Android avanzado

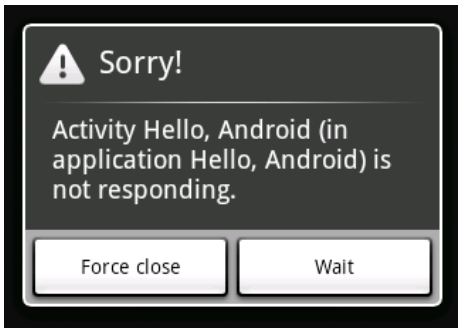
Índice

1. Hilos

1.1. 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.

1.1.1. 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){
                        //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){
                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;
                        }
                }
        }
}
```

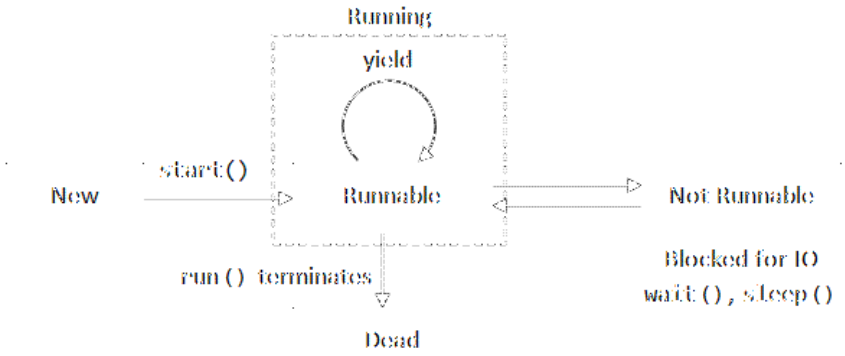
En el anterior ejemplo también se ha ejecutado el hilo desde su propio constructor, de manera que la creación del objeto `Hilo1` ha sido suficiente para ejecutarlo.

1.1.2. 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

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.

1.1.3. 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;
                        }
                }
        }
}
```

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.

1.1.4. 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.

1.2. Sincronización de hilos

1.2.1. 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){
        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.

1.2.2. 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
        // 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
```

1.2.2.1. 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;
        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()`.

1.2.3. 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`.

1.2.3.1. 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 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.

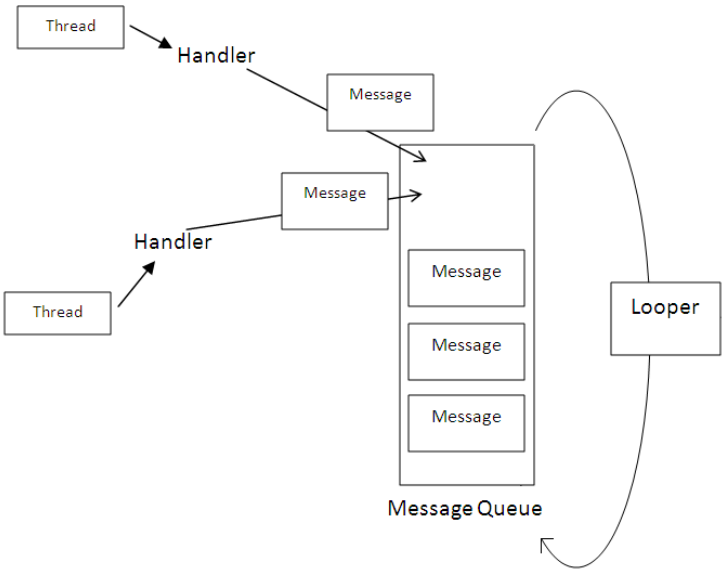
En el siguiente ejemplo se realiza una carga lenta de datos en otro hilo, al tiempo que se muestra una splash screen para indicar que se están cargando los datos. Una vez finalizada la tarea se envía un mensaje vacío. Este mensaje se captura con un Handler propio y se realiza el cambio de vistas de la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

Ejemplo: pantalla de inicialización

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

1.2.3.2. 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{
        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 `Runnables` 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:

```
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) {
                }
        }
}
```

1.2.4. 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"));
    }
}
```

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() );
```

1.3. 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.

```
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();
        }
}
```

1.3.1. 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(){
        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 `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.

2. Hilos - Ejercicios

2.1. 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.

2.2. (*) 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.

2.3. 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.

2.4. (*) 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.

2.5. 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.

2.6. 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.

3. Servicios

En Android un `Service` es un componente que se ejecuta en segundo plano, sin interactuar con el usuario. Cualquier desarrollador puede crear nuevos Service en su aplicación. Cuentan con soporte multitarea real en Android, ya que pueden ejecutar en su propio proceso, a diferencia de los hilos de las `Activity`, que hasta cierto punto están conectados con el ciclo de vida de las actividades de Android.

Para comunicarse con los servicios se utiliza el mecanismo de comunicación entre procesos de Android, Inter-Process Communication (IPC). La interfaz pública de esta comunicación se describe en el lenguaje AIDL.

3.1. Servicios propios

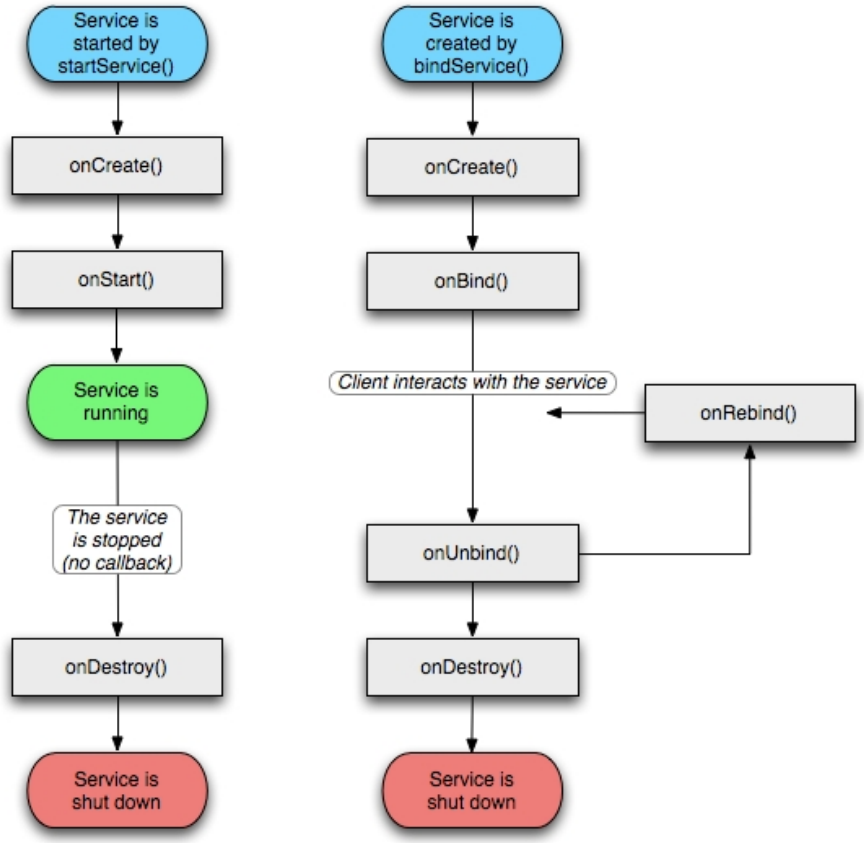
Podemos crear un servicio propio para realizar tareas largas que no requieran la interacción con el usuario, o bien para proveer de determinado tipo de funcionalidad a otras aplicaciones. Los servicios propios se deben declarar en el `AndroidManifest.xml` de la aplicación. En el código java se debe crear una clase que herede de la clase `Service` de forma directa o a través de alguna clase hija.

```
<service
        android:name="MiServicio"
        android:process=":mi_proceso"
        android:icon="@drawable/icon"
        android:label="@string/service_name"
        >
</service>
```

Para pedir permiso a otras aplicaciones para que utilicen el servicio se utiliza el atributo `uses-permission`.

Se puede especificar que un servicio debe ejecutarse en un proceso aparte a través del atributo `android:process`. De esta manera cuenta con su propio hilo y memoria y cualquier operación costosa que pueda tener no afectará al hilo principal donde estaría la interfaz gráfica. Si no lo declaramos con el atributo `android:process`, debemos utilizar hilos o `AsyncTask` para las tareas con procesamiento intensivo.

Una actividad puede iniciar un servicio a través del método `startService()` y detenerlo a través de `stopService()`. Si la actividad necesita interactuar con el servicio se utiliza el método `bindService()` del servicio. Esto requiere un objeto `ServiceConnection` que permite conectar con el servicio y que devuelve un objeto `IBinder`. Este objeto puede ser utilizado por la actividad para comunicarse con el servicio.



El ciclo de vida de un servicio de Android.

Una vez iniciado el servicio, se invoca su método `Service.onCreate()`. A continuación se invoca el método `Service.onStartCommand(Intent, int, int)` con la información del `Intent` proporcionado por la actividad. Para los servicios que se lanzan desde la aplicación principal, el método `onStartCommand()` se ejecuta en ese mismo hilo, el de la interfaz gráfica. Es bastante común crear un hilo nuevo y lanzarlo desde el `onStartCommand()` para realizar el procesamiento en segundo plano. Si se lanza ese hilo, la ejecución se devuelve rápidamente al método `onStartCommand()` dejándolo terminar en muy poco tiempo. A través del valor retorno de `onStartCommand()` podemos controlar el comportamiento de reinicio.

- `Service.START_STICKY` es el comportamiento estándar, en este caso el método `onStartCommand()` será invocado cada vez que el servicio sea reiniciado tras ser terminado por la máquina virtual. Nótese que en este caso al reiniciarlo el Intent que se le pasa por parámetro será `null`. `Service.START_STICKY` se utiliza típicamente en servicios que controlan sus propios estados y que se inician y terminan de manera explícita con `startService` y `stopService`. Por ejemplo, servicios que reproduzcan música u otras tareas de fondo.
- `Service.START_NOT_STICKY` se usa para servicios que se inician para procesar

acciones específicas o comandos. Normalmente utilizarán `stopSelf()` para terminarse una vez completada la tarea a realizar. Cuando la máquina virtual termine este tipo de servicios, éstos serán reiniciados sólo si hay llamadas de `starService` pendientes, de lo contrario el servicio terminará sin pasar por `onStartCommand`. Este modo es adecuado para servicios que manejen peticiones específicas, tales como actualizaciones de red o polling de red.

- `Service.START_REDELIVER_INTENT` es una combinación de los dos anteriores de manera que si el servicio es terminado por la máquina virtual, se reiniciará sólo si hay llamadas pendientes a `starService` o bien el proceso fue matado antes de hacer la llamada a `stopSelf()`. En este último caso se llamará a `onStartCommand()` pasándole el valor inicial del `Intent` cuyo procesamiento no fue completado. Con `Service.START_REDELIVER_INTENT` nos aseguramos de que el comando cuya ejecución se ha solicitado al servicio, sea completada hasta el final.

(En versiones anteriores a la 2.0 del Android SDK (nivel 5 de API) había que implementar el método `onStart` y era equivalente a sobrecargar `onStartCommand` y devolver `START_STICKY`).

El segundo parámetro del método `onStarCommand()` es un entero que contiene flags. Éstos se utilizan para saber cómo ha sido iniciado el servicio:

- `Service.START_FLAG_REDELIVERY` indica que el `Intent` pasado por parámetro es un reenvío porque la máquina virtual ha matado el servicio antes de ocurrir la llamada a `stopSelf`.
- `Service.START_FLAG_RETRY` indica que el servicio ha sido reiniciado tras una terminación anormal. Sólo ocurre si el servicio había sido puesto en el modo `Service.START_STICKY`.

Por ejemplo comprobamos si el servicio había sido reiniciado:

```
public class MiServicio extends Service {
        @Override
        public void onCreate() {

        }

        @Override
        public void onDestroy() {

        }

        @Override
        public int onStartCommand(Intent intent,
                                        int flags, int startId) {
                if((flags & START_FLAG_RETRY) == 0){
                        // El servicio se ha reiniciado
                } else {
                        // Iniciar el proceso de fondo
                }
                return Service.START_STICKY;
        }

        @Override
```

```
        public IBinder onBind(Intent arg0) {
                return null;
        }
}
```

El método `onBind()` debe ser sobrecargado obligatoriamente y se utiliza para la comunicación entre procesos.

3.1.1. Iniciar un servicio

Para iniciar un servicio de forma explícita se utiliza el método `startService()` con un `Intent`. Para detenerlo se utiliza `stopService()`:

```
        ComponentName servicio = startService(
                                new Intent(getApplicationContext(),
MiServicio.class));
        // ...
        stopService(new Intent(getApplicationContext(),
                                                servicio.getClass()));
```

3.1.1.1. Iniciar al arrancar

Hay aplicaciones que necesitan registrar un servicio para que se inicie al arrancar el sistema operativo. Para ello hay que registrar un `BroadcastReceiver` al evento del sistema `android.intent.action.BOOT_COMPLETED`. En el AndroidManifest tendríamos:

```
<uses-permission
 android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <receiver android:name=".OnBootReceiver">
      <intent-filter>
        <action
         android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

La actividad sobrecargaría el método `onReceive()`:

```
        public class MiReceiver extends BroadcastReceiver {
                @Override
                public void onReceive(Context context, Intent intent) {
                        Intent servicio = new Intent(context,
MiServicio.class);
                        context.startService(servicio);
                }
        }
```

Si la aplicación está instalada en la tarjeta SD, entonces no estará disponible en cuanto arranque el sistema. En este caso hay que registrarse para el evento `android.intent.action.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE`. A partir de

Android 3.0 el usuario debe haber ejecutado la aplicación al menos una vez antes de que ésta pueda recibir el evento `BOOT_COMPLETED`.

3.1.1.2. Servicios prioritarios

Es posible arrancar servicios con la misma prioridad que una actividad que esté en el foreground para evitar que Android pueda matarlos por necesidad de recursos. Esto es peligroso porque si hay muchos servicios de foreground se degrada el rendimiento del sistema. Por esta razón al iniciar un servicio en el foreground se debe notificar al usuario.

Los servicios de foreground se inician realizando desde dentro del servicio una llamada al método `startForeground()`:

```
int NOTIFICATION_ID = 1;
Intent        intent        =
        new Intent(this, MiActividad.class);
PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 1, intent, 0));
Notification  notification  =
        new Notification(R.drawable.icon,
                "Servicio prioritario iniciado",
                System.currentTimeMillis());
notification.setLatestEventInfo(this,
        "Servicio", "Servicio iniciado", pendingIntent);
notification.flags = notification.flags |
        Notification.FLAG_ONGOING_EVENT; //Mientras dure

startForeground(NOTIFICATION_ID, notification);
```

Esta notificación debe durar mientras el servicio esté en ejecución. Se puede volver del foregroudn con el método `stopForeground()`. En general los servicios no deben ser iniciados en el foreground.

3.1.2. Servicios y AsyncTask

Las operaciones lentas de un servicio deben realizarse en un hilo aparte. La manera más cómoda suele ser a través de una `AsyncTask`. En el siguiente ejemplo un servicio utiliza una `AsyncTask` para realizar una cuenta desde 1 hasta 100.

```
public class MiCuentaServicio extends Service {
        MiTarea miTarea;

        @Override
        public int onStartCommand(Intent intent,
                        int flags, int startId) {
                Log.i("SRV", "onStartCommand");
                miTarea.execute();
                return Service.START_STICKY;
        }

        @Override
        public void onCreate() {
                super.onCreate();
                Toast.makeText(this,"Servicio creado ...",
```

```
                        Toast.LENGTH_LONG).show();
        Log.i("SRV","onCreate");
        miTarea = new MiTarea();
}


@Override
public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this,"Servicio destruido ...",
                        Toast.LENGTH_LONG).show();
        Log.i("SRV","Servicio destruido");
        miTarea.cancel(true);
}

@Override
public IBinder onBind(Intent arg0) {

        return null;
}


private class MiTarea
                extends AsyncTask<String, String, String>{
        private int i;
        boolean cancelado;

        @Override
        protected void onPreExecute() {
                super.onPreExecute();
                i = 1;
                cancelado = false;
        }

        @Override
        protected String doInBackground(String... params) {
                for(; i<100; i++){
                        Log.i("SRV",
                                        "AsyncTask: "+"Cuento hasta "+i);
                        publishProgress(""+i);
                        try {
                                Thread.sleep(5000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        if(cancelado)
                                break;
                }
                return null;
        }

        @Override
        protected void onProgressUpdate(String... values) {
                Toast.makeText(getApplicationContext(),
                                "Cuento hasta "+values[0],
                                Toast.LENGTH_SHORT).show();
        }

        @Override
        protected void onCancelled() {
                super.onCancelled();
                cancelado = true;
        }

}
```

```
}
```

Para completar el ejemplo, el código de una actividad con dos botones que inicien y detenga este servicio tendría el siguiente aspecto:

```
public class Main extends Activity {
        Main main;

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

        ((Button)findViewById(R.id.Button01)).setOnClickListener(
        new OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                startService(new Intent(main,
                                        MiCuentaServicio.class));
                        }
                });
        ((Button)findViewById(R.id.Button02)).setOnClickListener(
        new OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                stopService(new Intent(main,
                                        MiCuentaServicio.class));
                        }
                });
    }
}
```

3.2. Broadcast receiver

3.2.1. Declaración y registro en el Manifest

Un receptor de broadcast es una clase que recibe `Intents` generados a través del método `Context.sendBroadcast()`. La clase debe heredar de `BroadcastReceiver` y debe implementar el método `onReceive()`. Sólo durante la ejecución de este método el objeto estará activo, por tanto no se puede utilizar para hacer ninguna operación asíncrona. Concretamente, no se podría mostrar un diálogo ni realizar un `bind` a un servicio. Las alternativas serían usar el NotificationManager en el primer caso y `Context.startService()` en el segudo.

La clase que herede de `BroadcastReceiver` estar declarada en el `AndroidManifest.xml`. También se declaran los `Intent` que la clase recibirá:

```
<application>
        <!-- [...]  -->

        <receiver
                android:name=".paquete.MiBroadcastReceiver"
                android:enabled="false">
```

```
           <intent-filter>
              <action
android:name="android.intent.ACTION_TIMEZONE_CHANGED" />
              <action android:name="android.intent.ACTION_TIME" />
           </intent-filter>
        </receiver>
</application>
```

La clase declarada en el ejemplo siguiente recibiría un intent al cambiar la hora (debido a un ajuste del reloj) o al cambiar la zona horaria:

```
public class MiBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context,
                                          Intent intent) {
        String action = intent.getAction();
        if (action.equals(Intent.ACTION_TIMEZONE_CHANGED)
                || action.equals(Intent.ACTION_TIME_CHANGED)) {

            //Ejemplo: Actualizar nuestro
            // Widget dependiente de la hora

        }
    }
}
```

Otro ejemplo típico es el de recibir el intent de una llamada de teléfono entrante. Habría que declarar el intent `android.intent.action.PHONE_STATE` en el `AndroidManifest.xml` y declarar el `BroadcastReceiver` con el siguiente código:

```
public class LlamadaReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle extras = intent.getExtras();
                if (extras != null) {
                        String state = extras.getString(
                                TelephonyManager.EXTRA_STATE);
                        if (state.equals(
TelephonyManager.EXTRA_STATE_RINGING)) {
                                String phoneNumber = extras.getString(
TelephonyManager.EXTRA_INCOMING_NUMBER);
                                Log.i("DEBUG", phoneNumber);
                        }
                }
        }
}
```

Hay intents para toda clase de eventos, como por ejemplo pulsar el botón de la cámara, batería baja, o incluso cuando se instala una nueva aplicación. Los componentes propios también pueden enviar `Broadcast`.

3.2.2. Registro dinámico

Un `BroadcastReceiver` se puede registrar a un `IntentFilter` dinámicamente en lugar de hacerlo a través del `AndroidManifest.xml`.

```
MiBroadcastReceiver intentReceiver = new MiBroadcastReceiver();

IntentFilter intentFilter =
        new IntentFilter(Intent.ACTION_CAMERA_BUTTON);
intentFilter.addAction(Intent.ACTION_PACKAGE_ADDED);

registerReceiver(intentReceiver, intentFilter);

unregisterReceiver(intentReceiver);
```

Es recomendable registrar el receiver en el método `onResume()` y desregistrarlo en el método `onPause()` de la actividad.

3.3. PendingIntents y servicios del sistema

Al pasar un `PendingIntent` a otra aplicación, por ejemplo, el Notification Manager, el Alarm Manager o aplicaciones de terceros, se le está dando permiso para ejecutar determinado código que nosotros definimos en nuestra propia aplicación.

El código a ejecutar lo colocaremos en un `BroadcastReceiver`:

```
public class MiBroadcastReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                Toast.makeText(context,
                        "Otra aplicación es la causa de esta tostada.",
                                Toast.LENGTH_LONG).show();
                // Código que queremos ejecutar
                // a petición de la otra aplicación
                // ...
        }

}
```

Sin olvidar declararlo en el `AndroidManifest.xml`, antes de cerrar la etiqueta de `</application>`:

```
<receiver android:name="MiBroadcastReceiver">
```

En este ejemplo se utilizará un `PendingIntent` para que el Alarm Manager pueda ejecutar el código de nuestro `BroadcastReceiver`. Este código pertenecería a algún método de la `Activity`:

```
Intent intent = new Intent(this, MiBroadcastReceiver.class);
PendingIntent pendingIntent = PendingIntent.getBroadcast(
                this.getApplicationContext(), 0, intent, 0);
AlarmManager alarmManager = (AlarmManager)
                                        getSystemService(ALARM_SERVICE);
alarmManager.set(AlarmManager.RTC_WAKEUP,
        System.currentTimeMillis()+5000, pendingIntent);
```

Android ofrece una serie de servicios predefinidos a los que se accede a través del método `getSystemService(String)`. La cadena debe ser una de las siguientes constantes:

`WINDOW_SERVICE`, `LAYOUT_INFLATER_SERVICE`, `ACTIVITY_SERVICE`, `POWER_SERVICE`, `ALARM_SERVICE`, `NOTIFICATION_SERVICE`, `KEYGUARD_SERVICE`, `LOCATION_SERVICE`, `SEARCH_SERVICE`, `VIBRATOR_SERVICE`, `CONNECTIVITY_SERVICE`, `WIFI_SERVICE`, `INPUT_METHOD_SERVICE`, `UI_MODE_SERVICE`, `DOWNLOAD_SERVICE`.

