

UNIDAD 5

CICLO DE VIDA E HILOS



Diseño de Interfaces Web



UD5. Ciclo de Vida e Hilos

1. Introducción
2. Ciclo de vida de una aplicación
3. Guardar y recuperar el estado de una actividad
4. Procesos en Hilos en Android
5. Menús de Android



UD5. Ciclo de Vida e Hilos

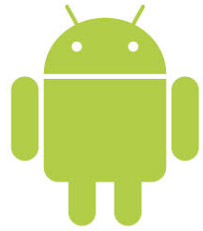
1. **Introducción**
2. Ciclo de vida de una aplicación
3. Guardar y recuperar el estado de una actividad
4. Procesos en Hilos en Android
5. Menús de Android



1. Introducción

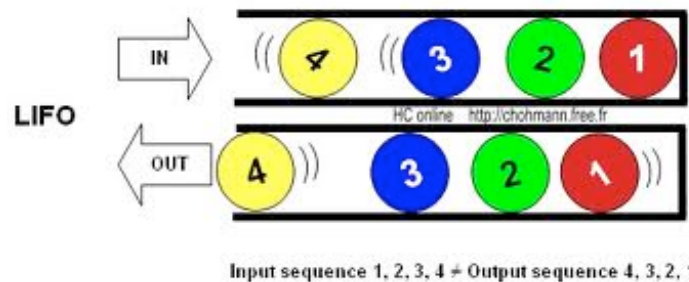
- Una **Actividad** (Activity) es un componente de Android que ofrece una pantalla con la que los usuarios pueden interactuar con la aplicación, como marcar el teléfono, sacar una foto, enviar un correo electrónico o ver un mapa.
- Cada Actividad tiene asociada una **ventana** en la que se dibuja la interfaz de usuario.
- Normalmente, esta ventana ocupa toda la pantalla, aunque puede ser menor que ésta o flotar sobre otras ventanas.



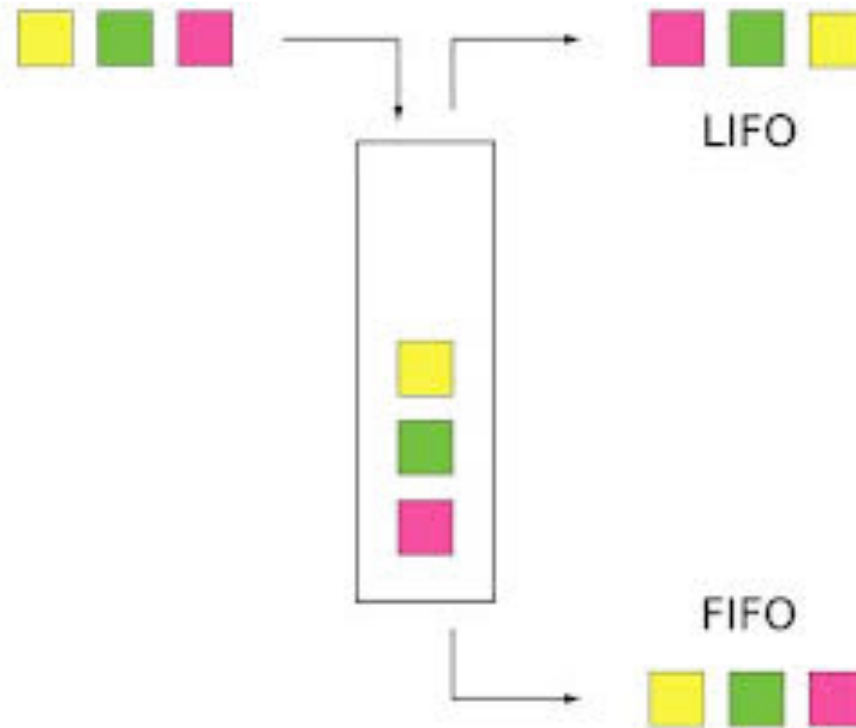
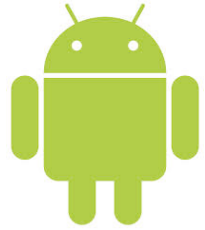


1. Introducción

- Una aplicación de Android consta de múltiples actividades que están más o menos ligadas entre sí.
- Habitualmente, se define una actividad "principal", que es la que se presenta al usuario cuando se inicia la aplicación por primera vez.
- Una actividad puede iniciar otra actividad con el fin de realizar diferentes operaciones.
- Cada vez que comienza una nueva actividad, la actividad anterior se detiene y la envía a una pila de retroceso ("**back stack**").
- Esta pila usa el mecanismo de cola **LIFO** ("last in, first out"), por lo que, cuando el usuario pulsa la tecla "Volver atrás" del móvil, se extrae de la pila la actividad anterior (destruyéndose la pila) y se reanuda.



1. Introducción





1. Introducción

- Cuando una actividad se para porque se inicia una nueva actividad, se le notifica este cambio de estado a través de los métodos de llamada **callback** del ciclo de vida de la actividad.
- Una función de llamada (en inglés callback) es una función que se remite a Android cuando se inicia una Actividad, para que el sistema operativo la “llame” durante la ejecución de esta Actividad.





1. Introducción

- Existen varios métodos de llamada *callback* que una actividad puede recibir debido a un cambio en su estado; por ejemplo, cuando el sistema crea la Actividad, cuando se reactiva o cuando se destruye.
- El programador puede aprovechar estos métodos para ejecutar sentencias específicas apropiadas para el cambio de estado.
- Por ejemplo, **cuando una Actividad se suspende es recomendable liberar de la memoria todos los objetos grandes.**
- Cuando la actividad se reanuda, se puede volver a reservar los recursos necesarios y continuar con las acciones que se interrumpieron.
- Estos cambios de estado forman parte del ciclo de vida de la actividad.



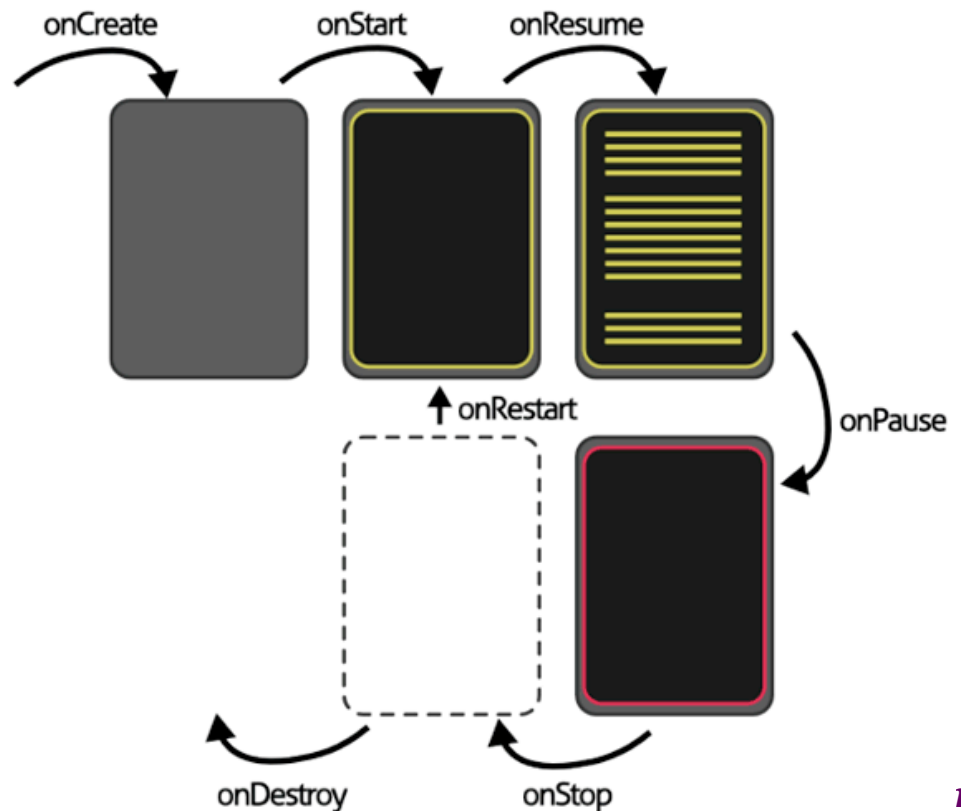
UD5. Multimedia y Ciclo de Vida

1. Introducción
2. **Ciclo de vida de una aplicación**
3. Utilizando multimedia en Android
4. Reproducir un video con VideoView
5. La clase MediaPlayer



2. Ciclo de vida de una aplicación

- Para crear una Actividad usamos la clase *Activity* de Android.
- En la subclase creada es necesario implementar los métodos *callback* que el sistema puede invocar cuando hay un cambio en su ciclo de vida, la actividad:
 - se está creando,
 - se detiene,
 - se reanuda
 - o se destruye.





