



Servicios

Telefonica
EDUCACIÓN DIGITAL

Índice



Servicios

1 Introducción	3
2 Implementación de un servicio	4
2.1 La clase Service	4
2.2 Registro de un servicio	5
2.3 Lanzamiento y detención de un servicio	6
3 Acceso a la interfaz gráfica desde un servicio	11
4 Tareas repetitivas en un servicio	17

1. Introducción

A lo largo este módulo hemos estudiado las actividades, el componente más importante para la creación de aplicaciones Android, que constituye la base para la creación de aplicaciones gráficas. También hemos analizado los proveedores de contenido, que son componentes que ofrecen datos a nuestras aplicaciones, así como los BroadcastReceiver para responder a eventos de aplicación.

El último componente para la construcción de aplicaciones Android que vamos a estudiar es el servicio. Mediante los servicios podemos crear tareas en una aplicación que se ejecuten en segundo plano, normalmente, tareas repetitivas o de larga duración.

Por ejemplo, mientras el usuario interacciona con las actividades de la aplicación, un servicio puede transferir cada cierto tiempo datos almacenados en algún fichero o base de datos de la aplicación a un servidor remoto.

Es importante recalcar que los servicios **no son procesos independientes**, se ejecutan en segundo plano, pero como parte del proceso de la aplicación de la que depende, otra cosa es que desde el servicio queramos lanzar un hilo independiente que se ejecute en paralelo con el proceso.

Veremos a lo largo de este apartado como implementar, lanzar y detener servicios en aplicaciones Android.

2. Implementación de un servicio

Para crear un servicio en una aplicación debemos extender la clase `android.app.Service`. En esta clase implementaremos la funcionalidad del servicio. Además de heredar `Service`, el servicio tendrá que ser registrado como tal para que pueda ser lanzado desde otras aplicaciones.

2.1 | La clase service

Esta clase proporciona una serie de métodos del ciclo de vida del servicio que podemos sobrescribir, estos son los más interesantes:

- **onBind().** Es un método abstracto y por tanto será obligatorio sobrescribirlo. Este método es llamado cuando otro componente, por ejemplo una actividad, quiere vincularse al servicio, devolviendo como resultado un objeto `IBinder` para manejar la vinculación. Un componente se vincula al servicio cuando quiere acceder a atributos y métodos del mismo. Como es algo que pocas veces haremos, en este método simplemente incluiremos la instrucción `return null`;
- **onStartCommand().** Este método es llamado cuando se lanza el servicio desde

otra parte de la aplicación. Por tanto, en este método será donde **programemos las operaciones que deba realizar el servicio.**

Entre los parámetros que recibe está el objeto `Intent` con los datos enviados desde el componente que lo lanza.

El método `onStartCommand()` devuelve un entero que indica a Android el estado del servicio. Entre los posibles valores de devolución están las siguientes constantes de `Service`:

- **START_STICKY.** Después de ejecutarse el método, el servicio continúa ejecutándose hasta que se detenga explícitamente.

- **START_NOT_STICKY**. Tras la ejecución de `onStartCommand()` del servicio no continua ejecutándose, por lo que tendrá que volver a iniciarse explícitamente

- **onDestroy()**. Este método es llamado cuando se recibe la orden de detener el servicio. Seguidamente estudiaremos como iniciar y detener un servicio desde una aplicación

2.2 | Registro de un servicio

Como indicamos anteriormente, la clase del servicio debe ser registrada para que se pueda utilizar. El registro de un servicio se realiza en el archivo de manifiesto a través del elemento `<service>`, en cuyo atributo `name` se indicará el nombre de la clase del servicio:

```
<service  
    android:name=".ClaseServicio"
```

Al igual que sucede con las actividades y `BroadcastReceiver`, es posible asociar una acción a un servicio para que pueda ser lanzado desde aplicaciones externas. La forma de hacerlo es a través de un `<intent-filter>`:

```
<service  
    android:name=".ClaseServicio">  
  
    <intent-filter>  
        <action android:name="com.ejercicios.servicios.miservicio"/>  
    </intent-filter>  
  
</service>
```