En general los servicios obtenidos a través de esta API pueden estar muy relacionados con el contexto (`Context`) en el cuál fueron obtenidos, por tanto no conviene compartirlos con contextos diferentes (actividades, servicios, aplicaciones, proveedores).

3.3.1. AlarmManager para programar servicios

`AlarmManager` no sólo se puede utilizar para programar otros servicios sino que en la mayoría de los casos **se debe** utilizar. Un ejemplo sería el de programar un servicio que comprueba si hay correo electrónico o RSS. Se trata de una tarea periódica y el servicio no tiene por qué estar en ejecución todo el tiempo. De hecho un servicio que sólo se inicia con `startService()` y nunca se finaliza con `stopService()` se considera un "antipatrón de diseño" en Android.

Si un servicio cumple ese antipatrón, es posible que Android lo mate en algún momento. Si un servicio de verdad requiere estar todo el tiempo en ejecución, como por ejemplo, uno de voz IP (require estar conectado y a la escucha todo el tiempo), entonces habría que iniciarlo como servicio foreground para que Android no lo mate nunca.

El `AlarmManager` es la analogía del cron de Unix. La diferencia importante es que cron siempre continúa con su anterior estado mientras que `AlarmManager` empieza en blanco cada vez que arranca. Por tanto estamos obligados a volver a registrar nuestros servicios durante el arranque.

3.4. Comunicación entre procesos

La comunicación entre procesos se puede realizar tanto invocando métodos como pasando objetos con información. Para pasar información de una actividad a un servicio se pueden utilizar los `BroadcastReceiver`, pasando información extra, `Bundle`, en el `Intent` que se utiliza para iniciar un servicio, o bien haciendo un binding al servicio.

Vamos a empezar por el binding de una actividad a un servicio:

3.4.1. Atar actividades a servicios

Un tipo posible de comunicación es el de invocar los métodos de un servicio.

Atar o enlazar (to bind) una actividad a un servicio consiste en mantener una referencia a la instancia del servicio, permitiendo a la actividad realizar llamadas a métodos del servicio igual que se harían a cualquier otra clase accesible desde la actividad.

Para que un serivicio de soporte a binding hay que implementar su método `onBind()`.

Éste devolverá una clase binder que debe implementar la interfaz `IBinder`. La implementación nos obliga a definir el método `getService()` del binder. En este método devolveremos la instancia al servicio.

```
public class MiServicio extends Service {

        private final IBinder binder = new MiBinder();

        @Override
        public IBinder onBind(Intent intent){
                return binder;
        }

        public class MiBinder extends Binder {
                MiServicio getService() {
                        return MiServicio.this;
                }
        }

        // ...
}
```

La conexión entre el servicio y la actividad es representada por un objeto de clase `ServiceConnection`. Hay que implementar una nueva clase hija, sobrecargando el método `onServiceConnected()` y `onServiceDisconnected()` para poder obtener la referencia al servicio una vez que se ha establecido la conexión:

```
private MiServicio servicio; //La referencia al servicio

private ServiceConnection serviceConnection =
                                new ServiceConnection() {
        public void onServiceConnected(
                ComponentName className, IBinder service) {
                servicio = ((MiServicio.MiBinder)service).getService();
        }
        public void onServiceConnected(ComponentName className) {
                servicio = null;
        }
};
```

Finalmente, para realizar el binding hay que pasarle el intent correspondiente al método `Activity.bindService()`. El intent servirá para poder seleccionar qué servicio devolver. Tendríamos el siguiente código en `Activity.onCreate()`:

```
Intent intent = new Intent(MiActividad.this,
                                                MiServicio.this);
bindService(intent, serviceConnection,
                                                Conext.BIND_AUTO_CREATE);
```

Una vez establecido el binding, todos los métodos y propiedades públicos del servicio estarán disponibles a través del objeto `servicio` del ejemplo.

3.4.2. Inter Process Communication

En Android cada aplicación se ejecuta en su propia "caja de arena" y no comparte la

memoria con otras aplicaciones o procesos. Para comunicarse entre procesos Android implementa el mecanismo IPC, Inter Process Communication. Su protocolo requiere codificar y descodificar los distintos tipos de datos y para facilitar esta parte, Android ofrece la posibilidad de definir los tipos de datos con AIDL, Android Interface Definition Language. De esta manera para comunicarse entre dos aplicaciones o procesos hay que definir el AIDL, a partir de éste implementar un Stub para la comunicación con un servicio remoto y por último dar al cliente acceso al servicio remoto.

La interfaz de datos se define en un archivo .aidl. Por ejemplo el siguiente archivo es `/src/es/ua/jtech/IServicio.aidl`:

```
package es.ua.jtech;

interface IServicio {
        // Los valores pueden ser: in, out, inout.
        String saludo(in String nombre, in String apellidos);
}
```

Una vez creado el anterior archivo, la herramienta Eclipse+AIDL generará un archivo .java, para el ejemplo sería el `/src/es/ua/jtech/IServicio.java`

En nuestro servicio implementaremos el método `onBind()` para que devuelva un stub que cumple la interfaz anteriormente definida.

```
public class Servicio extends Service{
        @Override
        public IBinder onBind(Intent intent) {
                return new IServicio.Stub() {
                        public int saludo(String nombre, String apellidos)
throws RemoteException {
                                return "Hola, "+ nombre + " " + apellidos;
                        }
                };
        }
        // ...
}
```

Ahora hay que obtener acceso al servicio desde la aplicación a través de `ServiceConnection`. Dentro de nuestra `Activity` tendríamos:

```
public class MiActividad extends Activity {
        IServicio servicio;
        MiServicioConnection connection;

        class MiServicioConnection implements ServiceConnection {

                public void onServiceConnected(ComponentName name,
                                                IBinder service) {
                        servicio = IServicio.Stub.asInterface(
                                                (IBinder) service);
                }

                public void onServiceDisconnected(
                                        ComponentName name) {
                        servicio = null;
```

```
                }

        }

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

                connection = new MiServicioConnection();

                // El intent se crea así porque
                // la clase podría estar en otra
                // aplicación:
                Intent intent = new Intent();
                intent.setClassName("es.ua.jtech",
                        es.ua.jtech.Servicio.class.getName());

                bindService(intent, connection,
                        Context.BIND_AUTO_CREATE);

                String saludo = servicio.saludo(
                                                "Boyan", "Bonev");
                // ...
        }

        @Override
        protected void onDestroy() {
                unbindService(connection);
        }

}
```

Los tipos que se permiten en AIDL son:

- Valores primitivos como int, float, double, boolean, etc.
- `String` y `CharSequence`
- `java.util.List` y `java.util.Map`
- Intefaces definidos en AIDL, requiere definir el import
- Clases Java que implementen la interfaz `Parcelable` que Android usa para permitir la serialización. También requiere definir el import

Si el servicio está en otro proyecto necesitaremos declararlo con un intent filter en su Manifest

```
<service android:export="true" name="es.ua.jtech.Servicio">
  <intent-filter>
    <action android:name="es.ua.jtech.IServicio"/>
  </intent-filter>
</service>
```

y crear el intent con ese nombre de acción:

```
Intent intent = new Intent("es.ua.jtech.IServicio");
```

3.4.3. Otras formas de comunicación

3.4.3.1. Broadcast privados

Enviar un `Intent` a través del método `sendBroadcast()` es sencillo pero por defecto cualquier aplicación podría tener un `BroadcastReceiver` que obtenga la información de nuestro intent. Para evitarlo se puede utilizar el método `Intent.setPackage()` que restringirá el broadcast a determinado paquete.

3.4.3.2. PendingResult

Otra manera de que un servicio envíe información a una actividad es a través del método `createPendingResult()`. Se trata de un método que permite a un `PendingIntent` disparar el método `Activity.onActivityResult`. Es decir, el servicio remoto llamaría a `send()` con un `PendingIntent` con la información, de manera análoga al `setResult()` que ejecuta una actividad que fue llamada con `startActivityForResult()`. Dentro de `Activity.onActivityResult` se procesaría la información del `Intent`.

Este es un mecanismo que sólo funciona con actividades.

4. Servicios - Ejercicios

4.1. Servicio con proceso en background. Contador

Los servicios se utilizan para ejecutar algún tipo de procesamiento en background. En este ejercicio vamos a crear nuestro propio proceso asociado a un servicio, que ejecute determinada tarea, en este caso, contar desde 1 hasta 100, deteniéndose 5 segundos antes de cada incremento. En cada incremento mostraremos un `Toast` que nos informe de la cuenta.

En las plantillas tenemos el proyecto `android-av-serviciocontador` que ya incluye la declaración del servicio en el manifest, la actividad que inicia y detiene el servicio, y el esqueleto del servicio `MiCuentaServicio`.

- En el esqueleto que se proporciona, viene definida una extensión de `AsyncTask` llamada `MiTarea`. Los métodos `onPreExecute`, `doInBackground`, `onProgressUpdate` y `onCancelled` están sobrecargados pero están vacíos. Se pide implementarlos, el primero de ellos inicializando el campo `i` que se utiliza para la cuenta, el segundo ejecutando un bucle desde 1 hasta 100, y en cada iteración pidiendo mostrar el progreso y durmiento después 5 segundos con `Thread.sleep(5000)`. El tercer método, `onProgressUpdate` mostrará el `Toast` con el progreso, y por último el método de cancelación pondrá el valor máximo de la cuenta para que se salga del bucle.
- En los métodos del servicio, `onCreate`, `onStartCommand` y `onDestroy`, introduciremos la creacion de la nueva `MiTarea`, su ejecución (método `execute()` de la tarea) y la cancelación de su ejecución (método `cancel()` de la tarea).

Una vez más, el servicio deberá seguir funcionando aunque se salga de la aplicación y podrá ser parado entrando de nuevo en la aplicación y pulsando Stop.

4.2. Dialer. Iniciar una actividad con un evento broadcast (*)

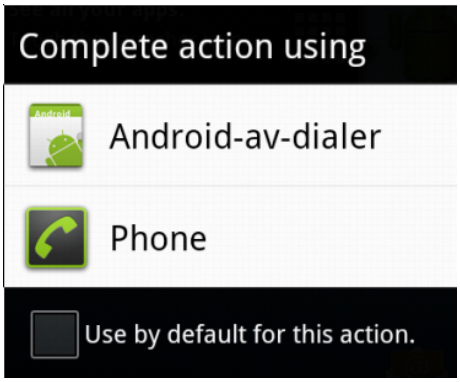
Se pide crear un proyecto `android-av-dialer` con el paquete `es.ua.jtech.av.dialer` y una activdad `Main` que muestre el texto "Dialer personalizado".

