# Contents

1   Thread of Execution

Android applications use a **single thread model**. A single thread of execution called **main**. It is started when a process is created.

- Handles and dispatches user interface events: drawing the interface, responding to interactions. E.g. onClick...()

- Handles activity lifecycle events: onCreate(), onDestroy. For all components in an application

- HandlerThread

2   Looper and handler

- HandlerThread

  - Extension of Thread with support for a Looper

- Looper

  - Each HandlerThread can have one Looper
  - A Java thread dies when the run method returns
  - Maintains a MessageQueue
  - Looper.loop(): loops through the MessageQueue and processes waiting Messages

- Message

  - A task to be completed
  - Might contain data, reference to a Runnable object

- Handler

  - Attached to a Looper
  - Enqueues messages in the Looper MessageQueue
  - Configurable delivery
  - Handles messages from the MessageQueue
  - Threadsafe
  - One Looper can have many Handlers associated with it
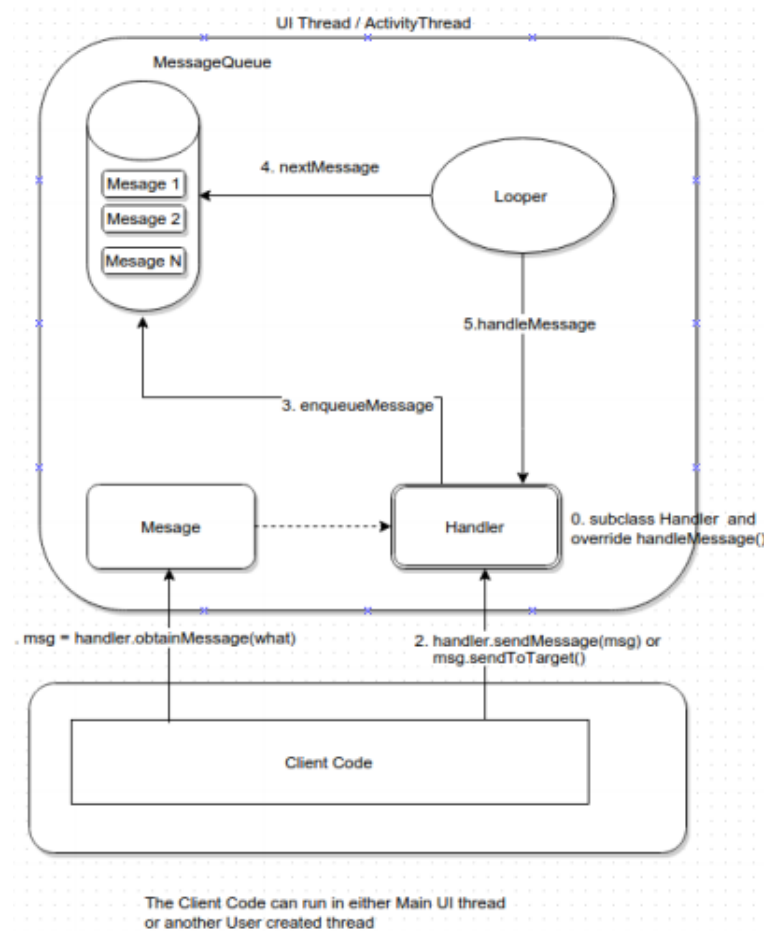
2.1   Splitting threads

- Long (ish) running code that does not involve the UI

  - E.g. an image download
  - Occurs in a separate thread of execution
  - Still tightly coupled to an activity
  - Not allowed to do network communication in the UI thread

- Instantaneous code that does involve the UI

  - E.g. drawing the image that has been downloaded
  - posted to the UI thread responsible for a particular View to execute, logically parceled up as a Runnable object
  - Risk of orphaned threads

3 AsyncTask

A convenience class for making complex asynchronous worker tasks easier. Worker / blocking tasks are executed in a background thread. Can get data back using **results callback**, and it's executed in the UI thread. With each AsyncTask that is spun off, a thread is created and destroyed, which might be a performance issue. We can solve this by implementing implement a thread pool.

4 Services

An Application Component that

- Has no UI

- Represents a desire to perform a longer-running operation. I.e. longer than a single-activity element of the task

- Threads are associated with the activity that started them i.e. could be orphaned

Activities are loaded/unloaded as users move around app, where as services **remain for as long as they are needed**. Can expose functionality for other apps: one service **may be used by many applications**, which allows to avoid duplication of resources

4.1 What services are not

- Not a separate process

  - Runs in the same process as the application in which it is declared (by default)

- Not a thread

  - One thread per Application
  - Handles events for all components
  - If you need to do things in the background, start your own thread of execution

4.2 Uses of Services

- MP3 Playback: Want to play audio while the user is doing other things

- Network Access: long download, sending email, polling email server for new mail.

- Anything that you dont want to interrupt the user experience for

4.3 Creating a Service

Services are designed to support communication with

- Local Activities (in the same process). For example: within VM

- Remote Activities (in a different process). For example IPC

- Multiple components

  - System services underpin much of Android core OS, but wrapped with various APIs

Services are components, similar to an Activities

- Register the service in the **manifest**

- Create a subclass of android.app.Service

- Handle the relevant **lifecycle methods**

4.4   Service lifecycle

There are two ways of spawning a service:

Started (loosely coupled)

- Send an Intent to explicitly start the service with startService()

- c.f. Messages, starting Activities

- Will run / exist in the background indefinitely / until kills itself (Does not return results). For example: C.f email checking.

