Hilos - Ejercicios

Índice

1. Hilos parables y pausables

En este ejercicio implementaremos primero un Thread y después un Runnable equivalente. Ambos reproducirán un tono cinco veces seguidas separadas por una pausa. Aunque el reproductor multimedia en realidad crea su propio hilo por separado, es reproducir un tono en cada iteración del método `run()` es una forma sencilla de comprobar que el hilo funciona sin tener que acceder a la interfaz gráfica.

A ambos hilos les proporcionaremos métodos para parar, pausar y reanudar su ejecución. En las plantillas hay un proyecto llamado `android-av-hilos-pausables` que proporciona una actividad principal con dos botones, uno para el Runnable y otro para el Thread. Además incluye un esqueleto del Thread, llamado `Hilo2Thread`.

En `MainActivity` rellenar el código de `button2` para permitir que inicie el hilo. Comprobar que al pulsarlo se reproduce repetidas veces el tono.

Comprobar que al pulsar tanto "Home" como "Atrás" la reproducción continúa, por tanto el programa no puede terminar a petición del usuario. Implementar el método `Hilo2Thread.termina()` que pondrá a true una variable que previamente haya sido establecida como campo del hilo, `boolean stopped=false`. En el método `run()`, comprobar en cada iteración el valor de dicha variable para salir del bucle si está a true. En el método `MainActivity.onDestroy()`, pedir que el hilo2 termine (`hilo2.termina()`), y hacer lo mismo si se pulsa el botón 2 y el hilo ya está iniciado.

Sustituir la llamada a `hilo2.termina()` por `hilo2.interrupt()` para comprobar que el hilo no se interrumpe. ¿Qué habría que añadir en el método `run()` para posibilitar su interrupción? Una vez hecho el cambio en el hilo, volver a dejar la llamada original al método `hilo2.termina()`.

Al pulsar "Home" los hilos no terminan, en cambio sí lo hacen al pulsar "Atrás" en el emulador. Para hacer que los hilos se bloqueen tenemos que implementar los métodos propios `pausa()` y `reanuda()`. Necesitamos una variable booleana `pauseme` que sea campo del hilo e indique si se ha solicitado la pausa. La comprobaremos en cada iteración y utilizaremos `wait()` para bloquear el hilo si es necesario. Antes de hacer el bloqueo utilizaremos el lock de un objeto pauselock para la sincronización de la sección crítica:

```
public void run() {
        for(int i=1; i<=5; i++){
            // ...

                synchronized (pauselock) {
                        while(pauseme){
                                try{
                                        pauselock.wait();
                                }catch(InterruptedException e){
                                        return;
                                }
                        }
```

```
                        }
                }
        }
```

En el método `pausa()` pondremos la variable `pauseme` a verdadero. En el método `reanuda()` la pondremos a falso y notificaremos los hilos bloqueados con `notifyAll()` que, como el caso de `wait()`, necesita estar en un bloque sincronizado:

```
        public void reanuda(){
                synchronized (pauselock) {
                        pauseme = false;
                        pauselock.notifyAll();
                }
        }
```

Realizar las llamadas a `reanuda()` y `pausa()` desde la actividad principal y comprobar que funcionan al pulsar la tecla "Home". Para reanudar la actividad hay que pulsar la tecla "Home" durante dos segundos y seleccionar la aplicación en la lista de aplicaciones recientes, o bien desde el menu de aplicaciones.

Una vez que `Hilo2Thread()` funcione bien, crear `Hilo1Runnable` con los mismos métodos y añadir las llamadas a la actividad.

Nota:

Puesto que `Runnable` es una interfaz, hubiera sido posible integrar los métodos de `Hilo1Runnable` en la propia clase de la actividad principal, sin necesidad de una clase aparte.

2. (*) Grupos de hilos y prioridades

En el proyecto `android-av-group-priorities` hay una actividad principal, un hilo y una estructura de datos `Lines` con getters y setters sincronizados que se utiliza para mostrar en la UI aquello que distintos hilos añaden a `Lines`. En el método `PrioritiesActivity.onResume()` se crea un hilo con un nuevo `Runnable` definido en línea, cuya intención es actualizar la interfaz gráfica cada 100 milisegundos. Se pide actualizar el `TextView` por `post()`. La actualización consistirá en poner el string `lines.getLines()` en el `TextView`.

