Hilos

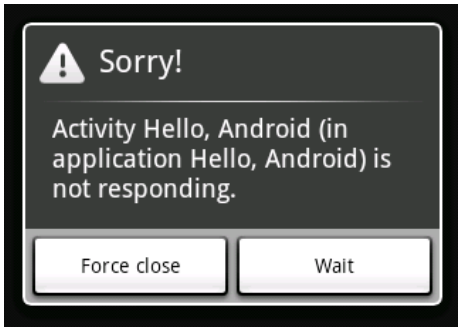
Índice

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

1. Hilos de ejecución

En la programación de dispositivos móviles se suele dar el caso de tener que ejecutar una operación en segundo plano para no entorpecer el uso de la aplicación, produciendo incómodas esperas. Algunas operaciones, como la descarga de un archivo grande, son lentas de forma evidente. Otras operaciones que pueden parecer rápidas, a veces también resultan lentas, si dependen del tamaño de un archivo o de algún factor externo como red. Los dispositivos continuamente pierden calidad de la señal o pueden cambiar de Wifi a 3G sin preguntarnos, y perder conexiones o demorarlas durante el proceso. Los hilos también sirven para ejecutar simultáneamente tareas o para operaciones que se ejecutan con una periodicidad temporal determinada.

En cuanto a la interfaz gráfica, los hilos son fundamentales para una interacción fluida con el usuario. Si una aplicación realiza una operación lenta en el mismo hilo de ejecución de la interfaz gráfica, el lapso de tiempo que dure la conexión, la interfaz gráfica dejará de responder. Este efecto es indeseable ya que el usuario no lo va a comprender, ni aunque la operación dure sólo un segundo. Es más, si la congelación dura más de dos segundos, es muy probable que el sistema operativo muestre el diálogo ANR, "Application not responding", invitando al usuario a matar la aplicaicón:



Mensaje ANR

Para evitar esto hay que crear otro hilo (`Thread`) de ejecución que realice la operación lenta.

1.1. Creación y ejecución

Un hilo o `Thread` es un objeto con un método `run()`. Hay dos formas de crearlos. Una es por herencia a partir de `Thread` y la otra es implementando la interfaz `Runnable`, que nos obliga a implementar un método `run()`.

```
public class Hilo1 extends Thread {
        @Override
        public void run() {
                while(condicion_de_ejecucion){
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
                        //Realizar operaciones
                        //...
                        try {
                                // Dejar libre la CPU durante
                                // unos milisegundos
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
}
```

```
public class Hilo2 implements Runnable {
        @Override
        public void run() {
                while(condicion_de_ejecucion){
                        //Realizar operaciones
                        //...
                        try {
                                // Dejar libre la CPU durante
                                // unos milisegundos
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
}
```

La diferencia está en la forma de crearlos y ejecutarlos:

```
Hilo1 hilo1 = new Hilo1();
hilo1.start();

Thread hilo2 = new Thread(new Hilo2());
hilo2.start();
```

Una forma todavía más compacta de crear un hilo sería la declaración de la clase en línea:

```
new Thread(new Runnable() {
    public void run() {
      //Realizar operaciones ...
    }
}).start();
```

Si el hilo necesita acceder a datos de la aplicación podemos pasárselos a través del constructor. Por ejemplo:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        new Hilo1(getApplicationContext());
}

