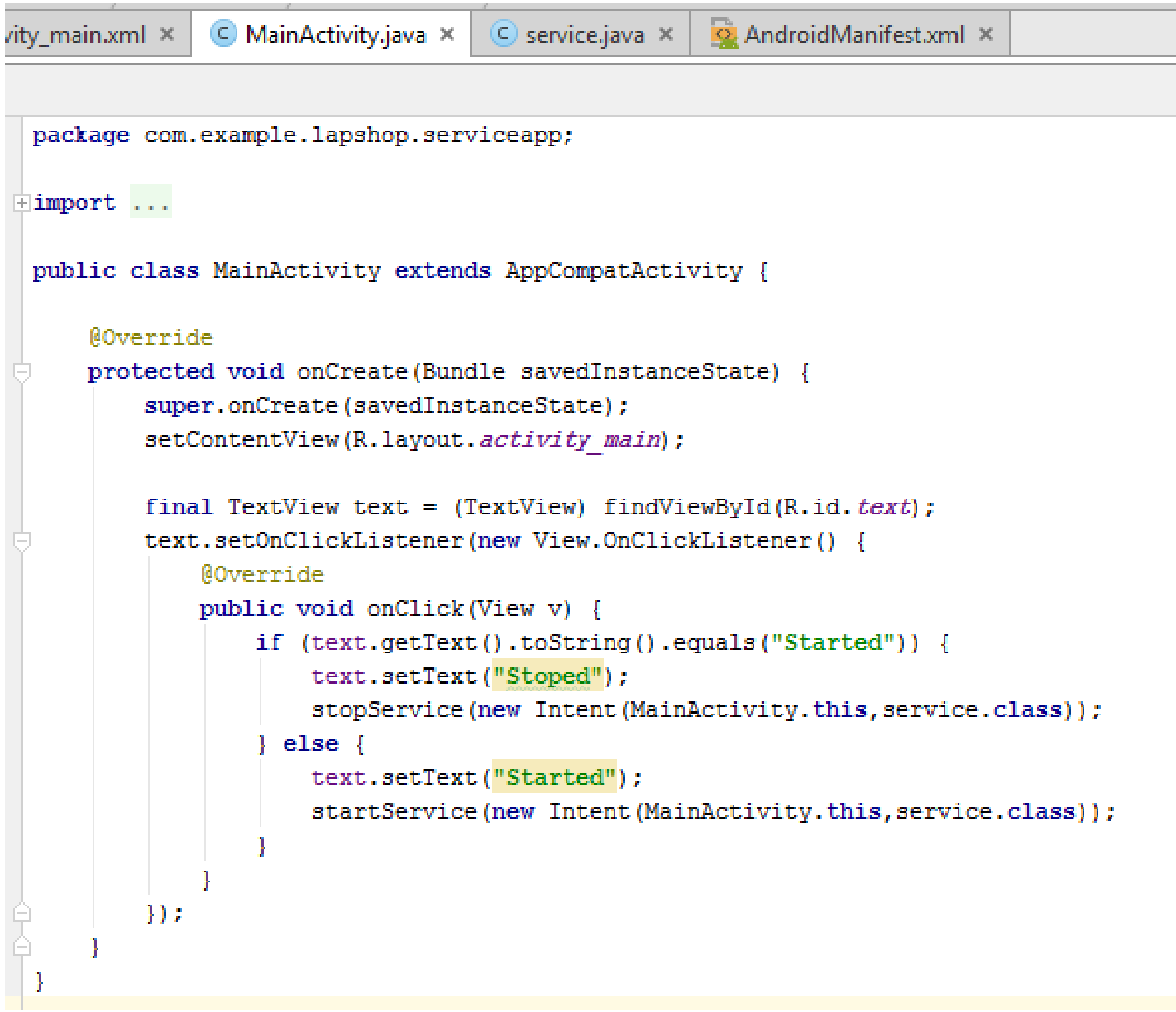
Service Android APP

1- Declaring a service in the manifest



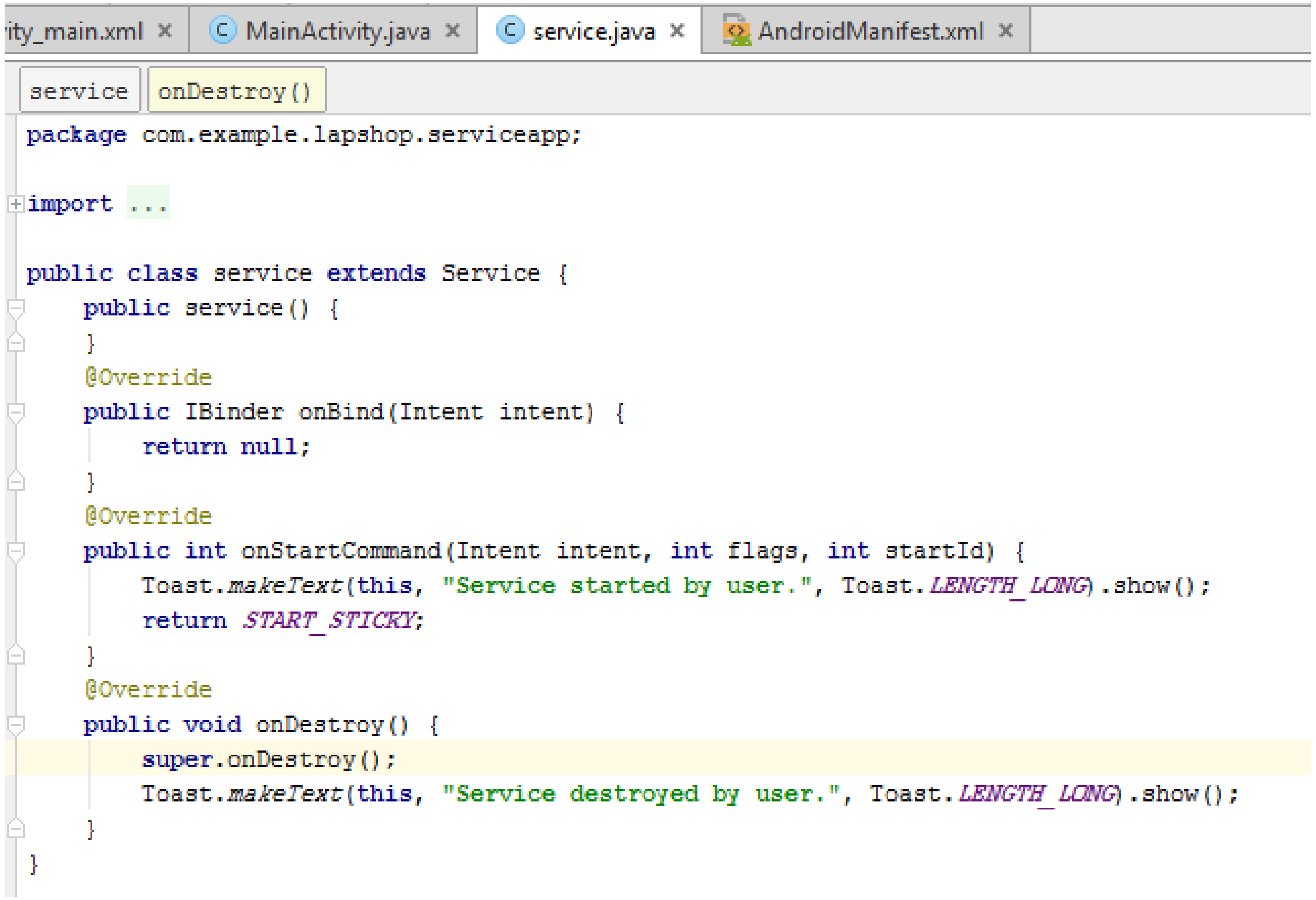
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.lapshop.serviceapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Service App"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".service"
            android:enabled="true"
            android:exported="true"></service>
    </application>

</manifest>
```

2- Starting and stopping a service in java



```
package com.example.lapshop.serviceapp;

import ...

public class MainActivity extends AppCompatActivity {

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

        final TextView text = (TextView) findViewById(R.id.text);
        text.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (text.getText().toString().equals("Started")) {
                    text.setText("Stoped");
                    stopService(new Intent(MainActivity.this,service.class));
                } else {
                    text.setText("Started");
                    startService(new Intent(MainActivity.this,service.class));
                }
            }
        });
    }
}
```

3- Creating a started service



```
package com.example.lapshop.serviceapp;

import ...

