

Programación Multimedia y Dispositivos Móviles

UD 6. Dialogs, tareas asíncronas y notificaciones

Javier Carrasco

Curso 2021 / 2022



Este obra está bajo una licencia de Creative Commons Reconocimiento 4.0 Internacional.
Última actualización: septiembre de 2021.

Versión impresa en Amazon: <https://www.amazon.es/dp/B08NM4XV4Q>

Dialogs, tareas asíncronas y notificaciones

6. Dialogs, tareas asíncronas y notificaciones.....	3
6.1. <i>Dialogs</i>	3
6.2. Utilizando <i>DialogFragment</i>	3
6.3. Utilizando <i>AlertDialog</i>	5
Añadir una lista al diálogo.....	7
Diálogos con listas persistentes de opción simple y múltiple.....	8
Cuadros de diálogo personalizados.....	11
6.4. Time y Date Picker.....	13
6.5. Progress Bar.....	15
6.6. Hilos.....	16
6.7. Tareas asíncronas.....	18
6.8. <i>Corrutinas</i>	22
6.9. Un ejemplo útil. Descargar imágenes.....	25
6.10. Notificaciones.....	29
6.11. Uso del vibrador.....	33

6. Dialogs, tareas asíncronas y notificaciones

En el capítulo anterior se pudieron ver unas pinceladas acerca del uso de los `AlertDialog`. En este capítulo se tratará de ampliar el manejo de los cuadros de diálogo en Android, así como la creación de tareas asíncronas y el uso de las notificaciones.

6.1. Dialogs

Un *Dialog* es un ventana de tamaño reducido con la que se indicará al usuario que debe tomar una decisión o, introducir algún tipo de información. Los cuadros de diálogo, *dialogs*, son de tamaño reducido, sin ocupar toda la pantalla y que, generalmente, se necesitarán para terminar de llevar a cabo una acción.

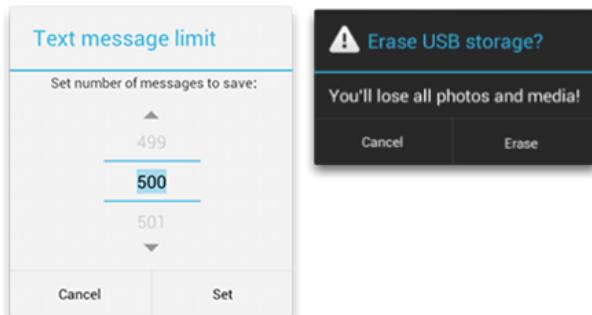


Figura 1

Android dispone de la clase `Dialog` para este objetivo, pero no se utiliza directamente, sino que se hace uso de las clases `AlertDialog` (de la que ya has visto algo) y `DatePickerDialog` o `TimePickerDialog`. Estas clases definen la estructura y el diseño de los diálogos.

También se puede hacer uso de la clase `DialogFragment` para gestionar los controles de un diálogo y tratar su estilo. El uso de `DialogFragment` garantiza trabajar correctamente con el ciclo de vida de la aplicación, como cuando se pulsa el botón Atrás.

6.2. Utilizando *DialogFragment*

Empieza creando un `DialogFragment` para construir dentro un `AlertDialog` básico. Viendo este procedimiento podrás pasar a crear diálogos más complejos. Se partirá de un nuevo proyecto con API mínima 21, *My Dialogs*, para los ejemplos que se verán a continuación.

Crea la clase `MyDialogFragment.kt` en *Kotlin* en el nuevo proyecto. Aquí se creará un *dialog* basado en `DialogFragment`. A continuación puedes ver el código que contendrá la clase `MyDialogFragment.kt` creada. Recuerda que para crear una nueva clase en Android Studio puedes utilizar la opción *File > New > Kotlin File/Class*.

4 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
1 class MyDialogFragment : DialogFragment() {
2     // Se crea la estructura del diálogo.
3     override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
4         return activity?.let {
5             val builder = AlertDialog.Builder(it)
6
7                 builder.setMessage(R.string.my_first_dialog)
8                     .setPositiveButton(android.R.string.yes)
9                 ) { dialog, which ->
10                     // Acciones si se pulsa SÍ.
11                     Log.d("DEBUG", "Acciones si SÍ.")
12                     Toast.makeText(
13                         it,
14                         "Acciones si SÍ",
15                         Toast.LENGTH_SHORT
16                     ).show()
17                 }
18                 .setNegativeButton(android.R.string.no)
19                 ) { dialog, which ->
20                     // Acciones si se pulsa NO.
21                     Log.d("DEBUG", "Acciones si NO.")
22                     Toast.makeText(
23                         it,
24                         "Acciones si NO",
25                         Toast.LENGTH_SHORT
26                     ).show()
27                 }
28             builder.create()
29         } ?: throw IllegalStateException("La Activity no puede ser nula")
30     }
31 }
```

En primer lugar, fíjate en el uso de una función de alcance, o *scope*, de Kotlin, concretamente `let`. En este ejemplo se utiliza para evaluar la creación del `DialogFragment`, si la creación es correcta se devuelve el objeto para poder mostrarlo, en caso contrario se devuelve el error.

Se define la variable `builder` para construir el `dialog`. Muy básico, se asigna el mensaje (`.setMessage`), un botón para la acción positiva (`.setPositiveButton()`) y otro para la acción negativa (`.setNegativeButton()`). Fíjate que se hace uso de los *resources*, tanto de la aplicación como de Android. Por último, se muestra el diálogo.

```
1 with(binding){
2     btnWithDialogFragment.setOnClickListener {
3         val myDialogFragment = MyDialogFragment()
4         myDialogFragment.show(fragmentManager!!, "teGusta")
5     }
6 }
```

Es probable que encuentres `supportFragmentManager` como primer parámetro del método `show()`, pero este no se recomienda utilizarlo a partir de la API 14.

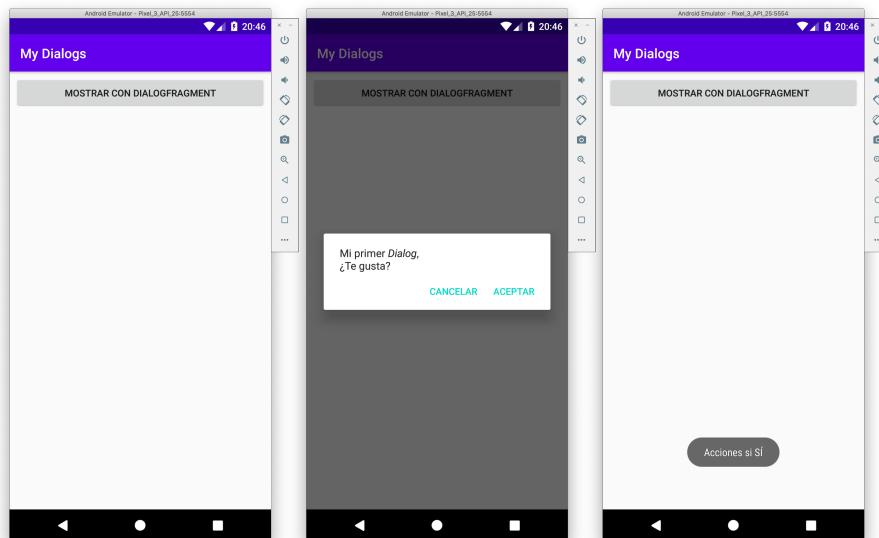


Figura 2

Al mostrar el *dialog* se usa el método `show()`, el segundo argumento, "teGusta", es una etiqueta única que el sistema podrá guardar y con la que restaurar el estado del *fragment* cuando sea necesario.

Cuando se utiliza este sistema, es la clase (`MyDialogFragment`) la que debe resolver las acciones para las respuestas dadas por el usuario.



Cuando se utiliza la clase `DialogFragment` para mostrar un diálogo desde un *fragment*, se puede detener, o paralizar brevemente, la ejecución y esperar a que el usuario responda. Esto puede ser útil para realizar operaciones después de la respuesta, sin importar lo que haya respondido el usuario. Para ello, bastará con utilizar el siguiente método antes de `show()`.

```
1 myDialogFragment.retainInstance = true
```

6.3. Utilizando AlertDialog

Es posible utilizar `AlertDialog` directamente sin necesidad de crear una clase. Este método (visto en el capítulo anterior), permite preguntar al usuario y recoger directamente desde la clase que hace la llamada la respuesta.

La estructura de los diálogos de alerta es la siguiente:

- **Título:** es opcional, y suele utilizarse cuando se proporciona un mensaje detallado, se utiliza una lista o se usa un diseño personalizado. Suele omitirse cuando es una simple pregunta.

6 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

- **Mensaje:** se muestra la información, la lista o el diseño personalizado.
- **Botones:** zona inferior, mostrará los botones, nunca más de tres. Existen tres tipos botones diferentes que se pueden agregar:
 - **Positivo:** puedes usar este botón para aceptar y continuar con la acción (la acción "Aceptar").
 - **Negativo:** este botón se utilizará para cancelar la acción.
 - **Neutral:** este botón se usa cuando el usuario no quiere continuar con la acción, pero no necesariamente quiere cancelar. Aparece entre los botones positivo y negativo. Por ejemplo, la acción podría ser "Recordarme más tarde".

A continuación se verá como crear un cuadro de diálogo de alerta de manera directa. Para este ejemplo se crea un nuevo método que se encargue de mostrar el cuadro de alerta. Añade el siguiente código dentro del `with(binding)` anterior.

