IAT359 Mobile Computing

Fall 2022

Instructor: Hanieh Shakeri

TA: Parnian Taghipour

lecture 7

- Processes and threads in Android
- Process priority in Android
- AsyncTask approach
- Android services

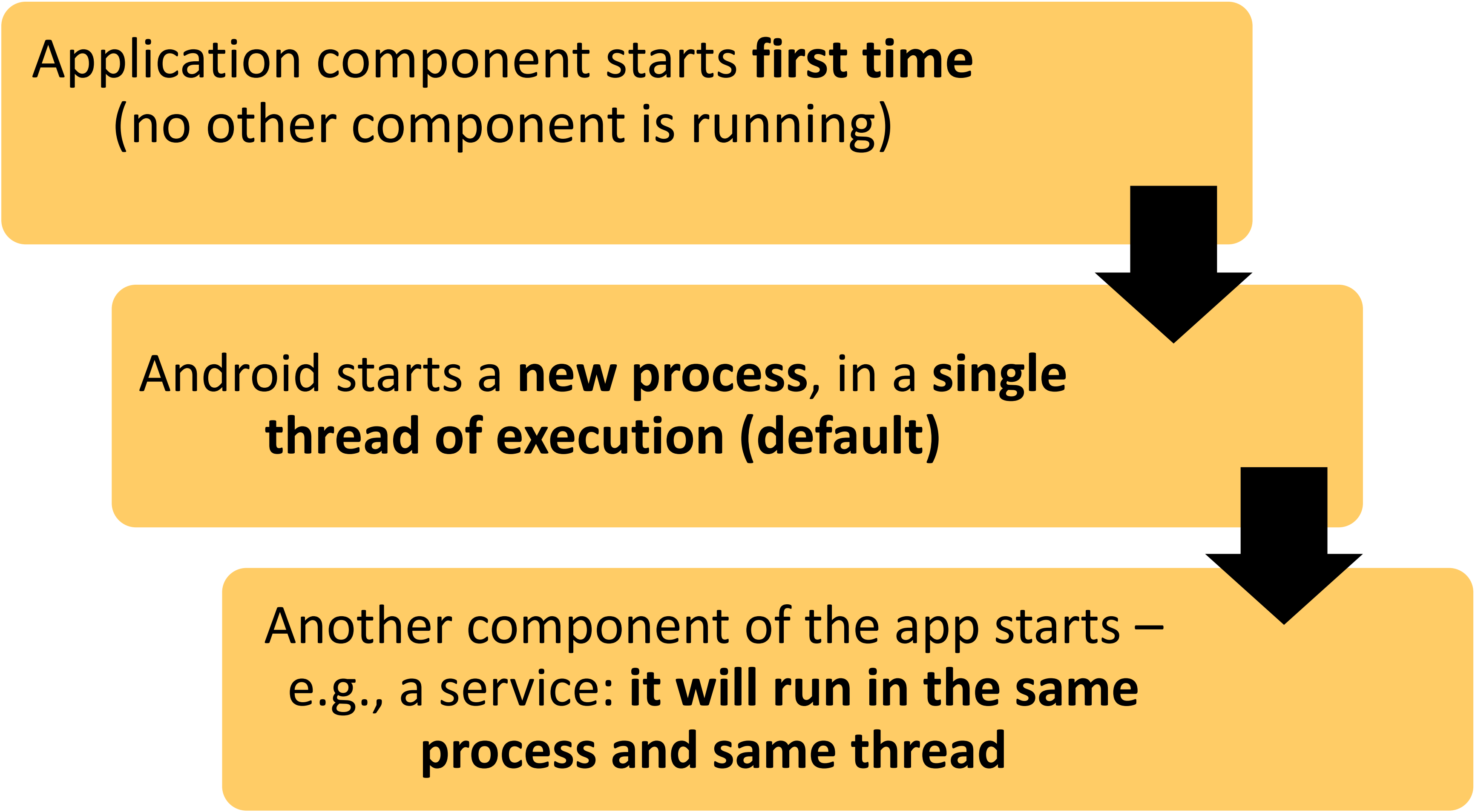
week 7 check-in

- Assignment 2 due Friday midnight
- Quiz 3 is next week – will open on Nov 1 at 2:20PM and will be due Nov 2 at 2:20PM
- Milestone 2 is due November 22 – this will be the majority of your implementation, so start early!
- Final exam date posted: December 17 at 8:30AM (a Saturday).

process and thread concepts

- **Processes** run in their own memory space
- A **process** can be considered an executing instance of an application
- **Threads** share the same memory space
- One **process** can be associated with several **threads**

processes and threads in Android



Application component starts **first time** (no other component is running)

Android starts a **new process**, in a **single thread of execution (default)**

Another component of the app starts – e.g., a service: **it will run in the same process and same thread**

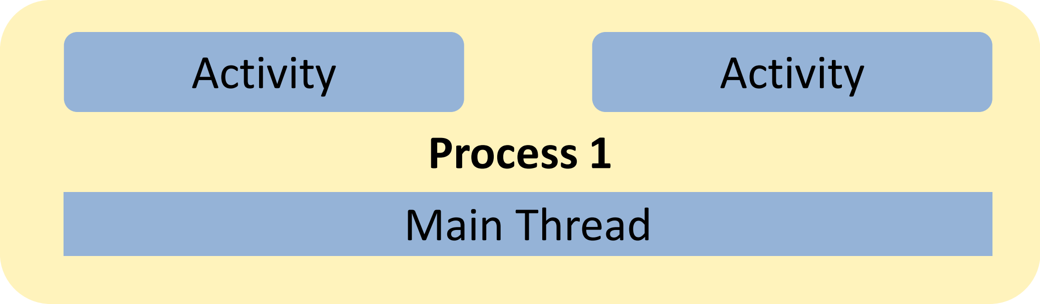
in android

- We can:
  - Arrange for different components in the app to run in separate processes
  - Create additional threads for any process

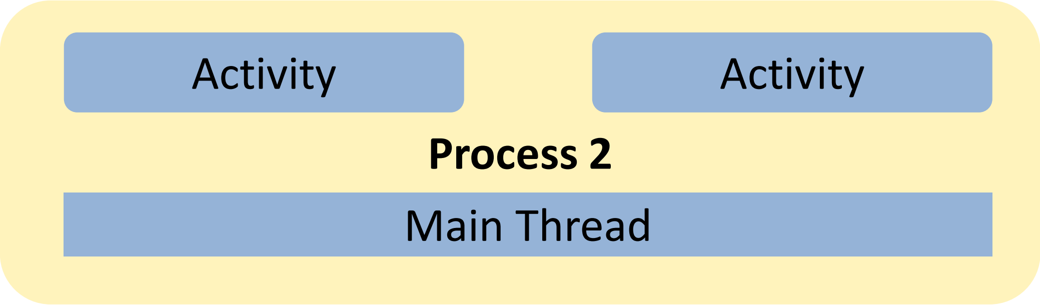
processes terminated (killed) by android

- Before terminating a process, Android assesses its importance / priority in reference to the user
- Comparison between processes
- The processes with which the user is interacting at that moment have higher priority

process priority



User is not interacting
with this process (not visible)



User is interacting
with this process

Process 2 has priority and will NOT be terminated first.

process termination – how?

- Process with lowest importance terminated first
- What processes have highest importance with respect to the user?
  - **FOREGROUND PROCESSES (Highest Priority)**

Activity – while in onResume() method User is interacting

Service – executing its lifecycle methods – foreground service or user interacting with service bound to the current activity

Broadcast Receiver – while executing onReceive() method

process importance

- 1. **Foreground processes** (previous slide)
  2. **Visible processes**
  - Activity onPause()
  - Service bound to a visible activity
- 3. **Service Process**
  - No direct user interaction
  - Service running (e.g., playing music)
- 4. **Background process**
  - No user interaction at all
  - The app most recently seen by the user is the last one to be destroyed
- 5. **Empty process**
  - No active component running
  - Still alive for caching purposes

keep in mind

- Empty process is the first one to be terminated.

- A process running a service is a Service Process. A Service Process is ranked higher than a process that does not have a service.

