Programación Multimedia y Dispositivos Móviles

UD 9. Cuadros de diálogo

Javier Carrasco

Curso 2024 / 2025

Cuadros de diálogo

9. Cuadros de diálogo

Este capítulo tratará de poner en conocimiento el uso de la clase `AlertDialog` para el manejo de los cuadros de diálogo en Android. Estos diálogos los puedes utilizar para notificaciones al usuario, confirmaciones y/o elección de datos.

9.1. Dialogs

Un *Dialog* es una 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, no suelen ocupar toda la pantalla y que, generalmente, se necesitarán para terminar de llevar a cabo una acción.

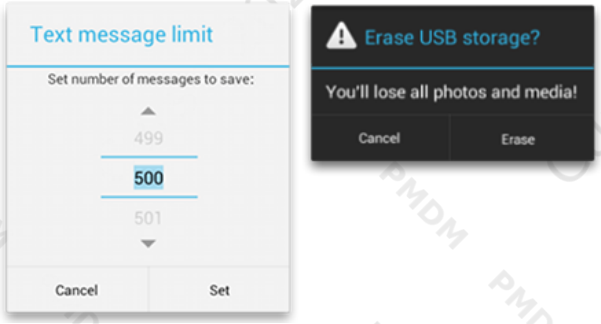

Figura 1

Android dispone de la clase `Dialog` para este fin, pero no se utiliza directamente, sino que se hace uso de las clases `AlertDialog`, `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*.

Se partirá de un nuevo proyecto con API mínima 24, *My Dialogs*, para los ejemplos que se verán a continuación.

9.2. Utilizando AlertDialog

Haciendo uso de la clase `AlertDialog` se puede preguntar al usuario y recoger la respuesta directamente desde el propio diálogo. La estructura de los cuadros de diálogo es la siguiente:

- **Título**: 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.

- **Mensaje**: 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 denegar la acción.
  - **Neutral**: este botón se usa cuando el usuario no quiere continuar con la acción, pero no necesariamente quiere denegar, usado para cancelar la acción. 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 manera directa. Para este ejemplo se creará un nuevo método que se encargue de mostrar el cuadro de alerta. Añade el siguiente código dentro de un bloque `with(binding)` para ahorrar al escribir.

```
btnAlertDialog.setOnClickListener {
    myAlertDialog(getText(R.string.my_first_dialog))
}
```

A continuación, el código para el método `myAlertDialog`, fíjate en tipo del parámetro *message*, al utilizar el método *getText* en la llamada se consigue obtener el formato del texto.

```
private fun myAlertDialog(message: CharSequence) {
    // Se crea el AlertDialog.
    AlertDialog.Builder(this).apply {
        // Se asigna un título.
        setTitle(android.R.string.dialog_alert_title)

        // Se asgina el cuerpo del mensaje.
        setMessage(message)

        // Se define el comportamiento de los botones.
        setPositiveButton(
            android.R.string.ok,
            DialogInterface.OnClickListener(function = actionButton)
        )
        setNegativeButton("No") { _, _ ->
            Toast.makeText(context, "No", Toast.LENGTH_SHORT).show()
            binding.root.setBackgroundColor(Color.RED)
        }
        setNeutralButton(android.R.string.cancel) { _, _ ->
            Toast.makeText(context, android.R.string.cancel, Toast.LENGTH_SHORT).show()
            binding.root.setBackgroundColor(Color.WHITE)
        }
    }.show() // Se muestra el AlertDialog.
}
```

En este ejemplo, puedes observar que los botones para la decisión negativa y neutral están estructurados igual, hacen lo mismo. También se utilizan los guiones bajos (_) que representan los parámetros *dialog* y *which* de la *lambda* que, como en este caso no se utilizan, en Kotlin se pueden sustituir por este carácter para indicar que no se utilizarán y evitar un *warning*.