En la actividad también se crea un array con dos hilos y a cada uno se le asigna una prioridad. Al ejecutar la aplicación se puede observar la salida de cada uno de ellos y el número del 1 al 10 que se muestra, indica la la prioridad del hilo. Ajustar el retardo de cada hilo de manera que no terminen los dos a la vez, pero que en pantalla queden mensajes de ambos hilos, y no sólo los mensajes del menos prioritario.

En la actividad se crea un grupo de hilos pero no se utiliza. Se pide pasar ese grupo por el constructor de `Hilo` y que la llamada al constructor padre del hilo sea `super(threadGroup,"Hilo")`. Así los hilos se añadirán al grupo de hilos. Establecer como prioridad máxima del grupo la normal:

`tGroup.setMaxPriority(Thread.NORM_PRIORITY)` y comprobar qué ocurre con la prioridad que antes estaba a 10.

3. Productor-consumidor

En las plantillas de la sesión se proporciona el proyecto `android-av-productor-consumidor` en el que hay una UI que representa el tamaño de un buffer de objetos a consumir/producir, y un slider que permite cambiar el coste temporal (simulado) de consumición y de producción. Las clases `Producer` y `Consumer` proporcionadas están incompletas. Se pide completarlas siguiendo estos pasos:

Sin modificar la clase `MainActivity`, observar cómo se crean los hilos de ejecución.

```
producers[i] = new Producer(queue);
```

Haría falta un `Thread` que podemos declarar en los propios Runnables del consumidor y del productor, creándolo a partir de sus respectivos Runnables e iniciándolo en el propio constructor del Runnable.

En MainActivity se declara una `BlockingQueue<Integer>` que se pasa como parámetro al construir los productores y los consumidores. Se trata de una cola bloqueante que hará de mecanismo de sincronización y de bloqueo entre los consumidores y los productores. Dentro de `Consumer` y `Producer`, crear un campo (en cada uno) para guardarse la referencia a la cola que se pasa por el constructor.

Para simular coste temporal de consumición utilizaremos los métodos `Integer Producer.generateNumberSomehow()` y `Consumer.process(Integer)`. La verdadera producción y consumición se realizará al acceder a la cola para encolar o para obtener un Integer, con los métodos `BlockingQueue.put(Object)` y `Object BlockingQueue.take()` respectivamente. Se pide realizar una producción (o consumición, respectivamente) en cada iteración de los bucles while. Puesto que las llamadas a la cola son bloqueantes, nos obligarán a capturar una excepción. En caso de recibir una excepción de interrupción se pide romper el bucle para así terminar la ejecución del hilo.

Ejecutar la actividad principal y variar la velocidad de producción y de consumición utilizando los controles gráficos. Si se pone el mismo coste temporal para ambas operaciones, el nivel del buffer tiende a crecer. Comprobar a qué se debe y cambiar alguna inicialización de variables en `MainActivity` para que el buffer tienda a quedarse igual si los retardos son iguales.

4. (*) Descarga de imágenes con hilos y Looper

El proyecto `android-av-download-images` incluye una actividad que dispone 12

botones sobre la pantalla y les asocia listeners para pulsación que cargan una imagen y la muestran en el botón pulsado. Para cargar la imagen se utiliza el Runnable `ImageLoader` que simula un retardo de 3 segundos y además reproduce un sonido que para al finalizar la carga. De esta manera si están en curso varias descargas a la vez, se puede apreciar por los sonidos entremezclados. Probar la aplicación pulsando en el orden que se quiera los botones. Se puede observar que si se pulsan muy rápido varios botones seguidos sus imágenes aparecen casi a la vez. También es posible que al pulsar dos botones, primero salga la del segundo que se pulsó, según lo que tarde en responder la red.

Se pide implementar una clase hilo llamada `ImageLoaderLooper extends Thread` que se autoinicie en su constructor con `this.start()`. Añadirle un campo `Handler handler` que inicializaremos en el método `run()`. En dicho método se preparará el Looper, se creará el Handler y se iniciará el loop:

```
Looper.prepare();
handler = new Handler();
Looper.loop();
```

Crear un método público `post(Runnable runnable)` que haga un post al handler: `handler.post(runnable);`

Finalmente crear un método público terminate() que llame a `handler.getLooper().quit()`.

En `MainActivity` declarar, inicializar y hacer las llamadas pertinentes al `ImageLoaderLooper` que se ha creado. Están comentadas y sólo hay que descomentarlas. Esta vez en lugar de iniciar el nuevo hilo `ImageLoader` que se crea al pulsar un botón, lo que se hará será pasarle el looper el nuevo hilo, a través del método `ImageLoaderLooper.post(Runnable)`.