Esta actividad se iniciará cuando se inicie un marcado de llamada telefónica. Para ello debes añadir otro intent filter al `AndroidManifest.xml` para que además de iniciarse ante el evento de lanzamiento de la aplicación, se inicie ante un evento `DIAL`:

```
<intent-filter>
    <action android:name="android.intent.action.DIAL" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Al ejecutar la aplicación desde Eclipse no ocurrirá nada más que su instalación en el

dispositivo. Para ejecutarla se debe iniciar un marcado telefónico. Es posible que no funcione en el emulador con Android 4.0. Además, a partir de Android 3.1 es necesario ejecutar la actividad de la aplicación al menos una vez para que el BroadcastReceiver quede registrado.



Diálogo que ofrece realizar la llamada con nuestro dialer

Para implementar un programa marcador de verdad, necesitarás algunos de los permisos siguientes:

```
<uses-permission android:name="android.permission.CALL_PRIVILEGED" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"
/>
<uses-permission android:name="android.permission.CALL_PHONE" />
```

Se pide añadir dos botones para marcar dos números de teléfono favoritos (preferiblemente inventados). El marcado se puede iniciar de la siguiente manera:

```
Intent dialIntent = new
Intent(Intent.ACTION_DIAL,Uri.parse("tel:+3400000000"));
startActivity(dialIntent);
```

4.3. Arranque. Iniciar servicio con evento broadcast

En el proyecto `android-av-onboot` tenemos un servicio que se lanza al inicar la actividad, mostrando Toasts del 1 al 5 y finalizando.

Se pide registrar el servicio para que se inicie cuando el móvil haya terminado de arrancar. Para ello añadiremos el un permiso al `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"
/>
```

También registraremos un receptor de broadcast `MiBroadcastReceiver` con el intent filter para `android.intent.action.BOOT_COMPLETED`.

Implementaremos el `MiBroadcastReceiver` y en su método `onReceive()` introduciremos el código:

```
if( "android.intent.action.BOOT_COMPLETED".equals(intent.getAction())) {
        ComponentName comp = new ComponentName(context.getPackageName(),
                                        MiServicio.class.getName());
        ComponentName service = context.startService(
                                        new Intent().setComponent(comp));
        if (null == service){
                Log.e(MiBroadcastReceiver.class.toString(),
                  "Servicio no iniciado: " + comp.toString());
        }
} else {
        Log.e(MiBroadcastReceiver.class.toString(),
          "Intent no esperado " + intent.toString());
}
```

Tras ejecutar el proyecto desde Eclipse habrá que reiniciar el emulador para observar que el servicio se inicia al finalizar el arranque. Es necesario reiniciarlo sin cargar desde un estado anterior guardado para que el sistema Android arranque desde cero.

4.4. Localizador de móvil desaparecido

No sería fácil detectar por software que nuestro móvil ha desaparecido, pero podemos programar un servicio que responda a determinado evento que nosotros causemos a través de la red móvil. Lo más sencillo, para el caso de que no haya red, sería que al recibir un mensaje SMS con un contenido determinado, el teléfono respondiera con un SMS (a ese mismo número o a otro que especifiquemos) indicando su localización.

En las plantillas de la sesión contamos con el BroadcastReceiver `Localizador` que podemos añadir a un nuevo proyecto que llamaremos `android-av-localizador` con una actividad principal `Main` en el paquete `es.ua.jtech.av.localizador`.

Registraremos el receptor de broadcast `Localizador` a través del `AndroidManifest.xml`. Dejaremos también la actividad `Main` para que pueda ser lanzada desde el menú. Es necesaria a partir de Android 3.1 para que quede efectuado el registro. Podríamos poner algún texto explicativo, "Se ha registrado el servicio de localizador".

También harán falta los permisos RECEIVE_SMS, SEND_SMS, ACCESS_COURSE_LOCATION, ACCESS_FINE_LOCATION.

Para probar el programa se pueden abrir dos emuladores y enviar SMS al número de puerto que viene indicado en la barra de cada emulador. Para establecer una localización simulada podemos utilizar Eclipse con la vista DDMS/Emulator control, o bien

```
telnet localhost <puerto>
geo fix <longitude> <latitude> [<altitude>]
```

4.5. AIDL (*)

Crea un proyecto llamado `android-av-aidl-calculadora`. con un servicio `ServicioCalculadora` que ofrezca los métodos `float suma(float, float)` y `float eleva(float, int)`.

La actividad principal `CalcActivity` debe mostrar interfaz gráfica muy sencilla que nos permita llamar al servicio del otro proyecto. Es importante hacer el `unbindService()` al cerrarse la actividad.

5. Notificaciones y AppWidgets

Los servicios en segundo plano de Android desempeñan funciones que requieren poca interacción con el usuario. Aún así lo normal es que haya que informar de sucesos o de progreso. Un mecanismo de interfaz gráfica específico de Android son las notificaciones. Otro son los AppWidgets, o widgets de la pantalla de inicio, que a menudo realizan la actualización de sus vistas desde servicios en segundo plano.

5.1. Notificaciones

El típico mecanismo de comunicación con el usuario que los Servicios utilizan son las `Notification`. Se trata de un mecanismo mínimamente intrusivo que no roba el foco a la aplicación actual y que permanece en una lista de notificaciones en la parte superior de la pantalla, que el usuario puede desplegar cuando le convenga.

Desplegando la barra de notificaciones

Para mostrar y ocultar notificaciones hay que obtener de los servicios del sistema el `NotificationManager`. Su método `notify(int, Notification)` muestra la notificación asociada a determinado identificador.

```
Notification notification;
NotificationManager notificationManager;
notificationManager = (NotificationManager)getSystemService(
        Context.NOTIFICATION_SERVICE);
notification = new Notification(R.drawable.icon,
        "Mensaje evento", System.currentTimeMillis());
notificationManager.notify(1, notification);
```

El identificador sirve para actualizar la notificación en un futuro (con un nuevo aviso de notificación al usuario). Si se necesita añadir una notificación más, manteniendo la anterior, hay que indicar un nuevo ID.

Para actualizar la información de un objeto `Notification` ya creado, se utiliza el método

```
notification.setLatestEventInfo(getApplicationContext(),
                "Texto", contentIntent);
```

donde `contentIntent` es un `Intent` para abrir la actividad a la cuál se desea acceder al pulsar la notificación. Es típico usar las notificaciones para abrir la actividad que nos permita reconfigurar o parar el servicio. También es típico que al pulsar sobre la notificación y abrirse una actividad, la notificación desaparezca. Este cierre de la notificación lo podemos implementar en el método `onResume()` de la actividad:

```
@Override
protected void onResume() {
        super.onResume();
        notificationManager.cancel(MiTarea.NOTIF_ID);
}
```

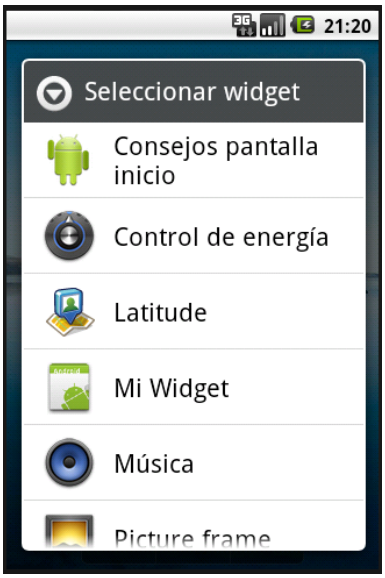
A continuación se muestra un ejemplo completo de notificaciones usadas por una tarea `AsyncTask`, que sería fácilmente integrable con un `Service`. (Sólo haría falta crear una nueva `MiTarea` en `Service.onCreate()`, arrancarla con `miTarea.execute()` desde `Service.onStartCommand(...)` y detenerla con `miTarea.cancel()` desde `Service.onDestroy()`).

```
private class MiTarea
        extends AsyncTask<String, String, String>{
    public static final int NOTIF1_ID = 1;
    Notification notification;
    NotificationManager notificationManager;

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
        notificationManager =
                (NotificationManager)getSystemService(
                                Context.NOTIFICATION_SERVICE);
        notification = new Notification(R.drawable.icon,
                "Mensaje evento", System.currentTimeMillis());
    }

    @Override
    protected String doInBackground(String... params) {
        while(condicionSeguirEjecutando){
                if(condicionEvento){
                        publishProgress("Información del evento");
                }
        }
        return null;
    }

    @Override
```

```
        protected void onProgressUpdate(String... values) {
                Intent notificationIntent = new Intent(
                        getApplicationContext(),
MiActividadPrincipal.class);
                PendingIntent contentIntent = PendingIntent.getActivity(
                        getApplicationContext(), 0, notificationIntent,
0);
                notification.setLatestEventInfo(getApplicationContext(),
                        values[0], contentIntent);
                notificationManager.notify(NOTIF_ID, notification);
        }
}
```

5.2. AppWidgets

Los widgets, que desde el punto de vista del programador son `AppWidgets`, son pequeños interfaces de programas Android que permanecen en el escritorio del dispositivo móvil. Para añadir alguno sobra con hacer una pulsación larga sobre un área vacía del escritorio y seleccionar la opción "widget", para que aparezca la lista de todos los que hay instalados y listos para añadir. En la versión 4 de Android para añadir widgets hay que ir al menú de aplicaciones y desde la sección de widgets seleccionar uno y pulsarlo prolongadamente para arrastrarlo a la zona deseada de la pantalla de inicio.



Seleccionar un (App) Widget

Este es un ejemplo de widget, el que muestra un reloj, incluido de serie con Android:

AppWidget reloj de Android

5.2.1. Crear un Widget

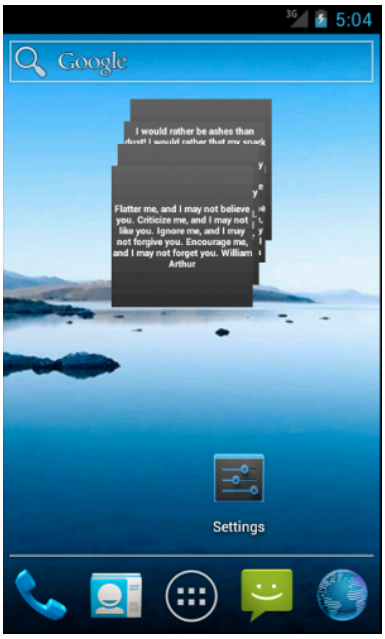
Para crear un widget se necesitan tres componentes: un recurso layout en xml que defina la interfaz gráfica del widget, una definición en XML de los metadatos del widget y por último un `IntentReceiver` que sirva para controlar el widget.

Los AppWidgets ocupan determinado tamaño y se refrescan con determinada frecuencia, metadatos que hay que declarar en el XML que define el widget. Se puede añadir como nuevo recurso XML de Android, y seleccionar el tipo Widget. Se coloca en la carpeta `res/xml/`. Por ejemplo, este es el `res/xml/miwidget.xml`:

```
<?xml version="1.0" encoding="utf-8"? >
<appwidget-provider
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:updatePeriodMillis="4320000"
    android:initialLayout="@layout/miwidget_layout"
    />
```

Este XML declara que el Layout del widget se encuentra en `res/layout/miwidget_layout.xml`. Por cuestiones de diseño de Android, relacionadas con seguridad y eficiencia, los únicos componentes gráficos permitidos en el layout del widget son los layouts `FrameLayout`, `LinearLayout`, `RelativeLayout`, los views `AnalogClock`, `Button`, `Chronometer`, `ImageButton`, `ImageView`, `ProgressBar` y `TextView`. Es interesante el hecho de que los `EditText` **no** están permitidos en el layout del widget. Para conseguir este efecto lo que se suele hacer es utilizar una imagen que imite este view y al hacer click, abrir una actividad semi transparente por encima que sea la que nos permita introducir el texto.

