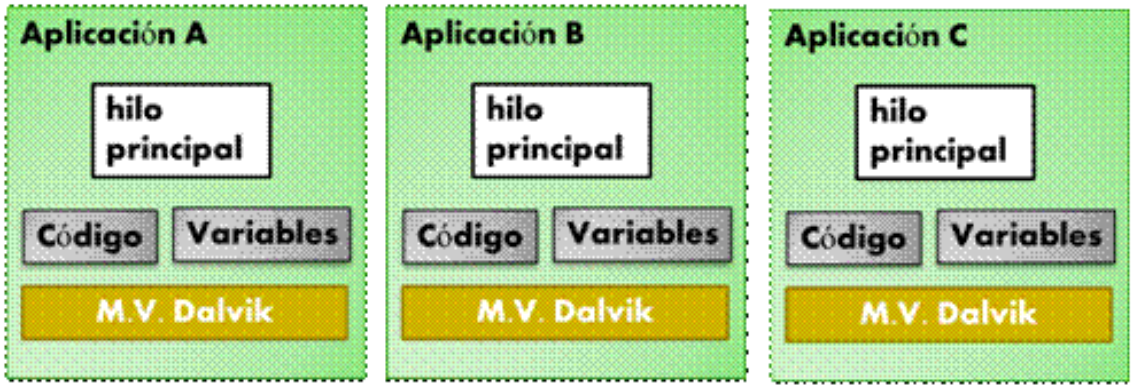
HILOS (Threads)

Uso de hilos de ejecución (Threads)

Introducción a los procesos e hilos de ejecución

Cada vez que se lanza una nueva aplicación en Android el sistema crea un nuevo proceso Linux para ella y la ejecuta en su propia máquina virtual Dalvik (Por supuesto si está programada en Java, si lo estuviera en código nativo no haría falta la máquina virtual). Trabajar en procesos diferentes nos garantiza que desde una aplicación no se pueda acceder a la memoria (código o variables) de otras aplicaciones.



Los Sistemas Operativos modernos incorporan el concepto de *hilo de ejecución* (*thread*). En un sistema multihilo un proceso va a poder realizar varias tareas a la vez, cada una en un hilo diferente. Los diferentes hilos de un proceso lo comparten todo: variables, código, permisos, ficheros abiertos, etc.

Cuando trabajamos con varios hilos, estos pueden acceder a las variables de forma simultánea. Hay que tener cuidado de que un hilo no modifique el valor de una variable mientras otro hilo está leyéndola. Este problema se resuelve en Java definiendo secciones críticas mediante la palabra reservada ***synchronized***.

Hilos de ejecución en Android

Cuando se lanza una nueva aplicación el sistema crea un nuevo hilo de ejecución (*thread*) para esta aplicación conocida como **hilo principal**. Este hilo es muy importante dado que se encarga de atender los eventos de los distintos componentes. Es decir, este hilo ejecuta los métodos `onCreate()`, `onDraw()`, `onKeyDown()`, … Por esta razón al hilo principal también se le conoce como **hilo del interfaz de usuario**.

El sistema no crea un hilo independiente cada vez que se crea un nuevo componente. Es decir, todas las actividades y servicios de una aplicación son ejecutados por el hilo principal.

Cuando una aplicación ha de realizar trabajo intensivo como respuesta a una interacción de usuario, hay que tener cuidado porque es posible que la **aplicación no responda de forma adecuada**. Por ejemplo, imagina que tienes que esperar para descargar unos datos de Internet, si lo haces en el hilo de ejecución principal este quedará bloqueado a la espera de que termine la descarga. Por lo que, no se podrá redibujar la vista (`onDraw()`) o atender eventos del usuario (`onKeyDown()`). Desde el punto de vista del usuario se tendrá la impresión de que la aplicación se ha colgado. Más todavía, si el hilo principal es bloqueado durante **más de 10-15 segundos**, el sistema mostrará un cuadro de dialogo al usuario ***"La aplicación no responde"*** para que el usuario decida si quieres esperar o detener la aplicación.

La solución en estos casos es crear un **nuevo hilo de ejecución**, para que realice este trabajo intensivo. De esta forma no bloqueamos el hilo principal, que puede seguir atendiendo los eventos de usuario. Es decir, cuando estés implementando un método del hilo principal (empiezan por `on`…) nunca realices una acción que pueda bloquear este hilo, como cálculos largos o que requieran esperar mucho tiempo. En estos casos hay que crear un nuevo hilo de ejecución y encomendarle esta tarea.