2. Ciclo de vida de una aplicación

Los dos métodos callback más importantes son:

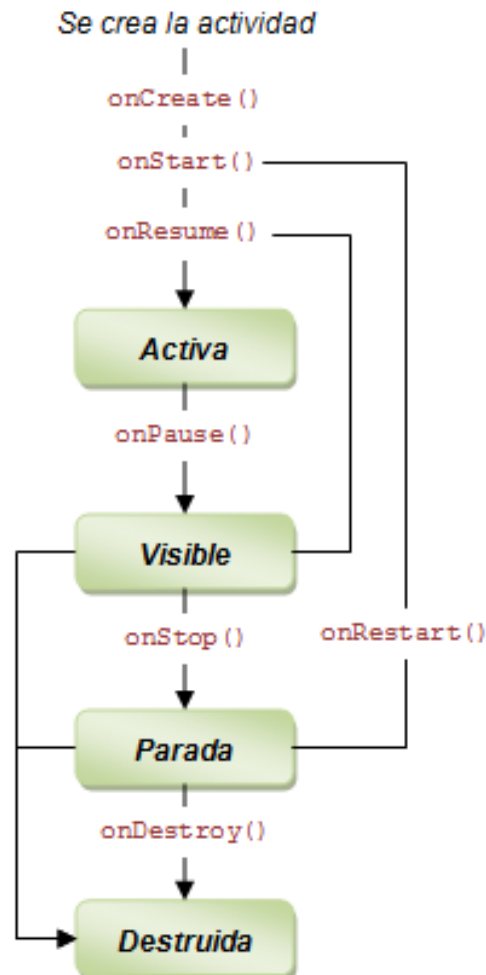
- **onCreate():** es obligatorio implementar este método ya que el sistema lo invoca cuando crea su actividad. Dentro de su implementación debemos iniciar los componentes esenciales de la actividad, como dibujar la interfaz de usuario empleado la función `setContentView()`.
- **onPause():** el sistema llama a este método cuando el usuario detiene la actividad, aunque no significa que la actividad se destruya. Aquí es donde el programador debe guardar todos los cambios que deben persistir en la siguiente sesión del usuario, ya que éste podría no volver a la Actividad y que ésta se destruyera.



2. Ciclo de vida de una aplicación

- Una Actividad puede mantenerse en cuatro estados:
 - **Activa** (*Running*): La actividad está encima de la pila, lo que quiere decir que es visible y tiene el foco.
 - **Visible** (*Paused*): La actividad es visible pero no tiene el foco. Se alcanza este estado cuando pasa a activa otra actividad con alguna parte transparente o que no ocupa toda la pantalla. Cuando una actividad está tapada por completo, pasa a estar parada. La actividad pausada sigue “viva” ya que se mantiene en memoria y conserva toda la información de estado, si bien el sistema operativo puede eliminarla en caso de memoria disponible muy baja.
 - **Parada** (*Stopped*): Cuando la actividad no es visible, se recomienda guardar el estado de la interfaz de usuario, preferencias, etc.
 - **Destruída** (*Destroyed*): Cuando la actividad termina al invocarse el método *finish()*, o es matada por el sistema Android, sale de la pila de actividades.

2. Ciclo de vida de una aplicación





2. Ciclo de vida de una aplicación

- **onCreate(Bundle):** Se llama en la creación de la actividad. Se utiliza para realizar todo tipo de inicializaciones, como la creación de la interfaz de usuario o la inicialización de estructuras de datos. Puede recibir información de estado de instancia (en una instancia de la clase Bundle), por si se reanuda desde una actividad que ha sido destruida y vuelta a crear.
- **onStart():** Nos indica que la actividad está a punto de ser mostrada al usuario.
- **onResume():** Se llama cuando la actividad va a comenzar a interactuar con el usuario. Es un buen lugar para lanzar las animaciones y la música.
- **onPause():** Indica que la actividad está a punto de ser lanzada a segundo plano, normalmente porque otra aplicación es lanzada. Es el lugar adecuado para detener animaciones, música o almacenar los datos que estaban en edición.



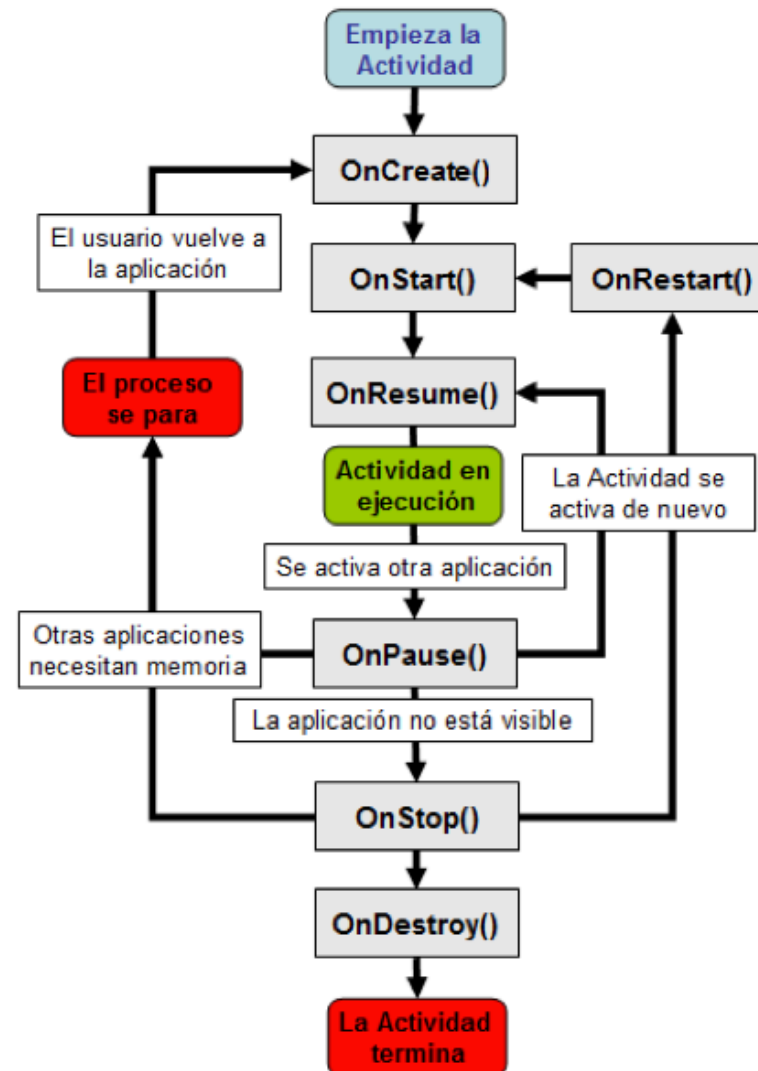
2. Ciclo de vida de una aplicación

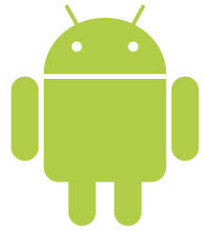
- **onStop()**: La actividad ya no va a ser visible para el usuario. Ojo si hay muy poca memoria, es posible que la actividad se destruya sin llamar a este método.
- **onRestart()**: Indica que la actividad va a volver a ser representada después de haber pasado por *onStop()*.
- **onDestroy()**: Se llama antes de que la actividad sea totalmente destruida. Por ejemplo, cuando el usuario pulsa el botón <volver> o cuando se llama al método *finish()*. Ojo si hay muy poca memoria, es posible que la actividad se destruya sin llamar a este método.