- Explicitly stop the service with stopService()

- User starts and stops it

Bound (tightly coupled)

- Bind to a service using bindService()

- Will run while any Activities are bound to it

- Actively using it

- Provides an interface (programmatic) for Activities to communicate with the Service

- Operating system starts and stops it

In both cases, if the service is not running it **will be created**. Note both are **the same service**. Different responsibilities for the lifecycle (If I start it, I have to stop it. If OS starts it, OS stops it when it decides to)
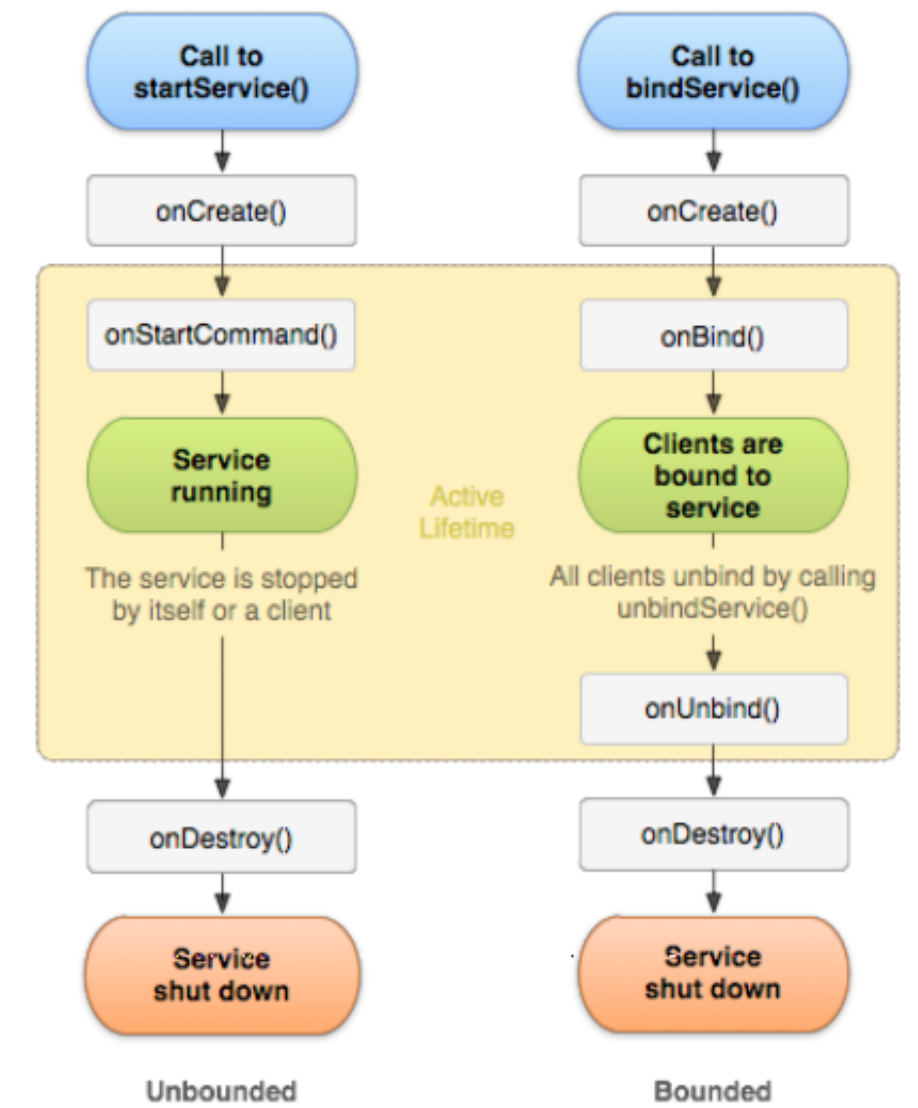
By nature, services are singleton objects (there can be only one). Service used by **many clients**. The Service sub-class object is instantiated if necessary

- `onCreate()` is called

- Either `onStartCommand` or `onBind` will be called depending on how the service has been "called"

- `onCreate` / `onStart` / `onBind` are called in the context of the main UI thread. It now must spawn a worker thread to do any significant work

- Something calls `stopService()`, (could be the OS or user again)

- `onDestroy` can now be used to save work.

4.5   Implementing a service

Generic started service

- Runs persistently (Or stops itself when all work is done)

- Receives messages asking for more work to be done (Delivered via onStartCommand)

IntentService

- A simple, unbound service.

    - It assumes we dont have multiple requests that need to be handled concurrently.
    - Creates a queue of work to be done.
    - HandlerThread, Looper, Handler again.

- Handles one intent at a time to onHandleIntent()

    - Intents delivered via onStartCommand added to a queue
    - Stops the service after all start requests have been handled
    - I.e. sending emails fire and forget

4.6   Terminating services

A Service runs in the background indefinitely, even if the component that started it is destroyed.

- Termination of a service

    - Self-termination (calling stopSelf())
    - stopService() via an Intent
    - System termination (i.e. memory shortage Last recently used again)

- Avoiding termination as a foreground service

    - This is something the user should really know about or is aware of
    - Active in the Status Bar / shows a Notification
    - Is treated as important as a foregrounded Activity
    - startForeground()

Because services run indefinitely, we can use onStartCommand where return value determines how the service should be continued if it is destroyed.

- `START_NOT_STICKY`

    - After onStartCommand returns, do not recreate the service unless there are intents to deliver

- `START_STICKY`

    - Recreate the service and call onStartCommand again, but do not redeliver the last intent

- `START_REDELIVER_INTENT`

    - Recreate the service and call onStartCommand again, redeliver the last intent. Immediately resume the previous job, i.e. downloading a file

Reference section

atomic

Something that "appears to the rest of the system to occur instantaneously" (One operation at a time).**Atomic operation** means an operation that appears to be instantaneous from the perspective of all other threads. You don't need to worry about a partly complete operation when the guarantee applies.