- It is better to use a service for long-running operations, rather than performing long-running operations in a thread.

main thread

- UI elements – updated on the main thread

- Clicking a button – sends a message on the main thread

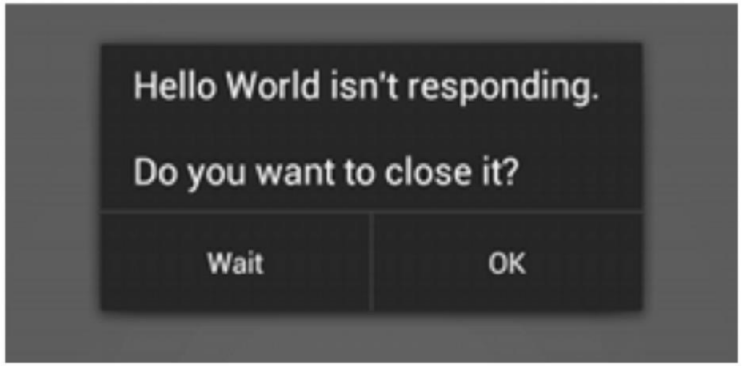
- startService() and service code – processed on the main thread

long-running operations in android

- Network access, database queries – block the UI
  - Main thread will be busy processing the long-running operations
  - Other UI operations (button clicks, etc) – will be dropped as messages inside the main thread queue

- UI thread blocked => no events can be dispatched => no drawing events can be dispatched

- From the user's perspective – the app appears frozen



Hello World isn't responding.

Do you want to close it?

avoid blocking UI thread (main thread)

DO NOT perform **long-running operations** on the main thread

- Working with files
- Network access
- Database access
- Complex calculations

Use **additional threads** for complex tasks

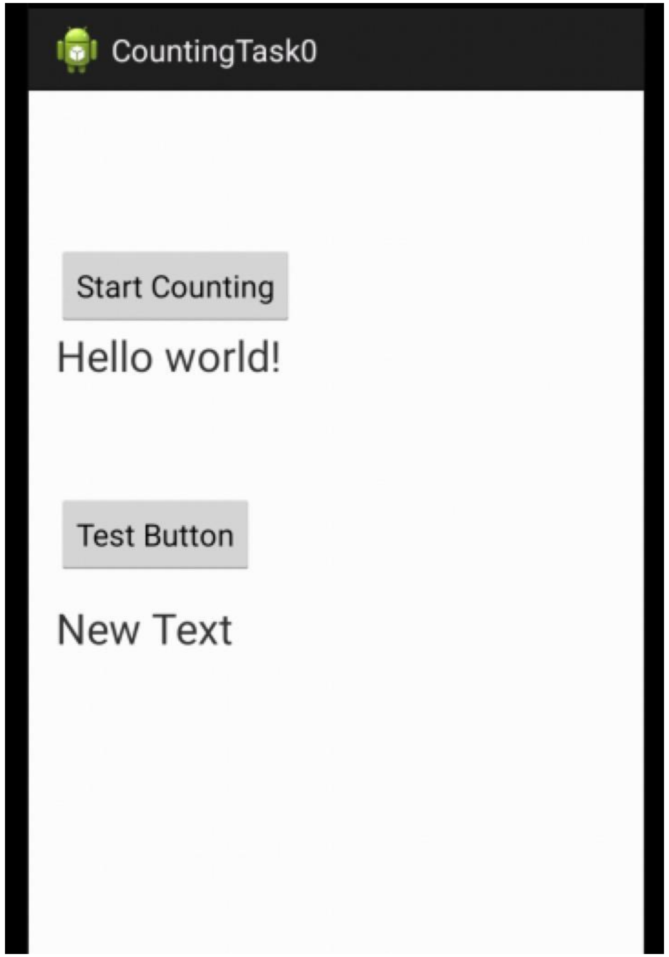
DO NOT modify UI from background threads

Use services

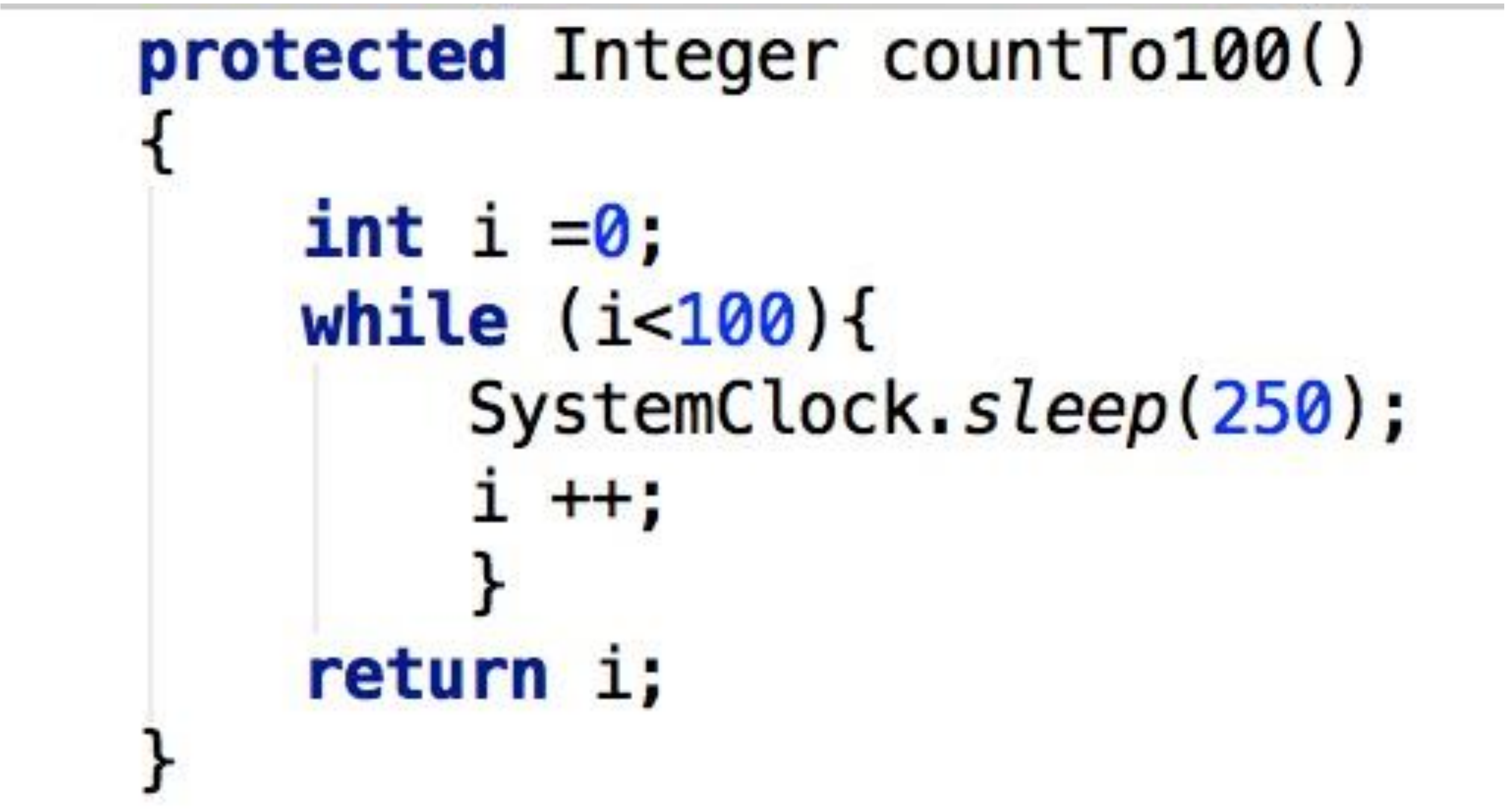
a simple example: CountingTask

Count to 100, with a delay of 250ms at each increase (StartCounting button will start the long operation of counting

Test Button: to test the responsiveness of the UI



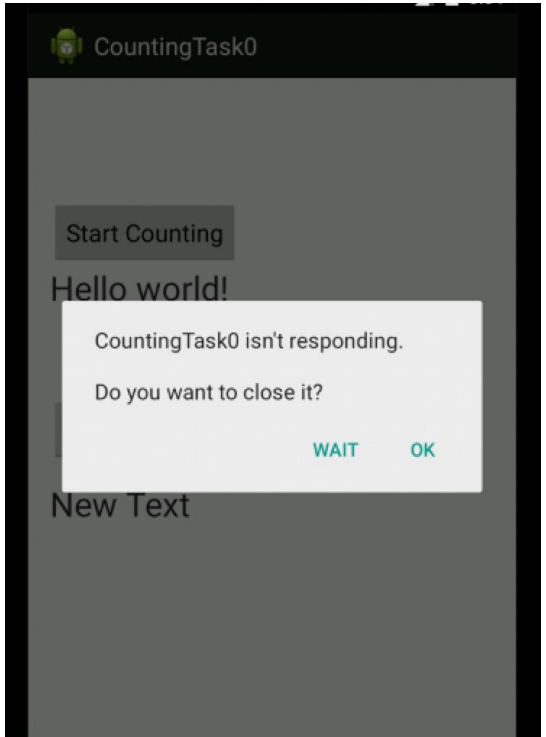
StartCounting()



```
protected Integer countTo100()
{
    int i =0;
    while (i<100){
        SystemClock.sleep(250);
        i ++;
        }
    return i;
}
```

responsiveness

- The TestButton is not responding while the long operation of counting is in progress
- The ANR (“Application Not Responding”) dialog appears