2. Ciclo de vida de una aplicación




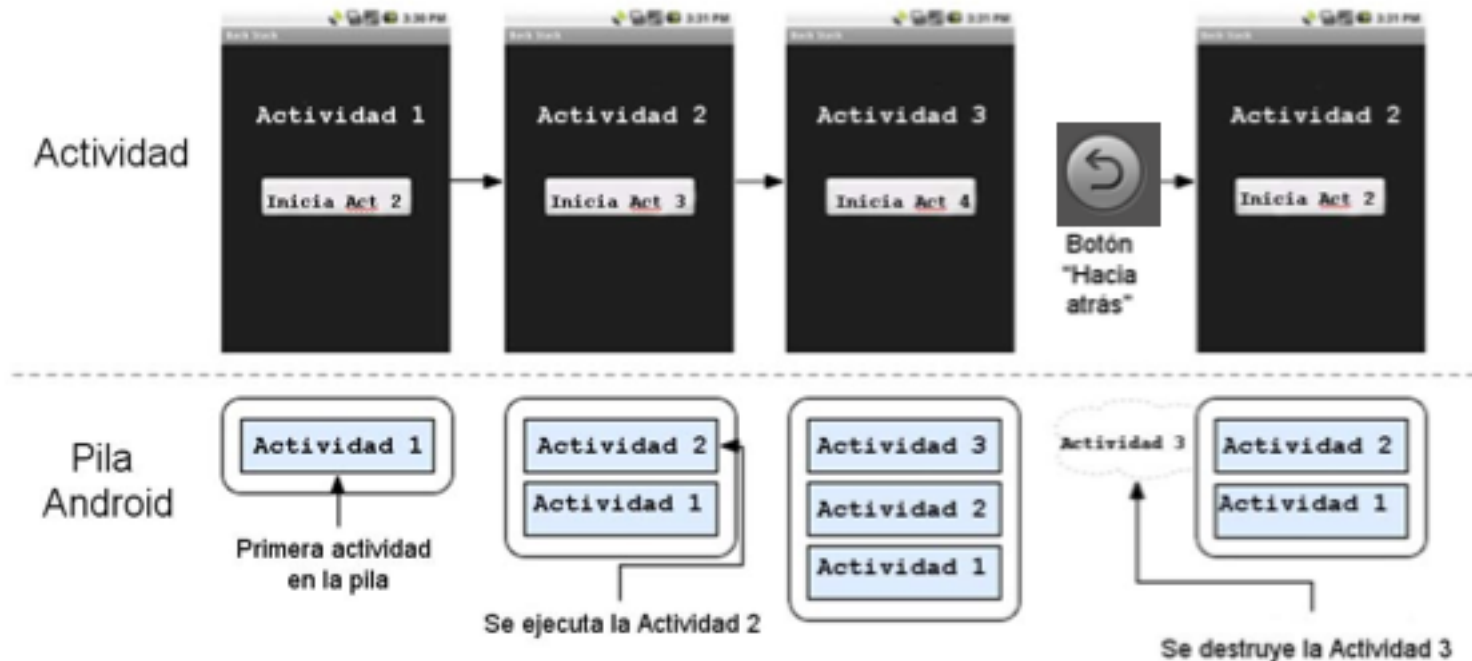
Ciclo vida Actividad





2. Ciclo de vida de una aplicación

- Si el usuario pulsa la tecla de retroceso  del teléfono, la Actividad actual se quita de la parte superior de la pila, se destruye y se reanuda la Actividad anterior en el estado que estuviera.
- Las actividades en la pila no se pueden reorganizar, se trata de una pila de tipo "último en entrar, primero en salir" ("last in, first out").





2. Ciclo de vida de una aplicación

- Si el usuario continúa presionando "Volver atrás", se extraerá una a una cada actividad de la pila mostrando la anterior, hasta que aparezca la pantalla de inicio u otra aplicación que se haya iniciado antes. Cuando esto ocurre, la tarea se destruye.
- Una tarea es una unidad compuesta de actividades que puede pasar a un segundo plano cuando los usuarios inician una nueva tarea o van a la pantalla de inicio. Cuando una tarea se encuentra en segundo plano, todas sus actividades se detienen, aunque su pila de ejecución se mantiene intacta hasta que el usuario decida que vuelva al "primer plano".
- Android denomina a esta manera de guardar el estado de las aplicaciones Multitarea (**Multitasking**), si bien no se trata de ejecución simultánea de las aplicaciones, sino de la posibilidad de seguir en el estado donde la dejamos cuando ejecutemos otra aplicación y volvamos a la inicial.



2. Ciclo de vida de una aplicación

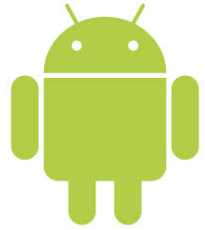
- Si el usuario continúa presionando "Volver atrás", se extraerá una a una cada actividad de la pila mostrando la anterior, hasta que aparezca la pantalla de inicio u otra aplicación que se haya iniciado antes. Cuando esto ocurre, la tarea se destruye.
- Una tarea es una unidad compuesta de actividades que puede pasar a un segundo plano cuando los usuarios inician una nueva tarea o van a la pantalla de inicio. Cuando una tarea se encuentra en segundo plano, todas sus actividades se detienen, aunque su pila de ejecución se mantiene intacta hasta que el usuario decida que vuelva al "primer plano".
- Android denomina a esta manera de guardar el estado de las aplicaciones Multitarea (**Multitasking**), si bien no se trata de ejecución simultánea de las aplicaciones, sino de la posibilidad de seguir en el estado donde la dejamos cuando ejecutemos otra aplicación y volvamos a la inicial.

2. Ciclo de vida de una aplicación

Práctica:

El ciclo de vida de una actividad





UD5. Ciclo de Vida e Hilos

1. Introducción
2. Ciclo de vida de una aplicación
3. **Guardar y recuperar el estado de una actividad**
4. Procesos en Hilos en Android
5. Menús de Android

3. Guardar y recuperar el estado de una actividad

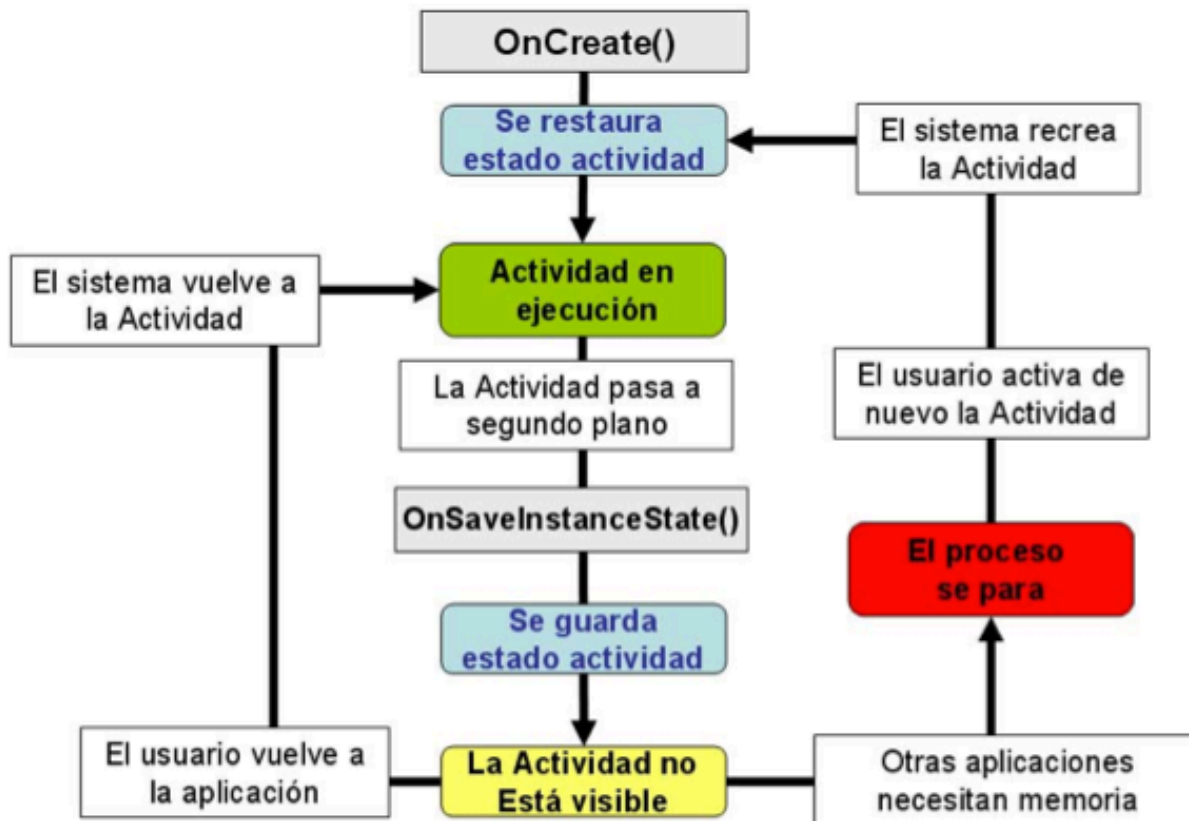


- La configuración de un teléfono puede cambiar mientras un usuario ejecuta una aplicación.
 - Por ejemplo, la orientación de la pantalla (vertical/horizontal), la disponibilidad de teclado, el idioma, etcétera.
- Cuando este cambio se produce, Android reinicia la Actividad usando el método *OnDestroy()* e inmediatamente seguido por *onCreate()*.
- Este comportamiento de reinicio está diseñado para que la aplicación se adapte a la nueva configuración de forma automática, por ejemplo, cuando cambia la posición de los componentes.