Desde la versión 3.0 de Android los widget cuentan con nuevas características y mayor interactividad. Los nuevos componentes que se pueden utilizar son `GridView`, `ListView`, `StackView`, `ViewFlipper`, `AdapterViewFlipper`. Por ejemplo `StackView` no estaba presente entre los componentes de las versiones anteriores y es un view que permite mostrar "tarjetas" e ir pasándolas hacia delante o hacia atrás.



Widget con StackView

Un ejemplo de layout podría ser el siguiente, en el que se coloca un reloj analógico y un campo de texto al lado:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:id="@+id/miwidget"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content">
    <AnalogClock
        android:id="@+id/analogClock1"
        android:layout_width="45dp"
        android:layout_height="45dp" />

        <TextView android:text=""
                android:id="@+id/TextView01"
                android:textSize="9dp"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:shadowColor="#000000"
                android:shadowRadius="1.2"
                android:shadowDx="1"
                android:shadowDy="1">
        </TextView>
</LinearLayout>
```

La pantalla de inicio de Android, por defecto está dividida en una matriz de 4x4 celdas, cada una de ellas con un mínimo de 74x74 píxeles independientes del dispositivo (dp). Para decidir la altura y la anchura del widget se pueden calcular el número de celdas que se desean ocupar, restándole a esa cantidad dos píxeles (dp) de borde. En los dispositivos donde las dimensiones mínimas que establezcamos no coincidan exactamente con las celdas de la pantalla, el tamaño de nuestro widget será extendido para llenar lo que queda

libre de las celdas.

Los AppWidgets no necesitan ninguna actividad, sino que se implementan como `IntentReceivers` con `IntentFilters` que detecten intents con acciones de actualización de widet como `AppWidget.ACTION_APPWIDGET_UPDATE,DELETED,ENABLED,DISABLED`, así como con acciones personalizadas. Se podría definir una clase que herede de `IntentReceiver` pero también se puede definir una clase que herede de `AppWidgetProvider` que es una alternativa proporcionada por Android para encapsular el procesamiento de los Intent y proveer de handlers para la actualización, borrado, habilitación y deshabilitación del widget.

Por tanto en el `AndroidManifest.xml` ya no necesitamos declarar una actividad principal. Lo que tenemos que declarar es el receptor de intents del widget:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      android:versionCode="1"
      android:versionName="1.0"
      package="es.ua.jtech.av.appwidget">
    <uses-sdk android:minSdkVersion="7" />
    <application android:icon="@drawable/icon"
                            android:label="@string/app_name">

        <receiver android:name=".MiWidget"
                        android:label="Frases Widget">
            <intent-filter>
                <action
android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <intent-filter>
                <action android:name="es.ua.jtech.av.ACTION_WIDGET_CLICK"
/>
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
                    android:resource="@xml/miwidget" />
        </receiver>
    </application>

</manifest>
```

En el anterior ejemplo de manifest se declara, por un lado, el intent filter que capturará el evento de actualización del widget y, por otro lado el intent que captura una acción personalizada, la `es.ua.jtech.av.ACTION_WIDGET_CLICK`.

5.2.2. RemoteViews

La clase `RemoteViews` se utiliza para definir y manipular una jerarquía de views que se encuentra en el proceso de una aplicación diferente. Permite cambiar propiedades y ejecutar métodos. En el caso de los widgets los views están alojados en un proceso (normalmente perteneciente a la pantalla del Home) y hay que actualizarlos desde el `IntentReceiver` que definimos para controlar el widget. Una vez definido el cambio a realizar por medio de `RemoteViews`, el cambio se aplica por medio del `AppWidgetManager` que podemos obtener a partir del contexto, como se muestra en la

última línea del siguiente código.

```
RemoteViews updateViews = new RemoteViews(
context.getPackageName(),
R.layout.miwidget_layout);

//Actualizar un campo de texto del widget
updateViews.setTextViewText(R.id.TextView01, "texto");

//Comportamiento al pulsar el widget
//On-click listener que envía un broadcast intent
Intent intent = new Intent(ACTION_WIDGET_CLICK);
PendingIntent pendingIntent = PendingIntent.getBroadcast(
                                                    context, 0,
intent, 0);
updateViews.setOnClickPendingIntent(
                                            R.id.miwidget,
pendingIntent);

ComponentName thisWidget = new ComponentName(
                                                    context,
MiWidget.class);
AppWidgetManager.getInstance(context).updateAppWidget(
                                                    thisWidget,
updateViews);
```

La actualización se realiza por medio de los métodos de la clase `RemoteViews`, tales como `.setTextViewText(String)`, `.setImageViewBitmap(Bitmap)`, etc, con el propósito de actualizar los valores del layout indicado en la creación del `RemoteViews`, `R.layout.miwidget_layout` en este caso.

`RemoteViews` proporciona métodos para acceder a propiedades como `setInt(...)`, `setBoolean(...)`, `setBitmap(...)`, etc, y métodos para acceder a Views específicos, como `setTextViewText(...)`, `setTextColor(...)`, `setImageViewBitmap(...)`, `setChronometer(...)`, `setProgressBar(...)`, `setViewVisibility(...)`.

Al pulsar el widget podríamos lanzar un intent con un navegador o con una actividad que debamos ejecutar:

```
//Comportamiento al pulsar el widget: lanzar alguna actividad:
Intent defineIntent = new Intent(....);
PendingIntent pendingIntent = PendingIntent.getActivity(
        getApplicationContext(), 0, defineIntent, 0);
updateViews.setOnClickPendingIntent(R.id.miwidget,pendingIntent);
```

5.2.3. Recibir los intent de actualización

Para actualizaciones periódicas se puede utilizar el `AlarmManager` que podría ser programado para forzar la actualización del widget, enviando periódicamente un broadcast con la acción que hayamos definido para tal fin.

En el actual ejemplo la acción personalizada sería `"es.ua.jtech.av.ACTION_WIDGET_CLICK"`, ya que en el manifest tenemos

```
    <intent-filter>
        <action android:name="es.ua.jtech.av.ACTION_WIDGET_CLICK" />
    </intent-filter>
```

Como su nombre indica, no es una acción para actualizaciones periódicas sino para detectar el click sobre un widget. Por tanto al hacer click sobre el widget, en lugar de abrir un navegador o una actividad, lo que haremos será crear un broadcast intent con la acción definida. Esta definición formaría parte de la actualización por medio de `RemoteViews`:

```
    Intent intent = new Intent("es.ua.jtech.av.ACTION_WIDGET_CLICK");
    PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0,
intent, 0);
    updateViews.setOnClickPendingIntent(R.id.miwidget, pendingIntent);
```

Para capturar este intent implementaremos no sólo el método `onUpdate()` del widget sino también el método `onReceive()`, ya que es un `IntentReceiver`.

```
public class MiWidget extends AppWidgetProvider {
        @Override
        public void onUpdate(
                        Context context,
                        AppWidgetManager appWidgetManager,
                        int[] appWidgetIds) {
                actualizar(context);
        }
        @Override
        public void onReceive(Context context, Intent intent) {
                super.onReceive(context, intent);
                if(intent.getAction().equals(
                                "es.ua.jtech.av.ACTION_WIDGET_CLICK")){
                        actualizar(context);
                }
        }
}
```

En el método `onReceive` comprobamos si el intent recibido es el que se corresponde con la acción que hemos definido y realizamos las acciones que haga falta, por ejemplo, actualizar el widget.

5.2.4. Servicio de actualización

Otra alternativa para refrescar la interfaz gráfica es el uso de un servicio que actualice el widget. Para ello habría que declararlo en el manifest y declarar la clase del servicio `UpdateService extends Service` dentro de la clase `MiWidget`.

A continuación se muestra un ejemplo de clase `MiWidget`que implementa un widget:

```
public class MiWidget extends AppWidgetProvider {

        @Override
        public void onUpdate(Context context, AppWidgetManager
                        appWidgetManager, int[] appWidgetIds) {
                // Inicio de nuestro servicio de actualización:
```

```
                context.startService(
                        new Intent(context, UpdateService.class));
        }

        public static class UpdateService extends Service {
                @Override
                public int onStartCommand(Intent intent,
                                                        int flags, int
startId) {
                        RemoteViews updateViews = new RemoteViews(
                                getPackageName(),
R.layout.miwidget_layout);

                        //Aqui se actualizarían todos los tipos de Views:
                        updateViews.setTextViewText(R.id.TextView01,
                                        "Valor con el que refrescamos");
                        // ...

                        //Y la actualización a través del updateViews
creado:
                        ComponentName thisWidget = new ComponentName(
                                        this, MiWidget.class);
AppWidgetManager.getInstance(this).updateAppWidget(
                                        thisWidget, updateViews);

                        return Service.START_STICKY;
                }

                @Override
                public IBinder onBind(Intent intent) {
                        return null;
                }
        }
}
```

5.2.5. Actividad de configuración

Algunos widgets muestran una actividad de configuración la primera vez que son añadidos a la pantalla. Cuando el sistema solicita la cofiguración de un widget se recibe un broadcast intent con la acción `android.appwidget.action.APPWIDGET_CONFIGURE`, por tanto definiremos en el manifest nuestra actividad con el `intent-filter` correspondiente, que la ejecutará al recibir la acción:

```
<activity android:name=".MiConfigurationActivity">
        <intent-filter>
                <action
android:name="android.appwidget.action.APPWIDGET_CONFIGURE"/>
        </intent-filter>
</activity>
```

La actividad debe devolver un Intent que incluya un extra con el identificador del App Widget, usando la constante `EXTRA_APPWIDGET_ID`. Ésta es incluída en el Intent que lanza la actividad.

```
        Intent resultado = new Intent();
        result.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
        setResult(RESULT_OK, resultado);
```

```
        finish();
```

La actividad de configuración también debe ser declarada en el xml que contiene los metadatos del widget:

```
<appwidget-provider
        ...
    android:updatePeriodMillis="3600000"
    android:configure="es.ua.jtech.av.appwidget.MiConfigurationActivity"
/>
```

Esta actividad será mostrada sólo al añadir el widget a la pantalla por primera vez.

6. Notificaciones y AppWidgets - Ejercicios

6.1. Servicio con notificaciones: Números primos

El proyecto `android-av-notificaciones` de la plantilla tiene un servicio con una tarea que va calculando números primos a un ritmo lento. Vamos a mostrar una `Notification` en la barra de tareas cada vez que se descubra un primo y la iremos actualizando con la llegada de cada nuevo número. Si salimos de la aplicación sin parar el servicio, seguirán apareciendo notificaciones, y si pulsamos sobre la notificación, volverá a lanzarse la actividad, cerrándose la notificación que hemos pulsado.