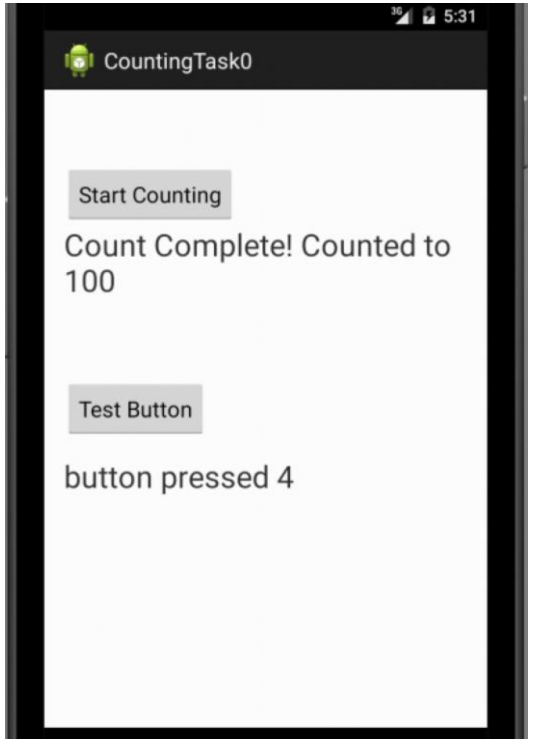


Hello world!

CountingTask0 isn't responding.

Do you want to close it?

New Text



Count Complete! Counted to 100

button pressed 4

improving responsiveness

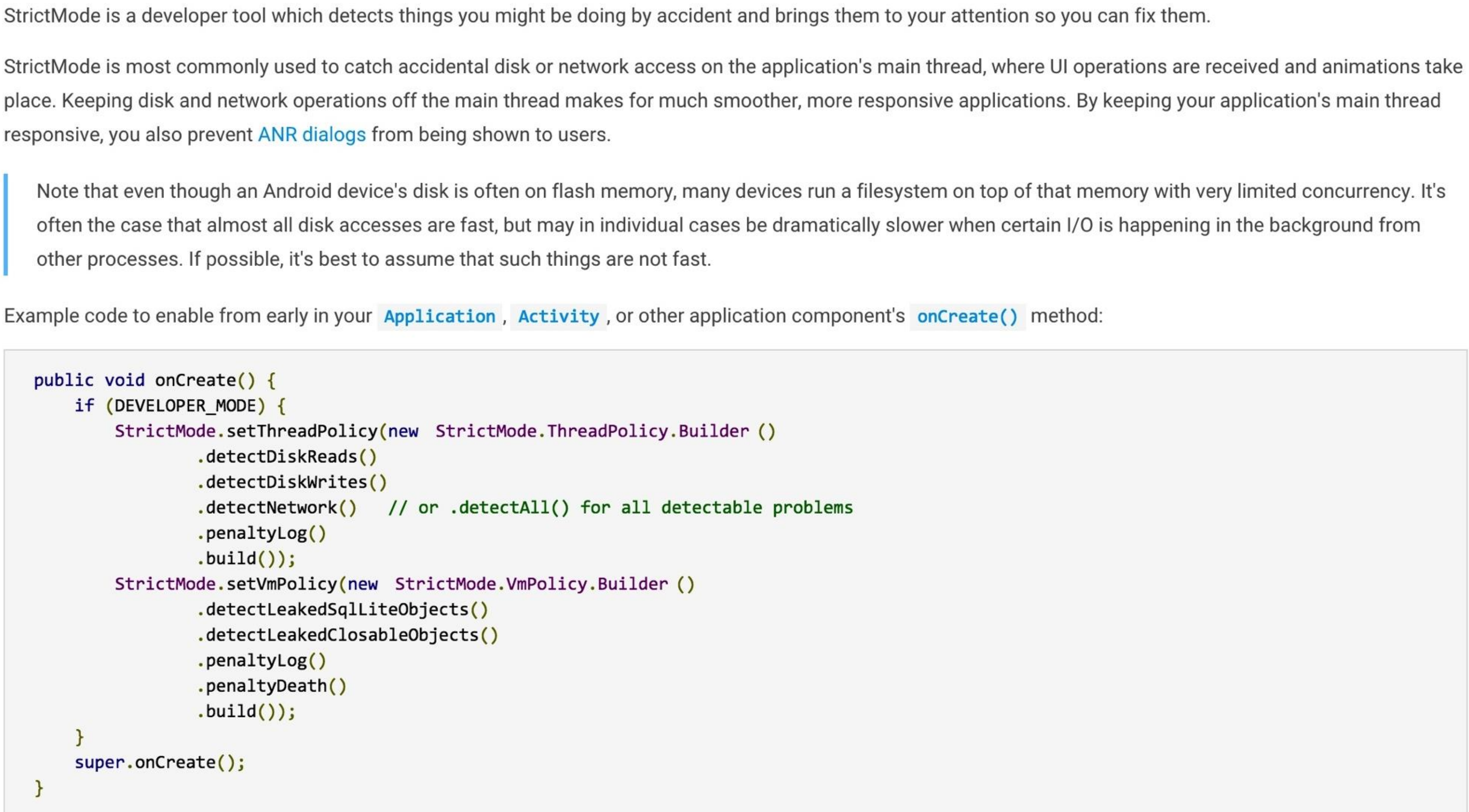
- Avoid performing blocking operations on the UI thread
- Blocking operations:
  - Complex calculations or rendering
  - Looking at data set of unknown size
  - Parsing a data set
  - Processing multimedia files
  - Accessing network resources
  - Accessing location based services
  - Access a content provider
  - Accessing a local database
  - Accessing a file

the UI thread

- For application that consist of Activity (or Activities) it is vital to not block the UI thread (main thread of execution)

- On API level 11 and later certain operations must be moved off the main UI thread
  - Code that accesses resources over a network
  - For example, HTTP requests on the main UI thread result in a NetworkOnMainThreadException
  - StrictMode:http://developer.android.com/reference/android/os/StrictMode.html

StrictMode



StrictMode is a developer tool which detects things you might be doing by accident and brings them to your attention so you can fix them.

StrictMode is most commonly used to catch accidental disk or network access on the application's main thread, where UI operations are received and animations take place. Keeping disk and network operations off the main thread makes for much smoother, more responsive applications. By keeping your application's main thread responsive, you also prevent ANR dialogs from being shown to users.

> Note that even though an Android device's disk is often on flash memory, many devices run a filesystem on top of that memory with very limited concurrency. It's often the case that almost all disk accesses are fast, but may in individual cases be dramatically slower when certain I/O is happening in the background from other processes. If possible, it's best to assume that such things are not fast.

Example code to enable from early in your `Application` , `Activity` , or other application component's `onCreate()` method:

```
public void onCreate() {
    if (DEVELOPER_MODE) {
        StrictMode.setThreadPolicy(new  StrictMode.ThreadPolicy.Builder ()
                .detectDiskReads()
                .detectDiskWrites()
                .detectNetwork()   // or .detectAll() for all detectable problems
                .penaltyLog()
                .build());
        StrictMode.setVmPolicy(new  StrictMode.VmPolicy.Builder ()
                .detectLeakedSqlLiteObjects()
                .detectLeakedClosableObjects()
                .penaltyLog()
                .penaltyDeath()
                .build());
    }
    super.onCreate();
}
```

enabling responsiveness

- **AsyncTask helper class** - recommended
  - Complete tasks asynchronously and communicate back to the main UI thread
- Java Thread class
  - Complete your processing as you would in any Java application
- Loader class
  - Facilitate the loading of data for use in an Activity or Fragment while still starting up quickly