3. Guardar y recuperar el estado de una actividad



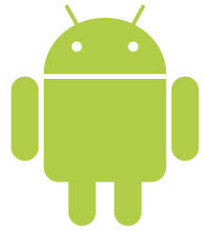
- La mejor manera de manejar un cambio de configuración para preservar el estado de la aplicación es usar los métodos **onSaveInstanceState()** y **onCreate()**.



3. Guardar y recuperar el estado de una actividad



- Para almacenar la información importante sobre el estado de ejecución de una Actividad podemos usar el método de llamada (callback) de Android **onSaveInstanceState()** y luego restaurar esta información cuando el sistema vuelva a crear la Actividad.
- El sistema llama a este método justo antes de que la Actividad se destruya y le pasa como parámetro un objeto de tipo Bundle.
- Este objeto Bundle es donde podemos almacenar la información del estado de la Actividad como pareja nombre-valor, utilizando el método `putString()`.
- De esta manera, si Android mata el proceso de la Actividad y el usuario vuelve a ejecutar la misma Actividad, el sistema pasa el objeto Bundle almacenado anteriormente como parámetro en el método `onCreate()`, para que pueda restaurar el estado de ejecución. Si no hay información sobre este estado, el objeto Bundle es nulo.



```
String var;
```

```
int pos;
```

```
@Override
```

```
protected void onSaveInstanceState(Bundle guardarEstado) {
```

```
    super.onSaveInstanceState(guardarEstado);
```

```
    guardarEstado.putString("variable", var);
```

```
    guardarEstado.putInt("posicion", pos);
```

```
}
```

```
@Override
```

```
protected void onRestoreInstanceState(Bundle recEstado) {
```

```
    super.onRestoreInstanceState(recEstado);
```

```
    var = recEstado.getString("variable");
```

```
    pos = recEstado.getInt("posicion");
```

```
}
```

3. Guardar y recuperar el estado de una actividad



- Si no queremos que un campo sea guardado, definimos el objeto con el atributo `android:saveEnabled="false"` para que no se almacene el campo.

```
<EditText android:id="@+id/passwordEditText"
    android:inputType="textPassword"
    android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:textSize="18sp" android:layout_marginLeft="3dip"
    android:layout_marginRight="3dip" android:layout_marginTop="3dip"
    android:saveEnabled="false">
</EditText>
```

3. Guardar y recuperar el estado de una actividad

Práctica:

Guardar el estado en una actividad



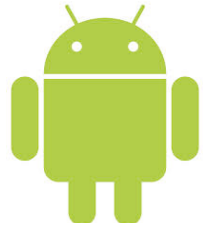
Introduce el usuario:

Introduce tu contraseña (no se guardará):



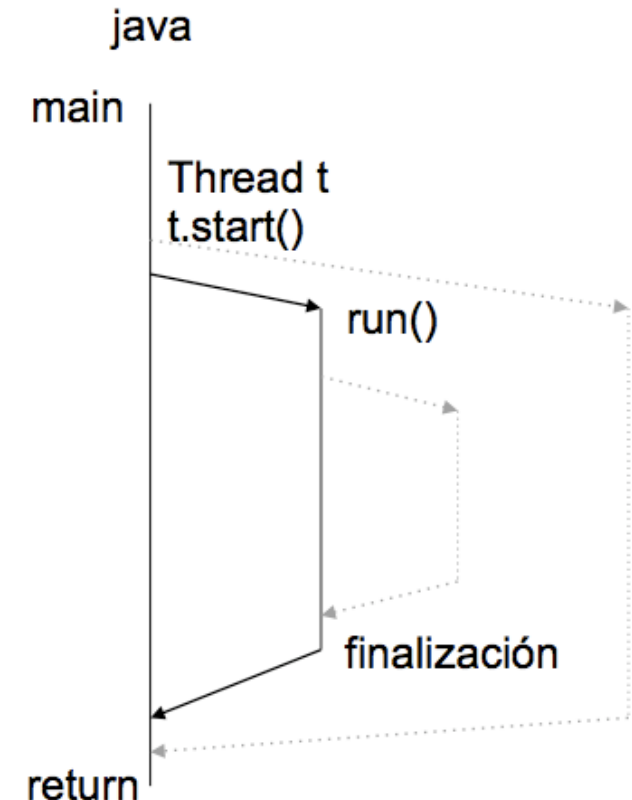
UD5. Ciclo de Vida e Hilos

1. Introducción
2. Ciclo de vida de una aplicación
3. Guardar y recuperar el estado de una actividad
4. **Procesos en Hilos en Android**
5. Menús de Android



4. Procesos en Hilos en Android

- Un hilo es una característica de la informática que permite a una aplicación realizar varias tareas a la vez (concurrentemente).
- Esencialmente, un hilo es una tarea que se ejecuta en paralelo con otra tarea.
- Todos los hilos que se ejecutan a la vez pueden compartir una serie de recursos, tales como la memoria, los archivos abiertos, etcétera.
- Esta técnica permite simplificar el diseño de una aplicación que puede así llevar a cabo distintas funciones simultáneamente.





4. Procesos en Hilos en Android

- Cuando un usuario ejecuta una aplicación en Android se inicia un nuevo proceso de Linux con un único hilo de ejecución.
- Por defecto, todos los componentes de una misma aplicación se ejecutan en el mismo hilo principal (en inglés se denomina "main thread").
- A este hilo también se lo denomina hilo de la interfaz de usuario (en inglés, "UI thread").
- Si se inicia una Actividad de una aplicación que ya se está ejecutando y, por lo tanto, ya existe un proceso asociado, entonces la nueva Actividad se inicia dentro de ese proceso y utiliza el mismo hilo de ejecución.
- Sin embargo, es posible diseñar aplicaciones que ejecuten sus diferentes componentes de la aplicación en procesos separados y el programador puede crear subprocesos adicionales para cualquier proceso principal.



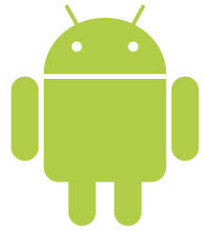
4. Procesos en Hilos en Android

- Cuando el usuario ejecuta una aplicación, Android crea un hilo de ejecución que se denomina principal (**main**).
- Este hilo es muy importante porque es el encargado de gestionar los eventos que dispara el usuario a los componentes adecuados e incluye también los eventos que dibujan la pantalla.
- Por esta razón, a este hilo principal también se le llama hilo de la interfaz de usuario (**UI thread**).
- **Este modelo de ejecución de las aplicaciones se denomina Modelo de ejecución de un único hilo** (en inglés, **Single Thread Model**).



4. Procesos en Hilos en Android

- Android no crea un subproceso independiente para cada componente de la aplicación.
- Todos sus componentes se ejecutan dentro del mismo proceso en el hilo principal.
- Por lo tanto, los métodos que responden a eventos del sistema, como `onTouch()`, siempre se ejecutan en este hilo principal o de la interfaz de usuario.
- Por ejemplo, cuando el usuario toca con el dedo un botón de la pantalla del teléfono, el hilo principal invoca el evento correspondiente en el widget apropiado y lo redibuja.