Las herramientas del interfaz de usuario de Android han sido diseñadas para ser ejecutadas desde un único hilo de ejecución, el hilo principal. Por lo tanto no se permite manipular el interfaz de usuario desde otros hilos de ejecución.

Ejemplo: Una tarea que bloquea el hilo principal.

En muchas ocasiones tenemos que realizar costosas operaciones o tenemos que esperar a que concluyan lentas operaciones en la red. En ambos casos, hay que tener la precaución de no bloquear el hilo principal. De hacerlo el resultado puede ser catastrófico, como se puede ver en el siguiente ejemplo.

1. Crea un nuevo proyecto y llámalo **Hilos**:
2. En el fichero de *layout* *activity_main.xml* escribiremos el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.example.amaia.ud03ejhilos.MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content">

        <EditText
            android:id="@+id/edEntrada"
            android:layout_width="0dip"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:inputType="numberDecimal"
            android:text="5">
            <requestFocus />
        </EditText>

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:onClick="calcularOperacion"
            android:text="Calcular Factorial"/>

    </LinearLayout>

    <TextView
        android:id="@+id/tvSalida"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text=""
        android:textAppearance="?android:attr/textAppearanceMedium"/>
</LinearLayout>
```

3. Y en el fichero Java *MainActivity.java* este otro código:

```
package com.example.ejthreads;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    private EditText edEnrada;
    private TextView tvSalida;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        edEnrada = findViewById(R.id.edEntrada);
        tvSalida = findViewById(R.id.tvSalida);

    }
```

```
    //Sin utilizar hilos
    public void calcularOperacion (View v){
        int n = Integer.parseInt(edEnrada.getText().toString());
        tvSalida.append(n + "! = ");
        int res = factorial (n);
        tvSalida.append(res + "\n");
    }

    private int factorial (int num){
        int resultado =1;
        for (int i=1; i<=num; i++){
            resultado *= i;
            SystemClock.sleep(1000);  //Esperamos un segundo
        }
        return resultado;
    }

}
```

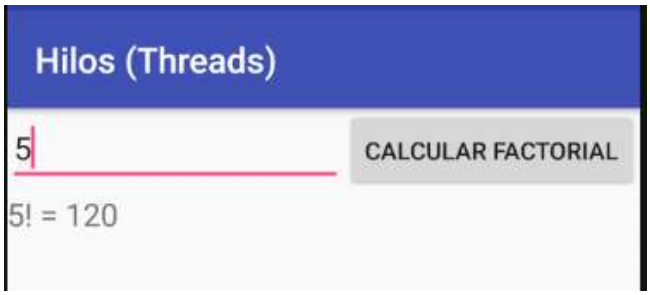
El método **`calcularOperacion()`** será llamado cuando se pulse el botón *Calcular Factorial*. Comienza obteniendo el valor entero introducido en `etEntrada` y muestra la operación a realizar por el `TextView tvSalida`. Luego, se llama al método **`factorial()`** y se muestra el resultado.

El método **`factorial()`** calcula la operación matemática factorial de un entero *n* (*n*!). Se calcula multiplicando todos los enteros desde uno hasta *n*. Por ejemplo:

$$\text{factorial}(5) = 5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

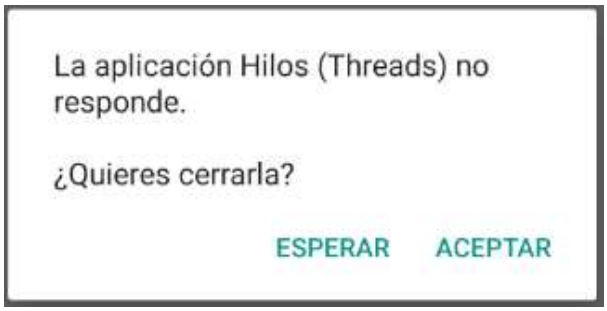
Esta operación se calcula con un bucle con la variable `i` tomando valores de 1 hasta `num`. Obviamente, esta operación se va a realizar de forma muy rápida y apenas bloquearemos el hilo principal una milésimas de segundo. Para que aparezca el problema a tratar, vamos a simular que se realizan un gran número de operaciones en cada pasada del bucle. Para esto llamaremos a `SystemClock.sleep(1000)` que bloqueará el hilo durante 1000 ms (1 segundos).

4. Ejecuta la aplicación. El resultado ha de ser similar al siguiente:



Observa cómo mientras se realiza la operación el usuario no puede pulsar el botón ni modificar el `EditText`. El usuario tendrá la sensación de que la aplicación está bloqueada.

5. Calcula ahora el factorial de 10. Si mientras se calcula tratas de interaccionar con el interfaz de usuario el sistema nos mostrará el siguiente error:



Mensajes del tipo "*La aplicación no responde*" son frecuentes en Android. Aparecen cuando el hilo principal se bloquea demasiado tiempo.

Creación de nuevos hilos con la clase Thread

Como acabamos de ver, siempre que tengamos que ejecutar un método que requiera bastante tiempo de ejecución, no podremos ejecutarlo en el hilo principal. Dado que este hilo ha de estar siempre disponible para atender los eventos generados por el usuario, nunca debe ser bloqueado. A continuación veremos cómo crear nuevos hilos usando la clase de Java **`Thread`**. El proceso es muy sencillo, no tendremos más que escribir una clase como la siguiente:

```
class MiThread extends Thread {

    @Override
    public void run() {
     . . .
    }
}
```

El método `run()` contiene el código que queremos que el hilo ejecute. Para crear un nuevo hilo y ejecutarlo escribiremos:

```
MiThread thread = new MiThread();
thread.start();
```

La llamada al método `start()` ocasionará que se cree un nuevo hilo y se ejecute el método `run()` en este hilo. La llamada al método `start()` es asíncrona. Es decir, continuaremos ejecutando las instrucciones siguientes, sin esperar a que el método `run()` acabe. Como veremos más adelante, si esperamos algún resultado de este método, será imprescindible establecer algún mecanismo de sincronización para saber cuándo ha terminado.

Ejercicio paso a paso: Crear un nuevo hilo con la clase Thread

En este ejemplo ejecutaremos el método `factorial()` en un hilo nuevo. Además, veremos algunas limitaciones de usar nuevos hilos, como la imposibilidad de acceder al interfaz de usuario.

1. En el proyecto creado anteriormente, dentro de `MainActivity` introduce el siguiente código:

```
class MiThread extends Thread {

    private int n, res;
```

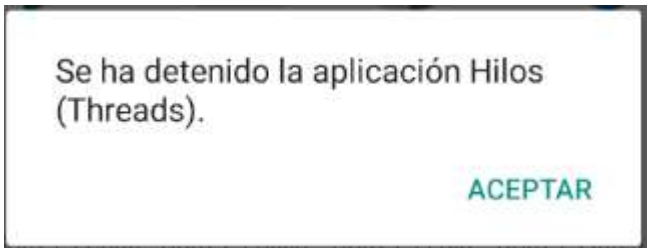
```
        public MiThread(int n) {
            this.n = n;
        }

        @Override
        public void run() {
            int res = factorial(n);
            tvSalida.append(res + "\n");
        }
    }
```

Siguiendo el esquema mostrado anteriormente, hemos creado un hilo que en su método `run()`, llama a `factorial()` y muestra el resultado por pantalla. Para realizar la operación necesitamos el parámetro de entrada `n`. Este no puede incorporarse en el método `run()` dado que ha de ser sobrescrito (`@Override`) sin alteración alguna. Para resolverlo hemos añadido un constructor a la clase, donde se indica este parámetro.

2. Reemplaza el método `calcularOperacion()` en el fichero `MainActivity.java`:

```
//Utilizando hilos
public void calcularOperacion (View v){
    int n = Integer.parseInt(edEnrada.getText().toString());
    tvSalida.append(n + "! = ");
    MiThread thread = new MiThread(n);
    thread.start();
}
```

3. Ejecuta la aplicación. Tras pulsar el botón el resultado ha de ser:



4. Abre la vista *LogCat* y busca el siguiente error:

```
11-15 09:41:35.722 5767-5805/com.example.ejthreads E/AndroidRuntime: FATAL EXCEPTION: Thread-286
    Process: com.example.ejthreads, PID: 5767
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
        at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:6556)
        at android.view.ViewRootImpl.invalidateChildInParent(ViewRootImpl.java:943)
```

Este mensaje indica que solo desde el hilo principal se puede interactuar con las vistas de la interfaz de usuario. **NOTA:** *También está prohibido desde otros hilos usar la clase Toast*

Una forma elegante de resolver este problema podría ser usar la clase `Callable`. Esta clase de Java nos permite llamar un método en un nuevo hilo, esperar a que este termine y recoger los resultados. No obstante, nuestro objetivo es mostrar las peculiaridades de Android con el manejo de hilos y vamos a resolver el problema de otra forma.

5. En el método `run()` reemplaza:

```
int res =factorial(n);
tvSalida.append(res + "\n");
```

por

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        int res = factorial(n);
        tvSalida.append(res + "\n");
    }
});
```

Quedando de la siguiente manera:

```
@Override
public void run() {

    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            int res = factorial(n);
            tvSalida.append(res + "\n");
        }
    });
}
```

De esta forma indicamos al sistema que ejecute parte de nuestro código en el hilo principal o hilo del interfaz de usuario.

6. Ejecuta la aplicación y comprueba que el resultado es satisfactorio.

Ejecutar una tarea en un nuevo hilo con AsyncTask

En Android es muy frecuente lanzar nuevos hilos. Habrá que hacerlo siempre que exista la posibilidad de que una tarea pueda bloquear el hilo del interfaz de usuario. Esto suele ocurrir en cálculos complejos o en accesos a la red.

Tras estudiar el uso de las herramientas estándares en Java para crear hilos; a continuación vamos a ver una clase creada en Android que nos ayudará a resolver este tipo de problemas **de forma más sencilla, la clase AsyncTask.**

Una tarea asincrónica (*asinc tasck*) **se define por un cálculo que se ejecuta en un hilo secundario y cuyo resultado queremos que se publique en el hilo del interfaz de usuario**. Para crear una nueva tarea asíncrona nos basamos en el siguiente esquema:

```
class MiTarea extends AsyncTask <Parametros, Progreso, Resultado> {

    @Override
    protected void onPreExecute() {
        …
    }
```

```
    @Override
    protected Resultado doInBackground(Parametros... par) {
        …
    }
    @Override
    protected void onProgressUpdate(Progreso... prog) {
        …
    }
    @Override
    protected void onPostExecute(Resultado result) {
        …
    }
}
```

donde **`Parametros`**, **`Progreso`** y **`Resultado`** han de ser reemplazados por **nombres de clases** según los tipos de datos con los que trabaje la tarea.

Los cuatro métodos que se pueden sobrescribir corresponden a los cuatro pasos que seguirá `AsyncTask` para ejecutar la tarea:

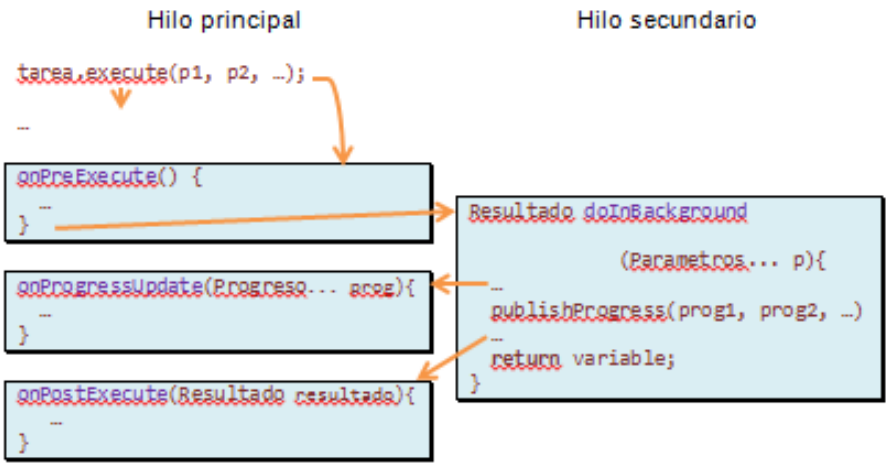
- **`onPreExecute()`**: En este método tenemos que realizar los trabajos previos a la ejecución de la tarea. Se utiliza normalmente para configurar la tarea y para mostrar en el la interfaz de usuario que empieza la tarea.
- **`doInBackground(Parametros...)`**: Es llamado cuando termina `onPreExecute()`. Es la parte más importante donde tenemos que realizar la tarea propiamente dicha. **Es el único método de los cuatro que no se ejecuta en el hilo del interfaz de usuario**. Lo va a hacer en un hilo nuevo creado para este propósito. Como hemos visto la clase `AsyncTask` ha de ser parametrizada con tres tipos de datos. Es decir, cuando se crea un `AsyncTask` la clase `Parametros` ha de ser reemplazada por una clase concreta que será utilizada para indicar la información de entrada a la tarea. Observa como en el parámetro de este método se han añadido tres puntos detrás de `Parametros`. Esto significa que se puede pasar al método un número variable de parámetros de esta clase.
- **`onProgressUpdate(Progress...)`:** Este método se utiliza para mostrar el progreso de la tarea al usuario. Se ejecuta en el hilo interfaz de usuario, por lo que podremos interactuar con las vistas. El progreso de una determinada tarea ha de ser controlado por el programador llamando al método `publishProgress(Progress...)` desde `doInBackground()`. La clase `Progress` es utilizada para pasar la información de progreso. Un uso frecuente es reemplazarla por `Integer` y representar el porcentaje de progreso en un valor entre el 0 y el 100.
- **`onPostExecute(Result)`:** Este método se usa para mostrar en el interfaz de usuario el resultado de la tarea. El parámetro de entrada (de la clase `Result`) corresponde con el objeto devuelto por el método `doInBackground()`.