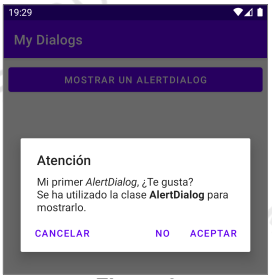

Figura 2

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.

```
private val actionButton = { dialog: DialogInterface, which: Int ->
    Toast.makeText(this, android.R.string.ok, Toast.LENGTH_SHORT).show()
    binding.root.setBackgroundColor(Color.GREEN)
}
```

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.

```
setPositiveButton(android.R.string.ok) { _, _ ->
    actionButton()
}
```

También existe una versión de *AlertDialog* para *Material Design*, bastará con sustituir la clase *AlertDialog* y su método *Builder* para construir el diálogo por la clase `MaterialAlertDialogBuilder`, el resto del código quedaría igual.

9.2.1. Añadir una lista al diálogo

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

```
private fun myAlertDialogList(names: Array<String>) {
    AlertDialog.Builder(this).apply {
        setTitle("My AlertDialog con lista")

        setItems(names) { _, which ->
            Toast.makeText(
                context,
                names[which],
                Toast.LENGTH_LONG
            ).show()
        }
    }.show()
}
```

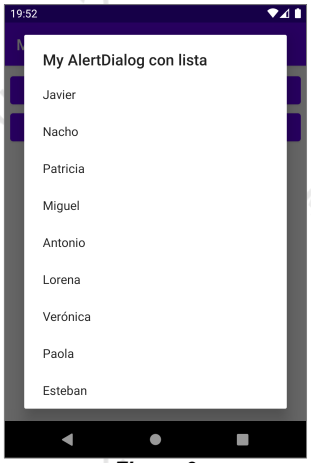

Figura 3

Como el contenido de la lista aparece en la zona del mensaje, esta opción no se utiliza, en su lugar, se utiliza `setItems` para mostrar los datos.

Debes fijarte que en este caso, sí se utiliza la variable `which`, esta devolverá el índice de la posición pinchada en la lista mostrada, que coincidirá con las posiciones del *array*.

También es importante saber que, al pulsar un elemento de la lista, el cuadro de diálogo se descarta, esto es debido a que no se trata de una lista persistente, motivo por el cual el cuadro de diálogo se cierra.

La llamada al método *myAlertDialogList* será como se muestra, pasándole un *array* como parámetro.

```
btnAlertDialogList.setOnClickListener {
    myAlertDialogList(resources.getStringArray(R.array.array_nombres))
}
```

9.2.2. Diálogos con listas persistentes de opción simple

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()`, dónde se indicará la lista a mostrar, el elemento que aparecerá seleccionado, -1 (ninguno) como primera instancia 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.

```
private fun myAlertDialogSinglePersistentList(names: Array<String>) {
    AlertDialog.Builder(this).apply {
        var selectedPosition = -1

        setTitle("My AlertDialog con lista simple")

        setSingleChoiceItems(R.array.array_nombres, -1) { _, which ->
            selectedPosition = which
            Log.d("DEBUG", names[selectedPosition])
        }

        setPositiveButton(android.R.string.ok) { dialog, _ ->
            if (selectedPosition != -1) {
                Toast.makeText(context, names[selectedPosition], Toast.LENGTH_SHORT).show()
            }
        }

        setNegativeButton(android.R.string.cancel) { _, _ ->
            Toast.makeText(context, android.R.string.cancel, Toast.LENGTH_SHORT).show()
        }
    }.show()
}
```

En este caso, se crea la variable `selectedPosition` para conocer que elemento ha seleccionado el usuario con cada pulsación de un ítem.

Si el método empleado para conocer la opción seleccionada no te convence, puedes utilizar la siguiente línea dentro del bloque *setPositiveButton* para obtener la posición navegando por la estructura jerárquica de vistas.

```
1  val selectedPosition = (dialog as AlertDialog).listView.checkedItemPosition
```