public class Hilo1 extends Thread {
        Context context;
        public Hilo2Thread(Context context){
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
                this.context = context;
                this.start();
        }
        @Override
        public void run() {
                while(condicion_de_ejecucion){
                        //Realizar operaciones
                        //...
                        try {
                                // Dejar libre la CPU durante
                                // unos milisegundos
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
}
```

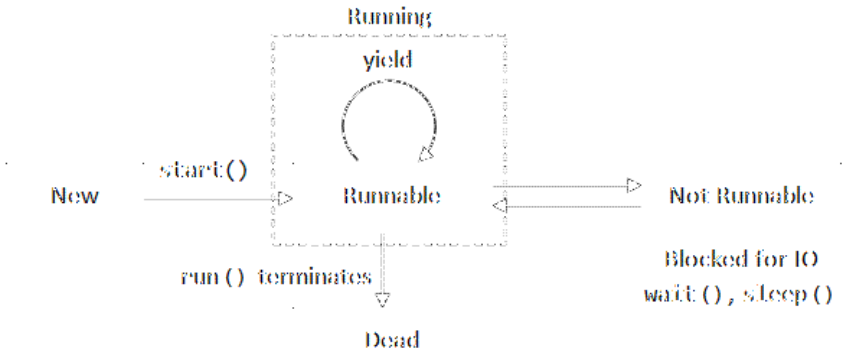
En el anterior ejemplo también se ha ejecutado el hilo desde su propio constructor, de manera que la creación del objeto `Hilo1` ha sido suficiente para ejecutarlo.

1.2. Ciclo de vida y finalización

Los hilos tienen un ciclo de vida, pasando por diferentes estados:

- New: El primer estado de un hilo recién creado. Permanece en este estado hasta que el hilo es ejecutado.
- Runnable: Una vez ejecutado pasa a este estado, durante el cuál ejecuta su tarea.
- Not runnable: Estado que permite al hilo desocupar la CPU en espera a que otro hilo termine o le notifique que puede continuar, o bien a que termine un proceso de E/S, o bien a que termine una espera provocada por la función `Thread.sleep(100);`. Tras ello volverá al estado Runnable.
- Dead: Pasa a este estado una vez finalizado el método `run()`.

El siguiente diagrama resume las transiciones entre los estados del ciclo de vida de un hilo.



Ciclo de vida de un hilo

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

Es muy importante asegurar que un hilo saldrá de la función `run()` cuando sea necesario, para que pase al estado dead. Por ejemplo, si el hilo ejecuta un bucle `while`, establecer a `true` una variable booleana como condición de que se siga ejecutando. En el momento que deba terminar su ejecución es suficiente con poner esa variable a `false` y esperar a que el hilo de ejecución llegue a la comprobación del bucle `while`.

No hay un método `stop()` (está deprecated, por tanto no hay que usarlo), en su lugar el programador debe programar el mecanismo que termine de forma interna la ejecución del hilo. Sin embargo si un hilo se encuentra en estado Not Runnable, no podrá hacer el `return` de su función `run()` hasta que no vuelva al estado Runnable. Una forma de interrumpirlo es usar el método `hilo1.interrupt();`.

En Android cada aplicación se ejecuta en su propia máquina virtual. Ésta no terminará mientras haya hilos en ejecución que no sean de tipo daemon. Para hacer que un hilo sea daemon se utiliza la función:

```
hilo1.setDaemon(true);
hilo1.start();
```

Aún así es recomendable salir de la función `run()` cuando la lógica del programa lo considere oportuno.

1.3. Hilos y actividades

La información que le haga falta a un hilo se puede pasar por parámetro al construir el objeto, o bien a posteriori con algún método setter del hilo. Otra alternativa más compacta sería que la propia actividad implemente `Runnable`.

```
public class HiloActivity extends Activity
                                        implements Runnable{
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                //Iniciar el segundo hilo de ejecución:
                (new Thread(this)).start();
        }

        @Override
        public void run() {
                while(condicion_de_ejecucion){
                        //Realizar operaciones
                        //...
                        try {
                                // Dejar libre la CPU
                                // durante unos milisegundos
                                Thread.sleep(100);
                        } catch (InterruptedException e) {
                                return;
                        }
                }
        }
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
}
```

Nótese que en el ejemplo anterior se podrían crear más hilos de ejecución creando más instancias del hilo e iniciándolas con `(new Thread(this)).start();` pero compartirían los mismos datos correspondientes a los campos de la clase `HiloActivity`.

Nota:
Al cerrar una `Activity` los hilos secundarios no terminan. Al pausarla tampoco se pausan. El programador debe encargarse de eso.

Es importante establecer bien la condición de terminación o de ejecución del hilo, ya que en una aplicación de Android, al cerrar una `Activity`, no se finalizan automáticamente los hilos secundarios de ejecución. De eso se tiene que encargar el programador.

Por defecto para cerrar una actividad hay que pulsar el botón hacia atrás del dispositivo Android. Sin embargo si se pulsa la tecla Home o se recibe una llamada de teléfono la actividad pasa a estado pausado. En este caso habría que pausar los threads en el método `onPause()` y reestablecerlos en `onResume()`, con los mecanismos de sincronización `wait` y `notifyAll`. Otra manera aplicable en ciertos casos sería terminar los hilos y volver a crearlos al salir del estado de pausa de la actividad.

1.4. Prioridades

Cada hilo tiene asociada una prioridad que se puede cambiar con el método:

```
hilo1.setPriority(prioridad)
```

El parámetro es un valor entero entre `Thread.MIN_PRIORITY` (que vale 1) y `Thread.MAX_PRIORITY` (que vale 10). El efecto de la prioridad de los hilos sólo puede llegar a apreciarse si varios hilos están simultáneamente en estado "Running". Si el hilo tiene estados "Waiting", o bien para liberar CPU, o bien porque esperan E/S, o bien porque esperan otro hilo, en estos intervalos de tiempo el procesador estaría liberado para ejecutar el otro hilo, independientemente de su prioridad.

2. Sincronización de hilos

2.1. Acceso a datos compartidos

Cuando varios hilos pueden acceder simultáneamente a algún dato, hay que asegurarse de que no lo hagan simultáneamente en el caso de estar modificándolo. Para ello utilizaremos el modificador `synchronized` sobre métodos:

```
        public synchronized void incrementa(int incremento){
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
                this.valor += incremento;
        }
```

De esta manera si un hilo empieza a ejecutar este método, todos los demás hilos que intenten ejecutarlo tendrán que esperar a que el primero que entró salga del método.

Una manera alternativa de escribirlo hubiera sido utilizando de manera explícita la variable `lock` del objeto:

```
        public synchronized void incrementa(int incremento){
            synchronized(this){
                        this.valor += incremento;
                }
        }
```

Hay estructuras de datos cuyos métodos ya están sincronizados, por ejemplo `Vector` que es la versión sincronizada de `List`. Sin embargo hay que llevar cuidado si las operaciones que realizamos no son atómicas, como por ejemplo obtener un valor, tratarlo, y después modificarlo. Si varios hilos acceden a esta secuencia de código, hay que sincronizarla para evitar llegar a estados inconsistentes de los datos.

Cuando se accede a otro tipo de recursos, como a un archivo en disco, también es fundamental asegurar la sincronización. Sin embargo si la sección crítica va a ser muy grande hay que tener en cuenta que esto puede disminuir el rendimiento. En general no hay que sincronizar si no es necesario.

2.2. Esperar a otro hilo

En el acceso a datos compartidos se ha visto cómo hacer que los demás hilos esperen a que otro salga de una sección crítica. Java también proporciona en la clase `Thread` métodos que permiten bloquear un hilo de manera explícita y desbloquearlo cuando sea necesario, son los métodos `wait()` y `notify()` (o `notifyAll()`).

Para evitar interbloqueos ambos métodos deben ser llamados siempre desde secciones de código sincronizadas. En el siguiente ejemplo una estructura de datos está preparada para ser accedida simultáneamente por varios hilos productores y consumidores. Es capaz de almacenar un único objeto pero si un consumidor accede y la estructura no contiene ningún objeto, dicho consumidor queda bloqueado hasta que algún productor introduzca un objeto. Al introducirlo, desbloqueará de manera explícita al hilo consumidor. Durante dicho bloqueo la CPU está completamente libre del hilo consumidor.

```
public class CubbyHole {
    private Object cubby;

    //El productor llamaría a este método.
    public synchronized Object produce(Object data) {
        Object ret = cubby;
        cubby = data;

        // Desbloquear el consumidor que
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
        // esté esperando en consume().
        notifyAll();

        return ret;
    }

    //El consumidor llamaría a este método
    public synchronized Object consume()
                    throws InterruptedException {
        // Bloquear hasta que exista
        // un objeto que consumir
        while (cubby == null) {
            // Libera el lock del
            // objeto mientras espera y
            // lo cierra al seguir ejecutándose
            wait();
        }

        Object ret = cubby;
        cubby = null;

        return ret;
    }
}
```

Otra forma de sincronización es la de esperar a que un hilo termine para continuar con la ejecución del presente hilo. Para este propósito se utiliza el método `join(Thread)`:

```
        Thread descarga1 = new Thread(new DescargaRunnable(url1));
        Thread descarga2 = new Thread(new DescargaRunnable(url2));
        //Bloquear hasta que ambas se descarguen
        descarga1.join();
        descarga2.join();
        //Descargadas, realizar alguna operación con los datos
```

2.2.1. Pausar un hilo

En las aplicaciones de Android los hilos no se pausan automáticamente cuando la `Activity` pasa a estado de pausa. Puede ser adecuado pausarlos para ahorrar recursos y porque el usuario suele asumir que una aplicación que no esté en primer plano, no consume CPU.

Igual que para detener un hilo suele ser conveniente tener una variable booleana que indique esta intención, en el siguiente ejemplo se añade una variable `paused` para indicar que el hilo debe ser bloqueado. La llamada a `wait()` bloquea el hilo utilizando el lock de un objeto `pauseLock`. Cuando se debe reanudar el hilo, se realiza una llamada a `notifyAll()`. Ambas llamadas están en una sección `synchronized`.

```
class MiRunnable implements Runnable {
    private Object pauseLock;
    private boolean paused;
    private boolean finished;

    public MiRunnable() {
        pauseLock = new Object();
        paused = false;
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
        finished = false;
    }

    public void run() {
        while (!finished) {
            // Realizar operaciones
                    // ...
            synchronized (pauseLock) {
                while (paused) {
                    try {
                        pauseLock.wait();
                    } catch (
                      InterruptedException e) {
                            break;
                    }
                }
            }
        }
    }
        //Llamarlo desde Activity.onPause()
    public void pause() {
        synchronized (pauseLock) {
            paused = true;
        }
    }
        //Llamarlo desde Activity.onResume()
    public void resume() {
        synchronized (pauseLock) {
            paused = false;
            pauseLock.notifyAll();
        }
    }
        //Llamarlo desde Activity.onDestroy()
    public void finish() {
       finished = true;
    }
}
```

Hay que prestar atención a que el método `Activity.onResume()` no sólo es llamado tras un estado de pausa sino también al crear la actividad por primera vez. Por tanto habrá que comprobar que el hilo no sea `null`, antes de hacer la llamada a `MiRunnable.resume()`.

2.3. Mecanismos específicos en Android

Android cuenta con un framework de mensajería y concurrencia que está basado en las clases `Thread`, `Looper`, `Message`, `MessageQueue` y `Handler`. Por conveniencia existe también la clase `AsyncTask`, para usos comunes de actualización de la UI. Un `Looper` se utiliza para ejecutar un bucle de comunicación por mensajes en determinado hilo. La interacción con el `Looper` se realiza a través de la clase `Handler`, que permite enviar y procesar un `Message` y objetos que implementen `Runnable`, asociados a una cola `MessageQueue`.

2.3.1. Mensajes mediante Handlers

`Handler` permite enviar y procesar un `Message` y objetos que implementen `Runnable`. Cada instancia de `Handler` se asocia con un único hilo y su cola de mensajes. Hay que

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

crear el Handler desde el mismo hilo al que debe ir asociado. Un hilo puede tener asociados varios `Handler`.

Mediante el uso de `Handler` se pueden encolar una acciones que que deben ser realizadas en otro hilo de ejecución distinto. Esto permite programar acciones que deben ser ejecutadas en el futuro, en un orden determinado.

El envío se realiza mediante `post(Runnable)` o `postDelayed(Runnable, long)`, donde se indican los milisegundos de retardo para realizar el post.