question

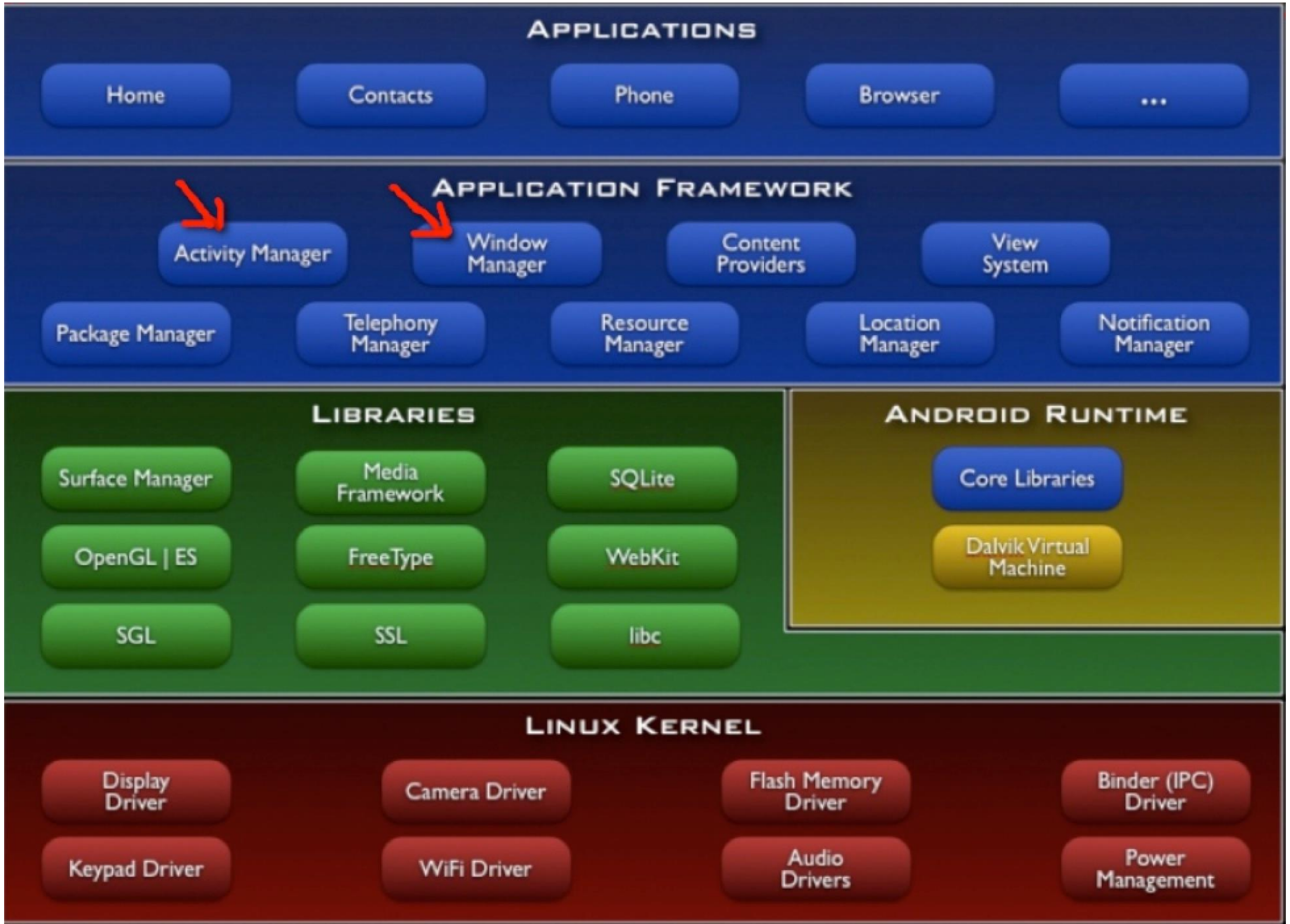
What is the difference between a thread and a process?

AsyncTask Approach

challenges re: responsiveness

Main Thread	• Handle events from different Views and components
ActivityManager **WindowManager**	• Monitor the application for responsiveness
Each task	• If it takes more than 5 seconds, the ANR dialog is triggered

activity manager/window manager



AsyncTask / thread limitations

- Good for tasks lasting between 100ms to a few seconds
- For tasks longer than a few seconds: **use services**
- **Note:** if longer tasks use threads, they may not execute successfully as Android may kill those threads under critical conditions

why use AsyncTask

- Easier to implement in comparison to thread and handler

- **Automates** the following:
  - Creation / termination of background thread
  - Managing of message queues
  - Updating on progress

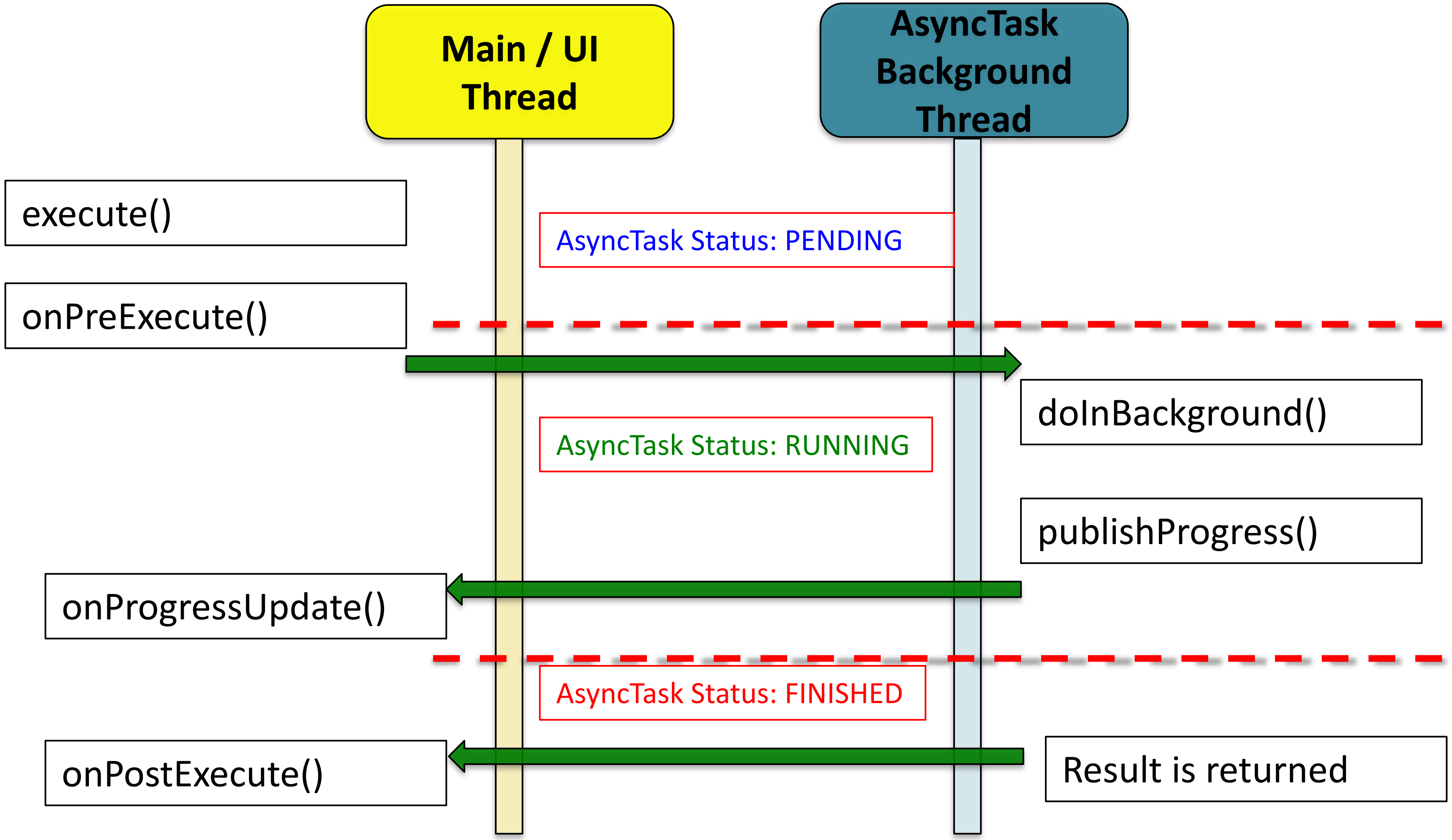
steps – implementing AsyncTask

Create a subclass of AsyncTask

Override one or more AsyncTask methods to:

- Do work inside the background thread
- Handle UI updates on the main thread
- When needed, create an instance of AsyncTask subclass and call execute() to have it begin doing its work

steps for AsyncTask



Main / UI Thread

AsyncTask Background Thread

execute()

AsyncTask Status: PENDING

onPreExecute()

doInBackground()

AsyncTask Status: RUNNING

publishProgress()

onProgressUpdate()

AsyncTask Status: FINISHED

onPostExecute()