```
1 btnAlertDialog.setOnClickListener {  
2     myAlertDialog(  
3         "Este es el segundo cuadro de diálogo, " +  
4         "se utiliza la clase AlertDialog para mostrarlo."  
5     )  
6 }
```

A continuación se muestra el código del método `myAlertDialog`.

```
1 private fun myAlertDialog(message: String) {  
2     val builder = AlertDialog.Builder(this)  
3     // Se crea el AlertDialog.  
4     builder.apply {  
5         // Se asigna un título.  
6         setTitle("My AlertDialog!!")  
7         // Se asigna el cuerpo del mensaje.  
8         setMessage(message)  
9         // Se define el comportamiento de los botones.  
10        setPositiveButton(  
11            android.R.string.ok,  
12            DialogInterface.OnClickListener(function = actionBarButton)  
13        )  
14        setNegativeButton(android.R.string.no) { _, _ ->  
15            Toast.makeText(  
16                context,  
17                android.R.string.no,  
18                Toast.LENGTH_SHORT  
19            ).show()  
20            binding.root.setBackgroundColor(Color.RED)  
21        }  
22        setNeutralButton("No sé") { _, _ ->  
23            Toast.makeText(  
24                context,
```

```

25         "No sé",
26         Toast.LENGTH_SHORT
27     ).show()
28     binding.root.setBackgroundColor(Color.WHITE)
29 }
30 }
31 // Se muestra el AlertDialog.
32 builder.show()
33 }

```

En este ejemplo se observa que los botones negativo y neutral están estructurados igual. Los guiones bajos (_) representan las variables *dialog* y *which*, como en este caso no se utilizan, en Kotlin se pueden sustituir por el guión bajo.

En el caso del botón positivo, se ha optado por crear una variable que se tratará como una función que ejecuta un bloque de código, esto puede ser buena idea si se trata de un bloque de instrucciones grande.

```

1 private val actionBar = { dialog: DialogInterface, which: Int ->
2     Toast.makeText(
3         this,
4         android.R.string.ok,
5         Toast.LENGTH_SHORT
6     ).show()
7     binding.root.setBackgroundColor(Color.GREEN)
8 }

```

También se puede crear un método al que puedes llamar directamente como se muestra a continuación. Este método te permitirá conseguir un código más limpio y fácil de entender.

```

1 setPositiveButton(android.R.string.ok) { _, _ ->
2     actionBar()
3 }

```



Figura 3

Añadir una lista al diálogo

Es posible que, en ocasiones, interese mostrar una lista al mostrar un *AlertDialog*, para eso se deberá modificar ligeramente la creación del cuadro de diálogo.

```

1 private fun myAlertDialogList() {
2     val builder = AlertDialog.Builder(this)
3     val namesArray = resources.getStringArray(R.array.array_nombres)
4
5     builder.apply {
6         setTitle("My AlertDialog con lista")
7         setItems(R.array.array_nombres) { _, which ->
8             Toast.makeText(

```

8 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
9             context,
10            namesArray[which],
11            Toast.LENGTH_LONG
12        ).show()
13    }
14
15    builder.show()
16
17 }
```

Como la lista aparece en la zona del mensaje, esta opción no se utiliza. Debes fijarte que en este caso, si se utiliza la variable `which`, ésta devolverá el índice de la posición pinchada en la lista mostrada.

También es importante saber que, al pulsar un elemento de la lista, se descarta el diálogo, no es una lista persistente, por lo que el cuadro de diálogo se cerrará.

Diálogos con listas persistentes de opción simple y múltiple

En el siguiente ejemplo se creará un cuadro de diálogo con una lista de opción simple (las que utilizan *radio button*). La diferencia con el ejemplo anterior es el método `setSingleChoiceItems()`, donde se indicará la lista a mostrar, el elemento que aparecerá seleccionado, en este caso -1 (ninguno), y a continuación las acciones que se pueden hacer cuando se seleccione un ítem, en este caso se escribirá una línea en el *log*.

Al tratarse de un cuadro de diálogo con una lista persistente, es necesario añadir un botón para recoger, o confirmar, la selección. En este ejemplo se puede ver un botón para aceptar la selección y otro para cancelar.

Para terminar, se deberá recoger la opción seleccionada por el usuario, para ello se utilizará la instrucción `val selectedPosition = (dialog as AlertDialog).listView.checkedItemPosition` y así poder tratarla después según convenga.

```
1 private fun myAlertDialogSinglePersistentList(names: Array<String>) {
2     val builder = AlertDialog.Builder(this)
3
4     builder.apply {
5         setTitle("My AlertDialog con lista simple")
6         setSingleChoiceItems(R.array.array_nombres, -1) { _, which ->
7             Log.d("DEBUG", names[which])
8         }
9
10        setPositiveButton(android.R.string.yes) { dialog, _ ->
```

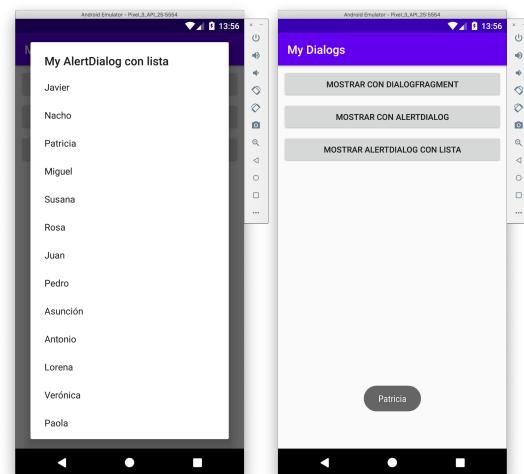


Figura 4

```

11     val selectedPosition = (dialog as AlertDialog)
12         .listView.checkedItemPosition
13     Toast.makeText(
14         context,
15         names[selectedPosition],
16         Toast.LENGTH_SHORT
17     ).show()
18 }
19
20     setNegativeButton(android.R.string.no) { _, _ ->
21         Toast.makeText(
22             context,
23             android.R.string.no,
24             Toast.LENGTH_SHORT
25         ).show()
26     }
27 }
28 builder.show()
29 }
```

Fíjate que para este ejemplo se ha modificado la llamada, pasándose la lista de nombres por parámetro al método encargado de crear el cuadro de diálogo.

```

1 val namesArray: Array<String> = resources.getStringArray(R.array.array_nombres)
2
3 with(binding) {
4     ...
5     btnAlertDialogSinglePersistentList.setOnClickListener {
6         myAlertDialogSinglePersistentList(namesArray)
7     }
8 }
```

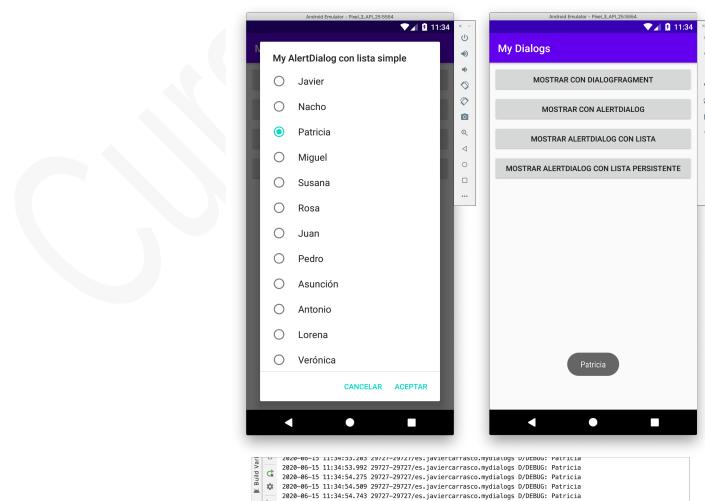


Figura 5

10 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

A continuación se verá como crear un cuadro de diálogo con una lista de opción múltiple (las que utilizan *checkbox*). Para este caso se utilizará el método `setMultiChoiceItems()`.

```
1  private fun myAlertDialogMultiPersistentList(names: Array<String>) {
2      val builder = AlertDialog.Builder(this)
3      val selectedItems = ArrayList<Int>()
4
5      builder.apply {
6          setTitle("My AlertDialog con lista multiple")
7          setMultiChoiceItems(R.array.array_nombres, null){ _, which, isChecked ->
8              if (isChecked) {
9                  selectedItems.add(which)
10                 Log.d("DEBUG", "Checked: " + names[which])
11             } else if (selectedItems.contains(which)) {
12                 selectedItems.remove(which)
13                 Log.d("DEBUG", "UnChecked: " + names[which])
14             }
15         }
16
17         setPositiveButton(android.R.string.yes){ _, _ ->
18             var textToShow = "Checked: "
19             if (selectedItems.size > 0) {
20                 for (item in selectedItems) {
21                     textToShow = textToShow + names[item] + " "
22                 }
23             } else textToShow = "No items checked!"
24             Toast.makeText(
25                 context,
26                 textToShow,
27                 Toast.LENGTH_SHORT
28             ).show()
29         }
30
31
32         setNegativeButton(android.R.string.no){ _, _ ->
33             Toast.makeText(
34                 context,
35                 android.R.string.no,
36                 Toast.LENGTH_SHORT
37             ).show()
38         }
39     }
40
41     builder.show()
42 }
```

En este caso, se necesita utilizar el botón positivo para confirmar la selección. Fíjate en la variable `selectedItems`, ésta será un `ArrayList` en el que se almacenarán los elementos seleccionados de la lista.