4. Procesos en Hilos en Android

- Si un usuario interacciona mucho con una aplicación, este modelo de ejecución con un único hilo puede dar lugar a poca fluidez.
- Llevar a cabo operaciones que van a tardar cierto tiempo, como acceder a Internet o consultar una base de datos, bloquearán la interfaz de usuario.
- Cuando el hilo principal está bloqueado la pantalla de aplicación se bloquea, desde el punto de vista del usuario, la aplicación se bloquea.
- Si el hilo principal está bloqueado durante 5 segundos, Android muestra al usuario una ventana con el mensaje "La aplicación no responde"





4. Procesos en Hilos en Android

- Es decir, en Android se puede manipular la interfaz de usuario desde otro subproceso. Por lo tanto, hay tener en cuenta dos reglas a la hora de diseñar aplicaciones en Android:
 - No bloquear nunca el hilo principal o de la interfaz de usuario.
 - No acceder a la interfaz de usuario de Android desde un hilo exterior.



4. Procesos en Hilos en Android

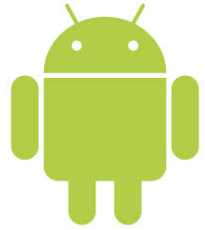
- **Es vital que no se bloquee el hilo principal** (o de interfaz de usuario) para que la interfaz de usuario de la aplicación responda a la interacción del usuario.
- Si la aplicación tiene que realizar operaciones que no son instantáneas, hay que ejecutarlas en hilos separados en segundo plano.
- Por ejemplo, a continuación se muestra el código para que se descargue de Internet una imagen cuando el usuario hace clic en un botón usando un hilo separado en segundo plano:

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://pag.es/imagen.png");  
            miImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```



4. Procesos en Hilos en Android

- A primera vista, el código anterior parece funcionar bien ya que crea un nuevo hilo en segundo plano para descargar la imagen.
- Sin embargo, no cumple la segunda regla del modelo de un hilo único: no acceder a la interfaz de usuario de Android desde un hilo exterior a esta interfaz de usuario.
- Esto puede producir un comportamiento inesperado en la aplicación y muy difícil de localizar.
- Para solucionar este inconveniente, Android ofrece varias formas de acceder al hilo de la interfaz de usuario desde otros hilos en segundo plano:
 - **Activity.runOnUiThread(Runnable)**
 - **View.post(Runnable)**
 - **View.postDelayed(Runnable, long)**



4. Procesos en Hilos en Android

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            final Bitmap b=loadImageFromNetwork("http://pag.es/imagen.png");  
            miImageView.post(new Runnable() {  
                public void run() {  
                    miImageView.setImageBitmap(b);  
                }  
            });  
        }  
    }).start();  
}
```

La mejor solución es extender la clase AsyncTask que simplifica la ejecución de las operaciones en segundo plano.



2. Hilos de ejecución

Creación de nuevos hilos con la clase Thread

- Como acabamos de ver siempre que tengamos que ejecutar un método que requiera bastante tiempo de ejecución, no podremos ejecutarlo en el hilo principal.
- Dado que este hilo ha de estar siempre disponible para atender los eventos generados por el usuario, **nunca debe ser bloqueado**.
- Crearemos una clase:

```
class MiThread extends Thread {  
  
    @Override  
  
    public void run() {  
  
        ...  
  
    }  
  
}
```



2. Hilos de ejecución

Creación de nuevos hilos con la clase Thread

- Y para crear el nuevo hilo y ejecutarlo:

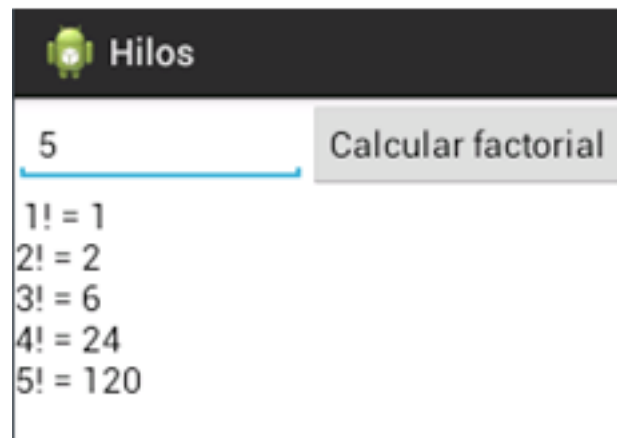
```
MiThread hilo = new MiThread();  
hilo.start();
```

- La llamada al método start() ocasionará que se cree un nuevo hilo y se ejecute el método run() en este hilo.
- La llamada al método start() es asíncrona: continuaremos ejecutando las instrucciones siguientes, sin esperar a que el método run() concluya.
- Si esperamos algún resultado de este método, será imprescindible establecer algún mecanismo de sincronización para saber cuando ha terminado.

4. Procesos en Hilos en Android

Práctica:

Tutorial: hilos de ejecución en Android



<http://www.androidcurso.com/index.php/tutoriales-android/36-unidad-5-entradas-en-android-teclado-pantalla-tactil-y-sensores/271-hilos-de-ejecucion-en-android>



4. Procesos en Hilos en Android

La clase AsyncTask

- La clase AsyncTask permite realizar operaciones asíncronas en la interfaz de usuario.
- Lleva a cabo las operaciones que bloquean la pantalla al usuario en un proceso de segundo plano y devuelve su resultado al hilo principal de la interfaz de usuario de manera sencilla.
- Para usar esta funcionalidad de Android hay que extender la clase AsyncTask e implementar los siguientes métodos callback:
 - **doInBackground():** inicia los procesos en segundo plano.
 - **onPostExecute():** actualiza la interfaz de usuario desde el hilo principal de la interfaz de usuario.

4. Proc

```
// Método asociado al botón "Descargar"

public void descargarImagen(View view) {

    imagen.setVisibility(View.INVISIBLE);

    cargando.setVisibility(View.VISIBLE);

    // Iniciamos la tarea de descarga

    TareaDescargaImagen tarea = new TareaDescargaImagen();

    tarea.execute(imagenURL);

}

// Clase que descarga una imagen de Internet como una tarea asíncrona.
// Es decir, podemos seguir usando la interfaz de usuario.

private class TareaDescargaImagen extends AsyncTask<String, Void, Bitmap> {

    // Método que se ejecuta en segundo plano

    protected Bitmap doInBackground(String... urls) {

        return loadImageFromNetwork(urls[0]);

    }

    // Cuando la tarea ha acabado se invoca automáticamente este método

    protected void onPostExecute(Bitmap resultado) {

        cargando.setVisibility(View.INVISIBLE);

        // Cargamos la imagen si se ha podido descargar la imagen de Internet

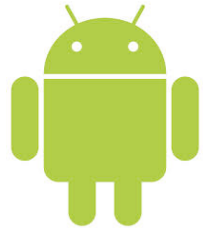
        if (resultado!=null) imagen.setImageBitmap(resultado);

        else imagen.setImageResource(R.drawable.error);

        imagen.setVisibility(View.VISIBLE);

    } // end onPostExecute

}
```



4. Procesos en Hilos en Android



```
// Método asociado al botón "Descargar"

public void descargarImagen(View view) {

    imagen.setVisibility(View.INVISIBLE);

    cargando.setVisibility(View.VISIBLE);

    // Iniciamos la tarea de descarga

    TareaDescargaImagen tarea = new TareaDescargaImagen();

    tarea.execute(imagenURL);

}
```



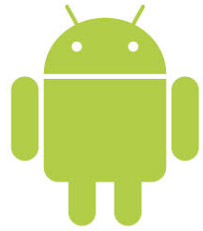
4. Procesos en Hilos en Android

```
// Cuando la tarea ha acabado se invoca automáticamente este método
protected void onPostExecute(Bitmap resultado) {
    cargando.setVisibility(View.INVISIBLE);

