Capítulo 10
Hebras y tareas asíncronas

Práctica 10.1: Actividades, aplicaciones y procesos

En la práctica 9.2 vimos el ciclo de vida de las actividades que es controlada por Android. Las actividades reciben notificaciones a través de diferentes métodos, que podemos sobrescribir para actuar en consecuencia.

Las actividades son un tipo de componente, como lo son los servicios, los proveedores de contenido o los broadcast receivers. En Android, no se lanzan "aplicaciones", sino que se lanzan componentes.

Los componentes de una aplicación se declaran en el fichero de manifiesto del APK que los contiene y, salvo que se indique lo contrario, todos los componentes se ejecutarán en el mismo proceso, y serán atendidos por la misma hebra.

Así, cuando lanzamos una actividad, Android comprueba si el proceso asociado a esa aplicación (APK) está o no lanzado ya. Si no lo está, lo lanza e inicializa, y le pide que construya y ejecute la actividad, iniciando su ciclo de vida. Si la actividad se cierra, Android normalmente no eliminará el proceso, dejándolo en suspendido por si llegara alguna solicitud nueva para él de lanzar un componente.

1. Crea un proyecto nuevo y llámalo ProcesosYComponentes.

2. Recupera la clase SpyActivity de la práctica 9.2 e inclúyela en el proyecto. Cambia el nombre del paquete para que pertenezca también a DM2E.procesosycomponentes como el resto de la aplicación. Haz que la actividad principal herede de ella.
3. Añade en el proyecto una nueva actividad. Llámala SecondaryActivity. Marca que quieres que sea una actividad para el lanzador Launcher activity para que podamos lanzar ambas. Como título, pon Proc. y comp. 2
4. Lanza la aplicación (se lanzará la primera actividad) y comprueba la aparición de los mensajes de log indicando el ciclo de vida.
5. Ve al directorio donde tengas instaladas las SDK de Android. Entra en el directorio platform-tools y ejecuta adb para abrir una línea de comandos en el dispositivo[1].
6. Ejecuta ps y busca el proceso DM2E.procesosycomponentes, que es el que está a cargo de nuestra ejecución.
7. Mira su identificador del proceso. Utiliza ps -t <id> para listar sus hebras.
8. Pulsa "Volver" en la actividad. Observa los mensajes del ciclo de vida, indicando que la actividad ha sido destruida.

---

1 Dependiendo de cuántos AVD tengas lanzados y dispositivos físicos tengas conectados necesitarás unos parámetros u otros. Por ejemplo, con un único dispositivo bastará con adb shell. Si tienes más de uno, tendrás que usar los parámetros -e, -d o -s <id>

9. Comprueba de nuevo con ps en la línea de comandos que el proceso sigue en ejecución.
10. Desde el lanzador, busca la segunda actividad y ejecútala. Con ps, comprueba que el proceso se mantiene y tiene el mismo identificador.
11. Cierra la actividad, comprueba que se han llamado a los métodos de su ciclo de vida, y que el proceso sigue vivo.

Práctica 10.2: La hebra principal y ANR

Aunque hemos visto que había varias hebras en el proceso, la gestión de todas las actividades de la aplicación es realizada desde la misma, la llamada hebra principal o hebra de UI (de nombre main en los listados).

1. Haz una copia de la práctica anterior.
2. Elimina la segunda actividad, dado que no la utilizaremos más.
3. Modifica el layout de la actividad principal para poner un botón que ocupe toda la pantalla, asocia un evento y crea el método.
4. Pon como código del evento un bucle infinito.

```
public void onPulsame(View v) {
        while(true)
        ;
}
```

5. Lanza la aplicación y pulsa el botón. ¿Qué ocurre?

Práctica 10.3: Hebras trabajadoras

Si queremos hacer tareas que lleven mucho tiempo, es preferible utilizar hebras.

1. Haz una copia de la práctica anterior.
2. Modifica el código del evento para que el bucle infinito se ejecute en una hebra nueva.

```
public void onPulsame(View v) {
Thread t;
t = new Thread(new Runnable() {
@Override
public void run() {
while(true);
}
});
t.start();
} // onPulsame
```

3. Lanza la aplicación. No pulses el botón aún.
4. En el shell con el dispositivo a través de adb, utiliza ps -t para ver las hebras. También puedes verlas con el Android Device Monitor.
5. Pulsa el botón. Ahora que la aplicación responde correctamente. Observa con ps que aparece una hebra nueva.
6. Pulsa "Volver" para cerrar la actividad. Comprueba que la hebra se mantiene.

