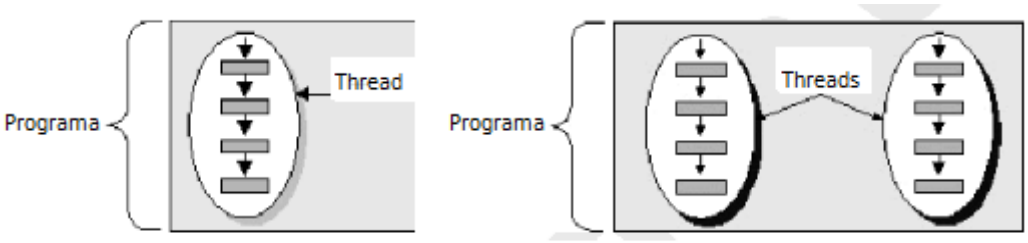
Threads

Una regla importante de las interfaces con el usuario, es que **siempre** deben responder a las acciones que el usuario realice, la pantalla jamás debe quedar congelada, de ser así se lanzará una excepción. Esto obliga al programador a realizar ciertas tareas que requieren tiempo (conexiones a internet, procesamiento de datos, etc.) en hilos de programa que se ejecuten en paralelo con la interfaz de usuario.



Si una Activity se bloquea por más de unos segundos, se lanzará una excepción y la aplicación se cerrará.

Para evitar este problema, y darle al usuario una sensación de que la aplicación esta respondiendo todo el tiempo, utilizaremos Threads, es decir, lanzaremos ejecuciones de programa en paralelo con el hilo principal.

Existen dos formas de utilizar Threads:

- Utilizando la clase Thread de Java
- Utilizando Asynctasks

Utilizando la clase Thread de Java

Hay dos formas de crear Threads, se puede definir una clase que herede de Thread, y hacer Override del método "run()" (en donde se ejecutará en paralelo el código que coloquemos allí dentro) o puede definirse una clase que implemente la interfaz Runnable, e implementar el método "run()" de la misma forma que antes. La diferencia entre las dos maneras, está en la forma de disparar el Thread:

Ejemplo heredando de Thread:

Definimos una clase que hereda de Thread

```
public class MyThread extends Thread{

	@Override
	public void run() {
		for(int i=0 ; i<10 ; i++){
			Log.d("1","Ejecucion desde Thread");
		}
	}
}
```

En la Activity, creamos un objeto de esta clase y ejecutamos el método "start()" de la clase padre, de este modo, el método "run()" comenzará a ejecutarse en un Thread aparte.

```
MyThread myThread = new MyThread();
myThread.start();
```

Ejemplo implementando Runnable:

Definimos una clase que implementa Runnable, e implementamos el método "run()" con el mismo código que antes:

```
public class MyThread implements Runnable{

	public void run() {
		for(int i=0 ; i<10 ; i++){
			Log.d("1","Ejecucion desde Thread");
		}
	}
}
```

En la Activity, debemos crear dos objetos: uno del tipo MyThread (nuestra clase) y otro del tipo Thread:

```
MyThread myThread = new MyThread();
Thread t = new Thread(myThread);
t.start();
```

Al objeto Thread le pasamos como parámetro nuestro objeto Runnable, luego llamamos el método "start()" del objeto Thread. El resultado es el mismo que en el ejemplo anterior.

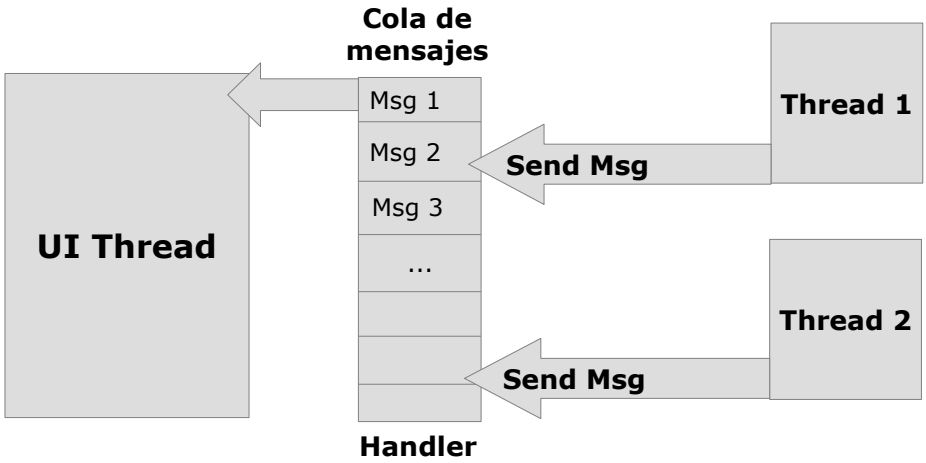
La interface Runnable proporciona un método alternativo a la utilización de la clase Thread, para los casos en los que no es posible hacer que nuestra clase herede de Thread. Esta situación puede darse cuando nuestra clase ya hereda de alguna otra, recordemos que en java no existe la herencia múltiple, pero sí pueden implementarse muchas interfaces, de modo que puede utilizarse Runnable variando ligeramente la forma en que se crean e inician los nuevos threads.