2.3 | Registro de un servicio

Lanzar un servicio es un proceso similar al lanzamiento de una actividad, se debe crear un objeto Intent asociado al servicio, añadir en el mismo los extras que queramos enviarle y finalmente llamar al método `startService()` de Context:

```
Intent intent=new Intent(this,ClaseServicio.class);  
  
intent.putExtra("codigo",2300);  
  
this.startService(intent);
```

La llamada a `startService()` provocará la ejecución de `onStartCommand()` en el servicio.

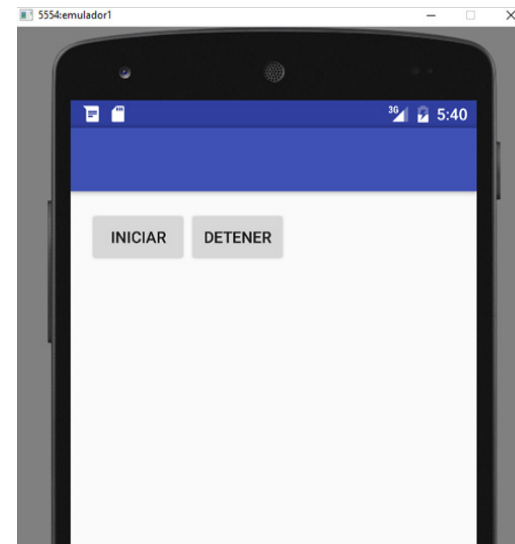
Una vez iniciado el servicio este quedará en ejecución indefinidamente hasta que se detenga. La detención del servicio se realizará a través del método `stopService()` de Context, a este método le pasaremos un objeto Intent asociado al servicio que queremos detener:

```
this.stopService(new Intent(this, ClaseServicio.class));
```

Cuando un servicio va a realizar una tarea de larga duración, lo habitual es implementar la misma dentro de un `AsyncTask` como hicimos en los casos de comunicación entre aplicaciones. La tarea se pondría en ejecución desde el `onStartCommand()` del servicio.

Ejemplo resuelto

A continuación presentaremos un pequeño ejemplo de implementación de un servicio. Se trata simplemente de una actividad con dos botones para iniciar y detener un servicio, cuya función será simplemente mostrarnos un mensaje de alerta cuando tengan lugar ambos sucesos.



Android resource file

Al servicio le daremos como nombre ServicioPrueba.

Se creará entonces una nueva clase extendiendo Service y el servicio quedará registrado en el AndroidManifest.xml a continuación de la actividad:

```
</activity>
<service
    android:name="com.example.ejemplo_servicio.ServicioPrueba"/>
```