    // Cargamos la imagen su se ha podido descargar la imagen de Internet
    if (resultado!=null) imagen.setImageBitmap(resultado);
    else imagen.setImageResource(R.drawable.error);

    imagen.setVisibility(View.VISIBLE);
} // end onPostExecute
}
```

para iniciar la tarea hemos usado el método `execute()`.



4. Procesos en Hilos en Android

- Para definir la tarea asíncrona hemos extendido la clase *AsyncTask<String, Void, Bitmap>*
 - Parámetros (Params). El tipo de información que se necesita para procesar la tarea.
 - Progreso (Progress). El tipo de información que se pasa dentro de la tarea para indicar su progreso.
 - Resultado (Result). El tipo de información que se pasa cuando la tarea ha sido completada.
- Si no se utiliza alguno, utilizar el tipo *void*.

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```



4. Procesos en Hilos en Android

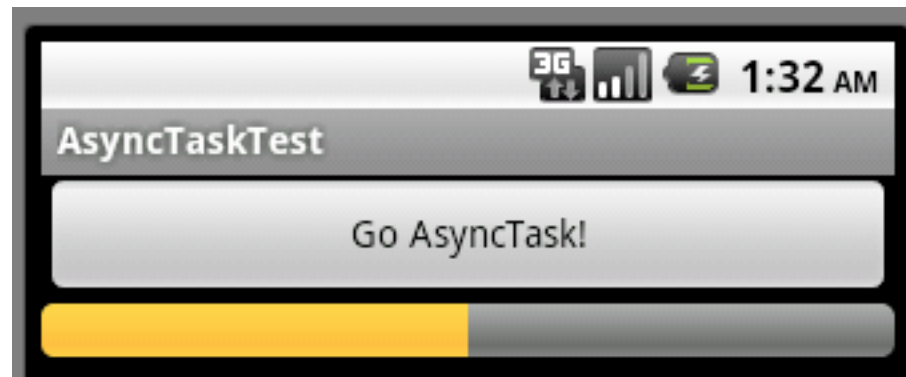
- Para poder trabajar sin problemas con la clase *AsyncTask*, existen algunas reglas acerca de los *threads* que debes tomar en cuenta:
 - La instancia de la tarea debe crearse en el *UI thread*.
 - El método *execute(Params...)* debe invocarse en el *UI thread*.
 - Los métodos *onPreExecute()*, *onPostExecute(Result)*, *doInBackground(Params...)*, *on ProgressUpdate(Progress...)* no deben llamarse de forma manual.
 - La tarea debe ser ejecutada sólo una vez (con excepción de que la ejecución corresponda a un segundo intento).



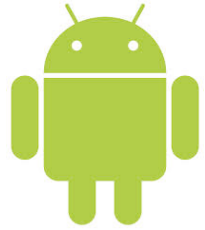
4. Procesos en Hilos en Android

- Cada vez que una tarea asíncrona se ejecuta, se pasa a través de 4 etapas:
- **onPreExecute()**. Se invoca en el *UI thread* inmediatamente después de que la tarea es ejecutada. Este paso se utiliza normalmente para configurar la tarea.
- **doInBackground(Params...)**. Se invoca en el subproceso en segundo plano inmediatamente después de que *onPreExecute()* termina de ejecutarse.
- **onProgressUpdate(Progress...)**. Se invoca en el *UI thread* después de una llamada a *publishProgress(Progress...)*. El momento de la ejecución es indefinido. Este método es utilizado para mostrar cualquier tipo de progreso en la interfaz de usuario mientras la tarea en segundo plano sigue ejecutándose.
- **onPostExecute(Result...)**. Se invoca en el *UI thread* después de que el proceso en segundo plano ha sido terminado.

4. Procesos en Hilos en Android



4. Procesos en Hilos en Android



```
<Button
    android:id="@+id/button1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Go AsyncTast!!" />

<ProgressBar
    android:id="@+id/progressbar1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    style="?android:attr/progressBarStyleHorizontal"
    android:max="100"
    android:progress="50"
```



4. Procesos en Hilos en Android

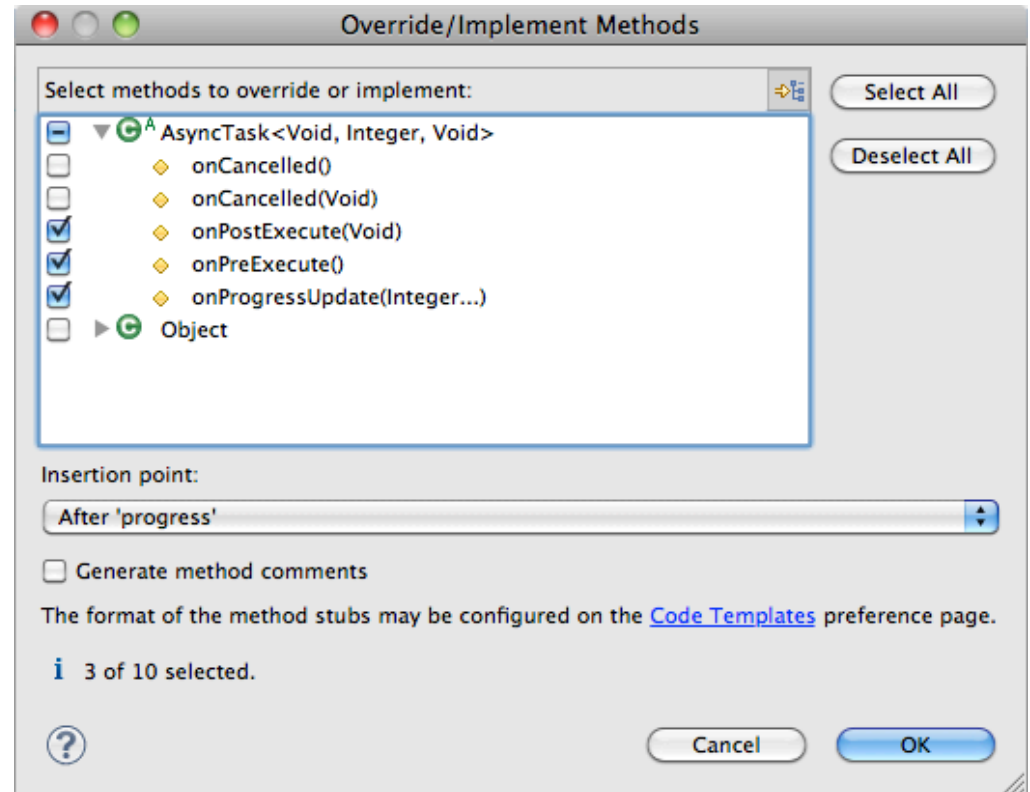
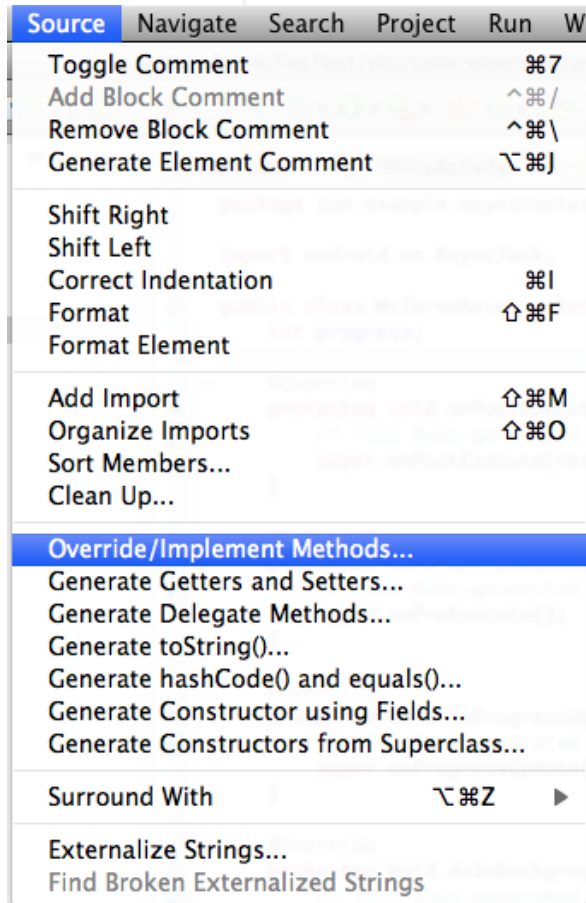
```
public class MiTareaAsinc extends AsyncTask<Void, Integer, Void>{  
  
}
```

- Definimos una subclase de AsyncTask que es una clase interna de la actividad principal; es decir, el siguiente código deberá ir dentro de las llaves que abren y cierran la definición de *MainActivity*.

4. Procesos en Hilos en Android



```
public class MiTareaAsinc extends AsyncTask<Void, Integer, Void>{
```





4. Procesos en Hilos en Android

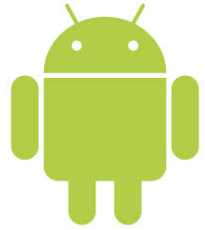
```
public class MiTareaAsinc extends AsyncTask<Void, Integer, Void>{
    int progress;

    @Override
    protected void onPostExecute(Void result) {
        // TODO Auto-generated method stub
        super.onPostExecute(result);
    }

