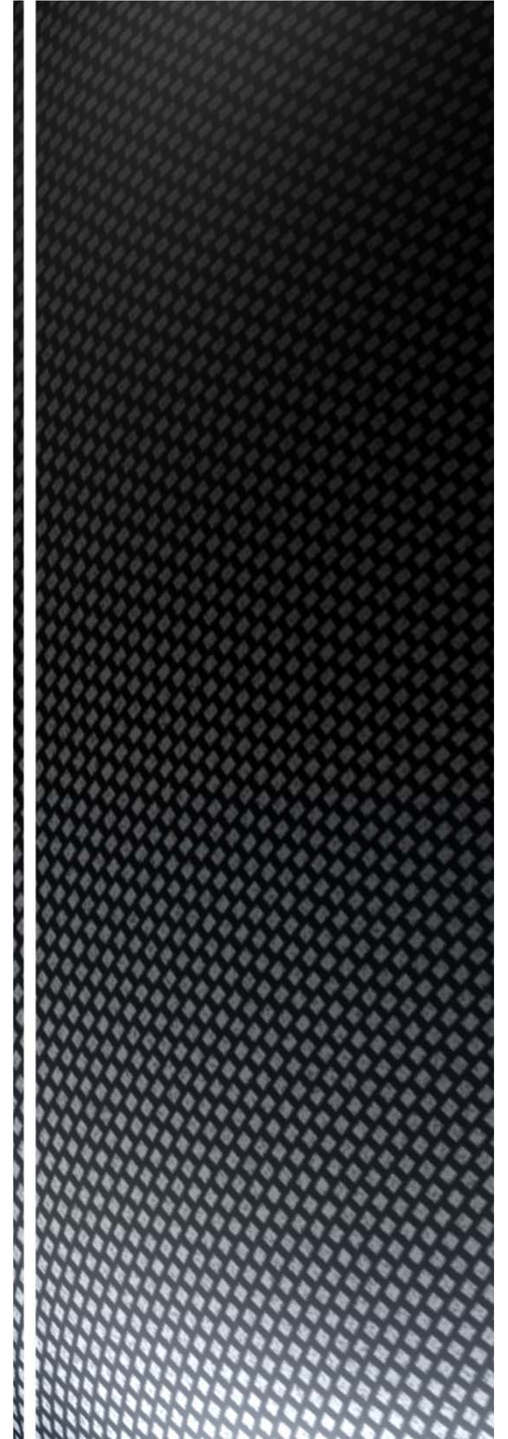
Android

THREADS



AsyncTask

AsyncTask habilita el uso fácil y correcto del UI Thread.

Es decir, permite dividir el trabajo entre el Background Thread y el UI Thread.

Permite realizar "operaciones pesadas" en background threads y publicar los resultados en el UI Thread sin necesisad de declarar ni manipular Threads o Handlers.

El Background Thread realiza la "operación pesada" y opcionalmente puede reportar su progreso.

El UI Thread, a través del AsyncTask es responsable de:

- El setup de la "operación pesada"
- La publicación del progreso cuando sea reportado
- De completar la operación tras la finalización del Background Thread

El Background Thread, a través del AsyncTask es responsable de:

- Realizar el trabajo
- Comunicar opcionalmente el avance del trabajo

AsyncTask

AsyncTask es una clase genérica que tiene tres parámetros

```
class AsyncTask<Params, Progress, Result> {
....
}
```

Params: Tipo de los parámetros de entrada al AsyncTask – Tipo usado en el Background

Progress: Tipo de los datos usados para informar del progreso

Result: Tipo del resultado que procesará el AsyncTask

AsyncTask Workflow

El método onPreExecute se ejecuta en el UI Thread antes de que arranque el método doInBackground

El método onPreExecute normalmente establece la “operación pesada”

El método doInBackground realiza la “operación pesada”.

Recibe una lista variable de parámetros de entrada y devuelve un resultado de tipo <Result>.