A continuación te presentamos el código de implementación del servicio, en el que como podemos ver, se sobreescribe el método abstracto onBind() y los métodos onStartCommand() y onDestroy(), que es en los que incluiremos el código del servicio:

```
public class ServicioPrueba extends Service {
    public ServicioPrueba() {
    }

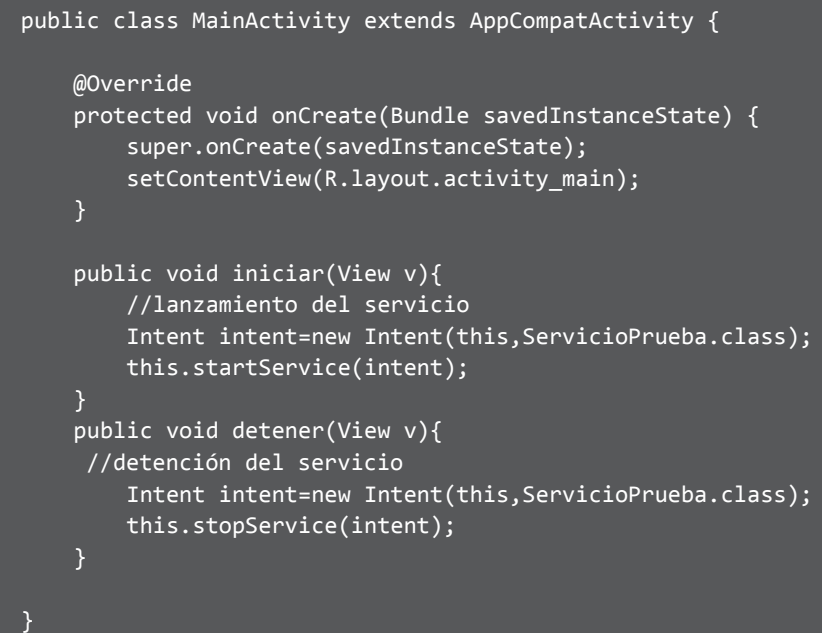
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
        Toast.makeText(this, "se ha iniciado el servicio", Toast.
LENGTH_LONG).show();
        return Service.START_STICKY;
    }

    @Override
    public void onDestroy() {

        super.onDestroy();
        Toast.makeText(this, "se ha destruido el servicio", Toast.
LENGTH_LONG).show();
    }
}
```

En cuanto a la actividad, implementará los dos métodos manejadores de los botones con que cuenta la actividad:



```
public class MainActivity extends AppCompatActivity {

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

    public void iniciar(View v){
        //lanzamiento del servicio
        Intent intent=new Intent(this,ServicioPrueba.class);
        this.startService(intent);
    }
    public void detener(View v){
        //detención del servicio
        Intent intent=new Intent(this,ServicioPrueba.class);
        this.stopService(intent);
    }

}
```

3. Acceso a la interfaz gráfica desde un servicio

En algunas ocasiones puede ocurrir que desde un servicio que se está ejecutando en segundo plano se quiera acceder a la interfaz gráfica de la actividad que lo ha lanzado para modificar el contenido de algún widget. El problema es que, desde la versión 3, **Android no permite acceder a los componentes de la actividad desde fuera de ella**, necesitaremos recurrir a los objetos Handler o manejadores de interfaz gráfica.

Para realizar modificaciones en la interfaz gráfica, definiremos en la actividad una subclase de `android.os.Handler` que sobrescriba el método `handleMessage()`, a través del cual se recibirán los mensajes desde los componentes externos. Su formato es el siguiente:

```
public void handleMessage(Message msg)
```

Este método será **invocado desde el servicio**, quien enviará un objeto `Message` con los datos para actualizar la interfaz gráfica de la actividad. Desde el interior de `handleMessage()`, se recogerán estos datos y se realizarán las modificaciones correspondientes en los widgets.

Para enviar mensajes al objeto Handler desde el componente externo, en este caso un servicio, se deberá primeramente obtener un objeto `Message` a través del siguiente método de Handler:

```
Message obtainMessage(int what, int arg1, int arg2, Object ob)
```

Los parámetros representan los cuatro datos que caracterizan a un `Message`, que son tres enteros y un objeto. El **significado de estos datos es el que le quiera dar el programador**, todo dependerá del uso que se haga de esos datos en el Handler.

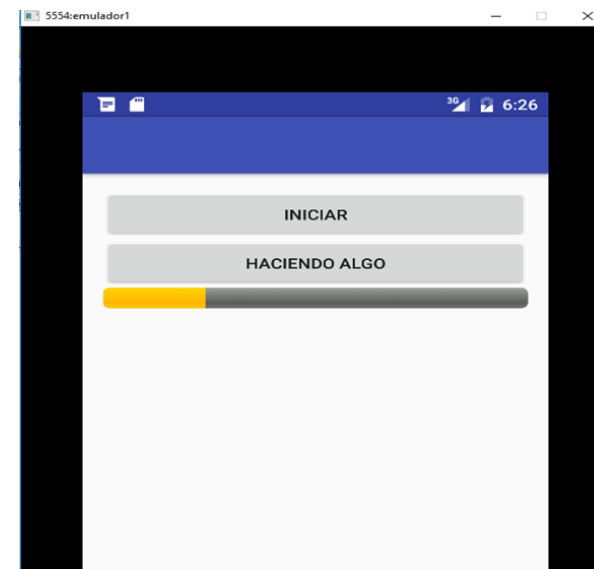
Una vez obtenido el objeto `Message`, llamaremos al método `sendToTarget()` de esta clase para proceder al envío del mensaje al `Handler`, en el que se ejecutará el método `handleMessage()`.