Una vez definida la clase descendiente de `AsyncTask` podremos arrancar una tarea de la siguiente forma:

```
MiTarea tarea = new MiTarea();
tarea.execute(p1, p2, p3);
```

Donde `p1, p2, p3` ha de ser una lista de objetos de la clase `Parametros`, pudiendo introducirse un número variable de parámetros. Hay que resaltar que `execute()` es un método asíncrono. Esto significa que, tras llamarlo, se pondrá en marcha la tarea en otro hilo, pero en paralelo se continuarán ejecutando las instrucciones que se hayan escrito a continuación de `execute`. El nombre `AsyncTask` se ha puesto precisamente por este comportamiento.

En el siguiente diagrama se muestra el orden de ejecución de estos métodos:



Ejemplo: Crear un nuevo hilo con AsyncTask

En este ejemplo resolveremos la operación del ejemplo anterior pero ahora usando `AsyncTask` en lugar de `Thread`.

1. Abre el proyecto **Hilos** creado anteriormente.
2. Dentro de `MainActivity.java` introduce el siguiente código:

```
class MiTarea extends AsyncTask <Integer, Void, Integer>{

    @Override
    protected Integer doInBackground(Integer... n) {
        return factorial (n[0]);
    }

    @Override
    protected void onPostExecute(Integer result) {
        tvSalida.append(result + "\n");
    }
}
```

Cuando usamos esta clase hemos de comenzar decidiendo los tres tipos de datos que utilizaremos usando el siguiente esquema **`AsyncTask<Parametros, Progreso, Resultado>`**. En nuestra tarea necesitamos **un entero como entrada, no usaremos información de progreso y devolveremos un entero**. Estos tres tipos de datos solo pueden ser clases. Si queremos utilizar un tipo simple, como `int`, tendremos que usar una clase envolvente, como `Integer`.

En el método `doInBackground()` se ha indicado como parámetro **`Integer...`**, de manera que se le podrán pasar una lista variable de enteros. Aunque, en nuestro caso solo nos interesa el primero (`n[0]`), estamos obligados a sobrescribir el método exactamente como se espera, y no podemos quitar los `...`. En este método nos limitamos a calcular el factorial y devolverlo. Al terminar este método será llamado `onPostExecute()`, pasándole como parámetro el valor devuelto. Para terminar, recuerda que el método `doInBackground()` se ejecutará en un nuevo hilo, mientras que `onPostExecute()` se ejecutará en el hilo del interfaz de usuario.

3. Comenta las dos últimas líneas del método **`calcularOperacion`** y añade las siguientes:

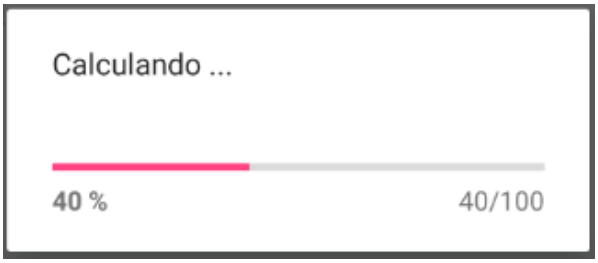
```
MiTarea tarea = new MiTarea();
tarea.execute(n);
```

La llamada a `execute()` provocará que los diferentes métodos definidos en *MiTarea* sean llamados en el orden adecuado.