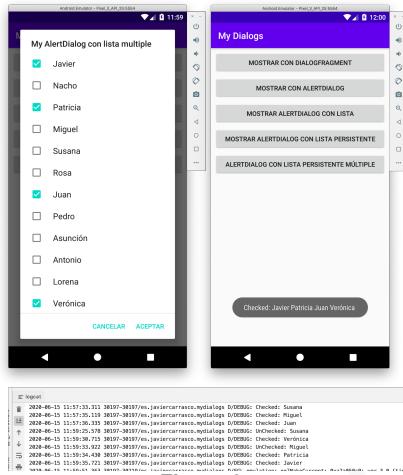


Figura 6

Cuadros de diálogo personalizados

En ocasiones, puede resultar útil mostrar un cuadro de diálogo con un diseño personalizado, que se adapte a las necesidades, para ello, se utilizará el método `setView()` para asignar el *layout* personalizado que se quiera mostrar.

En primer, se creará un diseño personalizado, para lo cual, se abrirá un nuevo *layout* llamado `dialog_layout.xml` con un contenido similar al siguiente.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3   xmlns:tools="http://schemas.android.com/tools"
4   android:layout_width="wrap_content"
5   android:layout_height="wrap_content"
6   android:orientation="vertical">
7
8   <ImageView
9     android:layout_width="match_parent"
10    android:layout_height="64dp"
11    android:background="#FFFFBB33"
12    android:contentDescription="@string/app_name"
13    android:scaleType="center"
14    android:src="@drawable/ic_launcher_foreground" />
15
16   <EditText
17     android:id="@+id/username"
18     android:layout_width="match_parent"
19     android:layout_height="wrap_content"
20     android:layout_marginLeft="4dp"
21     android:layout_marginTop="16dp"
```

12 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
22    android:layout_marginRight="4dp"
23    android:layout_marginBottom="4dp"
24    android:autofillHints=""
25    android:hint="@string/userName"
26    android:inputType="textEmailAddress"
27    tools:targetApi="o" />
28
29    <EditText
30        android:id="@+id/password"
31        android:layout_width="match_parent"
32        android:layout_height="wrap_content"
33        android:layout_marginLeft="4dp"
34        android:layout_marginTop="4dp"
35        android:layout_marginRight="4dp"
36        android:layout_marginBottom="16dp"
37        android:autofillHints=""
38        android:fontFamily="sans-serif"
39        android:hint="@string/password"
40        android:inputType="textPassword"
41        tools:targetApi="o" />
42    </LinearLayout>
```

A continuación, deberás escribir el siguiente código Kotlin que se encargará de crear el cuadro de diálogo personalizado haciendo uso del *layout* creado.

```
1  private fun myCustomAlertDialog() {
2      val builder = AlertDialog.Builder(this)
3
4      builder.apply {
5          val inflater = layoutInflater
6         .setView(inflater.inflate(R.layout.dialog_layout, null))
7
8          setPositiveButton(android.R.string.ok) { dialog, _ ->
9              // En este caso, se accede a los elementos del layout
10             // haciendo uso de Synthetic Binding.
11             val name = (dialog as AlertDialog).username.text
12             val pass = dialog.password.text
13             Toast.makeText(
14                 context,
15                 "User: $name\nPass: $pass",
16                 Toast.LENGTH_SHORT
17             ).show()
18         }
19         setNegativeButton(android.R.string.no) { dialog, _ ->
20             Toast.makeText(
21                 context,
22                 android.R.string.no,
23                 Toast.LENGTH_SHORT
24             ).show()
25             dialog.dismiss()
26     }
```

```

27     }
28     builder.show()
29 }
```

En primer lugar, se "inflará" la vista que aparecerá en el cuadro de diálogo utilizando la variable `inflater`. Al utilizar el método `setView()`, se añade la vista como primer argumento y, como segundo, se utiliza `null` para indicar que no existe elemento raíz en la vista.

Debes tener en cuenta como recoger los datos introducidos en el cuadro de diálogo. Esto se hace con las variables `name` y `pass`. Se hará uso de la variable `dialog`, utilizando un *casting* en la primera asignación. Como curiosidad, una vez hecho el cast, ya no es necesario volver a repetirlo dentro del mismo scope.

También se ha utilizado el método `dismiss()` en el botón negativo, se utiliza al cerrar el cuadro de diálogo y generar un evento, no es obligatorio. Este método se utiliza para recoger el evento mediante el uso del método `onDismiss()` y realizar más operaciones tras cerrar el diálogo, para ello sería necesario extender la clase con `DialogInterface.OnDismissListener`.

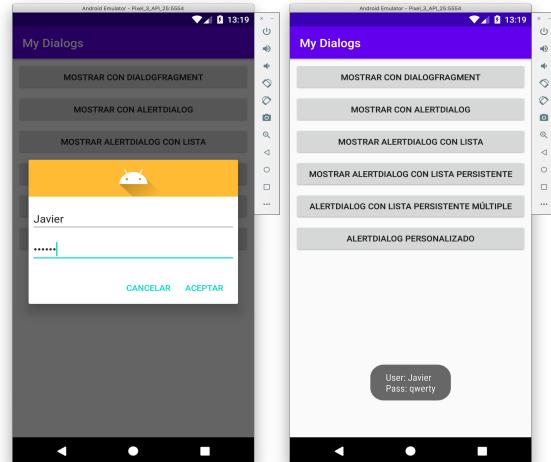


Figura 7

6.4. Time y Date Picker

Un **Time Picker** permite seleccionar una hora de manera fácil. El siguiente ejemplo muestra como lanzar un `TimePickerDialog` y asignar la hora seleccionada a un `TextView`.

```

1  btnTimePicker.setOnClickListener {
2      val cal = Calendar.getInstance()
3      val timeSetListerner = TimePickerDialog.OnTimeSetListener { _, hour, minute ->
4          cal.set(Calendar.HOUR_OF_DAY, hour)
5          cal.set(Calendar.MINUTE, minute)
6
7          tvTimePicker.text = SimpleDateFormat("HH:mm").format(cal.time)
8      }
9
10 // Al estar dentro del with(binding), se debe especificar el contexto
11 // con this@MainActivity.
12 TimePickerDialog(this@MainActivity,
13                 timeSetListerner,
14                 cal.get(Calendar.HOUR_OF_DAY),
15                 cal.get(Calendar.MINUTE),
16                 true
```

14 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
17     ).show()  
18 }
```

En primer lugar, debe crearse un *listener* para el *Time Picker*, éste permitirá recoger los valores unas vez cerrado el cuadro diálogo. Se utilizan los argumentos `hour` y `minute` para asignarlos a la variable `cal` y seguidamente asignarlo al *TextView*.

Por último, se instancia el *Time Picker Dialog*, los parámetros que utilizados son, el contexto, la hora y el minuto con los valores que contendrán al abrirse, además se indica a `true` el formato 24 horas.

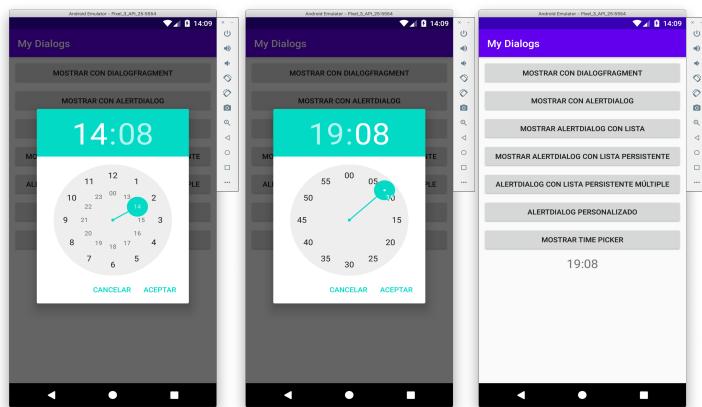


Figura 8

Un *Date Picker* funciona exactamente igual que el *Time Picker*. A continuación, puedes ver el código para poder crearlo y asignar el valor seleccionado por el usuario a un *TextView*.