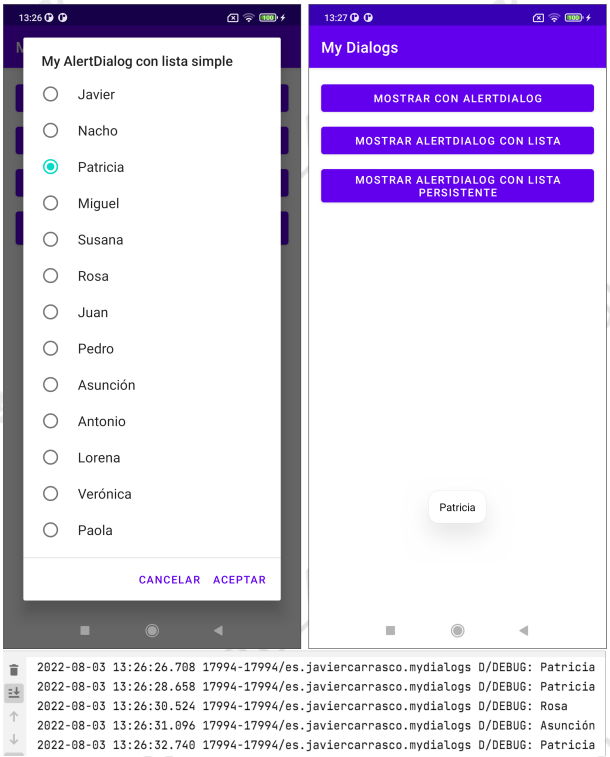

Figura 4

9.2.3. Diálogos con listas persistentes de opción múltiple

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()` de *AlertDialog* o *MaterialAlertDialogBuilder*, que son intercambiables.

```
1  private fun myAlertDialogMultiPersistentList(names: Array<String>) {
2      MaterialAlertDialogBuilder(this).apply {
3          val selectedItems = ArrayList<Int>()
4
5          setTitle("My AlertDialog con lista multiple")
```

```
        setMultiChoiceItems(R.array.array_nombres, null) { _, which, isChecked ->
            if (isChecked) {
                selectedItems.add(which)
                Log.d("DEBUG", "Checked: " + names[which])
            } else if (selectedItems.contains(which)) {
                selectedItems.remove(which)
                Log.d("DEBUG", "UnChecked: " + names[which])
            }
        }

        setPositiveButton(android.R.string.ok) { _, _ ->
            var textToShow = "Checked: "
            if (selectedItems.size > 0) {
                for (item in selectedItems) {
                    textToShow = textToShow + names[item] + " "
                }
            } else textToShow = "No items checked!"
            Toast.makeText(context, textToShow, Toast.LENGTH_SHORT).show()
        }

        setNegativeButton(android.R.string.cancel) { _, _ ->
            Toast.makeText(context, android.R.string.cancel, Toast.LENGTH_SHORT).show()
        }
    }.show()
}
```

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.

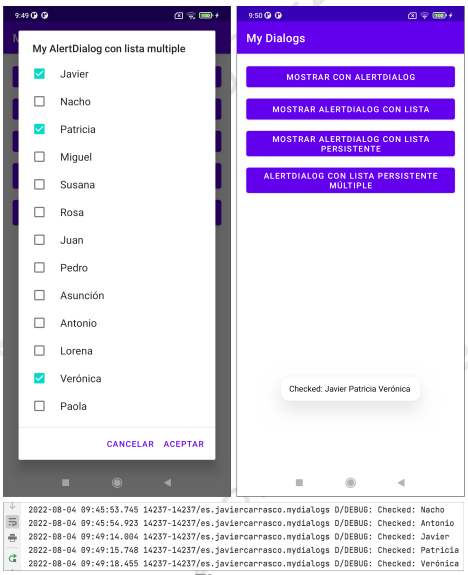

Figura 5

9.2.4. Cuadros de diálogo personalizados

En ocasiones, puede resultar útil mostrar un cuadro de diálogo con un diseño personalizado, que se adapte a las necesidades, como por ejemplo, un sistema de *login* o solicitar un correo electrónico. Para ello, se utilizará el método `setView()` para asignar el *layout* personalizado que se quiera mostrar.

En primer lugar, deberás crear un diseño personalizado, para lo cual, se abrirá un nuevo *layout* llamado `dialog_layout.xml`, intenta reproducir el cuadro de diálogo que se muestra como resultado para este ejemplo.