    @Override
    protected void onPreExecute() {
        // TODO Auto-generated method stub
        super.onPreExecute();
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        // TODO Auto-generated method stub
        super.onProgressUpdate(values);
    }

    @Override
    protected Void doInBackground(Void... params) {
        // TODO Auto-generated method stub
        return null;
    }
}
```



4. Procesos en Hilos en Android

- Es necesario implementar los cuatro métodos
 - **onPreExecute()**. Definimos el progreso de la barra en 0 para empezar.
 - **doInBackground()**. En este método definimos la forma en la que estará avanzando la *ProgressBar*, para lo cual nos auxiliamos de la clase [*SystemClock*](#), y para cada vez que nuestra variable *progress* aumente se actualice su avance en el *UI thread*.
 - **onProgressUpdate()**. Actualiza la *ProgressBar*.
 - **onPostExecute()**. Una vez que el proceso haya terminado, el botón podrá ser presionado nuevamente después de cambiar su atributo clickable.

4. Pro

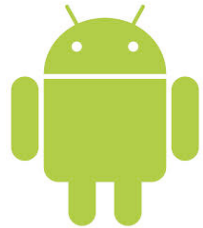
```
public class MiTareaAsinc extends AsyncTask<Void, Integer, Void>{
    int progreso;

    @Override
    protected void onPostExecute(Void result) {
        // TODO Auto-generated method stub
        boton.setClickable(true);
    }

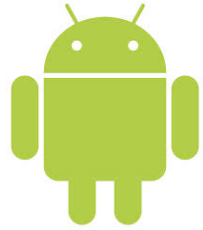
    @Override
    protected void onPreExecute() {
        // TODO Auto-generated method stub
        progreso=0;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        // TODO Auto-generated method stub
        progressBar.setProgress(values[0]);
    }

    @Override
    protected Void doInBackground(Void... params) {
        // TODO Auto-generated method stub
        while(progreso<100){
            progreso++;
            publishProgress(progreso);
            SystemClock.sleep(100);
        }
        return null;
    }
}
```



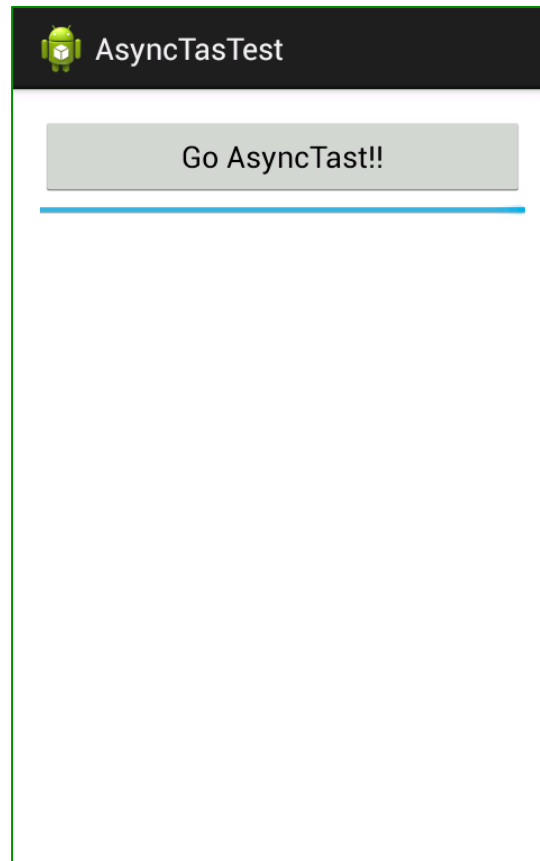
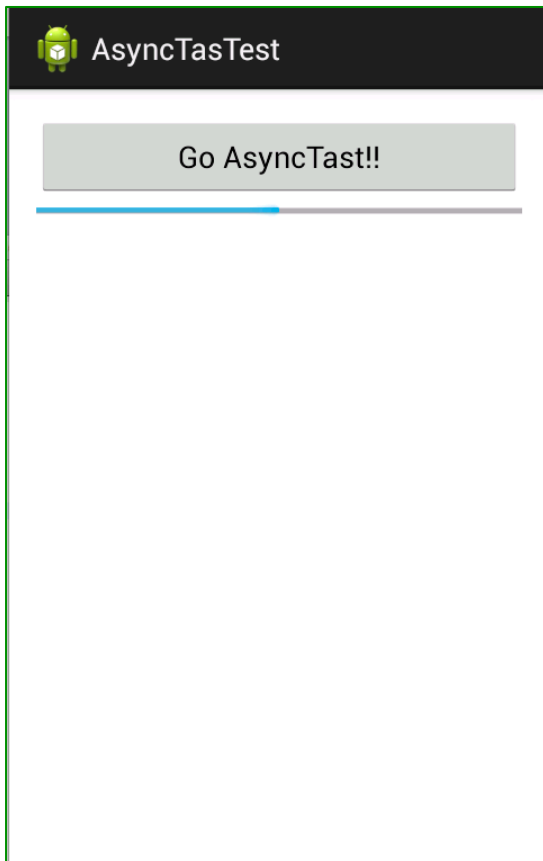
4. Procesos en Hilos en Android



```
boton.setOnClickListener(new OnClickListener() {  
  
    @Override  
    public void onClick(View v) {  
        // TODO Auto-generated method stub  
        boton.setClickable(false);  
        new MiTareaAsinc().execute();  
    }  
});
```

4. Procesos en Hilos en Android

Práctica: AsyncTask





4. Procesos en Hilos en Android

- Los Handlers implementan un mecanismo de paso de mensajes entre threads de forma que nos permiten enviar mensajes desde nuestro hilo al UIThread.
- Para ello se utiliza el método *sendMessage* de la clase Handler para avisar al UIThread por ejemplo cuando se tiene que hacer el repintado de la pantalla.



UD5. Ciclo de Vida e Hilos

1. Introducción
2. Ciclo de vida de una aplicación
3. Guardar y recuperar el estado de una actividad
4. Procesos en Hilos en Android
5. **Menús de Android**



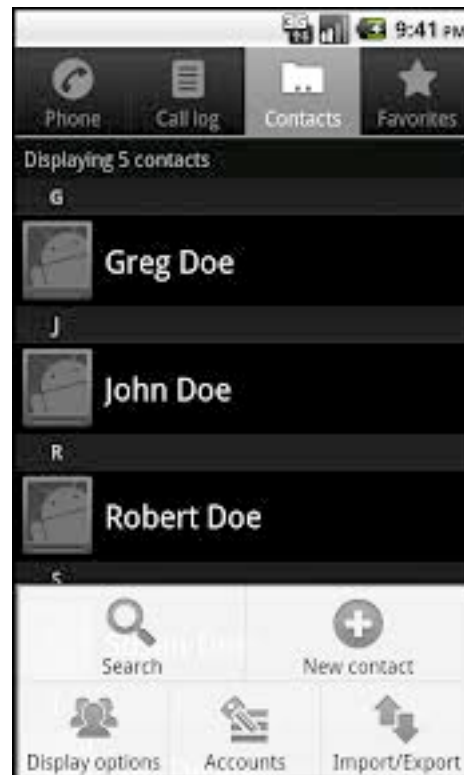
5. Menús en Android

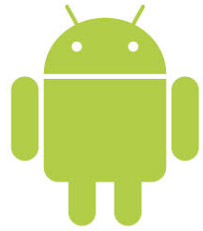
- En informática un Menú contiene una serie de opciones que el usuario puede elegir para realizar una determinada tarea.
- En las aplicaciones de Android podemos utilizar tres tipos de menús diferentes:
 - **Menús Principales:** son los usados con más frecuencia. Aparecen en la zona inferior de la pantalla al pulsar el botón Menú del teléfono.
 - **Submenús:** son los menús secundarios que aparecen al elegir una opción de un menú principal.
 - **Menús Contextuales:** son muy útiles y se muestran al realizar una pulsación larga sobre algún elemento de la pantalla. Es el equivalente al botón derecho del ratón en un PC.



5. Menús en Android

- Como es habitual en Android, existen dos formas de crear un menú en una aplicación Android:
 - definiendo el menú en un fichero XML e "inflándolo" después
 - creando el menú directamente mediante código Java.





5. Menús en Android

- Primero, crear un menú principal con un submenú a partir de su diseño en XML.
- Estos ficheros XML con el diseño del menú se deben guardar en la carpeta res\menu del proyecto y tienen una estructura de este tipo:

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">

  <item android:id="@+id/MenuOp1" android:title="Opción 1"
    android:icon="@drawable/menu_estrella"></item>
  <item android:id="@+id/MenuOp2" android:title="Opción 2"
    android:icon="@drawable/menu_brujula"></item>
  <item android:id="@+id/MenuOp3" android:title="Opción 3"
    android:icon="@drawable/menu_direccion">
    <menu>
      <item android:id="@+id/SubMenuOp1"
        android:title="Opción 3.1" />
      <item android:id="@+id/SubMenuOp2"
        android:title="Opción 3.2" />
    </menu>
  </item>
</menu>
```



5. Menús en Android

- Una vez definido el menú en el fichero XML, hay que implementar el método `onCreateOptionsMenu()` de la Actividad para que se cree en la pantalla.
- En este método debemos “inflar” el menú de forma parecida a como ya hemos hecho con otro tipo de componentes layouts.
- Primero obtenemos una referencia al objeto “inflador” mediante el método `getMenuInflater()` y, después, generamos la estructura del menú usando el método `inflate()` y pasándole como parámetro el ID del archivo XML de diseño del menú.
- Finalmente, el método debe devolver el valor `true` para indicar a la Actividad que debe mostrar el menú.