public class service extends Service {
    public service() {
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "Service started by user.", Toast.LENGTH_LONG).show();
        return START_STICKY;
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "Service destroyed by user.", Toast.LENGTH_LONG).show();
    }
}
```

Bound Services:
https://developer.android.com/guide/components/bound-services

Foreground Services:
https://developer.android.com/guide/components/foreground-services

Layouts

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of `View` and `ViewGroup` objects. A `View` usually draws something the user can see and interact with. Whereas a `ViewGroup` is an invisible container that defines the layout structure for `View` and other `ViewGroup` objects, as shown in figure 1.

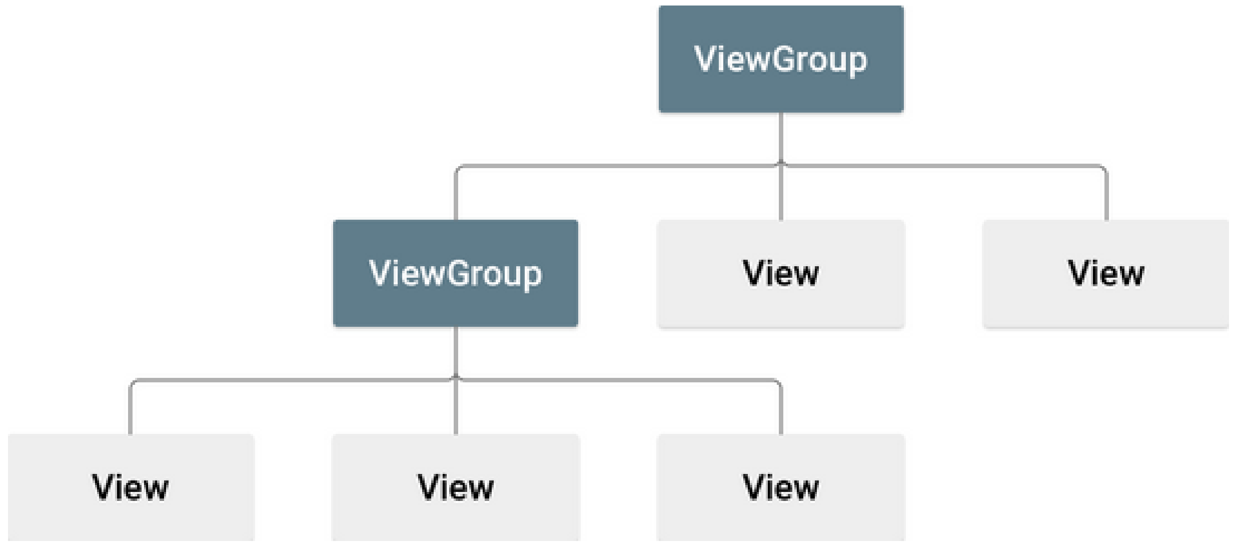
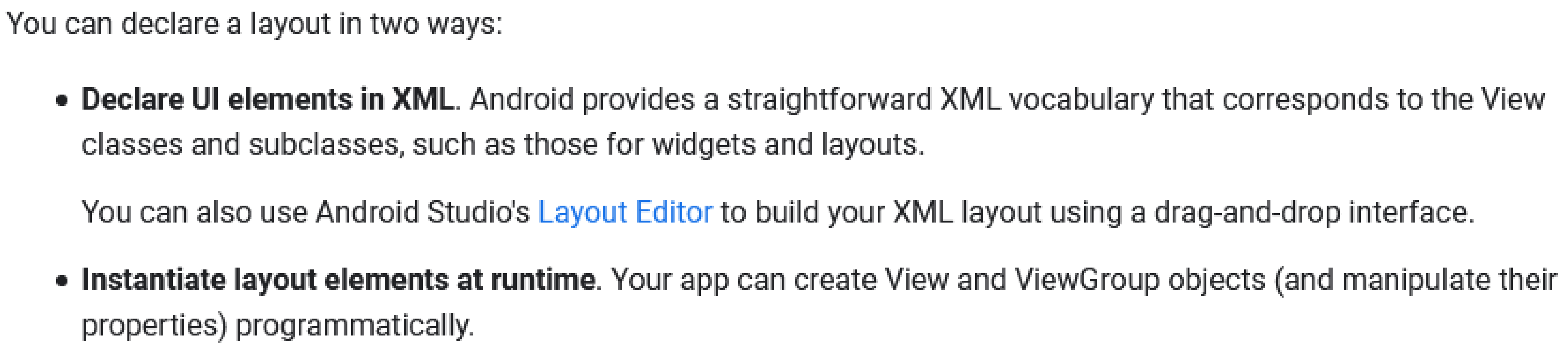


Figure 1. Illustration of a view hierarchy, which defines a UI layout

The `View` objects are usually called "widgets" and can be one of many subclasses, such as `Button` or `TextView`. The `ViewGroup` objects are usually called "layouts" can be one of many types that provide a different layout structure, such as `LinearLayout` or `ConstraintLayout`.

Layouts



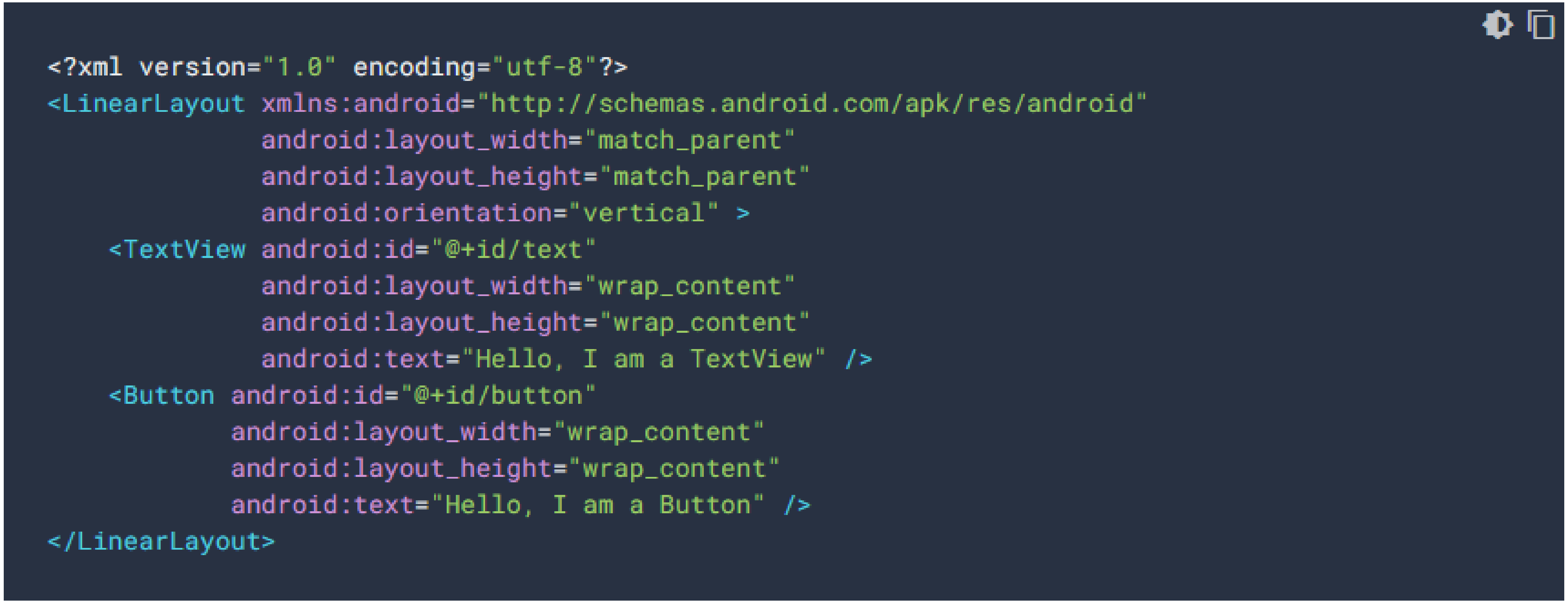
You can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

  You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.

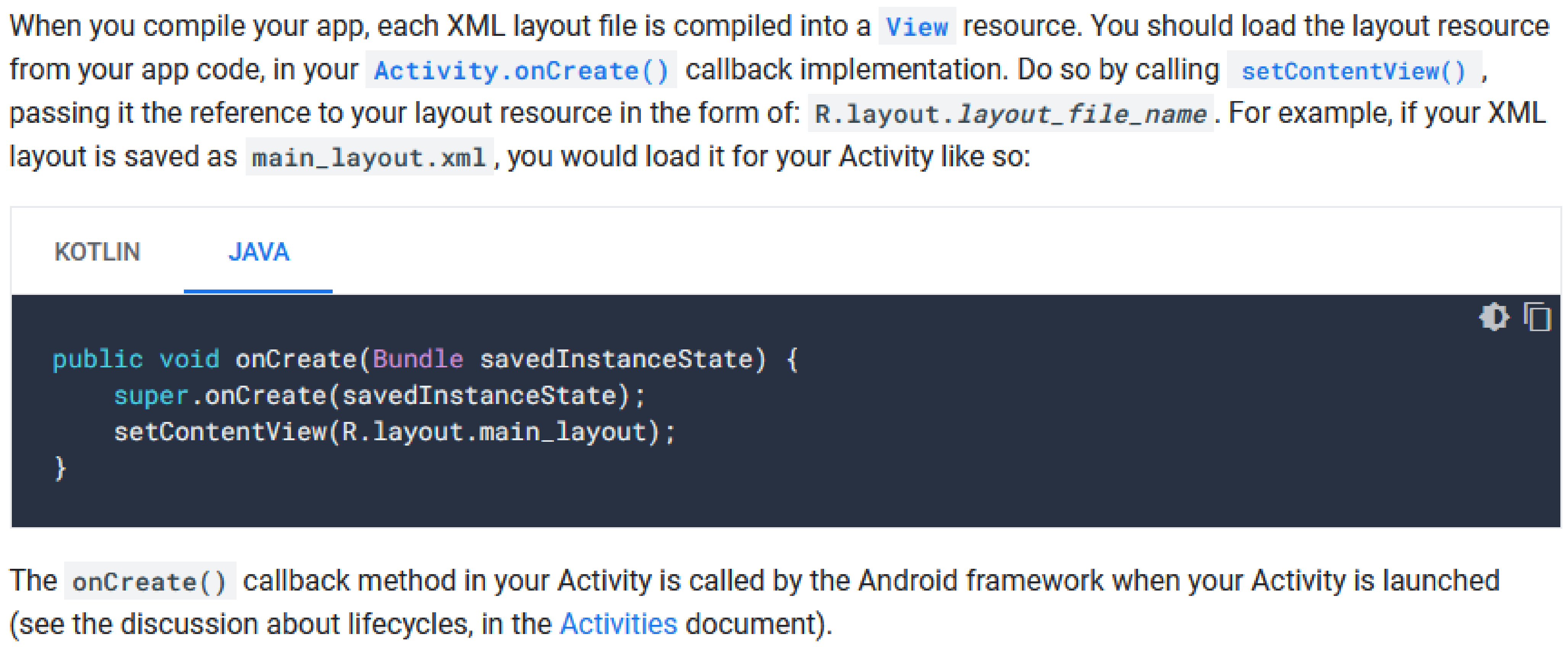
- **Instantiate layout elements at runtime**. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Write the XML



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Load the XML Resource



When you compile your app, each XML layout file is compiled into a `View` resource. You should load the layout resource from your app code, in your `Activity.onCreate()` callback implementation. Do so by calling `setContentView()`, passing it the reference to your layout resource in the form of: `R.layout.layout_file_name`. For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

KOTLIN JAVA

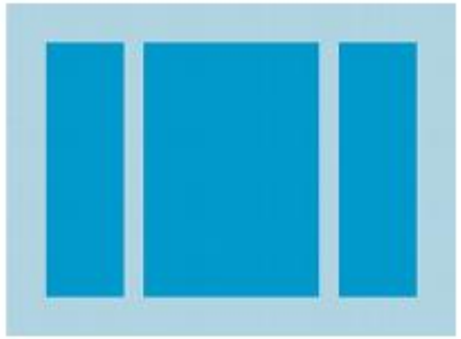
```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the Activities document).