4. Comprueba que funciona correctamente.

Mostrar un cuadro de progreso en un AsyncTask

Si estamos realizando una tarea que puede prolongarse en el tiempo, resulta muy importante mostrar al usuario cuando empieza, su progreso y cuando termina. En el siguiente ejercicio vamos a ver un ejemplo algo más complejo de `AsyncTask`, donde usaremos la clase `ProgressDialog` para mostrar la evolución de la tarea.



ProgressDialog.STYLE_HORIZONTAL

ProgressDialog.STYLE_SPINNER

Siguiendo con el ejemplo anterior, reemplaza la clase `MiTarea` por el código:

```
class MiTarea extends AsyncTask<Integer, Integer, Integer> {
    private ProgressDialog progreso;

    @Override
    protected void onPreExecute() {
        progreso = new ProgressDialog(MainActivity.this);
        progreso.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progreso.setMessage("Calculando ...");
        progreso.setCancelable(false);
        progreso.setMax(100);
        progreso.setProgress(0);
        progreso.show();
    }
```

```
        @Override
        protected Integer doInBackground(Integer... n) {
            int res = 1;
            for (int i=1; i<= n[0]; i++){
                res *= i;
                SystemClock.sleep(1000);
                publishProgress(i*100 / n[0]);
            }
            return res;
        }

        @Override
        protected void onProgressUpdate(Integer... porc) {
            progreso.setProgress(porc[0]);
        }

        @Override
        protected void onPostExecute(Integer result) {
            progreso.dismiss();
            tvSalida.append(result + "\n");
        }
    }
```

En esta nueva versión se han incluido los cuatro métodos principales de `AsynTask`. El primero en ejecutarse será **`onPreExecute()`**, donde creamos un `ProgressDialog` lo configuramos y lo mostramos. Cuando acabe, se ejecuará **`doInBackground()`**. En esta versión no podemos llamar simplemente a `factorial()`, dado que ahora queremos insertar en el bucle la sentencia `publishProgress(i*100/n[0])`. Lo que ocasionará una llamada a **`onProgressUpdate()`**, desde donde tendremos acceso al interfaz de usuario y podremos actualizar el `ProgressDialog`. Finalmente cuando `doInBackground()` termine se llamará a `onPostExecute()`, donde destruiremos el `ProgressDialog` y mostraremos el resultado.

Ejemplo:Cancelando un AsyncTask

Dado que `AsyncTask` se utiliza en tareas prolongadas (largas), es posible que el usuario no quiera esperar a que termine o que descubra en medio del proceso que no puede terminar la tarea. Se puede utilizar el método `cancel()` cuando ocurra esta circunstancia. Si se cancela una tarea el método `onPostExecute(Resultado)` no será llamado, y en su lugar se llamará a `onCanceled()`. El siguiente ejemplo lo vemos:

1. En el método `onPreExecute()` del `AsyncTask` cambia el parámetro `setCancelable()` de false a true;

```
progreso.setCancelable(false);    →    progreso.setCancelable(true);
```

2. Añade a continuación de la línea que acabas de modificar:

```
progreso.setOnCancelListener(new DialogInterface.OnCancelListener() {
    @Override
    public void onCancel(DialogInterface dialog) {
        MiTarea.this.cancel(true);
    }
});
```

Con esto conseguimos poner un escuchador de evento al `ProgressDialog`, para que cuando sea cancelado también cancele el `AsyncTask`.

3. Dentro de `doInBackground()` añade la siguiente condición de finalización en el bucle for:

   ```
   for (int i=1; i<= n[0] && !isCancelled(); i++){
   ```

   De esta forma no seguiremos realizando la tarea si nos cancelan.

4. Para terminar añade el siguiente método en la clase `AsyncTask`:

   ```
   @Override
   protected void onCancelled() {
       tvSalida.append("Cancelado\n");
   }
   ```