```
1 btnDatePicker.setOnClickListener {  
2     val cal = Calendar.getInstance()  
3     val dateSetListener = DatePickerDialog.OnDateSetListener { _, i, i2, i3 ->  
4         cal.set(Calendar.YEAR, i)  
5         // ENERO - 0, FEBRERO - 1, ..., DICIEMBRE - 11  
6         cal.set(Calendar.MONTH, i2)  
7         cal.set(Calendar.DAY_OF_MONTH, i3)  
8  
9         tvDatePicker.text = "${cal.get(Calendar.DAY_OF_MONTH)}" +  
10            "/${cal.get(Calendar.MONTH)}" +  
11            "/${cal.get(Calendar.YEAR)}"  
12     }  
13  
14     // Dentro de with(binding) se especifica el contexto con this@MainActivity.  
15     DatePickerDialog(  
16         this@MainActivity,  
17         dateSetListener,  
18         cal.get(Calendar.YEAR),  
19         cal.get(Calendar.MONTH),  
20         cal.get(Calendar.DAY_OF_MONTH)  
21     ).show()  
22 }
```

Si te fijas en el código, verás que los argumentos creados en el *listener* se corresponden con el año, el mes y día, en ese orden. El resto, funciona exactamente igual que en *Time Picker* visto anteriormente, pero fíjate que los meses a través de la clase `Calendar` comienzan por cero, observa la discrepancia en la figura.

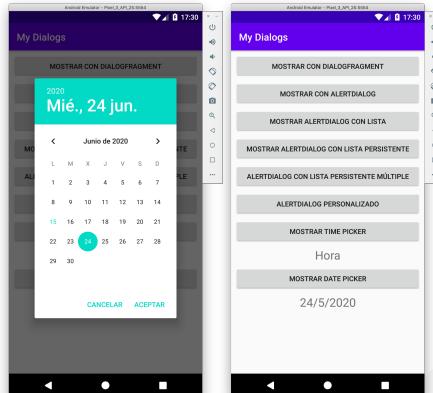


Figura 9

6.5. Progress Bar

Las *Progress Bar* son un elemento visual que permite mostrar al usuario el estado de una tarea en ejecución. Añade un componente *Progress Bar* a la actividad principal, *activity_main.xml*, junto con un botón para iniciar una tarea.

```

1 <ProgressBar
2     android:id="@+id/progressBar"
3     style="?android:attr/progressBarStyleHorizontal"
4     android:layout_width="0dp"
5     android:layout_height="wrap_content"
6     android:layout_marginStart="8dp"
7     android:layout_marginTop="16dp"
8     android:layout_marginEnd="8dp"
9     android:progress="0"
10    app:layout_constraintEnd_toEndOf="parent"
11    app:layout_constraintStart_toStartOf="parent"
12    app:layout_constraintTop_toBottomOf="@+id/tvDatePicker" />
```

El siguiente código muestra el uso de *Progress Bar*, además se introducirán los conceptos de hilos y tareas asíncronas.

La idea es que, al pulsar el botón "*Iniciar Progress Bar*", se creará una tarea ficticia mediante un hilo, que se encargará de actualizar la barra de progreso, mientras, se podrá utilizar el resto de la aplicación.

Una finalizada la tarea, será el propio *Thread* el encargado de avisar de su finalización mediante un *Toast*.

```

1 btnStartProgressBar.setOnClickListener {
2     var progressBarStatus = 0
3     var seccion = 0
```

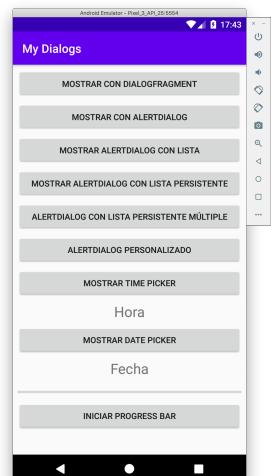


Figura 10

16 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
4     progressBar.progress = 0
5
6     // Se inicia el Thread.
7     // Se crea un hilo ficticio imitando una tarea.
8     Thread(Runnable {
9         while (progressBarStatus < 100) {
10             // Se actualiza el estado del Progress Bar.
11             progressBarStatus = seccion
12             progressBar.progress = progressBarStatus
13
14             // Operación que se realizará.
15             try {
16                 seccion += 10
17                 Thread.sleep(1000)
18             } catch (e: InterruptedException) {
19                 e.printStackTrace()
20             }
21         }
22
23         // Acciones que se realizarán al finalizar la tarea.
24         this@MainActivity.runOnUiThread {
25             Toast.makeText(
26                 this@MainActivity,
27                 "Tarea finalizada!!!",
28                 Toast.LENGTH_SHORT
29             ).show()
30         }
31     }).start()
32 }
```

6.6. Hilos

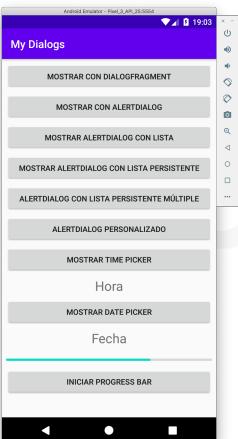


Figura 11

Los hilos son ejecuciones que se asocian a una aplicación. Están ligados a la programación concurrente, dando al usuario la sensación de multitarea. Los hilos son indicados para tareas con una media, o alta, tasa de datos y tareas con elevado consumo de CPU.

Pero, debe tenerse en cuenta que los hilos comparten el mismo espacio de memoria, la modificación de un dato por parte de un hilo, permitirá el acceso al dato modificado por otros hilos.

Según la documentación de Android, se recomienda no acceder directamente a objetos del hilo de UI desde los hilos creados manualmente. Avisan que pueden producirse anomalías debido a la ausencia de sincronización.



Figura 12

Observa el código, en un proyecto nuevo llamado *MyThreads*, en la actividad principal añade cuatro botones con una etiqueta a la derecha de cada uno de ellos.

A continuación, se creará un método que se encargue de crear un hilo. Cada uno de estos botones llamará al método para iniciar un hilo, mostrando en su *TextView* correspondiente el estado de la ejecución.

```

1 // Método encargado de crear un hilo.
2 private fun crearHilo(
3     boton: Button, duracion: Long,
4     vueltas: Int, visualizador: TextView
5 ) {
6     visualizador.text = "0"
7     visualizador.setBackgroundColor(Color.TRANSPARENT)
8     boton.isEnabled = false
9     Thread(Runnable {
10         // Se imita la realización de una tarea.
11         var contador = 0
12         while (contador < vueltas) {
13             try {
14                 contador++
15                 Thread.sleep(duracion)
16             } catch (e: InterruptedException) {
17                 e.printStackTrace()
18             }
19
20             // Permite actuar con elementos de la UI.
21             visualizador.post {
22                 visualizador.text = contador.toString()
23             }
24         }
25         // Acciones que se realizarán al finalizar la tarea.
26         runOnUiThread {
27             visualizador.text = "FIN"
28             visualizador.setBackgroundColor(Color.GREEN)
29             boton.isEnabled = true
30             Toast.makeText(
31                 this,
32                 boton.text,
33                 Toast.LENGTH_SHORT
34             ).show()
35         }
36     }).start()
37 }
```

Las llamadas podrían ser como se muestran a continuación, cada una de ellas variando la duración de la tarea.

```

1 with(binding) {
2     btnHiloUno.setOnClickListener {
3         crearHilo(btnHiloUno, 500, 10, tvHiloUno)
```

```

4
5     btnHiloDos.setOnClickListener {
6         crearHilo(btnHiloDos, 100, 20, tvHiloDos)
7     }
8     btnHiloTres.setOnClickListener {
9         crearHilo(btnHiloTres, 200, 30, tvHiloTres)
10    }
11    btnHiloCuatro.setOnClickListener {
12        crearHilo(btnHiloCuatro, 300, 40, tvHiloCuatro)
13    }
14 }

```

Puedes observar el resultado final de este código en la imagen de la figura, junto a los botones se mostrará el contador con la evolución de la tarea reproducida en cada *thread*.

6.7. Tareas asíncronas

Las tareas asíncronas liberan al programador de la creación de hilos, la sincronización entre ellos y la presentación de los resultados en el hilo principal. Estas tareas utilizan como sistema de procesamiento el paralelismo.

Este tipo de tareas son ideales para operaciones con poca carga, como tareas de red y de disco, además de tareas en las que el resultado deberá mostrarse al usuario en la actividad.