Common Layouts

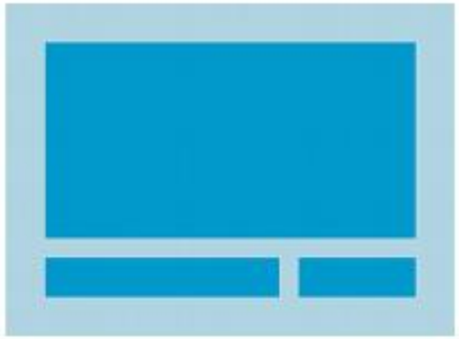
Each subclass of the `ViewGroup` class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Linear Layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

Web View

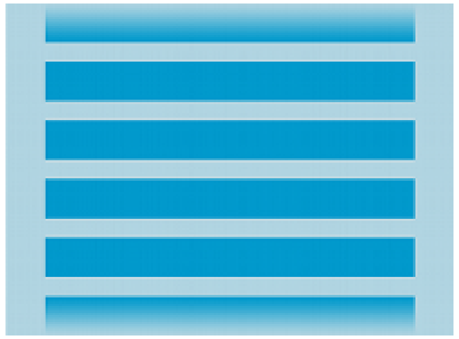
Displays web pages.

Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses `AdapterView` to populate the layout with views at runtime. A subclass of the `AdapterView` class uses an `Adapter` to bind data to its layout. The `Adapter` behaves as a middleman between the data source and the `AdapterView` layout—the `Adapter` retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the `AdapterView` layout.

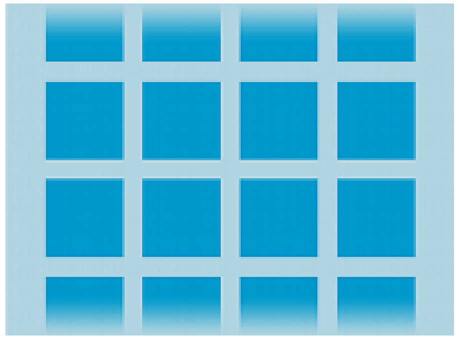
Common layouts backed by an adapter include:

List View



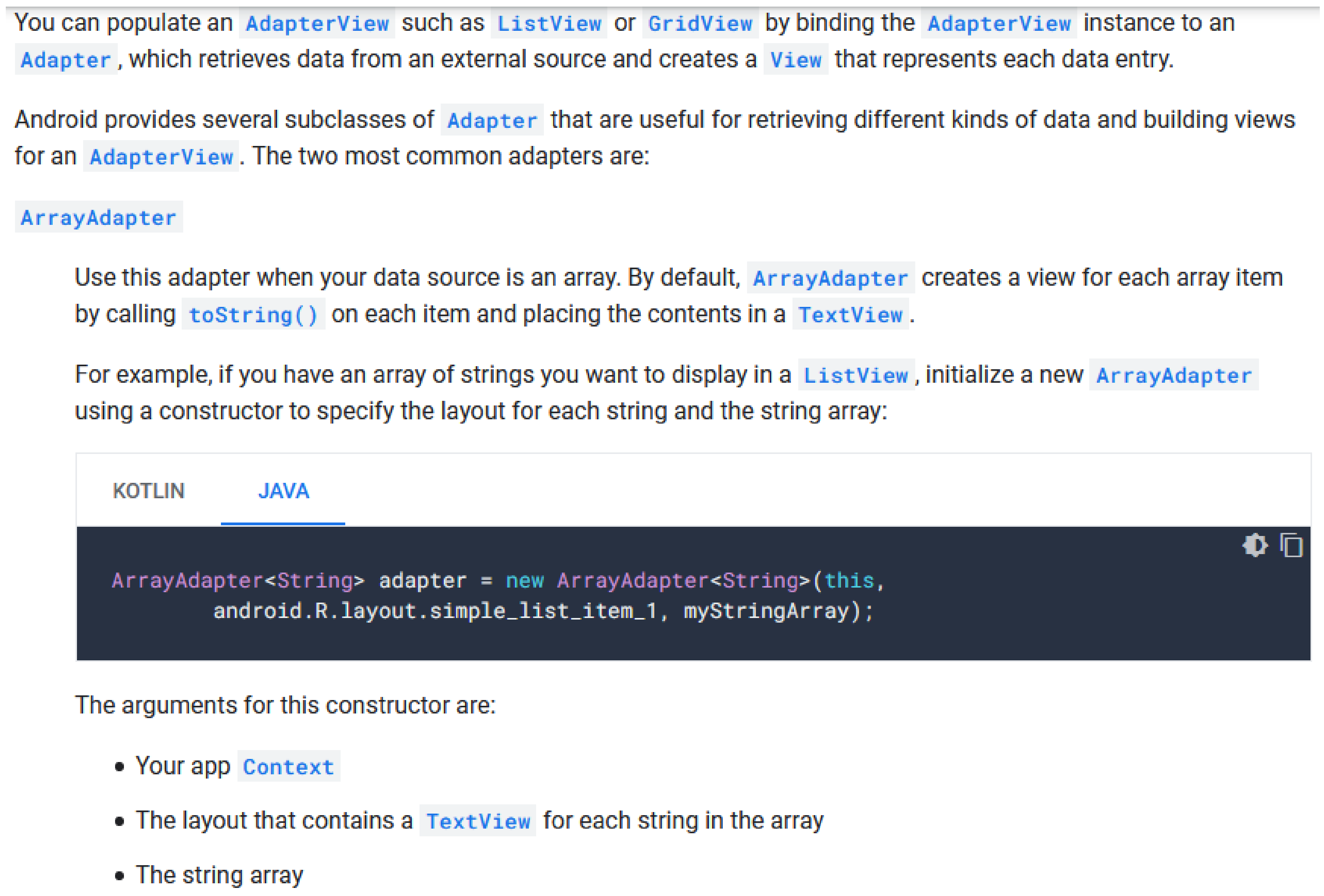
Displays a scrolling single column list.