Durante su ejecución puede opcionalmente llamar al método publishProgress, pasándole una lista de valores que indicarán sobre el progreso de la “operación costosa”

Si el Background Thread llama a publishProgress entonces se deben realizar llamadas a onProgreshUpdate en el UI Thread mientras el Background Thread siga ejecutando.

Finalmente onPostExecute es llamado en el UI Thread con el resultado proporcionado por el Background Thread.

E0903-ThreadingAsyncTask

AsyncTask Workflow

Cancelar un AsyncTask es posible vía la llamada al método cancel()

En cualquier momento se puede llamar desde el UI Thread al método cancel() del AsyncTask

- Esto ocasiona que las siguientes llamadas a isCancelled() devuelvan true.
- Si permitimos la cancelación de la tarea deberemos preguntar por isCancelled() durante la ejecución de la "operación pesada"
- Cuando la "operación pesada" termine le llamará a onCancelled(Object) en vez de a onPostExecute(Object)

Si no queremos que el usuario pueda interactuar con el UI Task mientras se ejecuta la "operación pesada" podemos hacer uso de un ProgressDialog.

Handler

La clase Handler está diseñada para dar soporte a la gestión del trabajo entre dos threads cualesquiera, no siendo el UI Thread necesariamente uno de ellos.

Un Handler está asociado con un thread específico.

Un thread puede **transferir trabajo** para otro thread mediante el envío de un Message o un Runnable al Handler asociado al thread.

Los Runnables los usamos cuando sabemos exactamente cuales son pasos de ejecución a realizar, pero queremos que sean ejecutados en el Thread receptor.

Un Message es una clase que puede contener datos tales como un código de mensaje, un objeto arbitrario o valores enteros.

Se usan por el thread emisor para indicarle al receptor qué operación realizar, dejando al Handler la implementación de ésta.

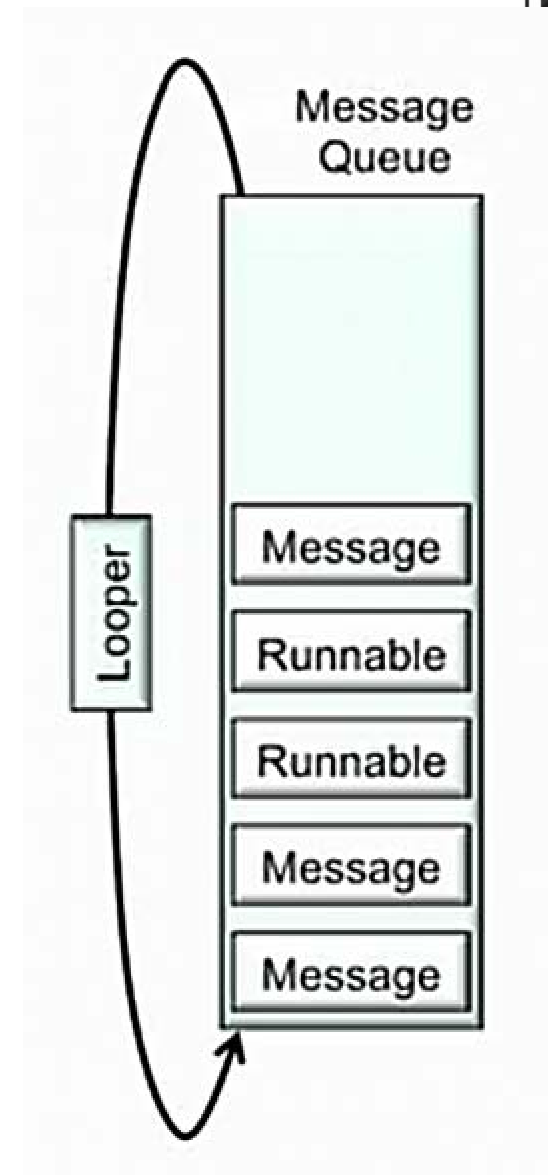
Handler Architecture

Los Handlers gestionan los Messages y los Runnables

Cada thread está asociado con un MessageQueue y un Looper.

El MessageQueue es una cola de datos que guarda Messages y Runnables.