7. Para matar la hebra, puedes usar kill desde el shell (sólo si era un AVD), el administrador de aplicaciones del móvil, o, sencillamente, volver a lanzar la ejecución de la aplicación desde el entorno de desarrollo. Ésto ocasiona la reinstalación de la aplicación, y Android detendrá antes el proceso asociado a ella.

Si, en lugar de un bucle infinito, quieres hacer algo más apreciable, puedes generar un pequeño "beep" cada segundo:

```
t = new Thread(new Runnable() {
@Override
public void run() {
int i = 0;
ToneGenerator tg;
tg = new ToneGenerator(AudioManager.STREAM_ALARM, 100);
while(true) {
tg.startTone(ToneGenerator.TONE_CDMA_ALERT_INCALL_LITE, 20);
android.util.Log.i("TAG", "MainActivity: " + i);
++i;
try {
Thread.sleep(2000);
} catch(InterruptedException ie) {}
}
}
});
```

Práctica 10.4: Hebras trabajadoras e interfaz

Vamos a modificar la práctica anterior para que en lugar de un bucle infinito, tengamos una hebra trabajadora que calcule algo, y que dé el resultado al terminar.

1. Haz una copia de la práctica anterior. Refactoriza el paquete para que se llame DM2E.cuantosprimos y cambia el texto de la cadena app_name para que sea Cuantos primos.
2. Añade en la actividad dos métodos estáticos, uno para decidir si un número es o no primo, y otro para contar cuántos primos hay entre 1 y un número.

```
public static boolean esPrimo(long i) {
if (i == 1) return false;
if (i < 4) return true;
if ((i % 2) == 0 || (i % 3) == 0) return false;
if (i < 9) return true;
long n = 5;
while (n*n <= i && i % n != 0 && i % (n + 2) != 0)
n += 6;
return (n*n > i);
}

public static long cuantosPrimos(long limite) {
long result = 0;
for (long i = 1; i <= limite; ++i)
if (esPrimo(i))
++result;
```

```
return result;
}
```

3. Modifica el layout para meter una etiqueta (TextView) encima del botón. Seguramente quieras cambiar a LinearLayout.

```
<LinearLayout
...
android:orientation="vertical">
<TextView android:id="@+id/tvResultado"
android:layout_width="match_parent"
android:layout_height="wrap_content" />
<Button ... />
</LinearLayout>
```

4. 4. Modifica el código asociado a la hebra secundaria para que llame al método cuantosPrimos().

```
t = new Thread(new Runnable() {
@Override
public void run() {
long r = cuantosPrimos(1000000); // 10^6
TextView tv = (TextView) findViewById(R.id.tvResultado);
tv.setText("" + r);
}
});
```

1. Lanza la aplicación. ¿Qué ocurre y por qué?

Práctica 10.5: Ejecución en la hebra principal

El problema de la práctica anterior es que la hebra principal no es reentrante (segura ante hebras). No podemos manipular nada del interfaz desde una hebra que no sea la principal, porque, por eficiencia, no se utilizan métodos sincronizados.

Para manipular el interfaz desde una hebra trabajadora, necesitamos crear un Runnable que encapsule el código que queremos ejecutar, y "encolarlo" para que la hebra principal lo ejecute en cuanto le sea posible hacerlo de manera segura.

1. Haz una copia de la práctica anterior.
2. Encierra el código que manipula el interfaz en un objeto Runnable, y pásalo al método runOnUiThread() de la actividad.

```
t = new Thread(new Runnable() {
@Override
public void run() {
	final long r = cuantosPrimos(1000000); // 10^6
	runOnUiThread(new Runnable() {
		@Override
		public void run() {
			TextView tv = (TextView) findViewById(R.id.tvResultado);
			tv.setText("" + r);
		}
```

```
        });
}
});
```

3. Ejecuta la aplicación. Comprueba que, ahora sí, funciona bien.

Práctica 10.6: AsyncTask

El código de la práctica anterior es un poco farragoso, y no es cómodo de escribir. Android proporciona, desde el API level 3 (1.5), la clase AsyncTask que encapsula una hebra y facilita la programación de tareas largas que se deben realizar en segundo plano.