5. Menús en Android



```
@Override
public boolean onCreateOptionsMenu(Menu menu) {

    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_principal, menu);
    return true;
}
```



5. Menús en Android

- Para implementar el diseño del menú programándolo con sentencias Java redefinimos el método `onCreateOptionsMenu()` añadiendo las opciones del menú mediante el método `add()` del objeto `Menu`, que es un parámetro del primer método.
- Este método `add()` se invoca con cuatro parámetros:
 - ID del grupo asociado a la opción: veremos qué es esto en el siguiente ejemplo con un menú contextual, por lo que establecemos el valor `Menu.NONE`.
 - ID único para la opción: declaramos unas constantes de la clase.
 - Orden de la opción: como no queremos indicar ninguno, utilizamos `Menu.NONE`.
 - Texto de la opción: texto que aparece en el menú.

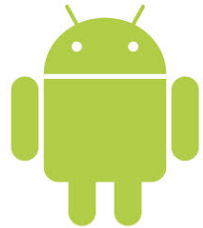
```
menu.add(Menu.NONE, MENU_OP1, Menu.NONE, "Opcion 1").setIcon(R.drawable.menu_estrella);  
menu.add(Menu.NONE, MENU_OP1, Menu.NONE, "Opcion 2").setIcon(R.drawable.menu_brujula);  
SubMenu submenu = menu.addSubMenu(Menu.NONE, MENU_OP1, Menu.NONE, "Opcion 3")  
                        .setIcon(R.drawable.menu_direccion);  
submenu.add(Menu.NONE, SMENU_OP1, Menu.NONE, "Opcion 3.1");  
submenu.add(Menu.NONE, SMENU_OP2, Menu.NONE, "Opcion 3.2");
```




5. Menús en Android

- Una vez construido el menú, es necesario implementar las sentencias que se ejecutan cuando el usuario selecciona una de las opciones,
- Para ello, usamos el evento **onOptionsItemSelected()** de la Actividad.
- Este evento recibe como parámetro el elemento de menú (MenuItem) que ha sido elegido por el usuario y cuyo ID podemos obtener con el método getItemId().
- En función de este ID podemos saber qué opción ha sido pulsada y ejecutar unas sentencias u otras.

5. Menús en Android



```
public boolean onOptionsItemSelected(MenuItem item) {  
    switch (item.getItemId()) {  
        case R.id.MenuOp1:  
            labelResultado.setText("Has pulsado la opcion 1");  
            return true;  
        case R.id.MenuOp2:  
            labelResultado.setText("Has pulsado la opcion 2");  
            return true;  
        case R.id.MenuOp3:  
            labelResultado.setText("Has pulsado la opcion 3");  
            return true;  
        case R.id.SubMenuOp1:  
            labelResultado.setText("Has pulsado la opcion 3.1");  
            return true;  
        case R.id.SubMenuOp2:  
            labelResultado.setText("Has pulsado la opcion 3.2");  
            return true;  
        default:  
            return super.onOptionsItemSelected(item);  
    }  
} // end onOptionsItemSelected
```



5. Menús en Android

- Los menús contextuales siempre están asociados a un componente en concreto de la pantalla y se muestra cuando el usuario lo pulsa un rato.
- Normalmente, se suele mostrar opciones específicas para el elemento pulsado.
- Por ejemplo, en un componente de tipo lista podríamos tener un menú contextual que aparezca al pulsar sobre un elemento en concreto de la lista y que permita editar su texto o eliminarlo de la lista.





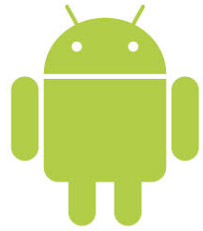
5. Menús en Android

- Lo primero es crear un ListView con elementos sobre los que pulsar y mostrar opciones de edición.

```
private String[] datos = new String[]{"Opcion 1 listado", "Opcion 2 listado",  
                                     "Opcion 3 listado", "Opcion 4 listado", "Opcion 5 listado"};
```

- Lo siguiente que debemos hacer es indicar en el método onCreate() de la Actividad que el listado tienen asociado un menú contextual usando la función **registerForContextMenu()**:

```
adaptador = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, datos);  
listadoPrincipal.setAdapter(adaptador);  
registerForContextMenu(listadoPrincipal);
```



5. Menús en Android

- Para crear las opciones disponibles con el método `onCreateOptionsMenu()`, vamos a construir los menús contextuales asociados a los diferentes componentes de la aplicación con el método `onCreateContextMenu()`.
- A diferencia del método `onCreateOptionsMenu()` Android invoca este método cada vez que es necesario mostrar un menú contextual.
- Este método lo implementaremos de misma forma que los menús básicos, inflándolo con un archivo de diseño XML o creándolo con sentencias Java.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/EditTextOp" android:title="Editar texto opción"></item>
    <item android:id="@+id/ReiniciaTextOp" android:title="Reiniciar texto opción"></item>
</menu>
```



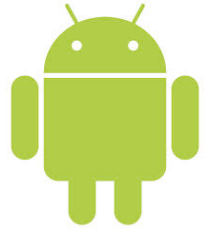
5. Menús en Android

- Si definimos varios menús contextuales en la misma Actividad, hay que tenerlo presente para implementar el método `onCreateContextMenu()`.

```
public void onCreateContextMenu(ContextMenu menu, View v,  
    ContextMenuInfo menuInfo)  
{  
    super.onCreateContextMenu(menu, v, menuInfo);
```

- Para hacerlo, obtenemos el ID del componente al que se va a ir asociado el menú contextual, que se recibe en el parámetro (View v) del método `onCreateContextMenu()` utilizando el método `getId()` de dicho parámetro:

```
MenuInflater inflater = getMenuInflater();  
if(v.getId() == R.id.labelResultado)  
    inflater.inflate(R.menu.menu_context_etiqueta, menu);  
  
else if(v.getId() == R.id.ListadoPrincipal)  
{  
    inflater.inflate(R.menu.menu_context_lista, menu);  
}
```



5. Menús en Android

- Para personalizar el título del menú contextual utilizamos el método `setHeaderTitle()`.

```
AdapterView.AdapterContextMenuInfo info =  
    (AdapterView.AdapterContextMenuInfo)menuInfo;  
menu.setHeaderTitle(  
    listadoPrincipal.getAdapter().getItem(info.position).toString());  
inflater.inflate(R.menu.menu_context_lista, menu);
```

- Para hacer esto es necesario conocer la posición del elemento seleccionado en el listado mediante el último parámetro `menuInfo`. Este parámetro contiene información adicional del componente sobre el que el usuario ha pulsado para mostrar el menú contextual. En este caso en particular el componente `ListView` contiene la posición del elemento pulsado. Para obtenerlo, hacemos un cambio de formato (typecasting) del parámetro `menuInfo` a un objeto del tipo `AdapterContextMenuInfo` y accedemos a su propiedad `position`.



5. Menús en Android

- Por último, para implementar las acciones que hay que ejecutar cuando el usuario selecciona una opción determinada del menú contextual vamos a implementar el método `onContextItemSelected()` de manera similar a cómo hacíamos con `onOptionsItemSelected()` para los menús básicos.

5. Menús en Android

Práctica:
Crear un menu