El Looper saca estos Messages y Runnables de la MessageQueue y los despacha según proceda.

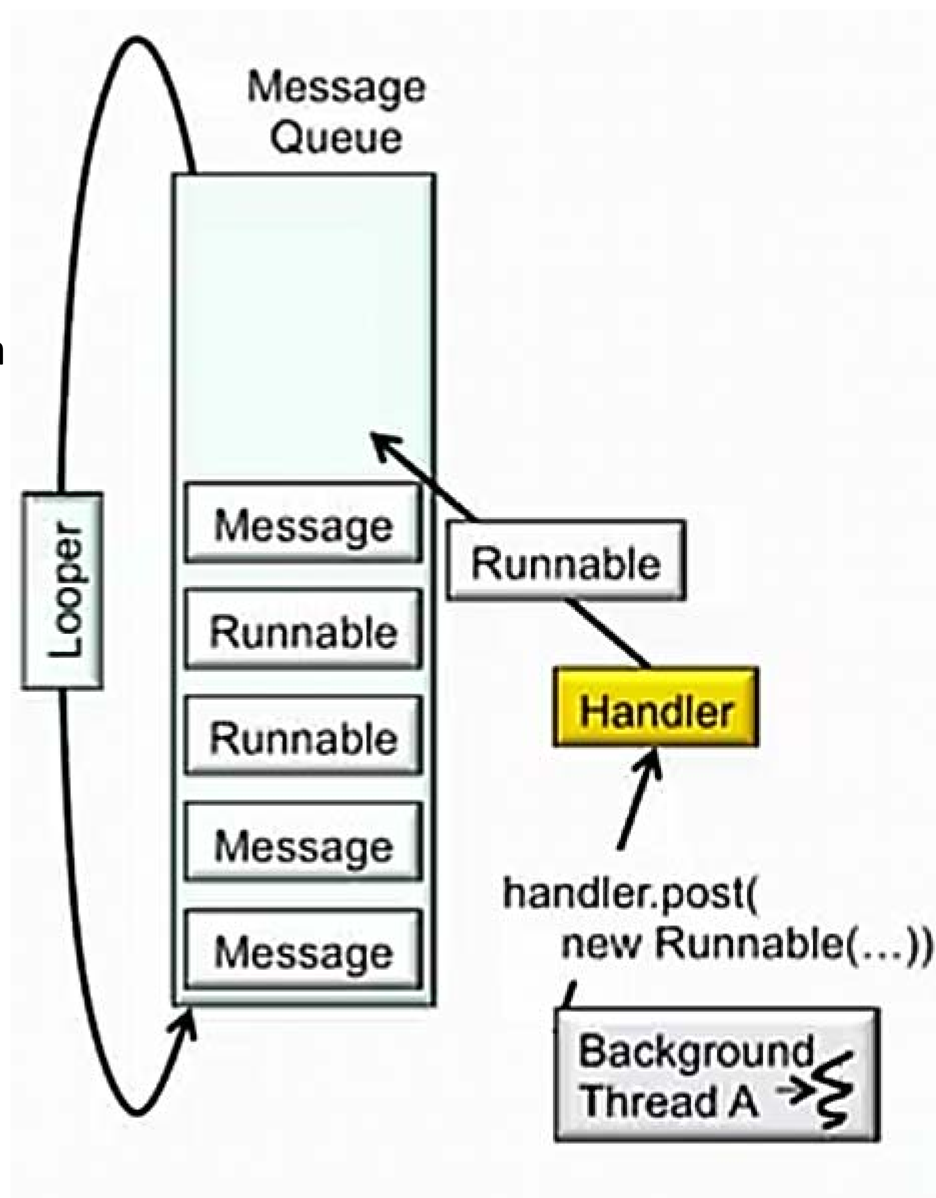


Handler Architecture

Un Thread A crea un Runnable.

El thread A utiliza un Handler para enviarlo a su thread asociado.

Esto hace que el Runnable se inserte en la MessageQueue del thread asociado al Handler.