Comprobar que por muy rápido que se pulsen los botones, las descargas siempre empiezan tras terminar la anterior, no pudiendo haber dos descargas simultáneas y respetándose siempre el orden.

5. Descarga de imágenes con Pool de hilos

En el anterior ejercicio hemos conseguido descargar una única imagen a la vez para evitar que haya muchas descargas simultáneas. ¿Y si queremos permitir `n` descargas simultáneas? Los pools de hilos nos ayudarán a gestionar la ejecución de los hilos de una forma más cómoda.

En las plantillas tenemos el proyecto `android-av-download-images-pool` que es idéntico al del ejercicio anterior pero en lugar de utilizar un hilo con `Looper` para cargar las imágenes se utilizará un `ThreadPoolExecutor` con la ayuda de una cola bloqueante `ArrayBlockingQueue` de Runables, ambos están declarados como campos de la actividad principal que se proporciona. Se pide:

Antes de la asociación de listeners con botones, inicializar la cola y el pool, permitiendo al pool ejecutar sólo dos hilos simultáneamente.

En lugar de iniciar el nuevo hilo que se crea al pulsar un botón, pasárselo al pool a través del método `pool.execute(Runnable)`.

Pulsar muy rápido cuatro botones seguidos para observar que las descargas ocurren de dos en dos.

6. Lector de RSS con AsyncTask

En el proyecto `android-av-rss` de las plantillas tenemos un lector de RSS que se limita a descargar el contenido de una URL dada, parsearla en búsqueda de los elementos típicos de noticias RSS y rellena una estructura de noticias con la descripción, título, fecha, url, link de una imagen, etcétera. No almacena los contenidos descargados ni los comprueba periódicamente con ningún servicio en segundo plano. Lo único que hace es mostrar la lista de noticias descargadas en una tabla en la UI. En la implementación que se proporciona la descarga se realiza en el mismo hilo de ejecución que la UI y se puede comprobar que la pantalla se congela durante la descarga. (Al hacer click largo en el campo de texto aparecen dos URLs de ejemplo).

Lector de RSS

Se pide declarar la clase `TareaDescarga` dentro de la clase `Main`:

```
public class Main extends Activity {

        // ...

        private class TareaDescarga extends AsyncTask<URL, String,
String>{

        }
}
```

Implementar sus métodos de manera que:

- Antes de ejecutar la tarea inicie el diálogo de progreso con `iniciaProgressDialog()`.
- En segundo plano realice la descarga y parseo: `bajarYParsear(params[0])` y después devuelva "Carga finalizada".
- Si se cancela lo indique con `textViewTitulo.setText("Descarga cancelada")`.
- Al finalizar la tarea actualice la interfaz con `actualizaTablaNoticiasYParaProgressDialog()`.

Declarar la tarea como campo de la clase `Main` y en el método `accionDescargar()` crearla y ejecutarla, en lugar de llamar a los métodos de bajar, parsear y actualizar la interfaz:

```
public class Main extends Activity {
        TareaDescarga tarea;
        // ...

    void accionDescargar(){
                try {
                        tarea = new TareaDescarga();
                        tarea.execute(new URL(url));
                        //iniciaProgressDialog();
                        //bajarYParsear(new URL(url));
                        //actualizaTablaNoticiasYParaProgressDialog();
                } catch (MalformedURLException e) {
                }
   }
}
```

Probar la aplicación para comprobar que la tarea se lanza y que la descarga se realiza, y que el diálogo de progreso indefinido se muestra y desaparece al terminar la descarga.

Para mostrar el número de noticia descargada hay que notificar el progreso desde dentro del bucle realizado en `bajarYParsear(URL)`. La manera más fácil es pasar todo el código de dicho método al método `doInBackground() de la tarea`. Una vez hecho esto, hay que localizar la línea donde se incrementa el número de items cargados, `nItems++`, y después de ella solicitar la publicación de progreso con `publishProgress("Cargando item "+nItems)`. Para que ésta se lleve a cabo hay que sobrecargar el método `onProgressUpdate()`:

```
        private class TareaDescarga extends AsyncTask<URL, String,
String>{
                //...
                @Override
                protected void onProgressUpdate(String... values) {
                        super.onProgressUpdate(values);
                        textViewTitulo.setText(values[0]);
                }
        }
```

Ahora aparte del diálogo de progreso, en segundo plano debería observarse el número de noticia que se está cargando.