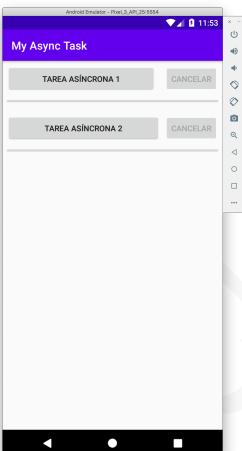


Figura 14

Para comenzara, en la actividad principal se añadirán cuatro botones, dos para lanzar las tareas y dos para cancelarlas, y dos barras de progreso para poder ver la evolución de las tareas.



Figura 13

Durante un tiempo, desde la aparición de Kotlin, han coexistido diferentes métodos. Uno de ellos ha sido el uso de la librería **Anko**¹, creada y mantenida por **JetBrains**. Esta librería ha sido un complemento ideal, ya que facilitaba mucho la creación de tareas asíncronas, dejándolo en un mero trámite.

Pero, a pesar de la facilidad que ofrecía Anko, fue *deprecated* en diciembre de 2019, aún así, puedes encontrar numerosos ejemplos que hacen uso de ella. Anko es reemplazado por la librería **Android KTX**² desarrollada por Google y forma parte de **Android JetPack**³.

En el proyecto que se muestra a continuación, podrá verse como crear tareas asíncronas haciendo uso de las instrucciones básicas que ofrece Android. Se utilizará para ello la clase **AsyncTask**⁴, pero cuidado, queda en desuso a partir de la API 30, se verá como solucionar esto más adelante.

1 Anko (<https://github.com/Kotlin/anko>)

2 Android KTX (<https://developer.android.com/kotlin/ktx>)

3 Android JetPack (<https://developer.android.com/jetpack>)

4 AsyncTask (<https://developer.android.com/reference/android/os/AsyncTask>)

A continuación, se creará una clase anidada dentro de la clase principal. Para poder interactuar con elementos de la *activity*, como el caso de la barra de progreso, se deberán pasar como parámetros los componentes que interesen, además del contexto.

```
1  /**
2   * Clase privada anidada para crear tareas asíncronas.
3   */
4  private inner class MyAsyncTask(
5      val contexto: Context,
6      val progressBar: ProgressBar,
7      val botonStart: Button,
8      val botonCancelar: Button) : AsyncTask<Int, Int, Int> {
9
10 /**
11  * Acciones antes de iniciar la tarea, se utiliza
12 * para inicializar o configurar la tarea.
13 */
14 override fun onPreExecute() {
15     super.onPreExecute()
16
17     Log.d(botonStart.text.toString(), "${botonStart.text}, iniciada!!")
18     progressBar.progress = 0
19     botonStart.isEnabled = false
20     botonCancelar.isEnabled = true
21 }
22
23 /**
24 * Este método realiza la tarea a ejecutar, vararg
25 * será un array con los parámetros indicados.
26 */
27 override fun doInBackground(vararg params: Int?): Int {
28     if (params.size == 2) {
29         var contador = 0
30         while (contador < params[0]!!) {
31             try {
32                 contador++
33                 Thread.sleep(params[1]!!.toLong())
34             } catch (e: Exception) {
35                 Log.println(
36                     Log.WARN,
37                     "doInBackground",
38                     e.message.toString()
39                 )
40             }
41         }
42         // Comprobamos si la tarea ha sido cancelada.
43         if (!isCancelled)
44             publishProgress(
45                 (((contador + 1) * 100 / params[0]!!).toFloat()).toInt()
46             )
47         else break
48     }
49 }
```

20 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
47         }
48         return 1
49     } else return -1
50 }
51
52 /**
53 * Método que se ejecutará al cancelarse la tarea.
54 */
55 override fun onCancelled(result: Int?) {
56     super.onCancelled(result)
57
58     progressBar.progress = 0
59     Log.d(botonStart.text.toString(), "${botonStart.text}, cancelada!!!")
60     Toast.makeText(
61         contexto,
62         "${botonStart.text}, cancelada!!!",
63         Toast.LENGTH_SHORT
64     ).show()
65
66     botonStart.isEnabled = true
67     botonCancelar.isEnabled = false
68 }
69 /**
70 * Permite mostrar información al usuario, se ejecuta
71 * cuando se utiliza el método publishProgress() desde el
72 * método doInBackground().
73 */
74 override fun onProgressUpdate(vararg values: Int?) {
75     super.onProgressUpdate(values[0])
76     progressBar.progress = values[0]!!
77 }
78 /**
79 * Se ejecuta al finalizar el método doInBackground()
80 * y se le pasa el resultado obtenido.
81 */
82 override fun onPostExecute(result: Int?) {
83     super.onPostExecute(result)
84     if (result == 1) {
85         progressBar.progress = 100
86         Toast.makeText(
87             contexto,
88             botonStart.text,
89             Toast.LENGTH_SHORT
90         ).show()
91     }
92
93     botonStart.isEnabled = true
94     botonCancelar.isEnabled = false
95 }
96 }
```

Evidentemente, no es necesario conocer todos los métodos de la clase `AsyncTask`, una vez declarada la clase, basta con utilizar la opción *Override Methods* del menú *Code*, o el botón derecho sobre el código y seleccionar *Generate*, tras lo cual deberás seleccionar los métodos que se deseen implementar.

Si te fijas en `AsyncTask<Int, Int, Int>`, para este caso, los parámetros de tipo utilizados son de tipo enteros, estos se podrán adaptar a nuestras necesidades. Cada uno de ellos representan la siguiente información `<Params, Progress, Result>`. Observa como quedaría el método `onCreate()` para poder iniciar las tareas.

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2     super.onCreate(savedInstanceState)
3
4     val binding = ActivityMainBinding.inflate(layoutInflater)
5     setContentView(binding.root)
6
7     // Se deshabilitan los botones cancelar hasta que se activen
8     // las tareas.
9     binding.btnCancelarUno.isEnabled = false
10    binding.btnCancelarDos.isEnabled = false
11
12    // Variables para crear las tareas asíncronas.
13    lateinit var myAsyncTask1: MyAsyncTask
14    lateinit var myAsyncTask2: MyAsyncTask
15
16    binding.btnAsyncTaskUno.setOnClickListener {
17        myAsyncTask1 = MyAsyncTask(
18            this,
19            binding.progressBarUno,
20            binding.btnAsyncTaskUno,
21            binding.btnCancelarUno
22        )
23        // Se lanza la tarea.
24        myAsyncTask1.execute(100, 20)
25    }
26
27    binding.btnCancelarUno.setOnClickListener {
28        myAsyncTask1.cancel(true)
29    }
30
31    binding.btnAsyncTaskDos.setOnClickListener {
32        myAsyncTask2 = MyAsyncTask(
33            this,
34            binding.progressBarDos,
35            binding.btnAsyncTaskDos,
36            binding.btnCancelarDos
37        )
38        myAsyncTask2.execute(200, 15)
39    }
40}
```

```

41     binding.btnCancelarDos.setOnClickListener {
42         myAsyncTask2.cancel(true)
43     }
44 }
```

Si ejecutas este código, verás que no se produce la ejecución en paralelo, primero se ejecutará la primera tarea iniciada y una vez acabada, comenzará la segunda. Esto es debido a que hasta la API 11 no se podía realizar ejecuciones en paralelo. Para poder hacerlo, se deberá realizar la llamada de ejecución comprobando la API utilizada.

```

1 if (android.os.Build.VERSION.SDK_INT >=
2     android.os.Build.VERSION_CODES.HONEYCOMB)
3     myAsyncTask1.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, 100, 20)
4 else myAsyncTask1.execute(100, 20)
```

En este caso, se sabe que la versión de la API es la 11, por lo que podrías sustituir el código `android.os.Build.VERSION_CODES.HONEYCOMB` por el número 11.

6.8. Corrutinas

Como se ha comentado al inicio de este punto, `AsyncTask` ha quedado en desuso a partir de la API 30 y, junto con la aparición de *Android KTX*, en la versión 1.3 de *Kotlin*, se añade lo que se conoce como *corrutinas*⁵⁶.

Las *corrutinas* son patrones de diseño que van a permitir simplificar el código que se quiera ejecutar de manera asíncrona. El objetivo principal de las *corrutinas* en Android es solucionar dos problemas:

- Evitar el bloqueo del proceso principal, impidiendo el parón de la aplicación, administrando más eficientemente las tareas largas.
- Aumentar la seguridad del subprocesso principal, siendo más seguras las operaciones de disco o de red.

Para poder utilizar *corrutinas* en un proyecto Android, se deberá añadir la siguiente dependencia al `build.gradle (Module: app)`.