Comunicando la Activity con el Thread

No es posible refrescar los elementos de la interfaz gráfica del usuario (GUI) desde un thread que no sea el de la interfaz gráfica del usuario. Si el programador hace esto, el OS lanzará una excepción y se cerrará la aplicación.

Esto es así, debido a que si se tienen más de dos Threads, es necesario sincronizar los métodos para evitar que modifiquen la interfaz de usuario al mismo tiempo, esto agrega una complejidad extra a la Activity, para evitarla, se diseñó el sistema para que no sea posible actualizar la GUI desde un Thread que no sea el de la GUI.

La forma que Android ofrece para comunicar Threads, es a través de las clases "Handler" y "Message"



El Thread de la GUI, creará un objeto Handler, y pasará una referencia de este objeto a todos los Threads con los que quiera comunicarse. Este objeto se encargará de "encolar" los mensajes de todos los Threads y de hacerlos llegar a la GUI de a uno, es decir, la sincronización queda encapsulada dentro de la clase Handler.

Ejemplo:

```
public class Pantalla1Activity extends Activity implements Callback{

	private Handler handler;

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

	  handler = new Handler(this);
	  Worker w = new Worker(handler);
	  Thread t = new Thread(w);
	  t.start();
	}

	@Override
	public boolean handleMessage(Message msg) {
		// Aca se reciben los mensajes
		return true;
	}
}
```

En el ejemplo, puede observarse que la Activity (GUI) crea un objeto del tipo Handler, este objeto, para ser creado, necesita un objeto del tipo Callback, que es el objeto en donde se depositarán los mensajes que se envían a través del Handler, en este caso, la propia Activity implementa la interfaz Callback (por eso pasa this en el constructor), y hace Override del método ”handleMessage()” el cual será llamado por objeto Handler para descargar los mensajes de su cola.

En este caso, creamos una clase llamada Worker que utiliza Threads, el objeto Handler es pasado como parámetro en el constructor, esta es la definición de la clase:

```
public class Worker implements Runnable{
	private Handler h;

	public Worker (Handler h)
	{
		this.h = h;
	}

	@Override
	public void run() {

		for(int i=0 ; i<10 ; i++)
		{
			sendMsg("Worker:"+i);

			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		sendMsg("Fin de thread worker");
	}

	private void sendMsg(String msg)
	{
		Message message = new Message();
		message.obj = msg;
		h.sendMessage(message);
	}
}
```

En esta clase, se recibe el objeto Handler en el constructor, y se guarda una referencia del mismo, ya que con él, se enviarán los mensajes a la Activity.

La clase Worker, tiene un método “run()” que itera 10 veces en un bucle con un delay de 1 segundo, es delay, se consigue mediante la llamada de un método estático de la clase Thread, llamado “sleep()”:

```
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
```

Por cada iteración, se llama al método "sengMsg()" definido en la propia clase. En este método, se utiliza el objeto Handler perteneciente a la GUI, para enviar un mensaje a la misma por medio del método "sendMessage()" el cual recibe como parámetro, un objeto del tipo Message.

```
Message message = new Message();
message.obj = msg;
h.sendMessage(message);
```

El objeto Message, posee 3 atributos que pueden ser cargados con información:

- **obj :** atributo tipo Object.
- **arg1 :** atributo tipo int.
- **arg2 :** atributo tipo int.

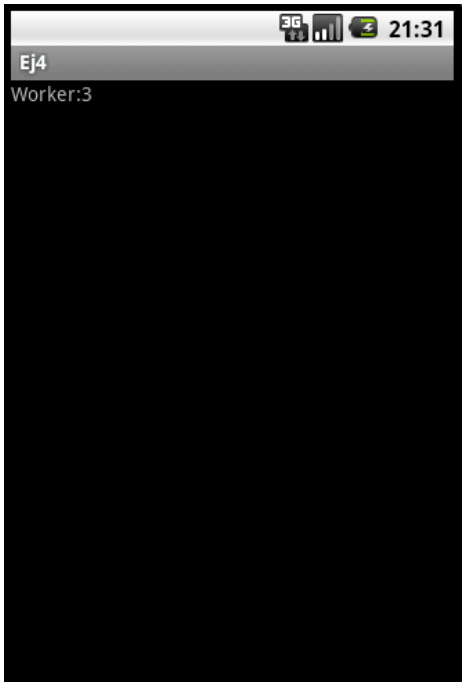
En el ejemplo, utilizamos el atributo "obj" para transmitir un objeto String desde el Thread Worker hacia la Activity.