- Dentro del servicio `MiNumerosPrimosServicio` se encuentra declarada la `AsyncTask` llamada `MiTarea`. En ella tenemos como campos de la clase una `Notification` y un `NotificationManager`. Hay que darles valores en el método `onPreExecute()`.
- El método `doInBackground(...)` ejecutará un bucle que irá incrementando `i` mientras su valor sea menor de `MAXCOUNT`. En cada iteración, si el número es primo (función incluida en la plantilla), pedirá que se muestre el progreso, pasándole como parámetro el nuevo primo encontrado.
- Implementar el método `onProgressUpdate(...)` para que muestre la notificación. Para ello habrá que actualizar la notificación con el método `setLatestEventInfo`, al cuál le pasaremos en un String la información del último primo descubierto y le pasaremos un `PendingIntent` para que al pulsar sobre la notificación, nos devuelva a la actividad de la aplicación, por si la hemos cerrado. Para crear el PendingIntent utilizaremos el método `PendingIntent.getActivity(...)` al cuál le tenemos que pasar un `new Intent(getApplicationContext(),Main.class)`.
- La aplicación debería funcionar en este punto, mostrando las notificaciones y relanzando la aplicación si son pulsadas, pero no cerrándolas al pulsarlas. Para ello simplemente tenemos que llamar al método `cancel(id)` del `notificationManager` y pasarle la constante NOTIF_ID para que la notificación no se muestre como una nueva, sino como actualización de la que ya habíamos puesto. Una manera de hacerlo es en un método estático del `MiNumerosPrimosServicio`, que ya está creado en las plantillas del ejercicio y se llama `cerrarMiNotificacion(NotificationManager nm)`. Debes invocar este método desde el `Main.onResume()`.

Notificación del servicio de números primos

6.2. IP AppWidget

Vamos abrir el proyecto `android-av-appwidget` para construir un AppWidget de Android, que nos muestre una frase célebre y la hora.

En el proyecto pulsamos con el boton derecho y añadimos un nuevo Android XML File, de tipo AppWidget Provider, que se llame `miwidget.xml`. El editor nos permite pulsar sobre el AppWidget Provider y editar sus atributos. Ponemos los siguientes:

```
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:updatePeriodMillis="4320000"
    android:initialLayout="@layout/miwidget_layout"
```

El layout `miwidget_layout.xml` no lo tenemos que crear porque ya está incluido en el proyecto.

Creamos una clase `MiWidget` que herede de `AppWidgetProvider`, en el paquete `es.ua.jtech.av.appwidget`. Sobrecargamos su método `onUpdate(...)` y su método `onReceive(...)`. En este último comprobaremos si se ha recibido un intent con una acción personalizada, la `es.ua.jtech.av.ACTION_WIDGET_CLICK`:

```
public class MiWidget extends AppWidgetProvider {
        public static final String ACTION_WIDGET_CLICK =
"es.ua.jtech.av.ACTION_WIDGET_CLICK";

        @Override
        public void onUpdate(Context context, AppWidgetManager
appWidgetManager,
                        int[] appWidgetIds) {
```

```
                        //TODO actualizar
        }

        @Override
        public void onReceive(Context context, Intent intent) {
                super.onReceive(context, intent);
                if(intent.getAction().equals(ACTION_WIDGET_CLICK)){
                        //TODO actualizar
                }
        }
}
```

En ambos casos la actualización sería la misma, así que vamos a extraerla en un método privado de la clase, `private void actualizar(Context context)` que llamaremos desde `onUpdate()` y desde `onReceive()`. El método utilizará los `RemoteViews` para actualizar el campo de texto. Lo actualizará con una frase aleatoria de entre las definidas dentro de un array de strings en el recurso `strings.xml` que viene incluido en el proyecto de las plantillas. También se asignará un comportamiento al hacer click sobre el widget que consistirá en enviar un broadcast intent con la acción que hemos definido:

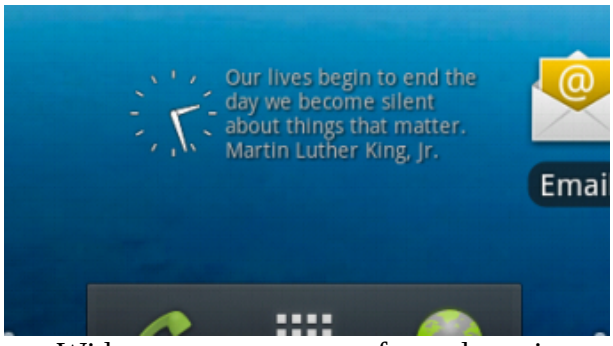
```
RemoteViews updateViews = new RemoteViews(context.getPackageName(),
R.layout.miwidget_layout);

//Seleccionar frase aleatoria
String frases[] = context.getResources().getStringArray(R.array.frases);
updateViews.setTextViewText(R.id.TextView01,
frases[(int)(Math.random()*frases.length)]);

//Comportamiento del botón
//On-click listener que envía un broadcast intent
Intent intent = new Intent(ACTION_WIDGET_CLICK);
PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0,
intent, 0);
updateViews.setOnClickPendingIntent(R.id.miwidget, pendingIntent);

ComponentName thisWidget = new ComponentName(context, MiWidget.class);
AppWidgetManager.getInstance(context).updateAppWidget(thisWidget,
updateViews);
```

En el `AndroidManifest.xml`, dentro de `<application>` declararemos el receiver de nuestro widget con dos intent filters, uno para la acción del sistema `android.appwidget.action.APPWIDGET_UPDATE`, y otro para la que utilizamos para forzar la actualización desde la clase `MiWidget`:

```
        <receiver android:name=".MiWidget" android:label="Frases Widget">
            <intent-filter>
                <action
android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <intent-filter>
                <action android:name="es.ua.jtech.av.ACTION_WIDGET_CLICK"
/>
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
                    android:resource="@xml/miwidget" />
        </receiver>
```

Ejecutamos el widget desde Eclipse, como aplicación android, y comprobamos que no ocurra ningún error en la consola de Eclipse. Ya se puede añadir el widget en el escritorio, efectuando una pulsación larga sobre una porción de área libre del escritorio, y seleccionando nuestro widget.

En Android 4.0 la pulsación larga no nos muestra la opción de añadir widget sino que hay que ir al menú de aplicaciones y seleccionar la pestaña de widgets. Para insertar uno se tiene que pulsar prolongadamente para arrastrarlo a una zona libre del escritorio. Añadimos el widget y observamos el resultado:



Widget que muestra una frase aleatoria

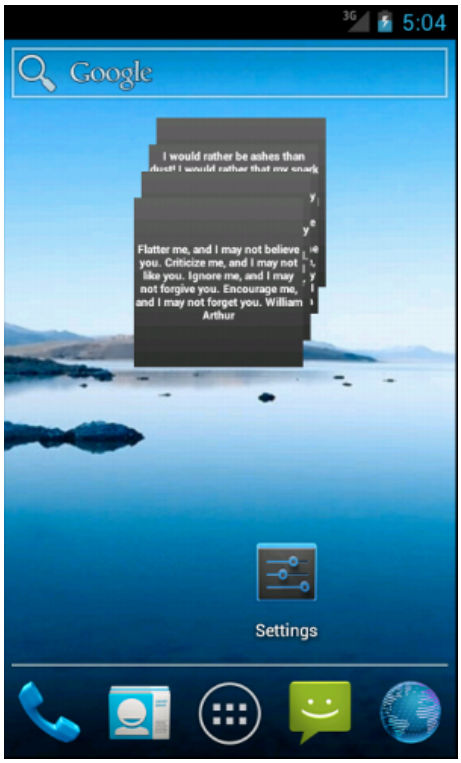
6.3. StackWidget (*)

Vamos a probar una de las características novedosas de los widgets a partir de Android 3.0, se trata del `StackView` que nos permite pasar "tarjetas" hacia delante y hacia atrás. Ejecuta el proyecto `android-av-stackwidget` de las plantillas y comprueba que funciona correctamente en un emulador con alguna de las últimas versiones de Android.

Vamos a modificar el widget para que muestre las frases del ejercicio anterior. Para ello tenemos que copiar y pegar el array de strings del recurso `strings.xml`. A continuación editamos la clase `StackWidgetProvider` y le añadimos un campo `private String [] frases`. Las podemos inicializar en en el constructor:

```
frases = context.getResources().getStringArray(R.array.frases);
```

En el método `onCreate` sustituiremos el bucle con las llamadas a `mWidgetItems.add(new WidgetItem(...))` por un bucle que añada todas las cadenas de `frases`. El resultado debe quedar así:



StackWidget con frases

7. Depuración y pruebas

7.1. Depuración con Eclipse

7.1.1. Log y LogCat

Sin duda alguna el sistema más utilizado para depuraciones sencillas es la salida estándar y los logs. En Android podemos imprimir mensajes al log tras importar `android.util.Log`. Los mensajes de Log van clasificados por una etiqueta que les asignamos, aparte de otra información implícita, como la aplicación que los lanza, el PID, el tiempo y el nivel de debug. Se recomienda crear una constante con la etiqueta:

```
private static final String TAG = "MiActivity";
```

Para añadir al log un mensaje de nivel de información utilizaríamos `Log.i()`:

```
Log.i(TAG, "indice=" + i);
```

Según el nivel de depuración utilizaremos una llamada de las siguientes:

- Log.v(): Verbose
- Log.d(): Debug
- Log.i(): Info
- Log.w(): Warning
- Log.e(): Error

Con esta información el Log podrá ser mostrado filtrando los mensajes menos importantes, por ejemplo si establecemos la vista del Log a nivel de Info, los de Debug y los Verbose no se mostrarían, pero sí los de Warning y Error que son más graves que Info.

En Eclipse contamos con la vista LogCat (si no se muestra por defecto se puede añadir) donde podemos realizar dicho filtrado. También podemos realizar filtrado por etiquetas para ver sólo los mensajes que nos interesan. Los mensajes van apareciendo en tiempo real, tanto si estamos con un emulador como si estamos con un dispositivo móvil conectado por USB.

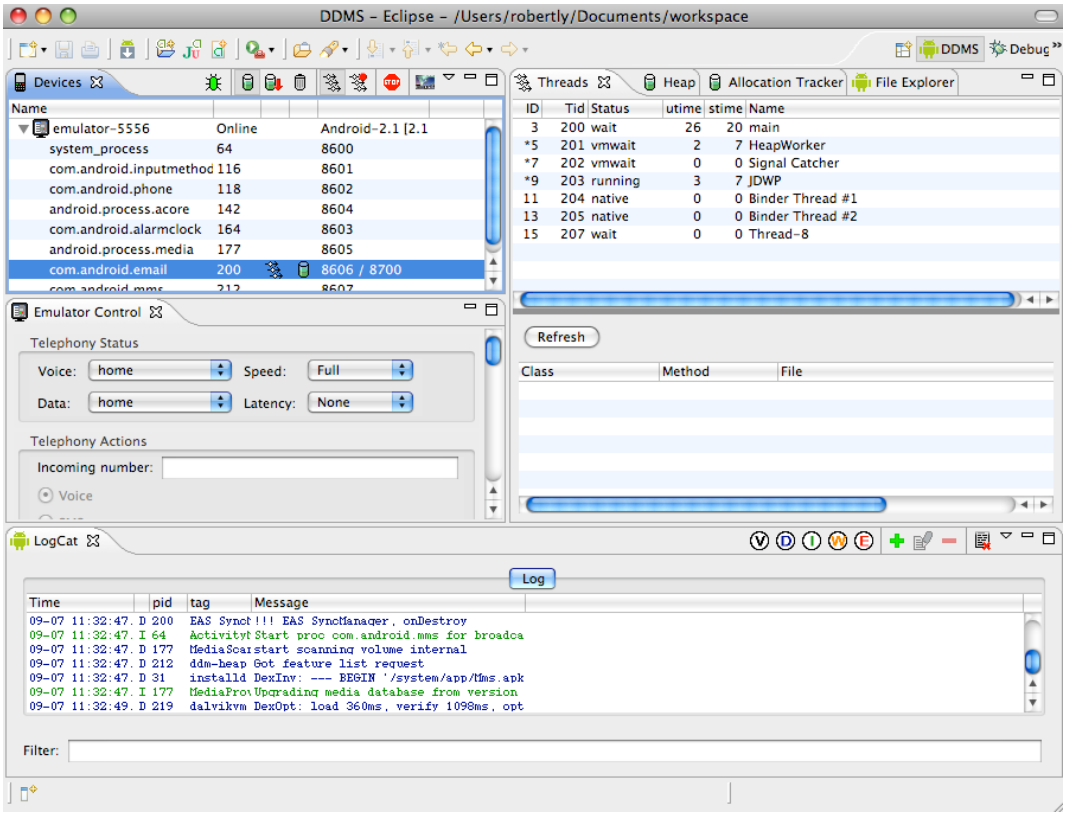
Se recomienda eliminar todas las llamadas a Log cuando se publica un programa en el Android Market, a pesar de que en los dispositivos de los usuarios no se vería ningún log ni salida estándar.