Los dos métodos más importantes de la clase, que habrá que sobrescribir, son:

- **protected RESULT doInBackground(PARAMS... params)**: contendrá el código que se ejecutará en una hebra secundaria. Recibe una lista de parámetros de tipo PARAMS.
- **protected void onPostExecute(RESULT result**): será llamado desde la hebra principal cuando el método anterior termine. El parámetro será el valor devuelto por doInBackground().

Como puedes ver, AsyncTask es una clase genérica, con tres argumentos, uno de los cuales aún no hemos visto:

- **PARAMS**: tipo de los parámetros recibidos al lanzar la tarea.
- **PROGRESS**: tipo de indicación del progreso, del que hablaremos en la práctica 10.8.
- **RESULT**: tipo del resultado que genera.

Para lanzar una tarea asíncrona, basta con crear un objeto y llamar, desde la hebra principal, al método execute(PARAMS... params), que se encargará de que el método doInBackground() se ejecute con los parámetros indicados en una hebra secundaria.

1. Haz una copia de la práctica anterior.
2. Crea una clase interna CalcularPrimos que herede de AsyncTask. El tipo PARAMS será Void, que se utiliza para indicar que no tenemos parámetros. El tipo RESULT será Long, y el tipo PROGRESS será, de momento, también Void. Mueve a ella los métodos estáticos del cálculo de primos (aunque ahora no podrán ser estáticos).
3. En el método doInBackground() llama a cuantosPrimos().
4. En el método onPostExecute() actualiza la etiqueta.

```
class CalcularPrimos extends AsyncTask<Void, Void, Long> {
public /*static*/ boolean esPrimo(long i) {
if (i == 1) return false;
if (i < 4) return true;
if ((i % 2) == 0 || (i % 3) == 0) return false;
if (i < 9) return true;
long n = 5;
while (n*n <= i && i % n != 0 && i % (n + 2) != 0)
n += 6;
```

```
return (n*n > i);
}
public /*static*/ long cuantosPrimos(long limite) {
long result = 0;
for (long i = 1; i <= limite; ++i)
if (esPrimo(i))
++result;
return result;
}
@Override
protected Long doInBackground(Void... params) {
return cuantosPrimos(10000000); // 10^7
}
@Override
protected void onPostExecute(Long resultado) {
TextView tv = (TextView) findViewById(R.id.tvResultado);
tv.setText(resultado.toString());
}
} // CalcularPrimos
```

5. Modifica el método del evento del ratón, crea un objeto de la clase, y lanza su ejecución.

```
public void onPulsame(View v) {
new CalcularPrimos().execute();
} // onPulsame
```

6. Prueba la aplicación.
7. Aumenta el límite en la búsqueda para que se cuenten más primos y el proceso tarde más. En el método doInBackground() pon, al principio, una notificación en el log:

```
protected Long doInBackground(Void... params) {
android.util.Log.d("CLICK", "Empiezo!");
return cuantosPrimos(10000000); // 10^7
}
```

8. Pulsa el botón dos veces. ¿Qué ocurre? ¿Cómo lo explicas?

Práctica 10.7: AsyncTask cancelable

Es posible "cancelar" una tarea asíncrona llamando a su método cancel(). La tarea no se detendrá por sí misma. Es responsabilidad nuestra comprobar periódicamente durante el trabajo en segundo plano si nos han pedido que paremos, llamando al método isCancelled(), y terminar en ese caso. Cuando una tarea termina habiendo sido cancelada, no se llamará al método onPostExecute(), sino al método onCancelled(RESULT), también desde la hebra principal.