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.5.0-native-mt'
```

Existen varias formas de crear *corrutinas*, este es un campo amplio, pero dada la naturaleza de este libro, no se puede profundizar mucho en ellas, pero sí tratar la que posiblemente sea la forma más utilizada y fácil de comprender de todas ellas, el *builder launch*.

Para hacer uso de las *corrutinas* es necesario conocer que éstas se basan en las llamadas **funciones de suspensión**. Estas funciones, que únicamente pueden utilizarse dentro de una *corrutina*, pueden detener la ejecución de la *corrutina* y devolverle el control tras finalizar su tarea, convirtiendo las *corrutinas* en ese lugar seguro en que no se bloqueará el hilo principal de la aplicación.

5 *Coroutines* (<https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html>)

6 Cómo mejorar el rendimiento de la app con las corrutinas de Kotlin (<https://developer.android.com/kotlin/coroutines>)

Para crear una función de suspensión bastará con indicarlo haciendo uso de la palabra reservada `suspend`.

```

1  /**
2   * Función de suspensión que detiene la ejecución de la corrutina
3   * hasta que finaliza.
4   */
5  suspend fun task(duracion: Long): Boolean {
6      Log.d("SUSPEND FUN", "Simulando una tarea!")
7      delay(duracion)
8      return true
9 }
```

Las *corrutinas* se ejecutan bajo un contexto, generalmente el principal, este contexto define las reglas de ejecución. Estas reglas son un *Map*, por tanto funcionan como clave-valor, y la que resulta más interesante de conocer es *dispatcher*, que permite identificar el hilo en el que se ejecutará. Debes tener en cuenta que un hilo puede ejecutar múltiples *corrutinas*, esto significa que el sistema tratará de optimizar al máximo los hilos existentes, no creará nuevos hilos para ejecutar las *corrutinas*.

Volviendo a los *dispatchers*, estos serán diferentes tipos de contextos que especificarán el tipo de hilo que se encargará de ejecutar la tarea. A continuación se describen los cuatro tipos de *dispatchers* existentes.

- **Default:** se utiliza si no se especifica lo contrario, es para tareas que hacen un uso intensivo de la CPU, cálculos de la aplicación, por ejemplo.
- **IO:** corresponde a las operaciones de entrada/salida, son tareas que bloquean el hilo hasta recibir una respuesta, como un acceso a base de datos, lectura de un fichero, peticiones a un servidor, etc.
- **Unconfined:** este se utilizará cuando el hilo en el que se ejecute no es importante.
- **Main:** este utiliza *dispatcher* utiliza librerías relacionadas con la interfaz de usuario, hace uso del hilo de UI.

Como ya se ha comentado, se utilizará el *builder launch* para la creación de *corrutinas*, éste necesita de un scope para funcionar, de momento se utilizará el *GlobalScope*, éste se utiliza para lanzar *corrutinas* de nivel superior que trabajan durante toda la vida útil de la aplicación y no se cancelan prematuramente.

Debes saber que este *builder* devolverá un objeto `Job` que contiene dos funciones que debes conocer.

- **join():** bloqueará la *corrutina* asociada hasta que todos sus hijos finalicen. Debe utilizarse dentro de una *corrutina*, ya que se trata de una función de suspensión.
- **cancel():** cancela la ejecución de los hijos de una *corrutina*.

Fíjate en el código que viene a continuación para comprender la creación y funcionamiento de las *corrutinas*.

24 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
1  /**
2   * Método encargado de crear una corriputina simulando una tarea.
3   */
4  private fun makeTask(
5      duracion: Int, btnStart: Button,
6      btnCancel: Button, progressBar: ProgressBar
7  ) = GlobalScope.launch(Dispatchers.Main) {
8      // Preparación de la corriputina.
9      btnStart.isEnabled = false
10     btnCancel.isEnabled = true
11     progressBar.progress = 0
12
13    withContext(Dispatchers.IO) { // Tarea principal.
14        var contador = 0
15        while (contador < duracion) {
16            if(task((duracion * 50).toLong())) {
17                contador++
18                progressBar.progress = (contador * 100) / duracion
19            }
20        }
21    }
22
23    // Finaliza la corriputina.
24    btnStart.isEnabled = true
25    btnCancel.isEnabled = false
26    progressBar.progress = 0
27
28    Toast.makeText(
29        this@MainActivity,
30        "${btnStart.text} finalizada!!",
31        Toast.LENGTH_SHORT
32    ).show()
33 }
```

Observa ahora la llamada que deberá hacerse para crear la *corriputina* desde un botón en la actividad principal de la aplicación, también puedes ver como se cancelaría la tarea.

```
1 // Las corriputinas devuelven un objeto Job al crearse.
2 var task1: Job? = null
3
4 binding.btnCorriputineUno.setOnClickListener {
5     task1 = makeTask(
6         10,
7         binding.btnCorriputineUno,
8         binding.btnCancelarUno,
9         binding.progressBarUno
10    )
11 }
12
13 binding.btnCancelarUno.setOnClickListener {
14     // LET ejecutará el contenido si task1 existe y/o no es nulo.
```

```

15     task1?.let {
16         task1?.cancel().apply {
17             Toast.makeText(
18                 this@MainActivity,
19                 getString(R.string.btnCoroutineUno) + " cancelada!!",
20                 Toast.LENGTH_SHORT
21             ).show()
22         }
23
24         binding.progressBarUno.progress = 0
25         binding.btnCancelarUno.isEnabled = false
26         binding.btnCoroutineUno.isEnabled = true
27     }
28 }
```

En la figura mostrada puedes ver una imagen de como podría quedar el ejemplo de *corrutinas* mostrado en este punto.

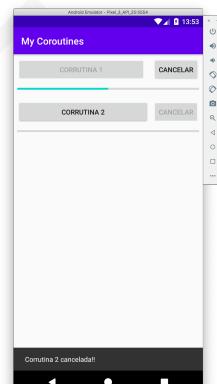


Figura 15

Ejercicios propuestos

6.1. Crea una aplicación **Android+Kotlin** llamada **My Async Tasks**, con API mínima 21. Esta aplicación deberá tener en su UI un botón, seis barras de progreso y una etiqueta para cada barra que la identifique (*hilo1*, *hilo2*, *asyncTask1*, *asyncTask2*, *coroutine1* y *coroutine2*).

Al pulsar el botón, deberán lanzarse seis tareas a la vez, 2 utilizando *threads*, 2 utilizando *AsyncTask* y 2 utilizando *corrutinas*. Sigue el esquema de los ejemplos y completa las barras de progreso según se ejecuten las tareas. Identifica el método más lento.

6.9. Un ejemplo útil. Descargar imágenes

En este último punto se verá un ejemplo con algo de utilidad, se creará una aplicación que permita descargar una serie de imágenes desde Internet y mostrarlas en pantalla, además, se realizará una acción conocida como "*inflado*". Crea un nuevo proyecto e indica en el fichero *AndroidManifest.xml* el permiso de uso de Internet.

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

Para el ejemplo, la actividad principal de la aplicación contendrá el siguiente código XML para formar la UI.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     xmlns:app="http://schemas.android.com/apk/res-auto"
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
```

26 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
7     android:layout_height="match_parent"
8     tools:context=".MainActivity">
9
10    <ScrollView
11        android:layout_width="match_parent"
12        android:layout_height="match_parent"
13        android:layout_marginStart="8dp"
14        android:layout_marginEnd="8dp"
15        app:layout_constraintBottom_toBottomOf="parent"
16        app:layout_constraintEnd_toEndOf="parent"
17        app:layout_constraintStart_toStartOf="parent"
18        app:layout_constraintTop_toTopOf="parent">
19
20        <LinearLayout
21            android:id="@+id/myLinearLayout"
22            android:layout_width="match_parent"
23            android:layout_height="wrap_content"
24            android:orientation="vertical">
25
26            <Button
27                android:id="@+id/button"
28                android:layout_width="match_parent"
29                android:layout_height="wrap_content"
30                android:text="@string/btn_imageDownloader" />
31
32            <ProgressBar
33                android:id="@+id/progressBar"
34                style="?android:attr/progressBarStyleHorizontal"
35                android:layout_width="match_parent"
36                android:layout_height="wrap_content" />
37
38            <TextView
39                android:id="@+id/tv_info"
40                android:layout_width="match_parent"
41                android:layout_height="wrap_content"
42                android:layout_marginBottom="10dp"
43                android:textAlignment="center"
44                tools:text="información" />
45        </LinearLayout>
46    </ScrollView>
47 </androidx.constraintlayout.widget.ConstraintLayout>
```

Fíjate que únicamente se utilizará un botón que cambiará de estado al pulsarse, una barra de progreso para mostrar el estado de la tarea, en este caso será descargar imágenes y un *TextView* que mostrará información sobre la tarea.

Observa que no hay ningún *ImageView* en la actividad, esto es porque se “inflarán” mediante código dentro del *LinearLayout*.

