Multithreading

Multithreading

- In the android applications that you have seen thus far you have mainly been dealing with a single thread of execution
  - However, when you start to build more complex applications you may add so much functionality and delays such that your application will suddenly crash for no apparent reason
  - This is because you will have run into a series of restrictions about responsiveness that are defined by the android system
  - So that it can maintian control of the device overall and maintain user responsiveness

Rules imposed by android

- There are two rules that android uses to determine if an app is unresponsive
  - An application must process a user input event in less than 5 seconds
  - A BroadcastReciever must finish processing in less than 10 seconds
    - Used for processing intents coming from other applications

Why android imposes these rules

- For every input event and every time a broadcast reciever is activated a timer is started to monitor the response time of that event
  - If it exceeds the limit then android considers it unresponsive and will kill the application immediately
  - The reason this is imposed is that an unresponsive app will prevent the user from using their device
  - Which leads to a frustrating experience

What happens if you do not respect those rules

- If you do not respect these rules and adapt your applications in response you can expect the following to happen to your application
  - It will be killed by android for taking too long to respond
  - Your application may lose data as a result
  - Eventually users will become frustrated with your application and will uninstall it and replace it by something else

How these rules affect you as a developer

- As a result you are required to keep your application as responsive as possible
  - The general mechanism for doing this is to use some form of multithreading
  - Where expensive and long running operations are run on a seperate background thread
  - Leaves the UI thread available for responding to user input events

Multithreading is not scary

- Multithreading is not a scary concept
  - Permits you to seperate concerns into different threads. One of the most common examples of this is to leave the UI on one thread and do all the networking on another thread
  - The only difficult bit is synchronising the threads
  - There are three forms of multithreading in android that we will explore here: Asynchronus tasks, threads and thread handlers, and finally services

Multicore architectures are now the norm

- Something else that you must accept as a developer now is that most if not all CPU architectures are multicore
  - Thus the other advantage you get from using multiple threads is that you can take advantage of parallel computation
  - Particularly if you are programming any form of mobile games
  - Thus multithreading will be a standard part of programming from here on in

Asynchronus tasks

- An asynchronus task provides a simple framework around the use of a single background thread
  - This background thread is expected to have well defined starting and ending points of execution
  - It will create the thread start executing and upon completion will return a result and kill the thread
  - To use an asynchronus task you are required to extend the AsyncTask class and provide at least one method definition.

What an Asynchronus task is used for

- As an asynchronus task denotes a single execution it is generally used for tasks that are expected to run only once and finish
  - For example a file download where a connection is setup and then torn down after the download
  - Or a simple request reply behaviour on a short network connection that does not happen very often
  - Thus the structure of the Asynchronus task maps to this set of operations well

Structure of an asynchronus task

- As a result the asynchronus task is constructed in such a way to reflect this behaviour
  - You are not required to implement all of the methods that you see here
  - The bare minimum that you are required to implement is the doInBackground() method
    - But generally most people will also implement the onPostExecute() method too

onPreExecute() method

- Called on the UI thread before the asynchronus task begins execution
  - Normally used to setup the task before it starts executing
  - Generally a good idea to setup progress bars here to give the user an indication as to how long the task has remaining
  - Tasks that show no indication of their progress will be assumed as to not be functioning by the user

onPostExecute() method

- Called on the UI thread after the asynchronus task finishes execution
  - The result of the background operation is passed as a parameter to this method
  - As it is assumed that it will be used to update the UI or that it will be used by the application
  - After this the AsyncTask has finished its work and will be destroyed.

doInBackground() method

- Invoked on the background thread after the onPreExecute() method has finished executing. This method is used to perform the background computation
  - Here is the method that you will use to perform your task such as downloading a file
  - All parameters that were specified in the construction of the task will be passed to this method to be processed in the background
  - This is where the majority of the processing of the AsyncTask will take place

onProgressUpdate() method

- Every now and again the doInBackground method should provide an update of the progress of the task. When it calls the publishProgress() method it will indirectly invoke onProgressUpdate()
  - Will be invoked on the UI thread whenever it is called
  - There is no guarentees as to when this method will be called for you
  - Only used to update the UI with the current progress of the task

How the async task fits in with the lifecycle handlers

- Generally because an asynchronus task is done in the background and will run only once there is no need to pause or stop it if the activity becomes paused or stopped
  - The async task only needs to be stopped if the application is to be destroyed.
  - There is the possibility though that a user may want to cancel a task (i.e. don't want to go ahead with operation or don't have time)
  - In which case there is a cancel method that can be called on the UI thread to stop the task at the earliest possible moment

Threads and handlers

- Threads are the bare unit of execution within an android application
  - Your applicaton is permitted to have as many threads as is necessary
  - These general threads have no restriction on their activity. Some threads will run only once and others will last for the lifetime of the application
- However background threads are not permitted to interact with the UI directly so a handler object is used as a communication system between the UI thread and all background threads.