7.1.2. Dalvik Debug Monitor Server (DDMS)

Android viene con el servidor de depuración DDMS que puede ser ejecutado desde la

terminal o desde Eclipse. Desde la terminal del sistema se tendría que ejecutar `./ddms` que se encuentra en la carpeta `tools`. No es necesario si utilizamos Eclipse, que cuenta con una vista DDMS para interactuar y visualizar resultados de depuración.

En Android cada aplicación se ejecuta en su propia máquina virtual (VM) y cada VM tiene un puerto al que el debugger se conecta. Cuando se inicia DDMS, éste se conecta a `adb`. Cuando conectamos un dispositivo se crea un servicio de monitorización entre `adb` y DDMS, que notifica a DDMS cuando una VM del dispositivo arranca o termina. Cuando una VM arranca, DDMS recoge su PID a través de `adb` y abre una conexión con el debugger de la VM. Aunque cada depurador se puede conectar a un único puerto, DDMS maneja múltiples depuradores conectados.



Vista de DDMS en Eclipse

Como se puede observar en la figura, DDMS nos permite ver los hilos que están en ejecución en dada máquina virtual. También se permiten todas las características de depuración que proporciona Eclipse para Java, como puntos de ruptura, visualización de variables y evaluación de expresiones.

DDMS cuenta con las siguientes funcionalidades y controles:

- Visualización del uso de memoria heap
- Seguimiento de reservas de memoria para objetos

- Trabajar con el sistema de ficheros del emulador o del dispositivo
- Examinar la información de hilos
- Profiling de métodos: seguimiento de medidas tales como número de llamadas, tiempo de ejecución, etc.
- LogCat
- Emulación de operaciones de telefonía y localización
- Cambiar el estado de red y simular red lenta

7.2. Pruebas unitarias con JUnit para Android

Las pruebas unitarias consisten en probar métodos y partes concretas del código, intentando independizarlas del resto del código lo máximo posible. Los proyectos de Test de Android se construyen sobre JUnit y nos proveen de herramientas que permiten realizar no sólo pruebas unitarias sino también pruebas más amplias.

Cuando creamos un proyecto de Android con el asistente de Eclipse tenemos la opción de incluir casos de prueba en el propio proyecto. Otra manera muy común es la de separar las pruebas en un proyecto aparte. Para ello creamos un nuevo proyecto `Android Test Project` y seleccionamos el proyecto de nuestro workspace que queremos probar.

A continuación creamos casos de prueba en el mismo paquete donde se encuentra el proyecto original, o en subpaquetes de éste. Se crean con `New / JUnit Test Case` pero en lugar de que la clase padre sea la de JUnit, utilizaremos `ActivityInstrumentationTestCase2`, que es la versión actualizada de `ActivityInstrumentationTestCase` cuyo uso ya está desaconsejado. `ActivityInstrumentationTestCase2` se parametriza con la clase de la actividad que vamos a probar. Por ejemplo, si vamos a probar `MainActivity`, tendremos:

```
package es.ua.jtech.av.suma.test;

import android.test.ActivityInstrumentationTestCase2;
import es.ua.jtech.av.suma.MainActivity;

public class MainActivityTest extends
                ActivityInstrumentationTestCase2<MainActivity> {

        public MainActivityTest() {
                super("es.ua.jtech.av.suma", MainActivity.class);
        }

        protected void setUp() throws Exception {
                super.setUp();
        }

        public void test1(){
                // asserts
        }

        public void test2(){
                // asserts
        }
```

```
        // ...

        protected void tearDown() throws Exception {
                super.tearDown();
        }
}
```

En el código anterior se llama al constructor de la clase padre con el paquete `"es.ua.jtech.av.suma"` y la clase `MainActivity` de la actividad a probar.

Con esta clase ya se pueden escribir métodos con pruebas. Sus nombres deben comenzar por "test", por ejemplo `testMiPrueba1()`. Dentro de estos métodos de pruebas podemos utilizar los métodos `assert` de JUnit que provocan el fallo si la aserción resulta falsa. Por ejemplo, `assertEquals(a,b)` compara los dos parámetros y si son distintos la prueba falla. Otro ejemplo es `assertTrue(c)` que comprueba si es verdadero el parámetro que se le pasa.

Con JUnit se pueden probar métodos de las clases del código pero en Android podemos necesitar probar partes de la interfaz gráfica. Para acceder a los views de la actividad lo normal sería declarar referencias a éstos como campos de la clase e inicializarlos en el método `setUp()`. Para obtener las referencias hay que acceder a ellos con `getActivity().findViewById(id))`:

```
        private Button bt;

        protected void setUp() throws Exception {
                super.setUp();

                MainActivity activity = getActivity();
                bt =
(Button)activity.findViewById(es.ua.jtech.av.suma.R.id.button1);
        }
```

Una vez obtenidas las referencias se pueden comprobar los valores que contienen.

```
        assertEquals("32.3", miTextView.getText().toString());
```

Otra necesidad es la de simular eventos de introducción de texto, de selección, de pulsación. Para ello se utiliza la clase `TouchUtils`

```
                TouchUtils.tapView(this, miEditText);
                sendKeys("S");
                sendKeys("i");
                sendKeys("NUMPAD_DOT");
                TouchUtils.clickView(this, bt);
```

Aparte de `ActivityInstrumentationTestCase2` hay otras alternativas que también heredan de las clases de JUnit:

- `AndroidTestCase` que sólo ofrece el contexto local y no el de la aplicación.
- `ServiceTestCase` que se usa para probar servicios.
- `ActivityUnitTestCase` que crea la actividad pero no la conecta al entorno, de

manera que se puede utilizar un contexto o aplicación mock.
- `ApplicationTestCase` para probar subclases propias de `Application`.

Se pueden utilizar todas las características de JUnit, tales como crear una Test Suite para agrupar distintos casos de pruebas.

7.3. Pruebas de regresión con Robotium

"Robotium es como Selenium pero para Android" es el eslogan de este software de pruebas que no está afiliado ni con Android ni con Selenium. Selenium es un software para pruebas de aplicaciones web que permite grabar una secuencia de acciones sobre una página web y después reproducirla. Una batería de pruebas de este tipo permite comprobar que tras realizar cambios en el software, éste sigue comportándose de manera idéntica en su interacción con el usuario.

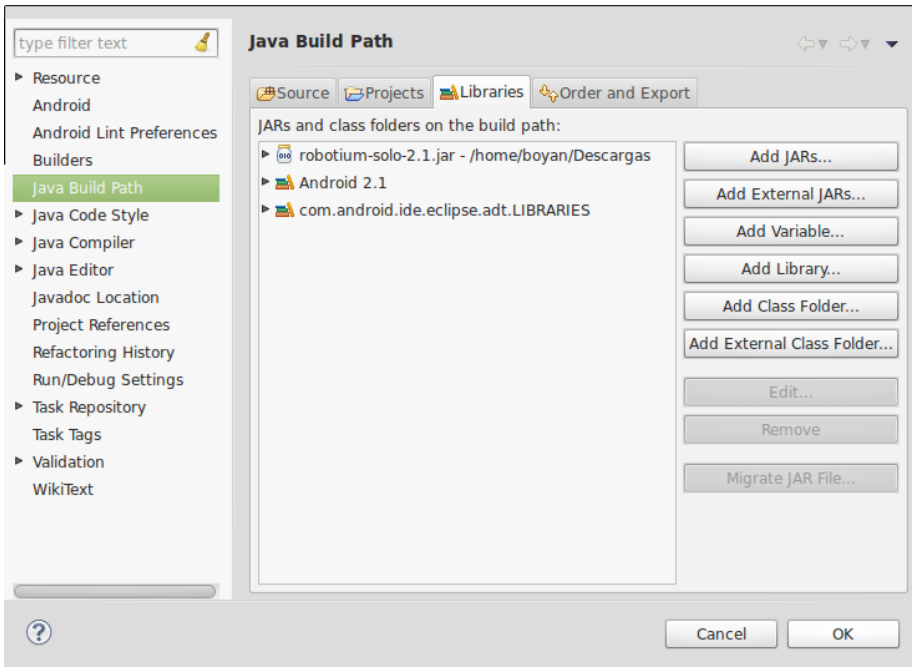
Robotium no permite grabar las acciones del usuario sino que la secuencia de acciones debe ser programada a través de sencillas llamadas a los métodos de Robotium. Da soporte completo para los componentes `Activity`, `Dialog`, `Toast`, `Menu` y `ContextMenu`.

Usar una herramienta como Robotium nos proporciona las siguientes ventajas:

- Desarrollar casos de prueba sin necesidad de conocer el funcionamiento interno de la aplicación probada.
- Automatizar el manejo múltiples actividades de Android.
- Pruebas realistas, al realizarse sobre los componentes GUI en tiempo de ejecución
- Integración con Maven y Ant para ejecutar pruebas como parte de una integración continua.

Para crear un conjunto de pruebas con Robotium debemos crear un nuevo proyecto de test de Android desde Eclipse: `New / Android Test Project`. Es importante crear los casos de prueba dentro del mismo paquete que el proyecto original, así que desde el asistente de creación del proyecto podemos introducir dicho paquete. También puede ser un subpaquete del original.

Añadiremos el JAR de Robotium en el build path del proyecto. Éste puede ser descargado de www.robotium.org y tendrá un nombre similar a `robotium-solo-3.1.jar` (según la versión). Lo añadiremos como jar tras incluírlo dentro del proyecto, en una carpeta `lib/`, por ejemplo.



Incluir Robotium en el Buildpath del proyecto de pruebas.

Una vez creado el proyecto de pruebas debemos crear un nuevo caso de prueba en el paquete correcto: `New / JUnit Test Case` en cuyo asistente pondremos el nombre del test que deseemos y cambiaremos la superclase del test por `ActivityInstrumentationTestCase2<Activity>` en lugar de `junit.framework.TestCase`. La Activity que pasamos como tipo puede ser directamente la clase de la actividad que vayamos a probar, por ejemplo `MainActivity`. Aunque pertenezca a otro proyecto Eclipse nos permite incluirla configurando automáticamente el path para referenciar al otro proyecto.