Ejemplo resuelto

Para aclarar mejor lo que acabamos de explicar, vamos a realizar un ejemplo práctico de implementación de un servicio desde el que enviaremos mensajes a un `Handler` para que pueda realizar modificaciones de la interfaz gráfica.

El servicio se encargará de realizar una tarea de larga duración consistente en el cálculo de la suma de un conjunto de números. Para simular la larga duración del proceso, después de cada suma el servicio se pondrá a dormir durante 100 milisegundos.

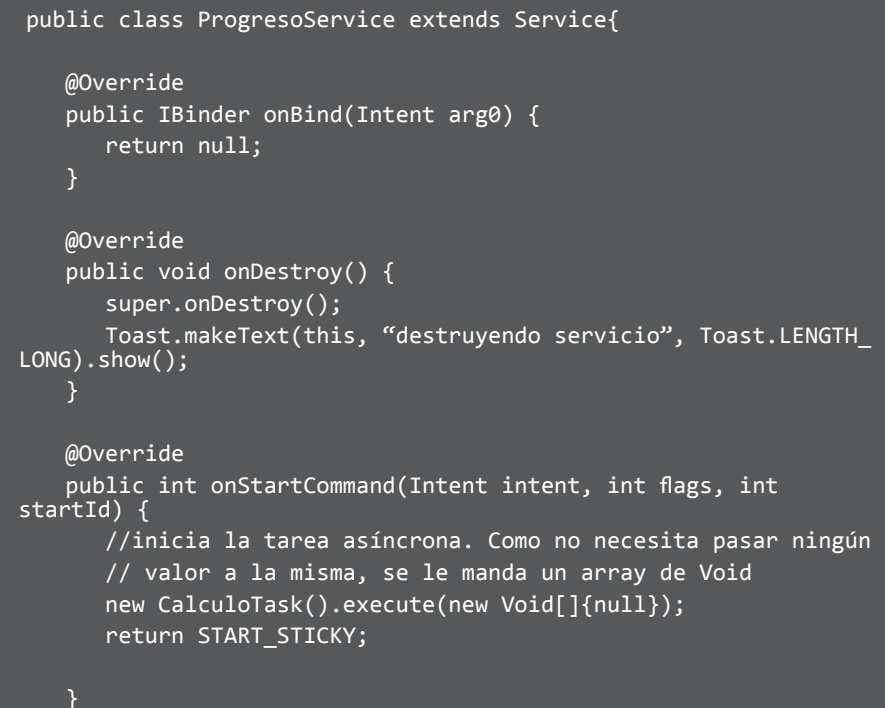
Por otro lado, tendremos una actividad que además de realizar el lanzamiento del servicio, contará con una barra de progreso que irá mostrando la evolución del proceso de cálculo.



Para mantener actualizado el estado de esta barra de progreso, el servicio tendrá que enviar periódicamente a un objeto Handler de la actividad el porcentaje de trabajo realizado, dicho valor será utilizado por el Handler para modificar el valor de la barra de progreso.

Respecto al botón “haciendo algo” de la actividad, simplemente se encargará de mostrar un mensaje de aviso, pues su objetivo no es más que demostrar que mientras el usuario interacciona con la actividad, el servicio en segundo plano sigue trabajando.