Para recibir esta información en la Activity, dentro del método "handleMessage()" el cual recibe como parámetro el objeto Message:

```
@Override
public boolean handleMessage(Message msg) {

        String text = (String) msg.obj;
        TextView tv = (TextView) findViewById(R.id.lbl1);
        tv.setText(text);
        return true;
}
```

En este caso, casteamos el objeto del mensaje a String y se lo asignamos a un TextView, esto es posible porque la asignación se lleva a cabo en la GUI.

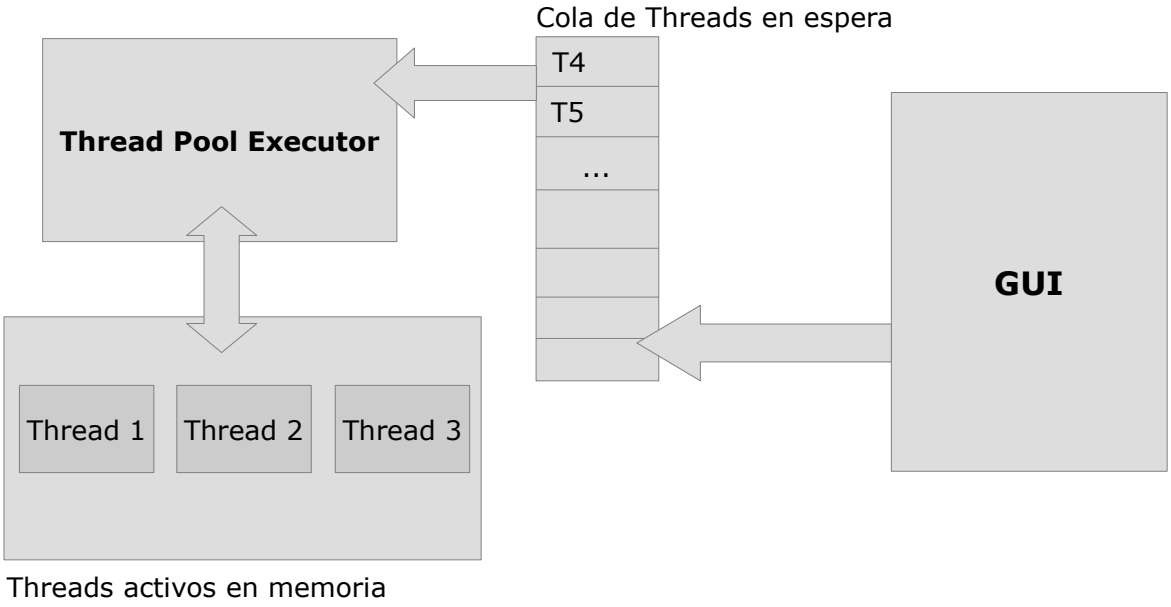
El resultado de la ejecución es la siguiente, el TextView se actualizará cada 1 segundo:



Pool de Threads

Imaginemos que tenemos una ListView, en donde la información de cada ítem es obtenida de Internet, supongamos que el programa disparará un Thread por cada ítem mostrado, el cual bajará información de Internet y la devolverá a la GUI principal, en donde se actualizará la pantalla. Esto es posible de lograr con todo lo mencionado anteriormente, pero supongamos que la cantidad de ítems de la lista es excesivamente grande, hay un riesgo de disparar demasiados Threads al mismo tiempo, sobrecargando al sistema.

Este problema es solucionado mediante el uso de un Pool de Threads, es decir, el sistema tendrá una cantidad prefijada de Threads en memoria, y una cola de Threads a ser ejecutados, si la cantidad que hay en la cola es mayor que la cantidad de Threads en memoria, los Threads quedarán en espera de que alguno de los que se este ejecutando termine, para comenzar.



Para el ejemplo, utilizaremos una Activity que muestra en pantalla 10 TextViews, los cuales son actualizados mediante Threads, pero el manejo de los Threads es llevado a cabo por la clase ThreadPoolExecutor, crearemos un objeto llamado "executor" de este tipo:

Se lanzará un Thread por cada TextView que exista en la lista, pero éstos se ejecutarán de a 3 a la vez, debido al manejo del ThreadPoolExecutor.

```
public class Pantalla1Activity extends Activity implements Callback{

      private ArrayList<TextView> listaTexts;
      private ThreadPoolExecutor executor;
      private Handler handler;

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

            // solo 3 threads a la vez
            executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
```

```
        listaTexts = new ArrayList<TextView>();
        listaTexts.add((TextView)findViewById(R.id.lbl1));
        listaTexts.add((TextView)findViewById(R.id.lbl2));
        listaTexts.add((TextView)findViewById(R.id.lbl3));
        listaTexts.add((TextView)findViewById(R.id.lbl4));
        listaTexts.add((TextView)findViewById(R.id.lbl5));
        listaTexts.add((TextView)findViewById(R.id.lbl6));
        listaTexts.add((TextView)findViewById(R.id.lbl7));
        listaTexts.add((TextView)findViewById(R.id.lbl8));
        listaTexts.add((TextView)findViewById(R.id.lbl9));
        listaTexts.add((TextView)findViewById(R.id.lbl10));

        handler = new Handler(this);

        // disparo todos los workers juntos, pero se ejecutan de a 3
        for(int i=0 ; i<listaTexts.size() ; i++)
        {
            Worker w = new Worker(handler,i);
            executor.execute(w);
        }
    }
      @Override
      public boolean handleMessage(Message msg) {

            TextView tv = listaTexts.get(msg.arg1);
            tv.setText((String)msg.obj);
            return true;
      }
}
```