What a general thread should be used for

- A general thread should be used for any application processing that can potentially last forever (e.g. game threads)
  - Another example of this is if you require an open network connection for the lifetime of your application
  - Because there is no definite end to tasks like these we need to use a general thread
  - However, because this is based on the Java threading model there are some quirks that must be accounted for (will be explained later)

What the purpose of a handler is

- The handler is used to generate and communicate messages from background threads to the UI thread
  - This mechanism is used by background threads to ask the UI thread to update the UI
  - Uses Message objects that contain at least two integers and potentially extra objects
  - As messages are used often the system will maintain a message pool to prevent allocation and deallocation of messages during the lifetime of all applications

Graphics contexts and the UI thread

- The reason we need such a communication mechanism is that every application is only permitted to have a single thread with access to the graphics context
  - The graphics context is what enables a thread to draw and update the screen
  - This is a restriction that is imposed by OpenGL
  - Thus if a background thread wants to update the UI it must communicate with the UI through the handler object

How to communicate between a thread and the UI thread

- The activity that contains the UI thread must create a subclass of a handler object that adds in definitions for the messages for communication between the various threads
  - This handler is automatically attached to the activity. And must be attached to the background threads
  - When the background thread wishes to communicate it will get a message from the pool and will add the necessary data to it before asking the handler to send the message
  - The handler will then be invoked on the UI thread in which case it must then go and update the UI as necessary.

How the java threading model works

- Java threads are expected to run once and only once
  - It is not possible to pause, resume or restart threads as all of these mechanisms have been deprecated.
  - The only thing you can do with a thread is temporarily pause execution by sending it to sleep for a specified time period
  - After it finishes executing you are required to kill the thread and use another in its place

The caveats of the java threading model

- The issue with a model like this is that whenever you have to temporarily pause a thread and have to restart it it is not easy
  - With the standard java Thread class everytime you would pause you would have to reove your threads and then create new ones in their place
  - Because the suspend and resume behaviour is depricated
  - You may want to have a look at the thread pools instead

How this fits in with the lifecycle handlers

- As was stated in a previous lecture threads fit into the lifecycle handlers in the pause and resume methods
  - Generally you will pause all of your threads in the onPause() method
  - And will resume them in the onResume() method

Services

- The other alternative is to use a service
  - Unlike asynchronus tasks or threads theses can be detached from an applicaton or activity and still run in the background
  - They will usually run in their own seperate process
  - Communication will either happen through IPC or RPC

What a service should be used for

- Generally a service is used when you need to keep something happening in the background while the application is not running
  - One of the most common examples of this is polling for notifications for a user
    - How the facebook app (amongst others) are capable of sending you notifications even though the application is not active
  - Or used for monitoring something without the need for a foreground application. Examples would include battery monitors

Services and the manifest file

- Like activities all services must be declared in the manifest file of an android application
  - Again if an application tries to start a service that is not listed there it is considered a security risk and is killed immediately
  - Using the <service /> tag

The Service class

- If you wish to create a service in android you are required to extend the service class or one of its derivatives
  - There are multiple methods that can be overridden and redefined in a service but we will only cover the most important ones here
  - onStartCommand(): used to start the service when another activity has asked android to start the service
  - onBind() called by the system whenever another object wishes to make a long term connection to the service. The binding will be in the form of an RPC mechanism

The Service class

- onCreate(): similar in function to that of the Activity class. Used to allocate the necessary resources for the service
- onDestroy(): does the opposite of onCreate()

- Services will either be stopped by an external activity or it is capable of stopping itself by calling stopSelf()

Why priorities are useful here

- Generally it is a good idea to give a background service a low process priority
  - As processing power should be given to the foreground activity
  - By assigning a lower priority to your service it will generally take CPU time when there is currently no user interaction with the device.
  - Helps to maintain a responsive system

Service bindings and the Binder class

- It is possible for other applications including your own to bind to your service by using an RPC mechanism
  - To expose such RPCs android will require developers to write an interface in AIDL (Android Interface Definition Language)
  - Running this through the aidl tool will generate a stub that will contain the necessary RPC mechanism
  - You will be required to build the skeleton around it that will define how each method in the interface works

Service bindings and the Binder class

- You will do this by including the AIDL interface and extending the binder class
  - Once this is done the stub can be exposed to clients. Clients can then bind to this interface and communicate with your service that way
  - Makes the communication look like standard object oriented method calls
  - To ease development effort