En primer lugar se creará una *mutableListOf* con las direcciones de las imágenes que se quieren descargar.

```

1 // Contiene las URLs de las imágenes a descargar.
2 private val urlImagenes = mutableListOf(
3     "https://live.staticflickr.com/7801/47562604261_4ff8522918_k.jpg",
4     "https://live.staticflickr.com/8345/28338120516_40e0c40f65_k.jpg",
5     "https://live.staticflickr.com/7449/28346482855_3d1b745518_k.jpg",
6     "https://live.staticflickr.com/3856/14648036959_72973ac0d3_k.jpg",
7     "https://live.staticflickr.com/5587/14648212427_b3bf236e4c_k.jpg"
8 )

```

El método `onCreate()` quedará como se muestra a continuación, se encargará de cambiar el estado del botón, limpiar el `LinearLayout` y crear y cancelar la tarea.

```

1 override fun onCreate(savedInstanceState: Bundle?) {
2     super.onCreate(savedInstanceState)
3     binding = ActivityMainBinding.inflate(layoutInflater)
4     setContentView(binding.root)
5
6     var tarea: Job? = null
7
8     binding.button.setOnClickListener {
9         if (binding.button.text == getString(R.string.btn_imageDownloader)) {
10             // Se comprueba la existencia de ImageViews, si existen se eliminan.
11             if (binding.myLinearLayout.childCount > 3) {
12                 binding.myLinearLayout.removeViews(
13                     3,
14                     binding.myLinearLayout.childCount - 3
15                 )
16             }
17             tarea = descargarImagenes()
18         } else {
19             tarea?.let {
20                 tarea?.cancel()
21                 binding.button.text = getString(R.string.btn_imageDownloader)
22                 binding.tvInfo.text = getString(R.string.txt_descargaCancelada)
23             }
24         }
25     }
26 }

```

A continuación, se creará el método `descargarImagenes()`, éste será el encargado de crear la corriente que se encargará de descargar las imágenes e “inflar” el `LinearLayout` que contendrá las imágenes.

```

1 private fun descargarImagenes() = GlobalScope.launch(Dispatchers.Main) {
2     binding.button.text = getString(android.R.string.cancel)
3     binding.tvInfo.text = getString(R.string.txt_descargando)
4
5     binding.progressBar.progress = 0
6
7     val imagenes = ArrayList<Bitmap>()
8

```

28 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
9 urlImagenes.forEach {
10     Log.d("URLs", it)
11
12     // Tarea asíncrona, se descargan las imágenes y se almacenan en un
13     // ArrayList<Bitmap>().
14     withContext(Dispatchers.IO) {
15         try {
16             val inputStream = URL(it).openStream()
17             imagenes.add(BitmapFactory.decodeStream(inputStream))
18         } catch (e: Exception) {
19             Log.e("DOWNLOAD", e.message.toString())
20         }
21     }
22     binding.progressBar.progress = (imagenes.size * 100) / urlImagenes.size
23 }
24
25 // Se añaden las imágenes a la vista una vez descargadas.
26 imagenes.forEach {
27     addImagen(it)
28 }
29
30 // Fin de la tarea.
31 binding.button.text = getString(R.string.btn_imageDownloader)
32
33 binding.tvInfo.text = getString(
34     R.string.txt_descargaCompleta,
35     imagenes.size
36 )
37 }
```

Fíjate como se cambia de contexto, utilizando el *Dispatcher* a **IO** para realizar la operación de descarga, las imágenes se almacenarán en un *ArrayList*. Por último, la función `addImage()` se encargará de realizar el “*inflado*”.

```
1 fun addImagen(image: Bitmap) {
2     val img = ImageView(this)
3
4     // Se carga la imagen en el ImageView mediante Glide y se ajusta el tamaño.
5     Glide.with(this)
6         .load(image)
7         .override(binding.myLinearLayout.width - 100)
8         .into(img)
9     img.setPadding(0, 0, 0, 10)
10
11    // Se infla el LinearLayout con una imagen nueva.
12    binding.myLinearLayout.addView(img)
13 }
```

En la siguiente figura puedes ver el resultado del código que se ha descrito sobre estas líneas.

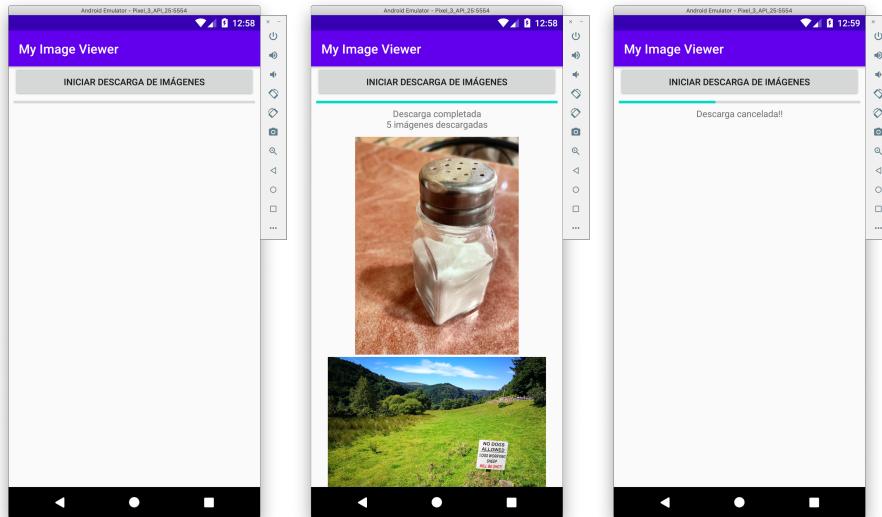


Figura 16

Ejercicios propuestos

6.2. Modifica el ejemplo visto para la descarga de imágenes, de forma que, cuando se cancele la tarea, se indique el número de imágenes que ha tenido tiempo de descargar y las muestre en el *LinearLayout*.

6.10. Notificaciones

Las notificaciones son un elemento de Android permiten mostrar información, en la barra de estado del dispositivo y de una manera breve al usuario. A continuación, se verá como crear notificaciones básicas, utilizando para ello características que pertenecen a la API 4 y superiores, concretamente, se hará hincapié en el cambio sufrido a partir de la API 26 con la aparición de los canales de notificación.

Se empezará creando un proyecto nuevo, que tendrá un único botón para comenzar, el cual, cada vez que se pulse creará una notificación que aparecerá en la barra de estado. Evidentemente, este no es el cometido de las notificaciones, ya que se utilizan más para notificar tareas acabas, mensajes recibidos, etc. Pero para poder ver como se crean y su funcionamiento servirá.

Ya en código, en el método `onCreate()` añade el siguiente código, éste utiliza el *builder* de la clase `NotificationCompat` para crear la notificación.

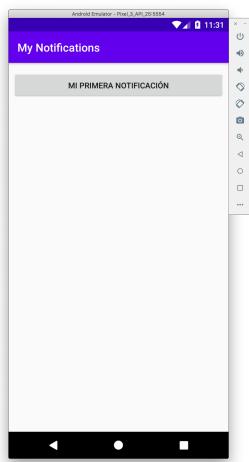


Figura 17

30 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
1 // Se crea una notificación utilizando el builder de NotificationCompat.  
2 val builder = NotificationCompat.Builder(this, CHANNELID)  
3 builder.apply {  
4     setSmallIcon(R.mipmap.ic_launcher_round)  
5     setContentTitle("Mi primera notificación")  
6     setContentText("Esta será la primera notificación creada.")  
7     priority = NotificationCompat.PRIORITY_DEFAULT  
8 }
```

Al instanciar la variable `builder` con la notificación, se le pasa el contexto y la variable `CHANNELID`, la cual está definida como una propiedad de clase principal con el nombre del paquete de la aplicación.

```
1 private val CHANNELID = "es.javiercarrasco.mynotifications"
```

El **canal** es algo que debe utilizarse siempre que el objetivo de la aplicación sea de la API 26 (Android 8.0) en adelante, a partir de esta versión, el sistema permite que el usuario tenga el control de las notificaciones de la aplicación, algo que se verá a continuación.

Volviendo a la variable `builder`, la creación es bastante intuitiva, se asigna un ícono a la notificación, un título y un contenido o descripción de la notificación. Por último, la prioridad que tendrá, `PRIORITY_DEFAULT`.

Fíjate ahora, como se lanzaría la notificación creada, en este caso, asociada a la pulsación sobre el botón creado.

```
1 binding.btnPrimeraNotificacion.setOnClickListener {  
2     with(NotificationManagerCompat.from(this)) {  
3         notify(notificationId, builder.build())  
4     }  
5 }
```

La variable `notificationId` contendrá el número que identifique a la notificación lanzada, este deberá ser diferente si las notificaciones son distintas, por ejemplo, un código para notificar una descarga, otra para recibir un mensaje, etc.

Para este ejemplo, el código utilizado es un número al azar, como el que se muestra a continuación.

```
1 private val notificationId = 123456
```

Pero recuerda que, si tu aplicación lanza más de una notificación diferente, o la misma varias veces, estas deberán tener distintos identificadores para poder tratarlas según la acción que se haga sobre ellas.