Para crear un objeto del tipo "ThreadPoolExecutor" utilizamos el método estático "newFixedThreadPool()" de la clase "Executors" pasándole como parámetro la cantidad de Threads activos en memoria que queremos que se ejecuten simultáneamente.

```
executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
```

Luego creamos un ArrayList de TextViews, y los referenciamos con los 10 TexViews que están en pantalla.

```
listaTexts = new ArrayList<TextView>();
listaTexts.add((TextView)findViewById(R.id.lbl1));
//...
```

Después de eso, creamos un Handler y pasamos a la propia Activity como objeto Callback que recibirá los mensajes de los Threads, por esta misma razón, hacemos Override del método "handleMessage()" al igual que en los ejemplos anteriores.

El método "handleMessage()" recibirá mensajes que contendrán la siguiente información:

- msg.obj : objeto String con texto desde el Thread
- msg.arg1 : valor "int" con el número de TextView

De esta forma, la Activity sabe cual TextView debe modificar de la lista, según el valor de "arg1".

Describiremos ahora la clase Worker, que a diferencia de los ejemplos anteriores, además de recibir el Handler, en su constructor recibe un valor del tipo "int", el cual corresponderá al número de TextView de la lista, al que esta asociado este Thread.

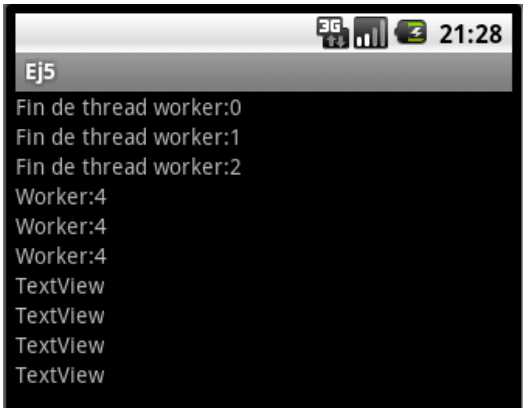
```
public class Worker implements Runnable{
	private Handler h;
	private int index;

	public Worker (Handler h,int index)
	{
		this.h = h;
		this.index=index;
	}
	@Override
	public void run() {

		for(int i=0 ; i<10 ; i++)
		{
			sendMsg("Worker:"+i);

			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}

		sendMsg("Fin de thread worker:"+index);
	}

	private void sendMsg(String msg)
	{
		Message message = new Message();
		message.obj=msg;
		message.arg1=index;
		h.sendMessage(message);
	}
}
```

Se observa que el constructor recibe el índice del tipo int, que luego es transmitido en el mensaje, en el atributo "arg1".

Por último, si volvemos a la Activity, veremos que el bucle "for" crea un thread por cada objeto que existe en la lista, pero en vez de ejecutar el método "start()" del Thread, como en los ejemplos anteriores, se pasan los Threads al objeto del tipo ThreadPoolExecutor mediante el método "execute()" :

```
// disparo todos los workers juntos, pero se ejecutan de a 3
for(int i=0 ; i<listaTexts.size() ; i++)
{
    Worker w = new Worker(handler,i);
    executor.execute(w);
}
```

La ejecución de la aplicación da el siguiente resultado:



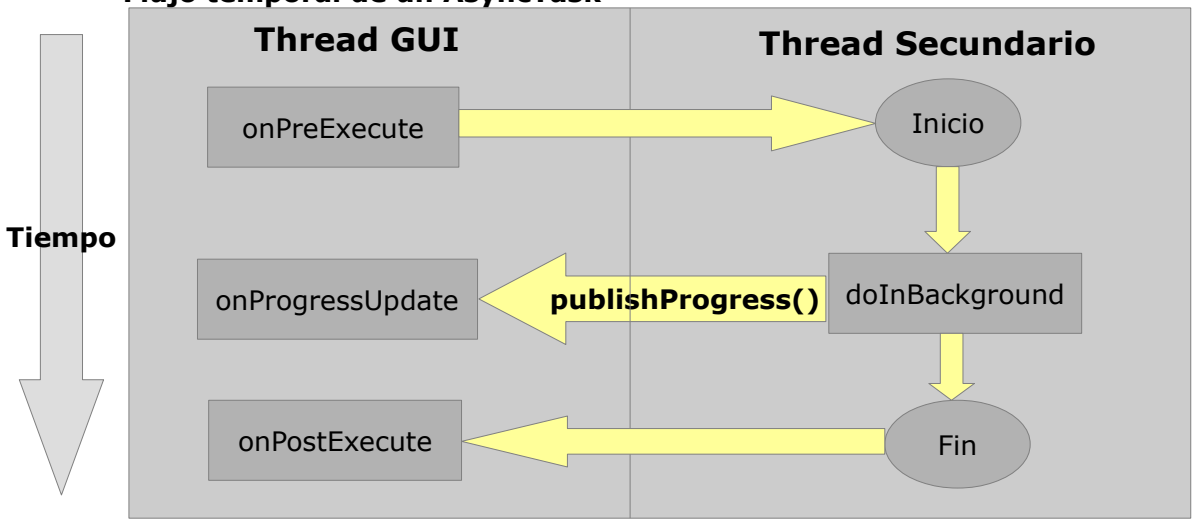
Utilizando AsyncTasks

Si lo que se pretende es utilizar un Thread que lleva a cabo una tarea, y comunica a la GUI el progreso y la finalización de la misma, utilizar AsyncTask provee una manera más simple de programar el código.

Para utilizar un AsyncTask, es necesario crear una clase que herede de AsyncTask, y hacer Override de 4 métodos:

- **onPreExecute() :** Se llamará al método antes de comenzar la tarea. Este método se ejecuta en el mismo thread que disparó el AsyncTask.
- **doInBackground() :** Aquí debe ponerse el código de la tarea a realizar. Este método se ejecuta en un Thread independiente, acá no puede actualizarse la GUI directamente.
- **onProgressUpdate() :** Se ejecuta al llamar a el método "publishProgress()", acá sí se puede actualizar la GUI, ya que este método también se ejecuta en el Thread que disparó al AsyncTask. El método "publishProgress()" y la clase padre, se encargan del manejo de los Handlers.
- **osPostExecute() :** Se llamará al método al finalizar "doInBackground()"

Flujo temporal de un AsyncTask



AsyncTask es una clase que utiliza Generics, posee 3 parámetros en donde hay que definir 3 tipos de datos:

```
AsyncTask<T,Q,R>
```

- T : Tipo de dato que recibirá el método "doInBackground()"
- Q : Tipo de dato que recibirá el método "onProgressUpdate()" (y en consecuencia, que recibirá "publishProgress()")
- R : Tipo de dato que devuelve "doInBackground()" y que recibe "onPostExecute()"

Es decir, que el valor que devuelve el método "doInBackground()", es cargado automáticamente como parámetro del método "onPostExecute()" antes de ser llamado. Este valor puede ser utilizado para determinar si la tarea terminó correctamente o no.

Por otro lado, también se deduce que el método "publishProgress()" recibirá un valor, que será pasado como parámetro al método "onProgressUpdate()" antes de ser llamado.

Ejemplo:

```
public class WorkerAsync extends AsyncTask<String,Integer,Boolean>
{
	TextView tv;

	public WorkerAsync(TextView tv)
	{
		this.tv=tv;
	}

	@Override
	protected void onPreExecute()
	{
		tv.setText("Inicio de thread worker");
	}
```

En el ejemplo podemos ver que hemos definido una clase llamada "WorkerAsync" la cual hereda de AsyncTask, de modo que pasamos los parámetros de los 3 tipos de datos que utilizaremos :

- doInBackground() recibirá un String.
- onProgressUpdate() recibirá un Integer (el valor del progreso).
- doInBackground() devolverá un Boolean ("true" si la tarea se llevó a cabo)

```
AsyncTask<String,Integer,Boolean>
```

En el constructor de nuestra clase, recibimos un objeto del tipo TextView, perteneciente a la Activity que disparará el AsyncTask, y que será modificado a lo largo de la tarea que realice la clase.

Luego, vemos el método "onPreExecute()", el cual modifica el texto del TexView, esto es posible porque este método se está ejecutando en el mismo Thread que la Activity, es decir, **todavía no se lanzó el nuevo Thread.**

A continuación, en la clase, de define el método "doInBackground()", el cuál devuelve el Boolean que se definió mediante generics, y recibe un String*. Este método simula una tarea haciendo un bucle de 10 iteraciones, una por segundo. Cada un segundo, llama a el método "publishProgress()" el cual recibe un Integer (como se definió mediante generics) que será pasado como parámetro al método "onProgressUpdate()", en este caso, el valor del Integer, corresponde con el numero de iteración (0 a 9). Al terminar la tarea, devuelve "true" este valor, será pasado como parámetro al método "onPostExecute()" el cual vuelve a modificar el TextView.

```
	@Override
	protected Boolean doInBackground(String... arg0)
	{
		for(int i=0 ; i<10 ; i++)
		{
			publishProgress(new Integer(i));
			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		return true;
	}

	@Override
	protected void onPostExecute(Boolean result)
	{
		tv.setText("Fin de thread worker:"+index+" result: "+result);
	}

	@Override
	protected void onProgressUpdate(Integer... values) {
		// TODO Auto-generated method stub
		super.onProgressUpdate(values);
		tv.setText("Valor de thread worker:"+values[0].intValue());
	}

}
```

Al llamar a "publishProgress()" y pasar el valor del progreso como parámetro, la clase padre creará los Handlers correspondientes, y hará todo el manejo que se mencionó anteriormente para la comunicación entre Threads, sin que el programador tenga que preocuparse por ello. Luego invocará al método "onProgressUpdate()" que se ejecutará en el Thread de la Activity, permitiendo actualizar la GUI dentro del mismo.