```
public class Actividad extends Activity {
        TextView textView;
        Handler handler;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                textView = (TextView) findViewById(R.id.textview1);

                handler = new Handler();
                handler.removeCallbacks(tarea1);
                handler.removeCallbacks(tarea2);
                handler.post(tarea1);
                handler.post(tarea2);
        }

        private Runnable tarea1 = new Runnable() {
                @Override
                public void run() {
                        textView.setText("Primer cambio");
                }
        };

        private Runnable tarea2 = new Runnable() {
                @Override
                public void run() {
                        textView.setText("Segundo cambio");
                }
        };

}
```

En el ejemplo anterior las tareas se ejecutarían por orden y lo harían en el mismo hilo de ejecución. El resultado visible sería el del segundo cambio.

En el siguiente ejemplo se realiza una carga lenta de datos en otro hilo, al tiempo que se muestra una splash screen para indicar que se están cargando los datos. Una vez finalizada la tarea se envía un mensaje vacío. Este mensaje se captura con un Handler propio y se realiza el cambio de vistas de la Actividad.

2.3.1.1. Ejemplo: pantalla de inicialización

```
public class Actividad extends Activity implements Runnable{
    @Override
    public void onCreate(Bundle savedInstanceState) {
```

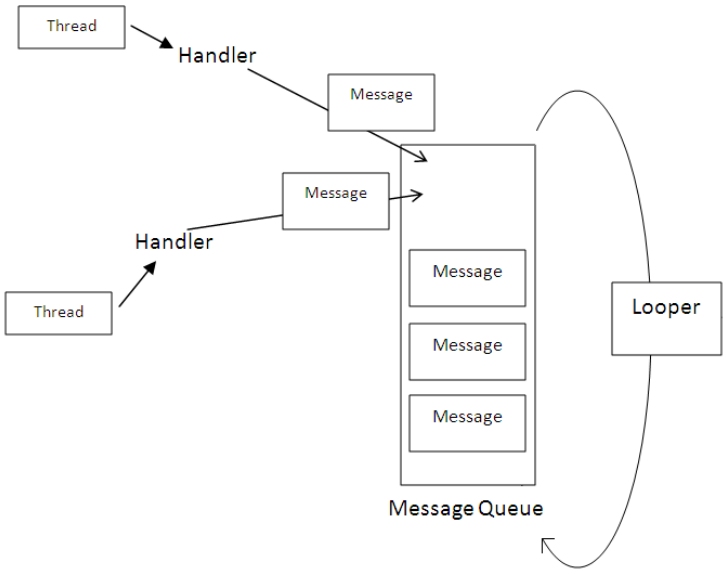

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
        super.onCreate(savedInstanceState);
        //debe estar definido en /res/layout:
        setContentView(R.layout.splashscreen);

        (new Thread(this)).start();
    }
    private Handler handler = new Handler(){
            public void handleMessage(Message msg){
                    setContentView(R.layout.main);
            }
    };
        @Override
        public void run() {
                //Carga lenta de datos...
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                }
                handler.sendEmptyMessage(0);
        }
}
```

2.3.2. Hilos con Looper

`Looper` se utiliza para ejecutar un bucle de comunicación por mensajes en determinado hilo. Por defecto los hilos no tienen ningún bucle de mensajes asociado. Para crearlo hay que llamar a `Looper.prepare()` desde el hilo correspondiente. En el siguiente ejemplo se implementa un hilo que cuenta con un método `post(Runnable)` para encolar las tareas al `handler`.



Paso de mensajes al Looper de un hilo.

```
class MiHiloLooper extends Thread{
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
        Handler handler; //message handler

        public MiHiloLooper(){
                this.start();
        }

        @Override
        public void run(){
                try{
                        Looper.prepare();
                        handler = new Handler();
                        Looper.loop();
                }catch(Throwable t){
                        Log.e("Looper","Error: ", t);
                }
        }

        public void terminate(){
                handler.getLooper().quit();
        }

        public void post(Runnable runnable){
                handler.post(runnable);
        }
}
```

Se usaría declarándolo y haciendo pasando a `post()` objetos `Runnable` cada vez que sea necesario. En el ejemplo siguiente no sería estrictamente necesario utilizar un looper, pero se puede pensar en un caso en el que no se sabe a priori qué `Runnable`s y cuándo habrá que ejecutar.

```
public class Actividad extends Activity {
        MiHiloLooper       looper;
        ImageView iv1,  iv2,  iv3;
        URL      url1, url2, url3;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                //... Inicializar views ...
                //... Inicializar urls  ...

                looper = new MiHiloLooper();
                looper.post(new Thread( new ImageLoader(url1, iv1) ));
                looper.post(new Thread( new ImageLoader(url2, iv2) ));
                looper.post(new Thread( new ImageLoader(url3, iv3) ));

        }

        @Override
        protected void onDestroy() {
                looper.terminate();
                super.onDestroy();
        }
}
```