En primer lugar, crearemos el servicio tal y como explicamos en el ejemplo resuelto anterior. El nombre de este servicio será ProgresoService y el código del mismo será el siguiente:

A screenshot of a code editor window with a dark background. The window has a title bar with standard minimize, maximize, and close buttons. The code is in Java and defines a class ProgresoService that extends Service. It includes three overridden methods: onBind, onDestroy, and onStartCommand. The onBind method returns null. The onDestroy method calls super.onDestroy() and shows a toast message "destruyendo servicio". The onStartCommand method starts an asynchronous task using CalculoTask and returns START_STICKY. An orange checkmark icon is visible at the bottom right of the code editor.

```
public class ProgresoService extends Service{

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

    @Override
    public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "destruyendo servicio", Toast.LENGTH_
LONG).show();
    }

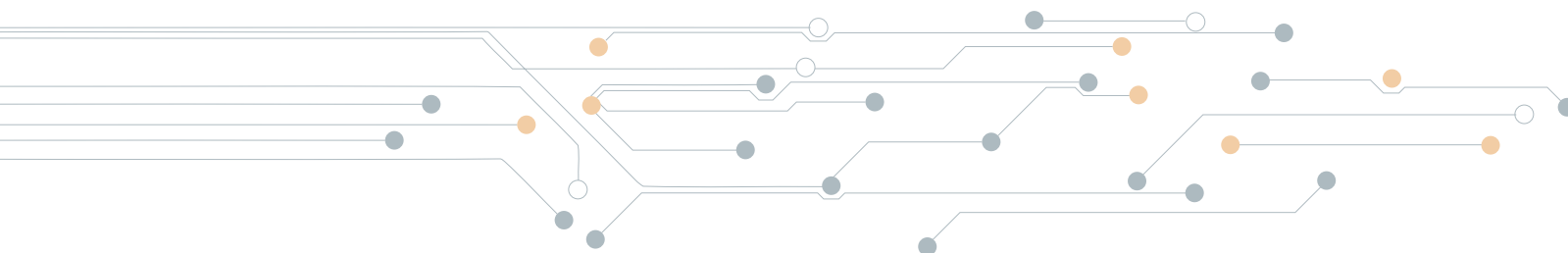
    @Override
    public int onStartCommand(Intent intent, int flags, int
startId) {
        //inicia la tarea asíncrona. Como no necesita pasar ningún
// valor a la misma, se le manda un array de Void
new CalculoTask().execute(new Void[]{null});
        return START_STICKY;
    }

}
```

```
private class CalculoTask extends AsyncTask<Void,Integer,Long>{
    @Override
    protected Long doInBackground(Void... arg0) {
        //simula una tarea de larga duración en donde
        //se tarda mucho en realizar la suma de los números
        //de 1 a 100
        long result=0;
        for(int i=1;i<=100;i++){
            result+=i;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //con cada suma, se hace la llamada a este método
            //que provocará la ejecución de onProgressUpdate
            publishProgress(i);
        }
        return result;
    }
}
```

```
@Override
protected void onPostExecute(Long result) {
    Toast.makeText(getBaseContext(), "calculo finalizado "+
        result, Toast.LENGTH_LONG).show();
}

@Override
protected void onProgressUpdate(Integer... values) {
    System.out.println(values[0]+" % calculado");
    //enviamos un mensaje al Handler con el porcentaje
    //de calculo realizado. Se lo pasamos en el segundo
    //parámetro
    MainActivity.manejador.
        obtainMessage(0,values[0],0,null).sendToTarget();
}
}
```



Al crear el servicio con el asistente de Android Studio, habrá quedado registrado como tal en el archivo de manifiesto:

```
<service android:name="servicios.ProgresoService"/>
```

En los casos de tareas asíncronas que hemos implementado hasta el momento siempre sobrescribíamos *doInBackground()* y *onPostExecute()*, en este caso, sobrescribimos también ***onProgressUpdate()***, método que es **llamado durante la ejecución de la tarea** (cada vez que llamamos a *publishProgress()* de *AsyncTask*) y que en este caso se encarga de enviar un mensaje al objeto *Handler* de la actividad con el porcentaje de trabajo realizado hasta el momento por el servicio.