***Nota acerca de el parámetro de "doInBackground()" y "onProgressUpdate()"**

Puede observarse que el parámetro esta definido así:

```
doInBackground(String... arg0)
```

En vez de:

```
doInBackground(String arg0)
```

Esto es llamado "Varargs" , anteriormente, un método que recibe una arbitraria cantidad de parámetros, solo era posible de programar, agrupando los parámetros en un array y pasando el array al método.

A partir de Java 5, es posible definir un método con una arbitraria cantidad de parámetros, los cuales son tratados como un array dentro del método, pero como parámetros independientes fuera del mismo.

Ejemplo:

```
public void metodo(String... parametros)
{
      String a = parametros[0];
      String b = parametros[1];
}


metodo("uno","dos");
```

De modo que los métodos "doInBackground()" y "onProgressUpdate()" pueden recibir una cantidad arbitraria de parámetros, no solo uno, y para acceder a los mismos, se debe tratar al parámetro como un array, como se ve en el método "onProgressUpdate()" :

```
values[0].intValue()
```

Si solo se pasó un parámetro, se utilizará el índice 0.

Disparando el AsyncTask desde la Activity

Para ejecutar el AsyncTask, solo debe crearse y llamar al método "execute()" el cual pertenece a la clase padre:

```
TextView tv = (TextView)findViewById(R.id.lbl1);

WorkerAsync w = new WorkerAsync(tv);

w.execute("parametro");
```

El parámetro que se le pasa al método "execute()" es el que recibe el método "doInBackground()".

Desventajas de la clase AsyncTask

- Esta clase, internamente, maneja un Pool de Threads, pero el programador no puede setear la cantidad de Threads que posee este Pool, de echo, esta cantidad depende de la version del OS.
- No pueden lanzarse Excepciones desde el método "doInBackground()"

Manejo de Threads frente a perdidas de referencias

Existen muchas razones por la que una Activity puede dejar de existir, por ejemplo, cuando se rota el dispositivo, y la pantalla pasa de "Portrait mode" a "Landscape mode", la Activity se destruye y una nueva es creada.

Supongamos que la primer Activity lanzó un Thread que comienza a realizar una tarea, luego el usuario rota la pantalla, la Activity se destruye y se vuelve a crear, de modo que el Thread tiene referencias a objetos que ya no existen (por ejemplo, el objeto Handler que se definía en la Activity y que era pasado al Thread para poder comunicarse).

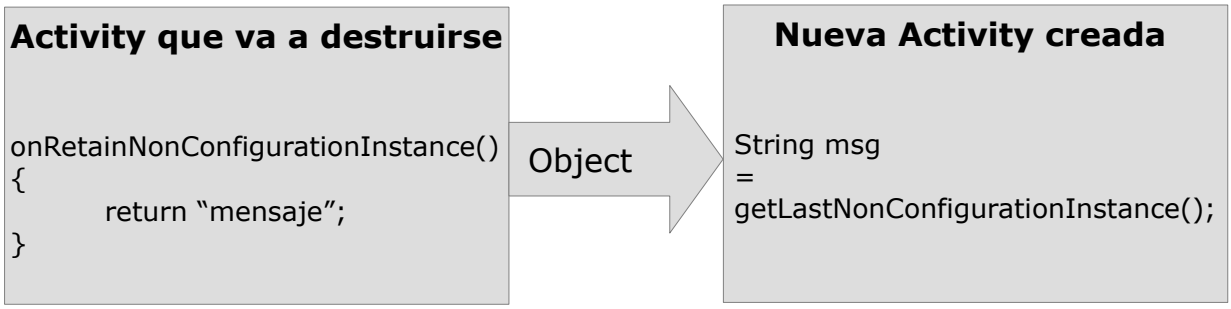
Cuando el Thread trata de usar el objeto des-referenciado, se lanzará una excepción y la aplicación se cerrará.

Esto genera dos problemas a solucionar:

- La nueva Activity no tiene la referencia del Thread que ya esta corriendo.
- El thread no tiene la referencia de la nueva Activity que se creó.