Grid View



Displays a scrolling grid of columns and rows.

Filling an adapter view with data



You can populate an `AdapterView` such as `ListView` or `GridView` by binding the `AdapterView` instance to an `Adapter`, which retrieves data from an external source and creates a `View` that represents each data entry.

Android provides several subclasses of `Adapter` that are useful for retrieving different kinds of data and building views for an `AdapterView`. The two most common adapters are:

`ArrayAdapter`

Use this adapter when your data source is an array. By default, `ArrayAdapter` creates a view for each array item by calling `toString()` on each item and placing the contents in a `TextView`.

For example, if you have an array of strings you want to display in a `ListView`, initialize a new `ArrayAdapter` using a constructor to specify the layout for each string and the string array:

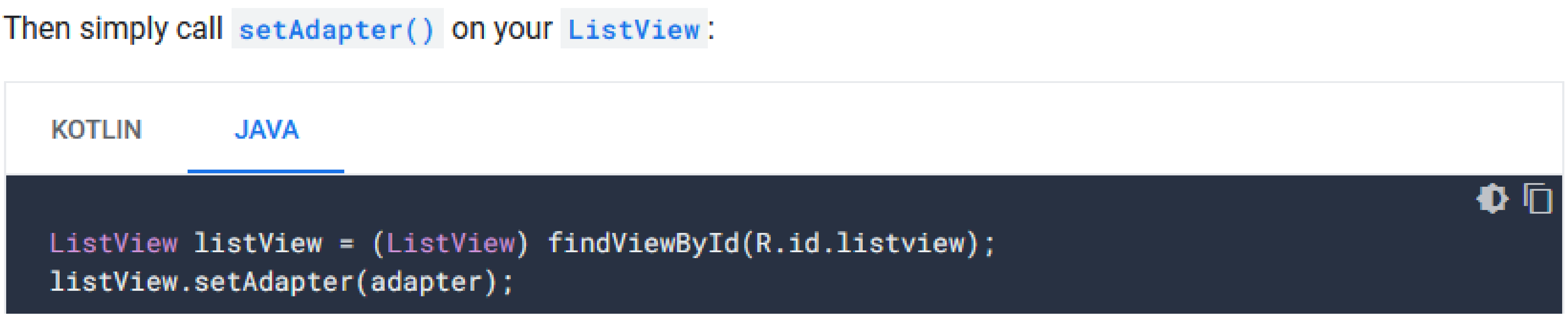
KOTLIN JAVA

```java
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, myStringArray);
```

The arguments for this constructor are:

- Your app `Context`
- The layout that contains a `TextView` for each string in the array
- The string array

Filling an adapter view with data



Then simply call `setAdapter()` on your `ListView`:

```java
ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

See you in next session

Broadcasts overview

- Android apps can send or receive broadcast messages from the Android system and other Android apps. These broadcasts are sent when an event of interest occurs.
- For example, the 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.
- Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow. However, you must be careful not to abuse the opportunity to respond to broadcasts and run jobs in the background that can contribute to a slow system performance.