Las tres imágenes se cargarían una detrás de otra (y no en paralelo) en el orden establecido. La clase ImageLoader sería un `Runnable` que tendría que descargar la imagen y ponerla en su `ImageView`. El acceso a la interfaz podría realizarse a través del método `ImageView.post()`, ya que en el anterior ejemplo no hemos declarado ningún Handler. Por ejemplo podría quedar así:

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
class ImageLoader implements Runnable{
        ImageView iv;
        URL      url;

        public ImageLoader(String url, ImageView iv){
                this.iv  = iv;
                this.url = url;
        }

        @Override
        public void run() {
                try {
                        InputStream is = url.openStream();
                        final Drawable drawable =
                                Drawable.createFromStream(is, "src");
                        button.post(new Runnable() {
                                @Override
                                public void run() {
                                        iv.setImageDrawable(drawable);
                                }
                        });
                } catch (IOException e) {
                        Log.e("URL","Error downloading image "+
                                url.toString());
                } catch (InterruptedException e) {
                }
        }
}
```

2.4. Pools de hilos

Los pools de hilos permiten no sólo encolar tareas a ejecutar, sino también controlar cuántas ejecutar simultáneamente. Por ejemplo, si se quieren cargar decenas de imágenes, pero máximo de dos en dos.

```
public class AvHilos8PoolExecutorActivity extends Activity {
    int  poolSize = 2;
    int  maxPoolSize = 2;
    long keepAliveTime = 3;

        ThreadPoolExecutor pool;
        ArrayBlockingQueue<Runnable> queue;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        queue = new ArrayBlockingQueue<Runnable>(5);
        pool = new ThreadPoolExecutor(
                poolSize, maxPoolSize, keepAliveTime,
                TimeUnit.SECONDS, queue);
        pool.execute(new MyRunnable("A"));
        pool.execute(new MyRunnable("B"));
        pool.execute(new MyRunnable("C"));
        pool.execute(new MyRunnable("D"));
        pool.execute(new MyRunnable("E"));
    }
}
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

Los parámetros que se pasan a `ThreadPoolExecutor` al crearlo son el número mínimo de hilos a mantener en el pool, incluso aunque no estén en ejecución, el número máximo de hilos permitido, el tiempo que se permite que permanezca un hilo que ya no esté en ejecución, la unidad de tiempo para el anterior argumento y por último la cola de tareas.

Cada `ThreadPoolExecutor` guarda estadísticas básicas, como por ejemplo el número de tareas completadas. Se pueden obtener a través de los métodos del pool y de la cola:

```
        queue.size() );
        pool.getActiveCount() );
        pool.getTaskCount() );
```

3. Hilos e interfaz de usuario

Una de las principales motivaciones para el uso de hilos es permitir que la interfaz gráfica funcione de forma fluida mientras se están realizando otras operaciones. Cuando una operación termina, a menudo hay que reflejar el resultado de la operación en la interfaz de usuario (UI, de user interface). Otro caso muy común es ir mostrando un progreso de una operación lenta.

Considérese el siguiente ejemplo en el cuál se descarga una imagen desde una URL y al finalizar la descarga, hay que mostrarla en un `imageView`.

```
ImageView imageView =
        (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
    public void run() {
      Drawable imagen = descargarImagen("http://...");
      //Desde aquí NO debo acceder a imageView
      //imageView.setImageDrawable(imagen)
      //daría error en ejecución
    }
}).start();
```

Tras cargar la imagen no podemos acceder a la interfaz gráfica porque la GUI de Android sigue un modelo de hilo único: sólo un hilo puede acceder a ella. Se puede solventar de varias maneras:

- `Activity.runOnUiThread(Runnable)`
- `View.post(Runnable)`
- `View.postDelayed(Runnable, long)`
- `Handler`
- `AsyncTask`

El método `Activity.runOnUiThread(Runnable)` es similar a `View.post(Runnable)`. Ambos permitirían que el `Runnable` que pasamos por parámetro se ejecute en el mismo hilo que la UI.