El siguiente listado corresponde al código de la actividad:

```
public class MainActivity extends AppCompatActivity {

    private static ProgressBar pb;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        pb=(ProgressBar)findViewById(R.id.progreso);
        //valor máximo a alcanzar en la barra de progreso
        pb.setMax(100);
    }
    public void onIniciar(View v){
        startService(new Intent(this, ProgresoService.class));
    }

    public void onProgreso(View v){
        //muestra un mensaje cualquiera
        Toast.makeText(this, "Haciendo otras cosas", Toast.
LENGTH_LONG).show();
    }
    //Utilizado para actualizar la interfaz gráfica
    // de la actividad principal
    public static Handler manejador = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            //el valor del segundo parámetro es recogido
            //mediante el atributo arg1
            int progreso = msg.arg1;
            //actualizamos el estado de la barra de
            //progreso
            pb.setProgress(progreso);
        }
    };
};
```

Como podemos observar, el objeto Handler se guarda en una variable pública y estática, a fin de que pueda ser invocado desde el servicio mediante la expresión *ClaseActividad.objeto*. Vemos como para recuperar el valor enviado desde el servicio recurrimos a la propiedad *arg1* de *Message*, que se corresponde al segundo parámetro indicado en el método *obtainMessage()*. Podríamos haber utilizado el primer parámetro, el tercero o incluso el cuarto, y es que como ya se indicó antes, el significado y utilidad de estos parámetros lo elegimos nosotros.

En algunas ocasiones puede ocurrir que desde un servicio que se está ejecutando en segunda plano se quiera acceder a la interfaz gráfica de la actividad que lo ha lanzado para modificar al contenido de algún widget. El problema es que, desde la versión 3, **Android no permite acceder a los componentes de la actividad desde fuera de ella**, necesitaremos recurrir a los objetos Handler o manejadores de interfaz gráfica.

4. Tareas repetitivas en un servicio

En numerosas ocasiones los servicios que se ejecutan en segundo plano tienen que realizar tareas repetitivas, como comprobar periódicamente el estado de algún sistema, realizar volcado de datos cada cierto tiempo, etc. Para este tipo de tareas nos apoyaremos en las clases **Timer** y **TimerTask** del paquete `java.util`.

Un objeto `Timer` es un temporizador y se crearía utilizando el constructor sin parámetros de la clase `Timer`:

```
Timer tm=new Timer();
```

A partir de ahí, podemos definir la ejecución periódica de un determinado bloque de instrucciones a través del método `schedule`, cuyo formato es el siguiente:

```
schedule(TimerTask task, long delay, long period)
```

El primer parámetro del método es un objeto `TimerTask` donde se define el código de la tarea repetitiva. La clase `TimerTask` dispone del método abstracto `run()`, que habrá que sobrescribir y codificar en él las operaciones a realizar por la tarea repetitiva.

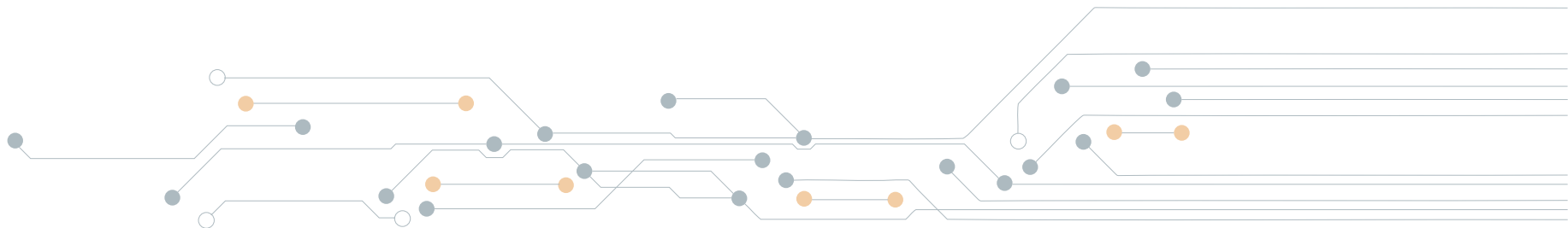
Los parámetros `delay` y `period` representan el retardo en el comienzo de la tarea y la periodicidad de ejecución, respectivamente. Ambos se miden en milisegundos.