5. Verifica el resultado.

```
class MiTarea extends AsyncTask<Integer, Integer, Integer> {
    private ProgressDialog progreso;

    @Override
    protected void onPreExecute() {
        progreso = new ProgressDialog(MainActivity.this);
        //progreso.setProgressStyle(ProgressDialog.STYLE_SPINNER);
        progreso.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progreso.setMessage("Calculando ...");
        progreso.setCancelable(true);
        progreso.setOnCancelListener(new DialogInterface.OnCancelListener(){
            @Override
            public void onCancel(DialogInterface dialog) {
                MiTarea.this.cancel(true);
            }
        });
        progreso.setMax(100);
        progreso.setProgress(0);
        progreso.show();
    }

    @Override
    protected Integer doInBackground(Integer... n) {

        int res = 1;
        for (int i=1; i<= n[0] && !isCancelled(); i++){
            res *= i;
            SystemClock.sleep(1000);
            publishProgress(i*100 / n[0]);
        }
        return res;
    }

    @Override
    protected void onProgressUpdate(Integer... porc) {
        progreso.setProgress(porc[0]);
    }

    @Override
    protected void onPostExecute(Integer result) {
        progreso.dismiss();
```

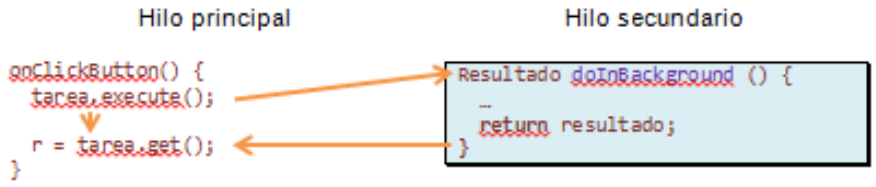
```
            tvSalida.append(result + "\n");
        }

        @Override
        protected void onCancelled() {
            tvSalida.append("Cancelado\n");
        }
    }
```

El método get() de AsyncTask

En caso de necesitar el resultado de esta tarea para poder continuar ejecutando tu programa puedes utilizar el siguiente método:

```
Resultado r = tarea.get();
```

Lo que hace es esperar a que termine la tarea y devuelve el resultado obtenido. Aunque parezca muy útil, este método no es recomendable usarlo dado que se bloquea hasta que termine la tarea, y esto es justo lo que queríamos evitar al crear el `AsyncTasck`. Veamos un ejemplo con el siguiente esquema:



Si el método **`onClickButton()`** es asociado a la pulsación de un botón y este se pulsa, has de tener claro que el hilo principal quedará bloqueado hasta que termine la tarea.

El método **`get()`** dispone de una sobrecarga alternativa que resulta muy práctica. En ella indicamos dos parámetros, donde fijamos un tiempo máximo de la tarea y en que unidades está este tiempo. Por ejemplo, si escribimos **`get(4, TimeUnit.SECONDS)`**, pasados 4 segundos se detendrá la tarea y se lanzará una excepción. **Usar este método puede resultar interesante cuando no tenemos más remedio que bloquear el hilo del interfaz del usuario hasta que termine una tarea.**

Vemos un ejemplo de uso. Si hemos creado un `AsyncTask` que valida un usuario en nuestro servidor podríamos usar el siguiente método:

```
public boolean onLogin(String usuario, contrasena){
        try{
              TareaLogin tarea = new TareaLogin();
              tarea.execute(usuario, contrasena);
              return tarea.get(4, TimeUnit.SECONDS);
        } catch(TimeoutException e) {
              Toast.makeText(contexto, "Tiempo excedido al validar",
                          Toast.LENGTH_LONG).show();
        } catch(CancellationException e) {
              Toast.makeText(contexto, "Error al conectar con servidor",
```

```
                        Toast.LENGTH_LONG).show();
            } catch(Exception e) {
                  Toast.makeText(contexto, "Error con tarea asíncrona",
                        Toast.LENGTH_LONG).show();
            }
      returnfalse;
}
```

Cuando este método sea invocado hay que tener claro que **el hilo del interfaz de usuario se va a bloquear hasta que termine la tarea**. Sin embargo, en este caso particular no queremos que el usuario realice ninguna acción hasta ser validado. Por lo que no se apreciará falta de interactividad. Además, estamos limitando este bloqueo a un máximo de 4 segundos, impidiendo que aparezca el error “La aplicación no responde”. En caso de producirse cualquier problema será tratado en la sección `catch`, donde se mostrará al usuario el error que se ha producido. `TimeoutException` ocurrirá si la tarea tarda más de 4 segundos. `CancellationException` ocurrirá si el método `cancel()` es invocado dentro de la tarea, supuestamente si el servidor no responde o si la contraseña no es correcta. Pueden producirse un par de tipos de excepciones más relacionadas con hilos de ejecución. Ambas son capturadas mediante la excepción genérica `Exception`.