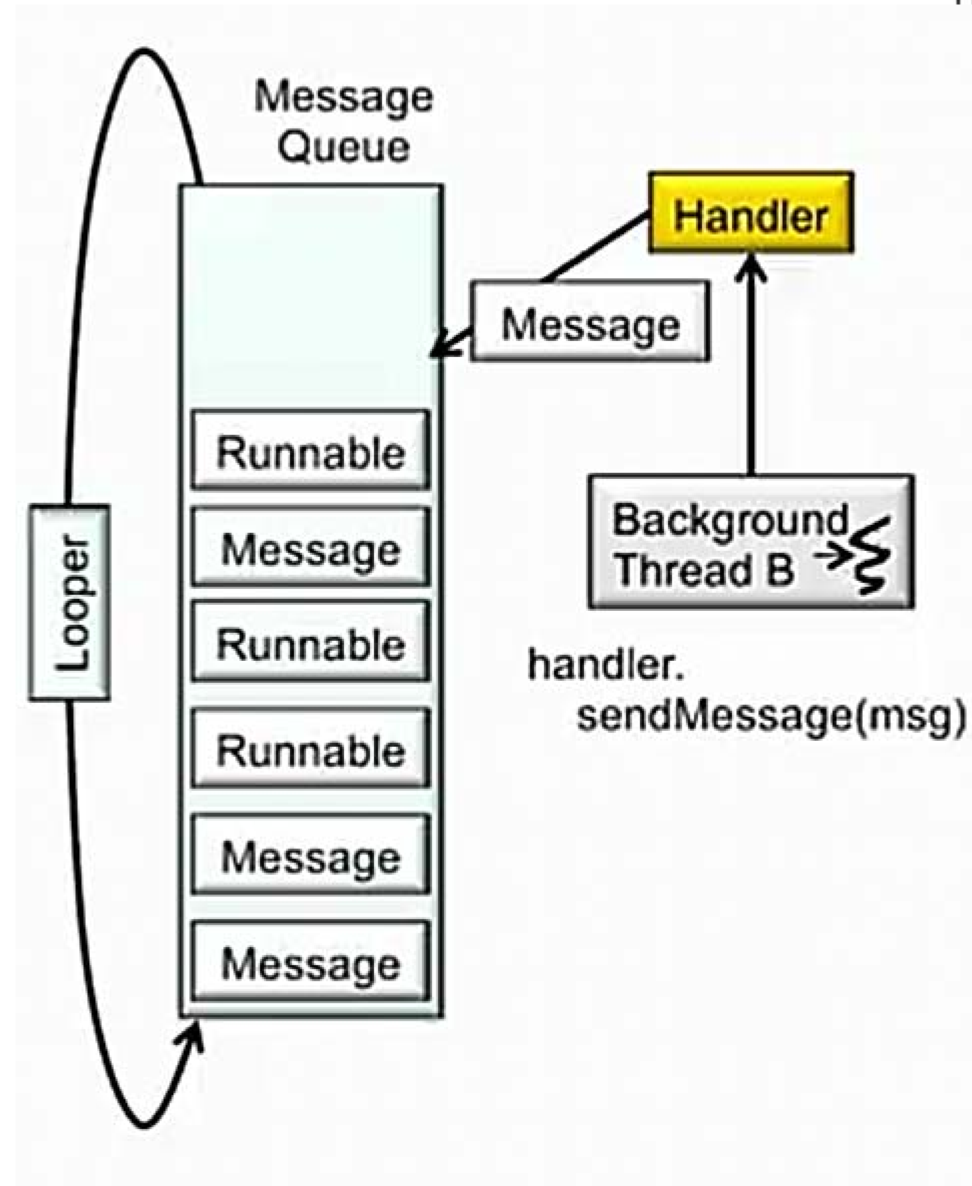
Handler Architecture

Con los Messages ocurre lo mismo.

Cuando un Thread B utiliza un Handler para enviar un Message a otro thread, el Message se coloca en el MessageQueue del thread receptor.

El Looper está esperando a la llegada de trabajos (Message o Runnables) al MessageQueue.