Por ejemplo, si queremos que un servicio nos muestre un mensaje de aviso cada minuto, el código que programaríamos en el método `onStartCommand()` para iniciar la tarea repetitiva sería el siguiente:

```
Timer tm=new Timer();
tm.scheduleAtFixedRate(new TimerTask(){
    public void run(){
        Toast.makeText(ClaseServicio.class,
            "Saludo",Toast.LENGTH_LONG).show();
    }
}, 0, 1000);
```

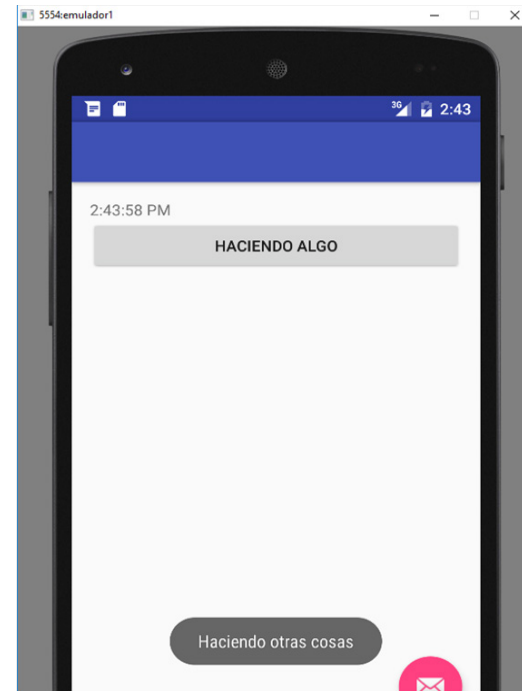
En este caso, se utiliza una clase anónima para extender `TimerTask` y proporcionar una implementación de `run()`. También vemos que, dado que `Service` extiende `Context`, se utiliza la propia clase del servicio como contexto en el método `makeText()`.

Para detener el temporizador llamaremos al método `cancel()` del objeto `Timer`.



Ejemplo resuelto

Veamos un nuevo ejemplo de servicio, esta vez de realización de tareas periódicas. En este caso se trata de un servicio que se encarga de calcular la hora del sistema cada medio segundo y pasarla a la actividad para que la muestre en un TextView. Así, mientras el usuario trabaja normalmente con la actividad, el servicio se encargará en segundo plano de mantener la hora actualizada en todo momento



El siguiente listado correspondería al código del servicio:

```
public class ServicioReloj extends Service {
    Timer t;

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

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        iniciarTemporizador();
        return START_STICKY;
    }

    @Override
    public void onDestroy() {
        //cancelamos el temporizador al finalizar el servicio
        t.cancel();
        super.onDestroy();
    }

    private void iniciarTemporizador(){
        t=new Timer();
        //inicia el temporizador de manera inmediata
        // con una repetición de 0.5 segundos
        t.schedule(new MiTimer(), 0,500);
    }

    private class MiTimer extends TimerTask{

        @Override
        public void run() {
```

```

        Date d=new Date();
        //creamos un dateformat para formatear
        //la hora en formato medio
        DateFormat df=DateFormat.getTimeInstance(DateFormat.
MEDIUM);
        //aplicamos el formato a la hora obtenida en Date
        String hora=df.format(d);
        //mostramos la hora en el TextView a
        //través del handler al que le enviamos un mensaje
        MainActivity.manejador.obtainMessage(0,0,0, hora).
sendToTarget();

    }

}
```

Según vemos en el listado anterior, el temporizador es iniciado al lanzarse el servicio y cuando este es destruido se procede a su cancelación.