1. Haz una copia de la práctica anterior.
2. Vamos a querer cancelar la tarea asíncrona cuando se pulse el botón una segunda vez. Para poder hacerlo, crea un atributo en la clase.
3. Cambia el código del evento del ratón para que cancele la tarea si se pulsó una segunda vez:

```
CalcularPrimos cp;
public void onPulsame(View v) {
if (cp == null) {
cp = new CalcularPrimos();
```

```
cp.execute();
}
else
cp.cancel(true);
} // onPulsame
```

4. Modifica la clase asíncrona para que compruebe de vez en cuando si se ha pedido que se termine. Para no comprobarlo contínuamente, podemos hacerlo únicamente cuando encontremos un número primo.

```
public long cuantosPrimos(long limite) {
long result = 0;
for (long i = 1; i <= limite; ++i)
if (esPrimo(i)) {
++result;
// Miramos si nos han cancelado. Sólo si es primo
// para mirarlo menos veces.
if (isCancelled())
break;
} // if esPrimo
return result;
}
```

5. Sobreescribe el método onCancelled(), refactorizando el código de onPostExecute() para no repetir.

```
@Override
protected void onPostExecute(Long resultado) {
terminar(resultado.toString());
}
@Override
protected void onCancelled(Long resultadoParcial) {
terminar("½Cancelado! Llevaba " + resultadoParcial);
}
protected void terminar(String s) {
TextView tv = (TextView) findViewById(R.id.tvResultado);
tv.setText(s);
cp = null;
}
```

6. Prueba la aplicación. Pulsa el botón mientras se están contando primos y comprueba que la operación se cancela.

Práctica 10.8: Progreso de la AsyncTask

Para poder mostrar una barra de desplazamiento, AsyncTask proporciona dos métodos más:

- **void publishProgress(PROGRESS... values)**: recibe una lista de parámetros del tipo PROGRESS, el tercer argumento del tipo genérico que dejamos pendiente en la práctica 10.6. Debe ser llamado desde doInBackground() cuando queremos informar de un avance en el progreso.
- **onProgressUpdate(PROGRESS... values):** se ejecuta en la hebra principal cada vez que desde la hebra trabajadora se llama al método publishProgress(...). Los parámetros recibidos son los que se pasaran a éste.

Por completar, AsyncTask tiene un último método, onPreExecute(), que puede ser sobreescrito y que será llamado desde la hebra principal en el momento de ir a lanzar el procesado.

1. Haz una copia de la práctica anterior.
2. Modifica el layout para añadir una barra de progreso horizontal debajo de la etiqueta, y déjala oculta.

```
<ProgressBar
android:id="@+id/pbCalcularPrimos"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:visibility="invisible"
android:max="100"
style="?android:attr/progressBarStyleHorizontal"/>
```

3. Modifica la clase interna CalcularPrimos para que tenga como segundo argumento el tipo Integer (será el que hemos llamado PROGRESS).
4. Añade como atributo una ProgressBar que nos evite tener que buscarla en la vista continuamente.
5. Sobreescribe el método onPreExecute() para que se incialice su valor (usando findViewById()), y se haga visible.

6. Sobreescribe el método onProgressBarUpdated(Integer...) para que se establezca como progreso en la barra el valor del parámetro. Ten en cuenta que al ser una lista de parámetros opcionales, lo recibirás como un array, que será de un solo elemento.
7. Modifica el método cuantosPrimos() para que se actualice la barra de progreso. Evita llamar a publishProgress() con el mismo valor para no forzar a repintados y sincronizaciones superfluas.
8. Modifica el método terminar() para que se oculte la barra de progreso.

```
protected ProgressBar pb;
public long cuantosPrimos(long limite) {
long result = 0;
long lenStep = limite / 100;
int prevProgress = -1;
for (long i = 1; i <= limite; ++i) {
if (esPrimo(i)) {
++result;
if (isCancelled())
```

```
break;
if (prevProgress != i / lenStep) {
prevProgress = (int)(i / lenStep);
publishProgress(prevProgress);
}
} // if esPrimo
} // for
return result;
} // cuantosPrimos
@Override
protected void onPreExecute() {
pb = (ProgressBar) findViewById(R.id.pbCalcularPrimos);
pb.setVisibility(View.VISIBLE);
}
[ ... ]
protected void terminar(String s) {
TextView tv = (TextView) findViewById(R.id.tvResultado);
tv.setText(s);
pb.setVisibility(View.INVISIBLE);
cp = null;
}

@Override
protected void onProgressUpdate(Integer... values) {
pb.setProgress(values[0]);
}
```