Solución al primer problema:

Existe una forma de pasar información entre la Activity que se destruye y la nueva, para ello, se debe hacer Override del método "onRetainNonConfigurationInstance()". En este método, que se ejecuta antes de destruirse la Activity, se devolverá un Object, el cual podrá ser obtenido por la nueva Activity creada, mediante el método "getLastNonConfigurationInstance()"



De esta forma, podemos pasar el Worker creado en la primer Actvity, a la segunda:

```
private WorkerAsync worker;

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

        // traigo el worker de la activity anterior
        worker =(WorkerAsync) getLastNonConfigurationInstance();
      }

    @Override
    public Object onRetainNonConfigurationInstance() {
      // paso el worker a la siguiente activity, cuando esta se destruye
      return worker;
    }
```

NOTA: onRetainNonConfigurationInstance y getLastNonConfigurationInstance están deprecados para la API >= 13 (Android 3.2) Para versiones nuevas se debe utilizar la API de Fragments con el método setRetainInstance()

Solución al segundo problema:

Ya pudimos obtener el objeto Worker, el cual tiene en ejecución un Thread aparte, en la nueva Activity creada, ahora necesitamos indicarle a este otro Thread, que hubo un cambio de referencias, debemos actualizar las referencias por las nuevas.
Para ello, definiremos dos métodos en la clase Worker:

- conectar()
- desconectar()

Antes de que la Activity se destruya (un buen lugar podría ser el método "onDestroy()") deberemos llamar al método del Worker, "desconectar()" para avisarle que, momentáneamente (hasta que la nueva Activity es creada), las referencias que posee no son validas.

Una vez obtenido el Worker en la nueva Activity, invocaremos al método "conectar()" de dicho objeto, pasándole a éste, las nuevas referencias, (Handlers, Views, la propia Activity, etc), dejando al Thread listo para seguir trabajando y terminar con la tarea.

El el Worker puede utilizarse un flag para saber si se debe seguir con la tarea o no, según si se ha llamado a "desconectar()" o no, la lógica de conectar y desconectar el worker queda a criterio del programador, solo debe tenerse en cuenta que mientras el worker este desconectado, no pueden enviarse mensajes a la Activity ya que la misma ha dejado de existir y se debe esperar a que se genere una nueva.

Ejemplo:

En la Activity:

```
@Override
protected void onDestroy() {
  super.onDestroy();
  // borro la referencia del worker a esta activity porque se va a destruir
  worker.desconectar();
}
```

y en el onCreate(), después de haber obtenido el Worker:

```
worker =(WorkerAsync) getLastNonConfigurationInstance();
if(worker==null){
      worker = new WorkerAsync(0);

      worker.conectar(this);
      worker.execute("parametro");
}
else{
   // conecto de nuevo el worker (actualizo la referencia a esta activity)
     worker.conectar(this);
}
```

En este ejemplo, el método conectar, recibe a la propia Activity. Puede observarse que si el worker es null, quiere decir que estamos en el caso de la primera Activity, todavía no se ha rotado la pantalla, de modo que el worker es creado, porque no existe uno previo.
En el caso de existir uno previo, solo se llama al método "conectar()" pasándole la referencia de la nueva Activity.

No mostraremos el código del worker debido a sus similitudes con el ejemplo anterior.

Solución alternativa: No recargar la Activity.

Si agregamos en el manifest, que la activity no se recargue al rotar la pantalla, ya no tendremos el problema mencionado anteriormente, para ello:

```
<activity
    android:name="utn.fra.prueba1.MainActivity"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:label="@string/app_name" >
```

La desventaja es que nuestro layout no se recargará para ajustarse a el nuevo tamaño de pantalla.

Pausando Threads

Supongamos que tenemos un Thread que realiza una tarea:

```
public class MyThread extends Thread{

	private boolean flagPausa=false;

	@Override
	public void run() {

		for (int i=0 ; i<50 ; i++)
		{
			Log.d("1","Ejecucion desde Thread,contador:"+i);

			// Sleep de 1 segundo
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			// ____________________

		}
	}
}
```

En el ejemplo, vemos que cada 1 segundo, este Thread imprime un mensaje con el valor de un contador que se va incrementando. Cuando el contador llega a 50, termina el método "run" y Thread termina su ejecución.

En la Activity, lanzamos el Thread de la siguiente manera:

```
		myThread = new MyThread();
		myThread.start();
```

Supongamos que queremos colocar 2 botones, uno para pausar la tarea del Thread y otro para reanudarla, escribimos en la Activity, los métodos "onClick" para cada uno de los botones:

```
	Button btnPausa = (Button)findViewById(R.id.btnPausa);
	Button btnContinuar = (Button)findViewById(R.id.btnContinuar);


	btnPausa.setOnClickListener(new OnClickListener() {

		public void onClick(View v) {
			// pausa

		}
	});

	btnContinuar.setOnClickListener(new OnClickListener() {

		public void onClick(View v) {
			// continuar

		}
	});
```

¿Cómo hacemos para pausar el Thread?