Podemos observar también como la hora es enviada en el parámetro Object del objeto Message generado con *obtainMessage()*.

El siguiente código corresponde a la actividad de la aplicación:

```
public class MainActivity extends AppCompatActivity {
    static TextView tvReloj;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        tvReloj=(TextView)findViewById(R.
id.tvReloj);
    }
    @Override
    protected void onResume() {
        startService(new Intent(this, ServicioReloj.class));
        super.onResume();
    }

    @Override
    protected void onPause() {
        stopService(new Intent(this, ServicioReloj.class));
        super.onPause();
    }
    public void progreso(View v){
        //muestra un mensaje cualquiera
        Toast.makeText(this, "Haciendo otras cosas",
```





```
Toast.LENGTH_LONG).show();
}

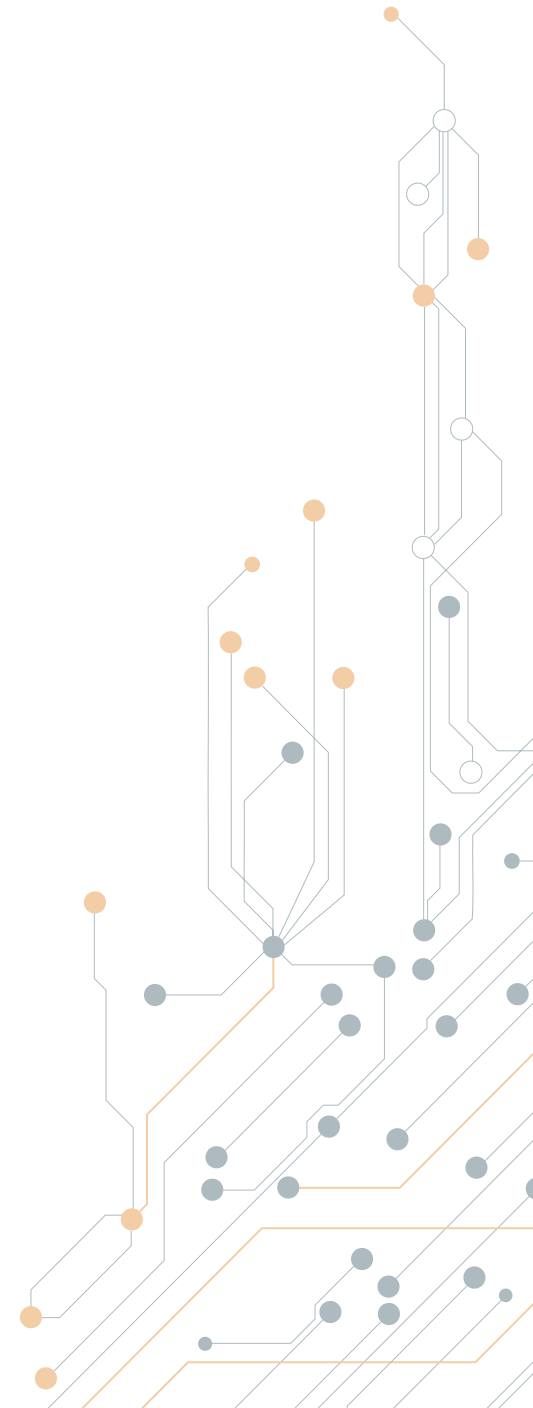
//se crea un objeto de una clase anónima, subclase de
Handler,
//que recibe mensajes procedentes de un hilo secundario
public static Handler manejador=new Handler(){

    @Override
    public void handleMessage(Message msg) {
        //recupera la hora en el parámetro
        //Object enviado a través del
        //objeto Message
        String hora=(String)msg.obj;
        tvReloj.setText(hora);

    }

};

}
```



Telefonica

EDUCACIÓN DIGITAL