Result is returned

parameters to be provided to AsyncTask

Param

- Type of parameter sent to the task for execution

Progress

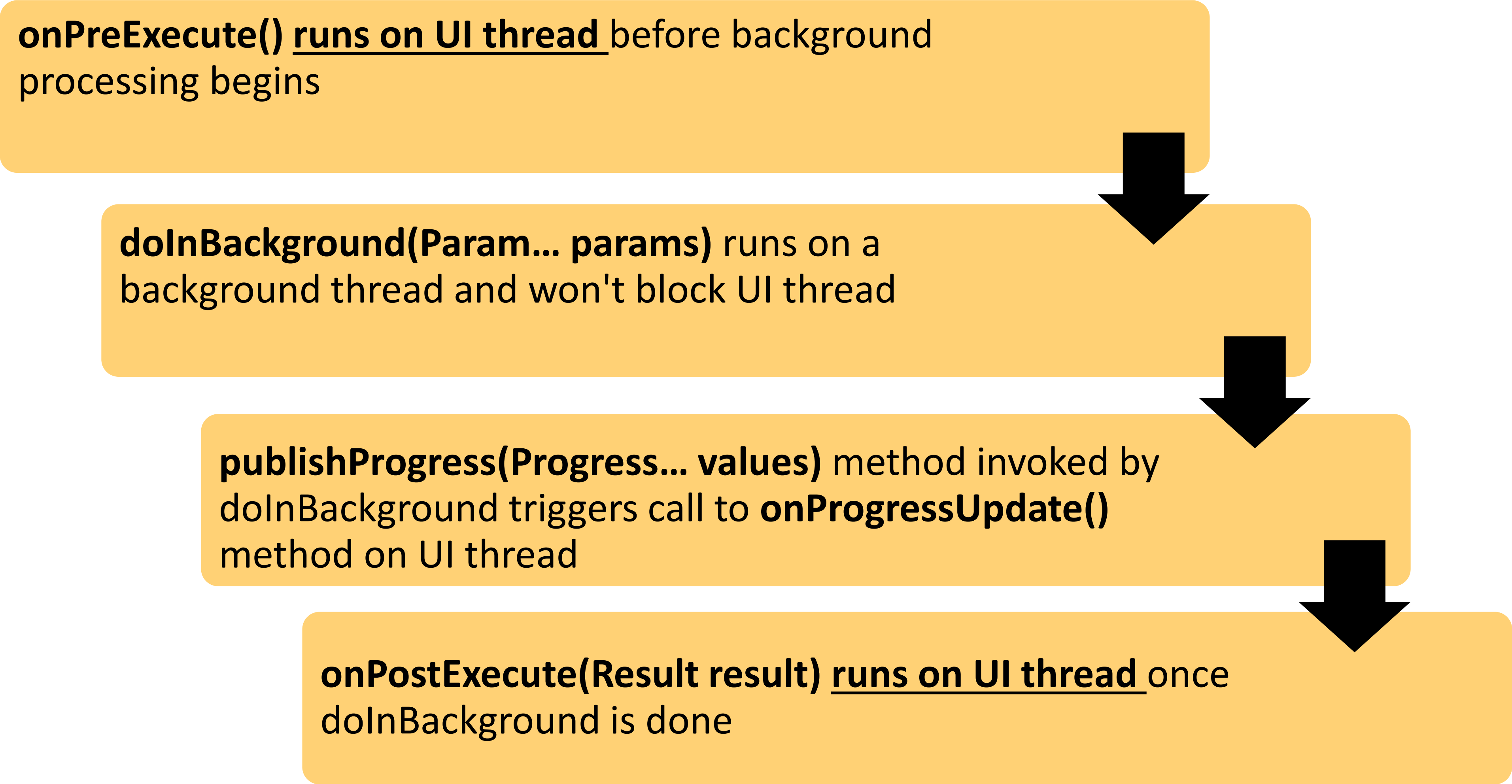
- Type of the info used to indicate progress

Result

- Type of the result received from the background task

Java Generics: https://docs.oracle.com/javase/tutorial/java/generics/why.html

steps of AsyncTask - methods



onPreExecute() runs on UI thread before background processing begins

doInBackground(Param... params) runs on a background thread and won't block UI thread

publishProgress(Progress... values) method invoked by doInBackground triggers call to **onProgressUpdate()** method on UI thread

onPostExecute(Result result) runs on UI thread once doInBackground is done

keep in mind

- AsyncTask class must be loaded on the UI thread

- The task instance must be created on the UI thread

- execute() method must be invoked on the UI thread

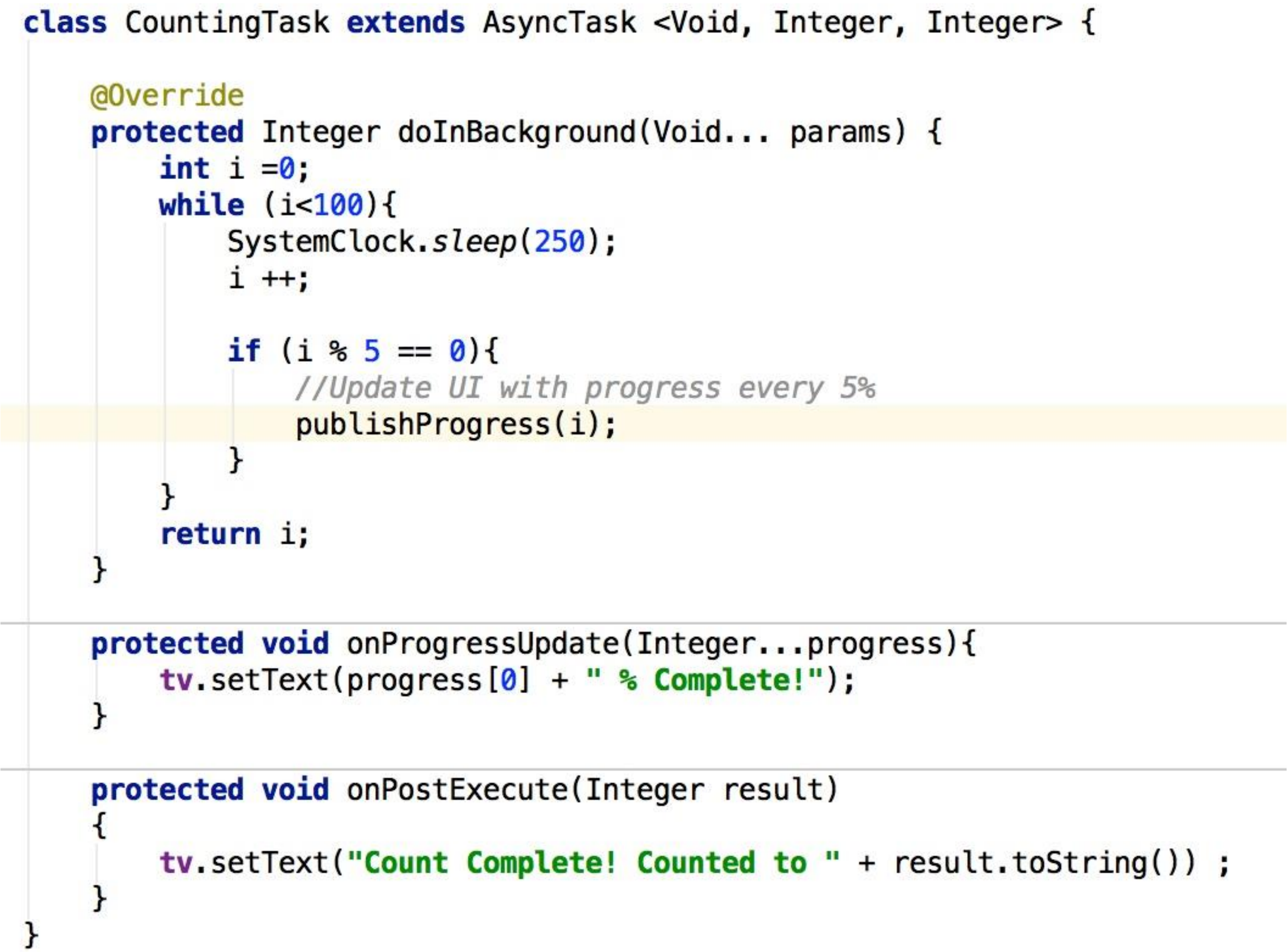
- onPreExecute(), onPostExecute(), doInBackground(), onProgressUpdate() are called automatically – do not call them in your code

question

- Why is it useful for the background thread to communicate about progress with the UI thread?