Cuando los trabajos llegan el Looper reacciona en función del tipo de trabajo de que se trate.

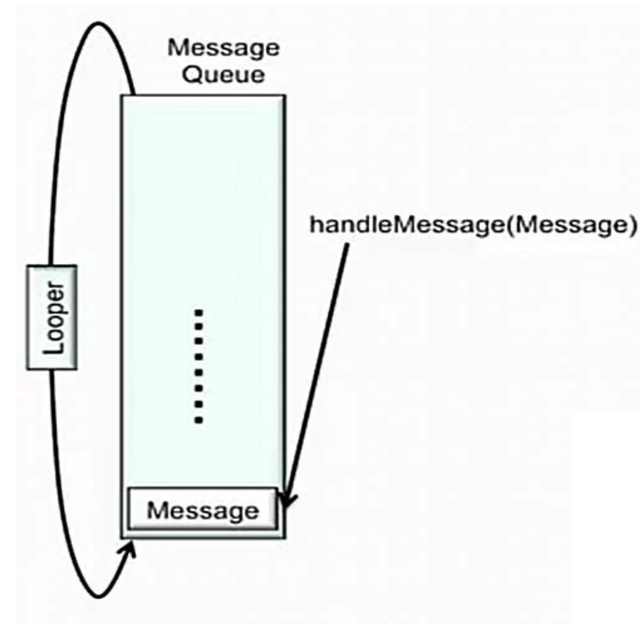


Handler Architecture

Si el trabajo es:

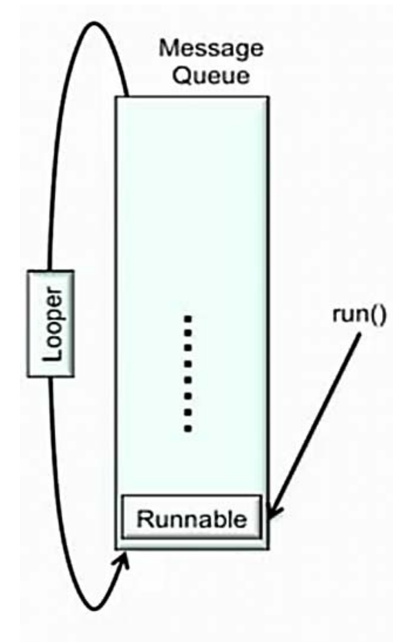
Un Message:

El Looper llamará al método handleMessage del Handler asociado al thread pasándole el Message.



Un Runnable:

Llamará al método run del propio Runnable.



Handler Architecture

Métodos utilizados para enviar Runnables a un Handler:

```
boolean post(Runnable r)
```

Añade el Runnable al MessageQueue

```
boolean postAtTime(Runnable r, long uptimeMilis)
```

Añade el Runnable al MessageQueue para ser ejecutado en un instante concreto.

```
boolean postDelayed(Runnable r, long uptimeMilis)
```

Añade el Runnable al MessageQueue para ser ejecutado pasado un tiempo definido.

Handler Architecture

Para enviar un Message primero hay que crearlo.

Llamando al método de la clase Handler, `Handler.obtainMessage()` obtenemos un Message para el cual el Handler ya está asociado.

Llamando al método de la clase Message, `Message.obtain()` que nos devuelve un Message del global pool sin tener asociado el Handler ni los datos. Hay muchas variantes para realizar esto, ver la documentación.

Para enviar el Message se puede usar:

```
public final boolean sendMessage (Message msg)
```

Para enviar instantaneamente el Message

```
public final boolean sendMessageAtFrontOfQueue (Message msg)
```

Para enviar instantaneamente y colocarlo en la cabeza de cola

```
public boolean sendMessageAtTime (Message msg, long uptimeMillis)
```

Para enviar el Message a la cola para su ejecución en un instante definido.

```
public final boolean sendMessageDelayed (Message msg, long
delayMillis)
```

Para enviar el Message a la cola para su ejecución tras un Delay a partir del momento de envío.

E0906-ThreadingHandlerRunnable

E0907-ThreadingHandlerMessages