Existen dos métodos para llevar a cabo la lógica:

- **wait :** Pausa un Thread por un tiempo indeterminado.
- **notify :** Saca a un Thread del estado "wait".

Utilizaremos el método "wait" desde dentro del Thread, para ingresarlo al modo "pausado" para ello, necesitaremos que el Thread chequee un flag que deberemos definir, y si ese flag es True, el Thread ejecutará el método "wait()" y la ejecución se pausará:

Ejemplo:

```
public class MyThread extends Thread{

	private boolean flagPausa=false;

	@Override
	public void run() {

		for (int i=0 ; i<50 ; i++)
		{

			Log.d("1","Ejecucion desde Thread,contador:"+i);

			// Sleep de 1 segundo
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			//____________________

			if(flagPausa)
			{
				flagPausa=false;
				try {
					synchronized (this) {
						Log.d("1","Entro en pausa");
						wait();
					}
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
			}
		}
	}

	public void pausar()
	{
		flagPausa=true;
	}
}
```

Definimos el flag de pausa

Tarea

Preguntamos por el flag

Ejecutamos wait

Método que se llama desde afuera para cambiar el estado del flag y que el Thread entre en pausa.

En la Activity, dentro del método "onClick" del botón de pausa, vamos a llamar al método "pausar()" de nuestro Thread.

```
btnPausa.setOnClickListener(new OnClickListener() {

        public void onClick(View v) {
            // pausa
            synchronized (myThread) {
                myThread.pausar();
            }
        }
    });
```

Para reanudar la ejecución del Thread, deberemos llamar al método "notify" de nuestro objeto Thread, esto hará que el Thread salga del estado "wait" y continúe con la ejecución del hilo, en la línea a continuación de donde se ejecutó el "wait":

```
btnContinuar.setOnClickListener(new OnClickListener() {

        public void onClick(View v) {
            // continuar
            synchronized (myThread) {
                myThread.notify();
            }
        }
    });
```

Bloque synchronized

Cuando ejecutamos el método "wait" dentro del Thread, y cuando ejecutamos "notify", encerramos estas líneas de programa en un bloque llamado "synchronized" y entre paréntesis aclaramos el objeto que queríamos "sincronizar". Si no hubiéramos hecho esto, al ejecutarse los métodos hubieran lanzado una Excepción.

El bloque synchronized nos asegura, que ningún otro Thread esta ejecutando la porción de código que estamos sincronizando, que involucre al objeto con el que estamos haciendo la sincronización.

Interrupciones

Cada vez que llamamos al método "sleep" debemos encerrar dicha llamada con un try-catch que captura la excepción "InterruptedException", esta excepción no tiene que ver con que "no se pudo ejecutar el sleep" el sleep siempre se podrá ejecutar.

```
// Sleep de 1 segundo
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
//____________________
```

Esta excepción fue creada para poder interrumpir el "sleep" y sacar al Thread de ese estado. Para ello, deberemos ejecutar el método "interrupt()" del objeto Thread en cuestión. Cuando hagamos esto, la ejecución del Thread volverá a activarse y se entrará al bloque "catch".

Del mismo modo que "sleep" el método "wait"
Generalmente, cuando se quiere finalizar un Thread y asegurarse de que éste termine, se utiliza una interrupción.

Ejemplo:

En nuestra activity, en el método "onDestroy" llamamos a el método "interrupt" de nuestro objeto Thread:

```
@Override
protected void onDestroy() {

  super.onDestroy();
  myThread.interrupt();// interrumpo el Thread para que termine
}
```

Y en nuestro Thread, deberemos considerar tres condiciones:

- 1 - Cuando se ejecuta "interrupt" el Thread se encontraba en el método "sleep".
- 2 – Cuando se ejecuta "interrupt" el Thread se encontraba en el método "wait".
- 3 – Cuando se ejecuta "interrupt" el Thread se encontraba en alguna línea dentro del bucle.

En el caso de que el Thread se encuentre en el método "sleep", se lanzará la interrupción "InterruptedException", de modo que dentro del catch, colocaremos un "return" para salir del método "run" y terminar el Thread.

```
try {
      Thread.sleep(1000);
} catch (InterruptedException e) {
      //e.printStackTrace();
      return; // se interrumpio el Thread, salgo del run
}
```

En el caso de que el Thread se encuentre en el método "wait", se lanzará la interrupción "InterruptedException", de modo que dentro del catch, colocaremos un "return" para salir del método "run" y terminar el Thread.

```
try {
       synchronized (this) {
             Log.d("1","Entro en pausa");
             wait();
       }
} catch (InterruptedException e) {
      //e.printStackTrace();
      return; // se interrumpio el Thread, salgo del run
}
```

Para el tercer caso, tenemos un método que nos devuelve True, si el Thread fue interrumpido, de modo que ponemos un "if" dentro del bucle preguntando por ello:

```
if(isInterrupted())
{
      return; // se interrumpio el Thread, salgo del run
}
```