En el caso de prueba creado declaramos `Solo` como campo privado de la clase y, si no lo están ya, sobrecargamos los métodos `setUp()`, `tearDown` y el constructor.

```
package es.ua.jtech.av.miproyecto.test;

import android.test.ActivityInstrumentationTestCase2;
import es.ua.jtech.av.miproyecto.MainActivity;
import com.jayway.android.robotium.solo.Solo;

public class TestMainActivity extends
ActivityInstrumentationTestCase2<MainActivity> {
        private Solo solo;

        public TestMainActivity(){
                super("es.ua.jtech.av.miproyecto", MainActivity.class);
        }

        @Override
        protected void setUp() throws Exception {
                super.setUp();
```

```
                solo = new Solo(getInstrumentation(), getActivity());
        }

        public void test1(){
                //...
                //assertTrue(comprobacion);
        }

        //...

        @Override
        protected void tearDown() throws Exception {
                solo.finishOpenedActivities();
        }
}
```

Para ejecutar las pruebas programadas hay que hacer click sobre `TestMain` y en el menú contextual seleccionar `Run as / Android JUnit Test`. El emulador arrancará si no está arrancado, la aplicación a probar será instalada y ejecutada, y las acciones programadas en las pruebas serán realizadas sobre la actividad. Los resultados serán reconocidos por las pruebas y mostradas en el resumen de resultados de JUnit.

Un ejemplo de test podría ser el de una sencilla calculadora que cuenta con dos `EditText` y con un botón de sumar. El código de la prueba sería el siguiente:

```
        public void test1(){
                solo.enterText(0,"10");
                solo.enterText(1,"22.4");
                solo.clickOnButton("+");
                assertTrue(solo.searchText("32.4"));
        }
```

Los `EditText` de la actividad se corresponen (por orden) con los parámetros 0 y 1. El botón se corresponde con el que contiene la cadena indicada como etiqueta. Finalmente se busca el resultado en toda la actividad.

Exsiten diferentes métodos con los que `Solo` da soporte a los componentes gráficos. Algunos ejemplos son:

- `getView(id)`
- `getCurrentTextViews(textView)`
- `setActivityOrientation(Solo.LANDSCAPE)`
- `sendKey(Solo.MENU)`
- `clickOnButton(text)`
- `clickOnText(text)`
- `clickOnEditText(text)`
- `clearText(text)`
- `enterText(text)`
- `goBack()`
- `sleep(millis)`

La información sobre todos los métodos está en www.robotium.org.

7.4. Pruebas de estrés con Monkey

Monkey es un programa para pruebas que simula input aleatorio del usuario. Su propósito es realizar una prueba de estrés de la aplicación para confirmar que, haga lo que haga el usuario con la GUI, la aplicación no tendrá un comportamiento inesperado. El input que realiza Monkey no tiene por qué tener sentido alguno.

A continuación solicitamos 1000 eventos simulados cada 100 milisegundos obteniendo la lista de ellos (opción -v) y afectará a las aplicaciones del paquete `es.ua.jtech.av`.

```
adb shell monkey -p es.ua.jtech.av -v --throttle 100 1000
```

Monkey simulará eventos de teclado, tanto qwerty como teclas hardware especializadas, movimientos de trackball, apertura y cierre del teclado, rotaciones de la pantalla.

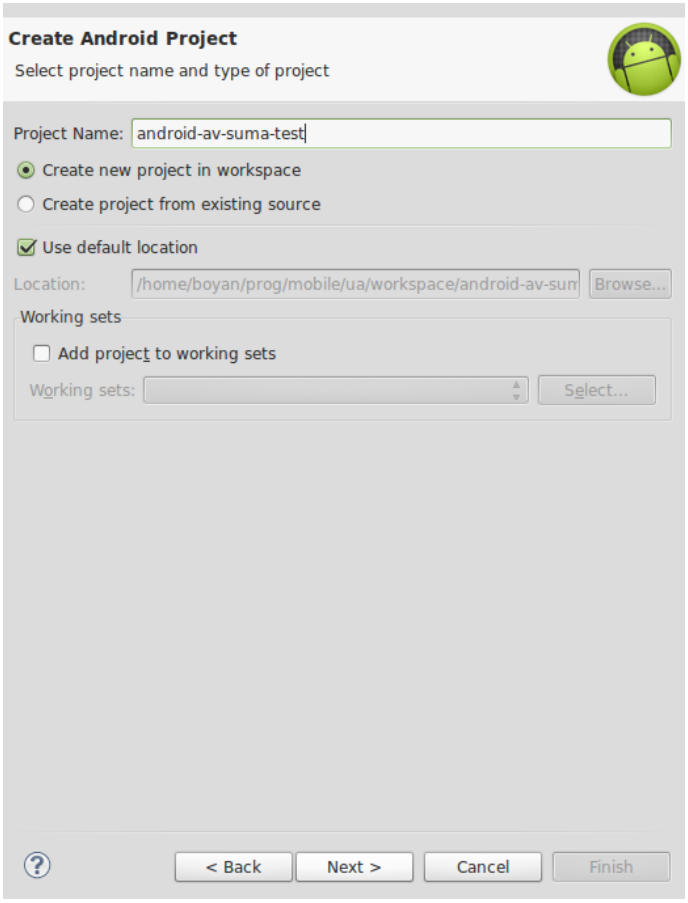
Si deseamos poder reproducir una serie de acciones aleatorias, podemos fijar la semilla de inicialización aleatoria con una propia, por medio de la opción `-s`. Así cada vez que ejecutemos con esta semilla, la secuencia de eventos aleatorios será la misma y podremos reproducir un problema tantas veces como haga falta hasta dar con la solución.

8. Depuración y pruebas - Ejercicios

8.1. Caso de prueba con JUnit para Android

En las plantillas contamos con el proyecto `android-av-suma` que consiste en una actividad con dos campos de texto para introducir dos números y a continuación pulsar el botón para visualizar el resultado en un `TextView`. A continuación se explica cómo crear un proyecto de pruebas JUnit.

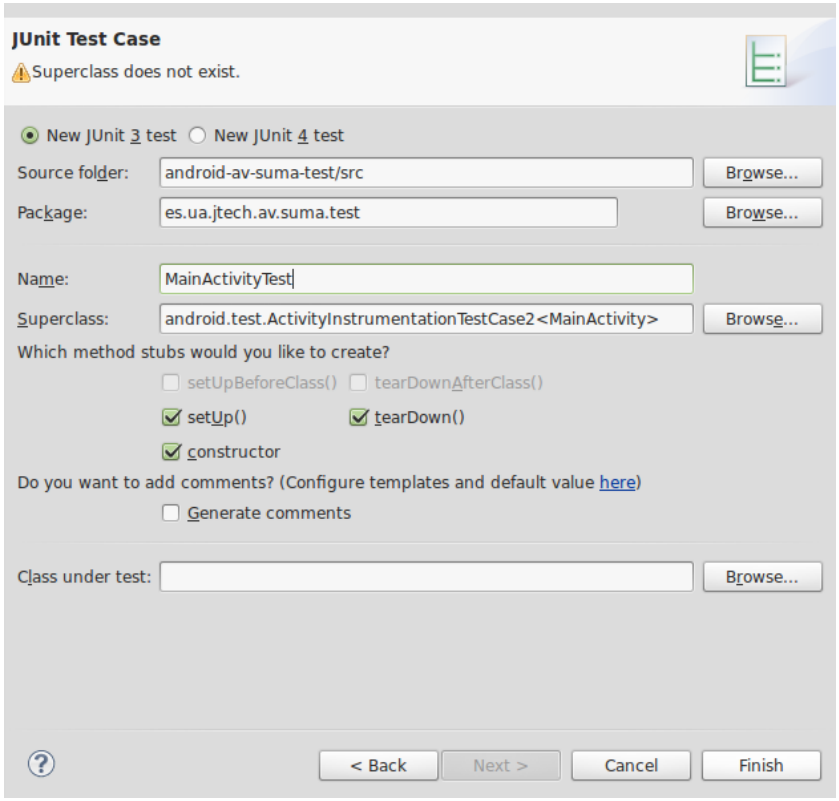
Crea un nuevo proyecto de tipo `Android Test Project` y deja marcada la opción de `Create new project in workspace`, llamándolo `android-av-suma-test`.



Crear nuevo proyecto de test de Android

Crea un nuevo caso de prueba con `New / JUnit Test Case` y cambia la clase de la que hereda a `ActivityInstrumentationTestCase2`. El nombre de la clase será `MainActivityTest` y el paquete el mismo que el del proyecto que estamos probando

pero terminado en `.test`. También podemos sustituir el tipo `<T>` por el de la clase de la actividad a probar, `<MainActivity>`:



Asistente para la creación de un caso de prueba

Dará error por la falta del import de `MainActivity`, pero se puede importar porque el otro proyecto está en el build path del proyecto de pruebas. También debemos cambiar el constructor para constuir el caso de prueba con el paquete y la actividad a probar:

```
	public MainActivityTest() {
		super("es.ua.jtech.av.suma", MainActivity.class);
	}
```

Vamos a añadir como campos de la clase objetos que referencien los views de nuestra actividad:

```
	private EditText et1, et2;
	private TextView tv;
	private Button   bt;
```

Los inicializaremos en el método `setUp()`, a partir de la actividad que se obtiene con `getActivity()`:

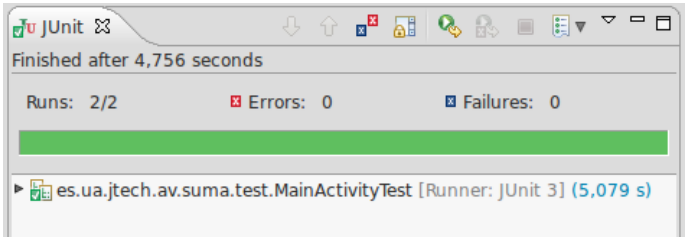
```
	protected void setUp() throws Exception {
```

```
                super.setUp();
                MainActivity activity = getActivity();
                et1 =
(EditText)activity.findViewById(es.ua.jtech.av.suma.R.id.editText1);
                et2 =
(EditText)activity.findViewById(es.ua.jtech.av.suma.R.id.editText2);
                tv  =
(TextView)activity.findViewById(es.ua.jtech.av.suma.R.id.textView3);
                bt  = (Button)
activity.findViewById(es.ua.jtech.av.suma.R.id.button1);
        }
```

Vamos a añadir dos métodos de test, uno de ellos va a comprobar que los componentes gráficos estén bien inicializados y el otro va a introducir unos datos en la interfaz y va a comprobar que el resultado es el correcto:

```
        @SmallTest
        public void testViewsCreados(){
                assertNotNull(et1);
                assertNotNull(et2);
                assertNotNull(tv);
                assertNotNull(bt);
                assertEquals("", et1.getText().toString());
                assertEquals("", et2.getText().toString());
                assertEquals(" ...", tv.getText().toString());
        }

        @SmallTest
        public void testSuma(){
                TouchUtils.tapView(this, et1);
                sendKeys("1");
                sendKeys("0");
                TouchUtils.tapView(this, et2);
                sendKeys("2");
                sendKeys("2");
                sendKeys("NUMPAD_DOT");
                sendKeys("3");
                TouchUtils.clickView(this, bt);
                assertEquals("32.3", tv.getText().toString());

        }
```

Para ejecutar la prueba hay que hacer click en `Run as/ Android JUnit Test`. El resultado debe salir en verde:



Resultado de JUnit en verde