El resultado de este ejemplo lo puedes ver en la figura, si reproduces el código, verás que, por mucho que pulses el botón, únicamente aparecerá una notificación, esto es debido al identificador.

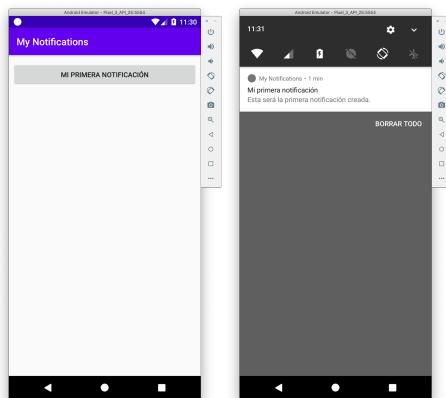


Figura 18

Ya puedes lanzar una notificación, pero si la pruebas, verás que al hacer clic sobre ella no pasa nada, eso es porque hay que decirle a la notificación qué hacer, que será indicarle que *intent* debe abrir al pulsar sobre ella. Crea una nueva actividad llamada `RequestNotification`. El *layout* únicamente contendrá un `TextView` con el texto "*Has recibido una notificación*".

Ahora, debes decirle a la notificación, que al crearse, tendrá un *intent* pendiente, esto hará que al pulsar sobre la notificación se abra la *activity* que se le indique. Estas líneas van justo antes de crear la variable `builder`.

```

1 // Se crea el intent que deberá abrirse al pulsarse la notificación.
2 val intent = Intent(this, RequestNotification::class.java).apply {
3     flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
4 }
5
6 val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0, intent, 0)

```

A continuación, se añaden las siguientes dos líneas en el *apply* de la creación del `builder` para indicar la nueva funcionalidad.

```

1 setContentIntent(pendingIntent)
2 setAutoCancel(true)

```

Se utiliza la propiedad `setContentIntent()` para especificar el *intent* que debe abrir y, la propiedad `setAutoCancel()` a *true* para que se cierre la notificación una vez pulsada.

Debes saber que este *intent* está fuera del ciclo de vida de la aplicación, por lo que si utilizas la propiedad *parent* en el *manifest* no funcionará. Tras pulsar sobre la notificación, verás como se abre la nueva actividad.

Volviendo a los **canales**⁸, como ya se ha comentado anteriormente, estos aparecieron con **Android 8.0. Oreo** (API 26). Básicamente permiten distinguir entre diferentes tipos de notificaciones dentro de una misma aplicación. De esta manera se dejará a elección del usuario que notificaciones desea recibir, aunque dependerá principalmente de la activación o no por parte del desarrollador.

Para los siguientes casos se utilizará un emulador con **API 28** para poder ver los canales en la gestión de notificaciones. Para hacer funcionar el ejemplo anterior, deberá registrarse al menos un canal de notificaciones para la aplicación, de lo contrario, no sería posible mostrar notificaciones, y cada vez que se intentase mostrar una, en el *log* podría verse el aviso de canal no registrado para la aplicación.

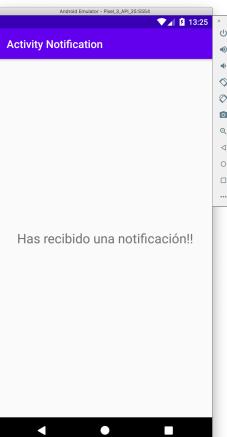


Figura 19

Se modifican las variables `CHANNELID` y `notificationId` por `CHANNELID1` y `notificationId1`, en previsión de próximas modificaciones. También se añade el método `createNotificationChannel()` para poder crear canales.

8 Crear canales en Android (<https://developer.android.com/training/notify-user/channels#CreateChannel>)

32 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
1 private fun createNotificationChannel(  
2     channel: String, name: Int,  
3     desc: Int, importance: Int  
4 ) {  
5     // Se crea el canal de notificación únicamente para API 26+.  
6     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
7         val name = getString(name)  
8         val descriptionText = getString(desc)  
9         val channel = NotificationChannel(channel, name, importance).apply {  
10             description = descriptionText  
11         }  
12         // Se registra el canal en el sistema.  
13         val notificationManager: NotificationManager =  
14             getSystemService(Context.NOTIFICATION_SERVICE)  
15             as NotificationManager  
16         notificationManager.createNotificationChannel(channel)  
17     }  
18 }
```

Justo antes de crear la notificación, se deberá registrar el canal en el sistema operativo. Esto puede hacerse nada más iniciar la aplicación. El resto de código quedará exactamente igual.

```
1 // Se registra el canal de notificación en el sistema.  
2 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {  
3     createNotificationChannel(  
4         CHANNELID1,  
5         R.string.txt_channel1_name,  
6         R.string.txt_channel1_desc,  
7         NotificationManager.IMPORTANCE_DEFAULT  
8     )  
9 }
```

Las siguientes imágenes muestran el canal de notificaciones de la aplicación, que sería lo que se tendría por defecto si no se crean más canales.

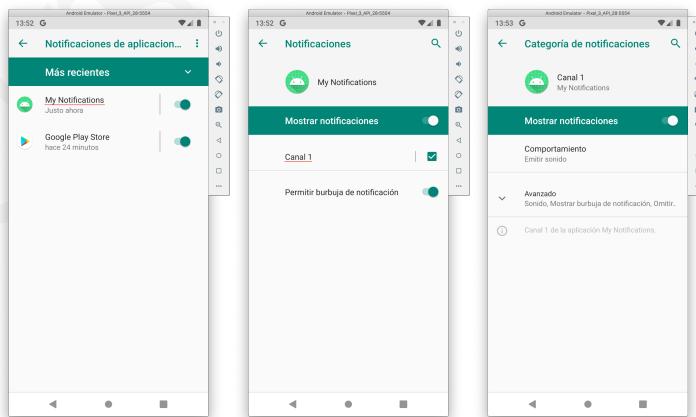


Figura 20

Ejercicios propuestos

6.3. Añade un segundo botón al ejemplo visto, la aplicación deberá registrar un segundo canal y mostrar una nueva notificación que sobre el nuevo canal.

6.4. Crea una nueva versión del ejemplo de **descarga de imágenes**, añade una notificación cuando finalice la descarga y otra diferente cuando se cancele la operación.

6.11. Uso del vibrador

Cuando se recibe una notificación, por lo general, se suele hacer vibrar el dispositivo. Para poder hacer uso del vibrador, al tratarse de un recurso físico del dispositivo, debe pedirse permiso indicándolo en el *manifest* del proyecto.

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

Para probar el vibrador se utilizará el proyecto creado para las notificaciones, la idea es hacer vibrar el dispositivo cuando llega una notificación.

El siguiente paso será añadir el código necesario para hacer vibrar el dispositivo. Evidentemente, esto no ocurrirá si se prueba en el emulador, pero notarlo deberás utilizar un dispositivo físico. Realmente no vibrará al entrar la notificación, ya que eso dependerá del sistema operativo, si no que lo hará en el momento de lanzar al pulsar el botón. Se creará el siguiente método para gestionar el uso de la vibración.

```
1 private fun vibrar() {
2     // Se instancia el objeto Vibrator.
3     val vibrador = getSystemService(VIBRATOR_SERVICE) as Vibrator
4
5     // Se comprueba la existencia de vibrador.
6     if (!vibrador.hasVibrator()) {
7         Toast.makeText(
8             applicationContext,
9             "No tienes vibrador!!",
10            Toast.LENGTH_SHORT
11        ).show()
12    } else {
13        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
14            val vibrationEffect =
15                VibrationEffect.createOneShot(
16                    1000,
17                    VibrationEffect.DEFAULT_AMPLITUDE)
18            vibrador.vibrate(vibrationEffect)
19        } else {
20            vibrador.vibrate(1000) // Deprecated API 26
21            Toast.makeText(
22                applicationContext,
23                "Deprecated version",
```

34 UNIDAD 6 DIALOGS, TAREAS ASÍNCRONAS Y NOTIFICACIONES

```
24         Toast.LENGTH_SHORT  
25     ).show()  
26 }  
27 }  
28 }
```

Fíjate que no es necesario pedir permiso al usuario para poder utilizar la vibración del dispositivo, pero si debe controlarse la versión para utilizar el método `vibrate()`, ya que desde la API 26 está *deprecated* en según la instancia que se utilice.

Ya solo faltaría añadir la llamada al método `vibrar()` desde el *listener* del botón encargado de crear la notificación.

```
1 binding.btnPrimeraNotificacion.setOnClickListener {  
2     with(NotificationManagerCompat.from(this)) {  
3         notify(notificationId1, builder.build())  
4         vibrar()  
5     }  
6 }
```

Si en algún momento necesitas cancelar la vibración, basta con utilizar el método `cancel()` del objeto `Vibrator` creado, por ejemplo, al salir de la aplicación.

```
1 override fun onDestroy() {  
2     super.onDestroy()  
3     vibrador.cancel()  
4 }
```