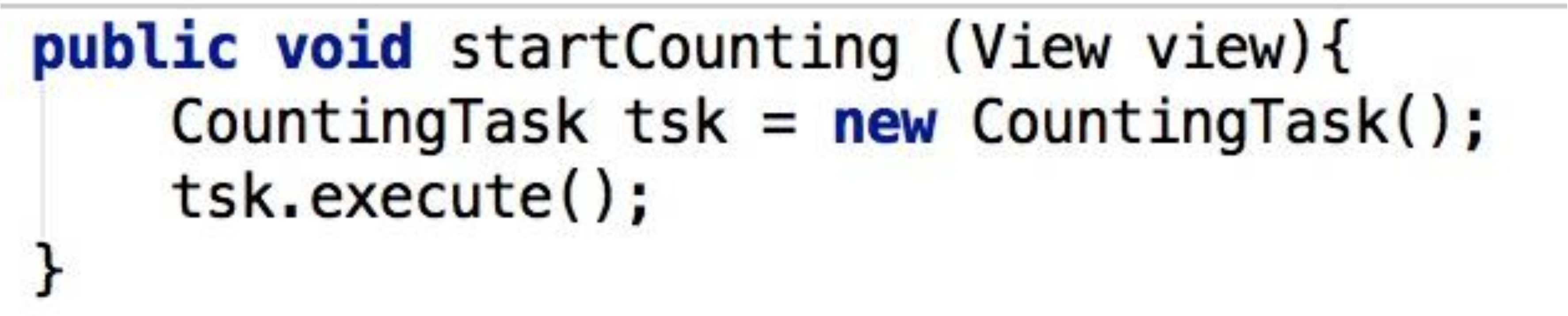
example – implementing AsyncTask

- CountingTaskWithThread



```
class CountingTask extends AsyncTask <Void, Integer, Integer> {

    @Override
    protected Integer doInBackground(Void... params) {
        int i =0;
        while (i<100){
            SystemClock.sleep(250);
            i ++;

            if (i % 5 == 0){
                //Update UI with progress every 5%
                publishProgress(i);
            }
        }
        return i;
    }

    protected void onProgressUpdate(Integer...progress){
        tv.setText(progress[0] + " % Complete!");
    }

    protected void onPostExecute(Integer result)
    {
        tv.setText("Count Complete! Counted to " + result.toString()) ;
    }
}
```

execute() method invoked from UI thread



```
public void startCounting (View view){
    CountingTask tsk = new CountingTask();
    tsk.execute();
}
```

10 min break

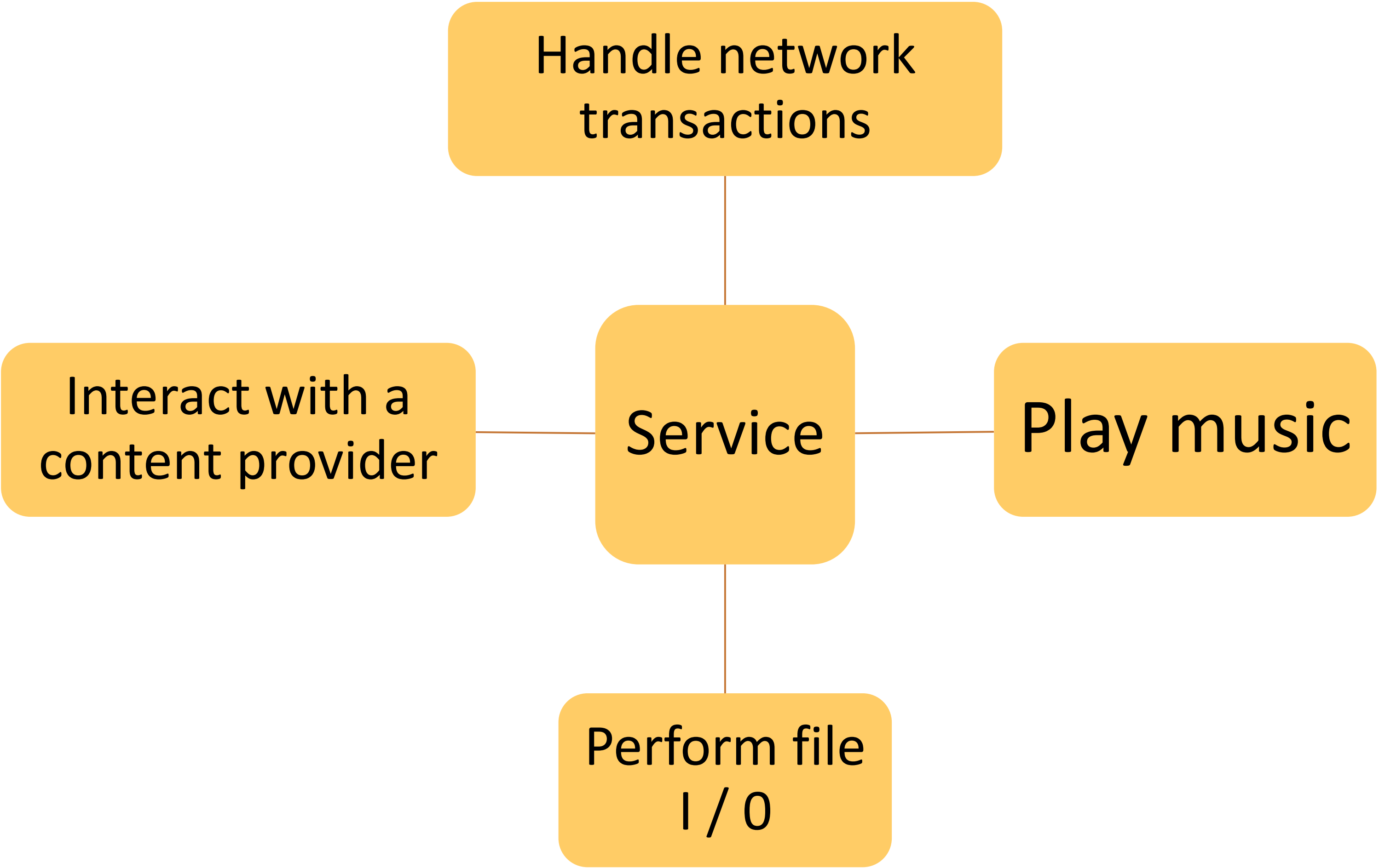
services

- App component
  - Long-running operations in the background
  - No UI

- Another app component can start a service
  - Service will continue to run in the background even if the user switches to another app

- Another app component can bind to a service and interact with it

examples



Handle network transactions

Interact with a content provider

Service

Play music

Perform file I / 0

types of services

Started

- Started by an app component startService()
- Runs in the background *even if the component that started it is destroyed
- Single operation
- Does not return a result to the caller

Bound

- App bind to it: bindService()
- Client-server interface: send requests / get results
- Runs only as long as a component bound to it

We can also have a started service (to run indefinitely) that allows binding

where does the service run

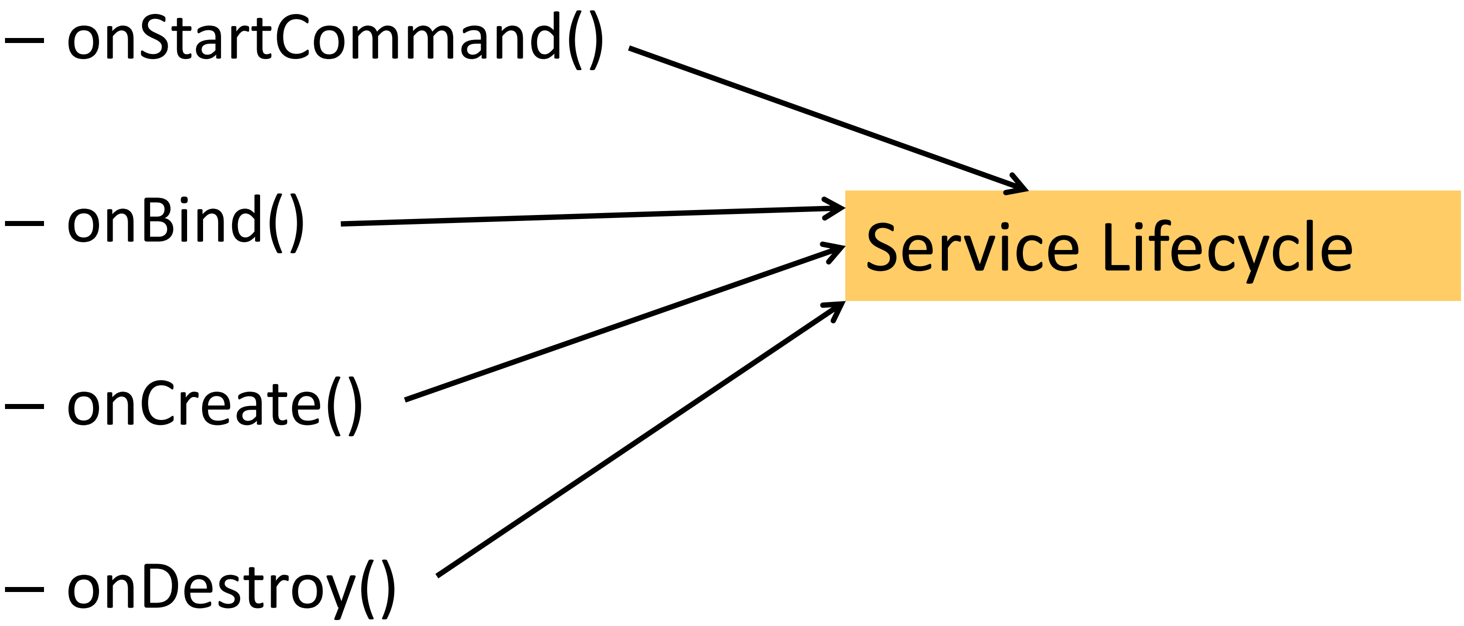
- Service runs in the **main thread** of its hosting process