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
ImageView imageView =
        (ImageView)findViewById(R.id.ImageView01);
new Thread(new Runnable() {
    public void run() {
      Drawable imagen = descargarLaImagen("http://...");
      imageView.post(new Runnable() {
        public void run() {
          imageView.setDrawable(imagen);
        }
      });
    }
}).start();
```

Otra manera sería por medio de Handlers, con el mismo fin: que la actualización de la UI se ejecute en su mismo hilo de ejecución. El `Handler` se declara en el hilo de la UI (en la actividad) y de manera implícita se asocia al Looper que contiene el hilo de la UI. Después en el hilo secundario se realiza un `post` a dicho `Handler`.

```
public class Actividad extends Activity {
        private Handler handler;
        private ImageView imageView;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                // ... Inicializar views ...

                handler = new Handler();

                Runnable runnable = new Runnable() {
                        @Override
                        public void run() {
                                Drawable imagen =
                                        descargarLaImagen("http://...");
                                handler.post(new Runnable() {
                                        @Override
                                        public void run() {
imageView.setDrawable(imagen);
                                        }
                                });
                        }
                };
                new Thread(runnable).start();
        }
}
```

3.1. AsyncTask

Otra manera es utilizar una `AsyncTask`. Es una clase creada para facilitar el trabajo con hilos y con interfaz gráfica, y es muy útil para ir mostrando el progreso de una tarea larga, durante el desarrollo de ésta. Nos facilita la separación entre tarea secundaria e interfaz gráfica permitiéndonos solicitar un refresco del progreso desde la tarea secundaria, pero realizarlo en el hilo principal.

```
TextView textView;
ImageView[] imageView;

public void bajarImagenes(){
```


Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

```
        textView = (TextView)findViewById(R.id.TextView01);
        imageView[0] = (ImageView)findViewById(R.id.ImageView01);
        imageView[1] = (ImageView)findViewById(R.id.ImageView02);
        imageView[2] = (ImageView)findViewById(R.id.ImageView03);
        imageView[3] = (ImageView)findViewById(R.id.ImageView04);

        new BajarImagenesTask().execute(
                "http://a.com/1.png",
                "http://a.com/2.png",
                "http://a.com/3.png",
                "http://a.com/4.png");
}
private class BajarImagenesTask
                extends AsyncTask<String, Integer, List<Drawable>> {
     @Override
     protected List<Drawable> doInBackground(String... urls) {
              ArrayList<Drawable> imagenes =
                      new ArrayList<Drawable>();
              for(int i=1;i<urls.length; i++){
                cargarLaImagen(urls[0]);
                publishProgress(i);
         return imagenes;
     }

     @Override
     protected void onPreExecute() {
       super.onPreExecute();
       textView.setText("Cargando imagenes...");
     }

     @Override
     protected void onProgressUpdate(String... values) {
             textView.setText(values[0] +
                     " imagenes cargadas...");
     }

     @Override
     protected void onPostExecute(List<Drawable> result) {
       for(int i=0; i<result.length; i++){
         imageView[i].setDrawable(result.getItemAt(i));
       }
       textView.setText("Descarga finalizada");
     }

     @Override
     protected void onCancelled() {
             textView.setText("Cancelada la descarga");
     }
}
```

Nota:

La notación `(String ... values)` indica que hay un número indeterminado de parámetros, y se accede a ellos con `values[0], values[1],` ..., etcétera. Forma parte de la sintaxis estándar de Java.

Lo único que se ejecuta en el segundo hilo de ejecución es el bucle del método `doInBackground(String...)`. El resto de métodos se ejecutan en el mismo hilo que la interfaz gráfica. La petición de publicación de progreso, `publishProgress(...)` está resaltada, así como la implementación de la publicación del progreso, `onProgressUpdate(...)`. Es importante entender que la ejecución de

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

`onProgressUpdate(...)` no tiene por qué ocurrir inmediatamente después de la petición `publishProgress(...)`, o puede incluso no llegar a ocurrir.

El uso de `AsyncTask` es una práctica recomendable en Android porque el método `post()` puede hacer el código menos legible. Esta estructura está en Android desde la versión 1.5.

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.

Copyright © 2011-12 Dept. Ciencia de la Computación e IA All rights reserved.