- Service does not create its own thread

- Service does not run in a separate process

creating a service

- Create a subclass of service
  - Override some callback methods



  - onStartCommand()
  - onBind()
  - onCreate()
  - onDestroy()

Service Lifecycle

service lifecycle

- Much simpler than the lifecycle of activity
- Important: how the service is created and destroyed
  - The service can run in the background without the user being aware

service lifecycle: paths

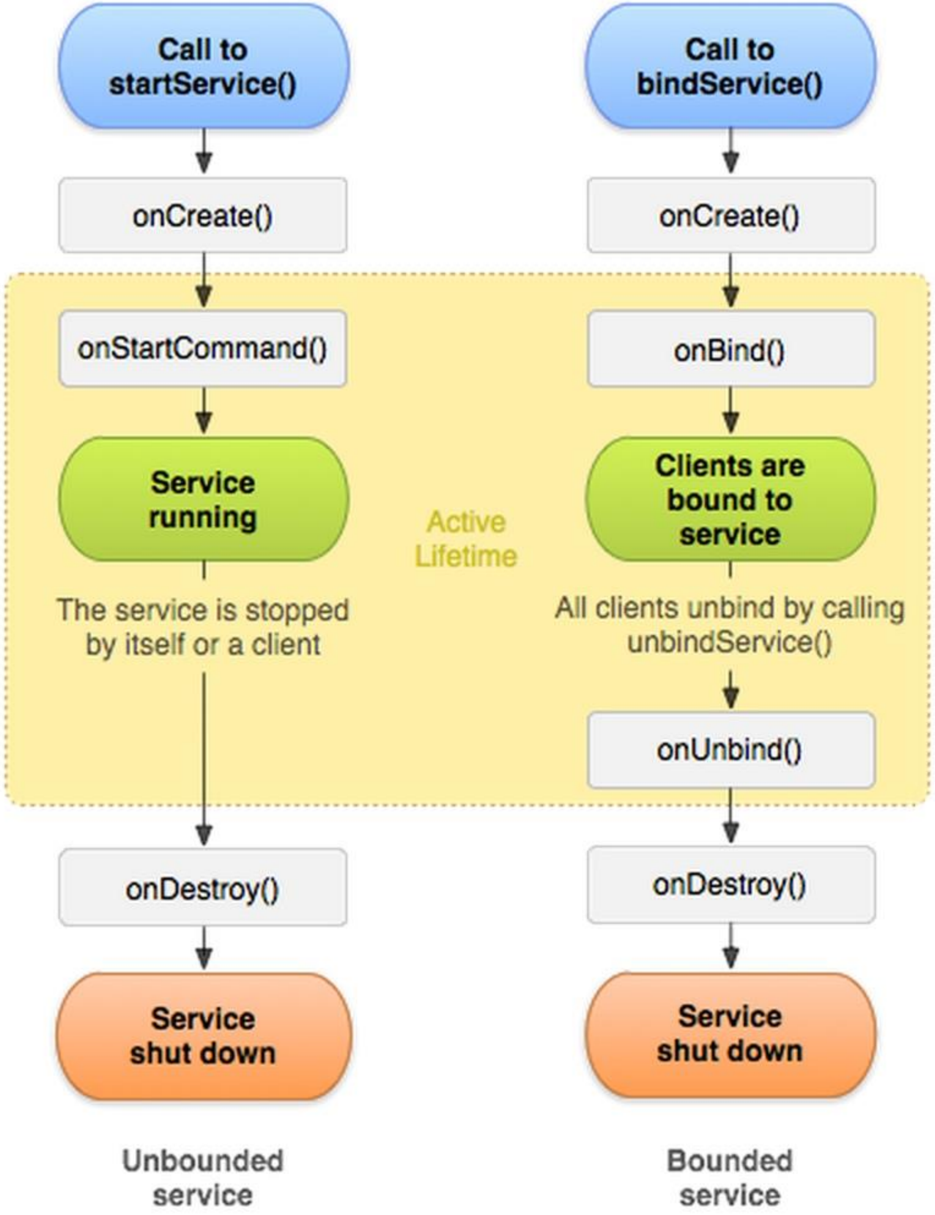
A started service

- 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 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.)

service lifecycle



creating a started service

- Another component starts the service by calling startService()
  - This will result in a call to the service’s onStartCommand() method

- Service has a lifecycle independent of the component that started it

- Service can run in the background indefinitely

- Stopping the service:
  - stopSelf(): service stops itself
  - stopService(): another component can stop it

started service: classes to extend

Service class

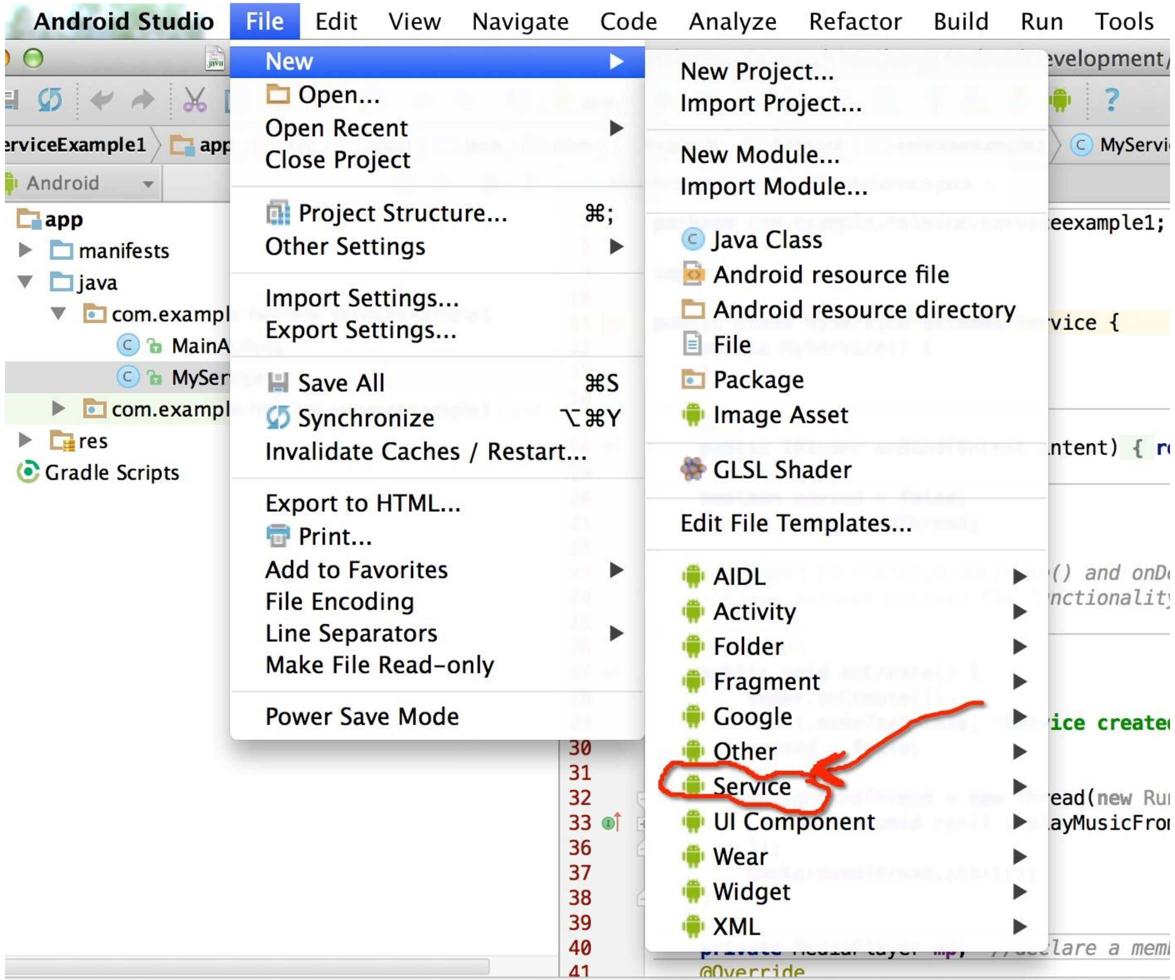
- Base class for all services
- Create new thread to perform the service's intensive work

IntentService class

- Subclass of **Service**
- Uses a worker thread to handle all start requests, one at a time.
- Implement **onHandleIntent()**, which receives the intent for each start request

example: creating a service

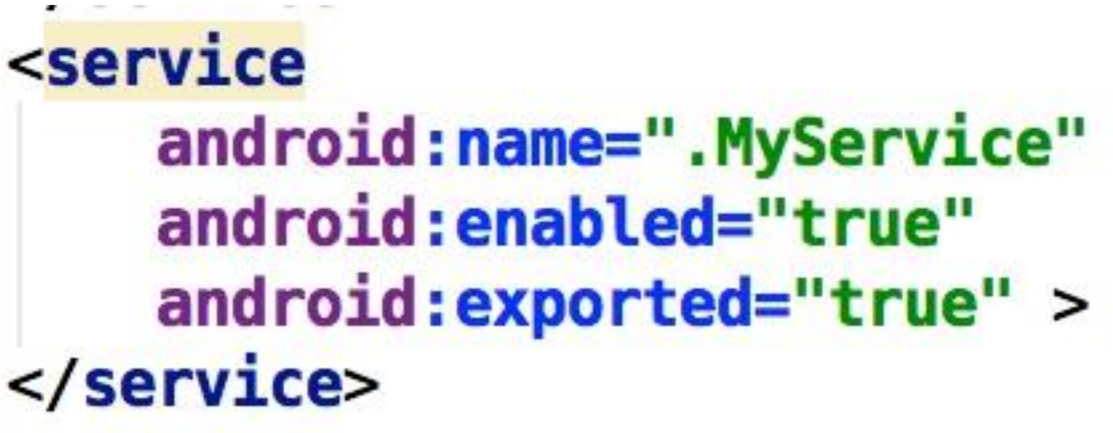
- Create a class to extend Service



service – manifest file

- Declare the Service in the Manifest file

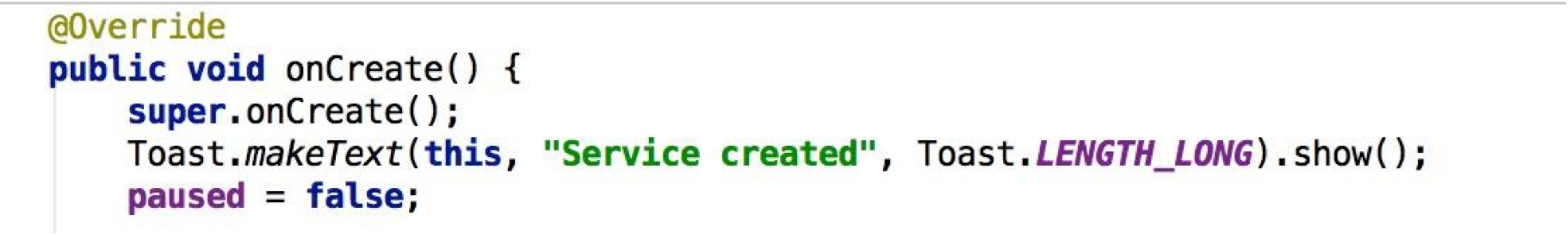
(this step is done automatically in Android Studio, with previous step)



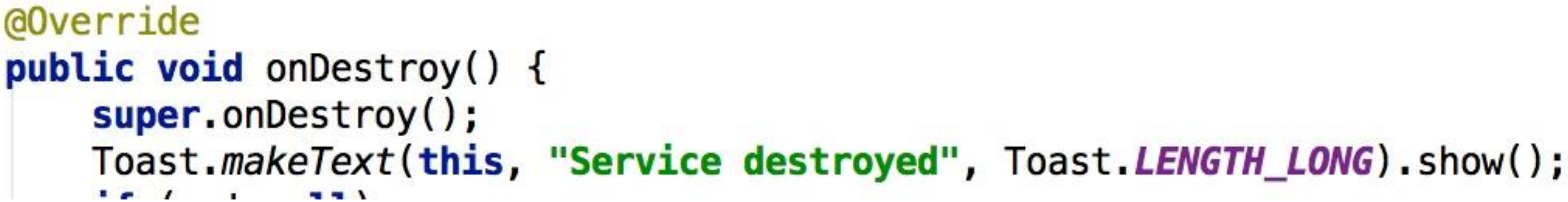
```
<service
    android:name=".MyService"
    android:enabled="true"
    android:exported="true" >
</service>
```

override methods

- Override the onCreate() and onDestroy() methods
  - These methods contain the functionality of the service when started or stopped



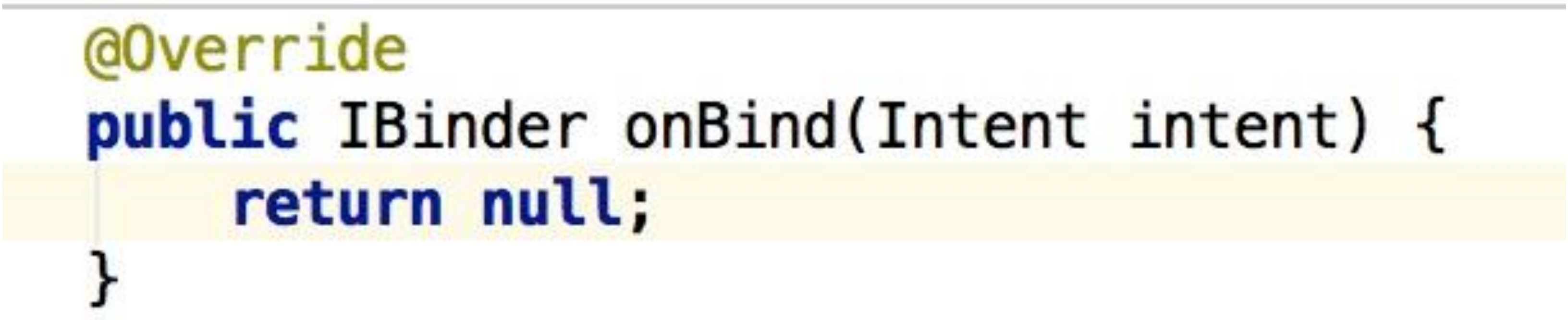
```
@Override
public void onCreate() {
    super.onCreate();
    Toast.makeText(this, "Service created", Toast.LENGTH_LONG).show();
    paused = false;
```



```
@Override
public void onDestroy() {
    super.onDestroy();
    Toast.makeText(this, "Service destroyed", Toast.LENGTH_LONG).show();
```

onBind() methods

- Override the onBind() method
  - This is done for cases when a new component binds to the service after it has been already created



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

activate the service

- Activate the service from an external trigger

- The service cannot run by itself
- It needs to be activated by a separate component in some way

- For example: a component can create an intent to start or stop the service using startService() or stopService()

question

- When would it be useful to have a service stop itself?
- When would it be useful to have another component stop the service?

code example: play music

- Service started or stopped: Toast shown

- onBind(): we override this method, but it is not used (no component binds to the service)

- Create a thread to play music to not block the UI

service behaviour

- The service does not stop when the activity is destroyed (change screen orientation)

- The service does not stop when the activity is paused (press the home button)

- The service, although launched by the activity, runs as its own entity

quiz 3 prep

- Week 5:
  - SharedPreferences
  - Storing and retrieving data from SharedPreferences
- Week 6:
  - SQLite Databases
  - SQLiteOpenHelper
  - Database Queries
- Week 7:
  - AsyncTask
  - Processes and Threads
  - Services

resources

- Processes and Threads: http://developer.android.com/guide/components/processes-and-threads.html
- What is the difference between a process and a thread: http://www.programmerinterview.com/index.php/operating-systems/thread-vs-process/
- Keeping your app responsive: http://developer.android.com/training/articles/perf-anr.html
- Android Services: http://developer.android.com/guide/components/services.html
- AsyncTask: http://developer.android.com/reference/android/os/AsyncTask.html