A continuación, reproduce el código *Kotlin* que se encargará de crear el cuadro de diálogo personalizado haciendo uso del *layout* personalizado y, recoger los datos introducidos por el usuario.

```kotlin
private fun myCustomAlertDialog() {
    // Se infla el layout personalizado del diálogo.
    val bindingCustom = DialogLayoutBinding.inflate(layoutInflater)

    AlertDialog.Builder(this).apply {
        setView(bindingCustom.root)

        setPositiveButton(android.R.string.ok) { _, _ ->
            Toast.makeText(
                context,
                "User: ${bindingCustom.etUsername.text}\n" +
                        "Pass: ${bindingCustom.etPassword.text}",
                Toast.LENGTH_SHORT
            ).show()
        }

        setNegativeButton(android.R.string.cancel) { dialog, _ ->
            Toast.makeText(context, android.R.string.cancel, Toast.LENGTH_SHORT).show()
            dialog.dismiss()
        }
    }.show()
}
```

En primer lugar, se *"inflará"* la vista personalizada en el cuadro de diálogo utilizando la variable `bindingCustom`. Al utilizar el método `setView()` se añade la vista como argumento. Debes tener en cuenta como recoger los datos introducidos en el cuadro de diálogo. Esto se hará mediante *ViewBinding*.

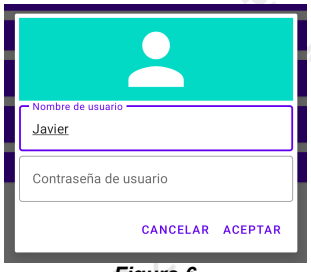

Figura 6

También se hace uso del 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 cancelar, para ello sería necesario extender la clase con `DialogInterface.OnDismissListener`.

Puedes controlar el comportamiento de los botones para que, por ejemplo, hasta que el usuario y contraseña no tengan un valor, no pueda cerrarse mediante el botón aceptar, obligando así a utilizar cancelar para cerrar. El método anterior cambiará ligeramente.

```
private fun myCustomAlertDialog() {
    // Se infla el layout personalizado del diálogo.
    val bindingCustom = DialogLayoutBinding.inflate(layoutInflater)

    val dialog = AlertDialog.Builder(this).apply {
        setView(bindingCustom.root)

        setPositiveButton(android.R.string.ok, null)
        setNegativeButton(android.R.string.cancel) { dialog, _ ->
            Toast.makeText(context, android.R.string.cancel, Toast.LENGTH_SHORT).show()
            dialog.dismiss()
        }
    }.create()

    dialog.setOnShowListener {
        dialog.getButton(AlertDialog.BUTTON_POSITIVE).setOnClickListener {
           if (bindingCustom.etUsername.text.isNullOrBlank()) {
               bindingCustom.textInputUsername.error = getString(R.string.warning_username)
           } else {
               bindingCustom.textInputUsername.error = ""

               Toast.makeText(
                   this@MainActivity,
                   "User: ${bindingCustom.etUsername.text}\n" +
                           "Pass: ${bindingCustom.etPassword.text}",
                   Toast.LENGTH_SHORT
               ).show()

               dialog.dismiss()
           }
       }
    }

    dialog.show()
}
```

Se cambia la llamada del *setPositiveButton* indicando en el segundo parámetro, el *listener*, que será nulo, para a continuación añadirle un ***OnShowListener*** de la clase ***DialogInterface***, de esta forma, cuando se muestre el diálogo se podrá capturar la pulsación sobre el botón que se decida, el resto ya lo conoces.

También debes fijarte que para esta estrategia, se crea el diálogo como un objeto, para después de crearlo añadir el *listener*, tras lo cual ya puede mostrarse (*show*).

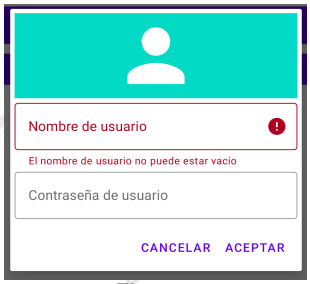

Figura 7

9.3. Time Picker

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

```
btnTimePicker.setOnClickListener {
    val cal = Calendar.getInstance()
    val timeSetListener = TimePickerDialog.OnTimeSetListener { tp, hour, minute ->
        cal.set(Calendar.HOUR_OF_DAY, hour)
        cal.set(Calendar.MINUTE, minute)
        tvTimePicker.text = SimpleDateFormat("HH:mm", Locale("es", "ES")).format(cal.time)
    }

    // Dentro del with(binding), se especifica el contexto con this@MainActivity.
    TimePickerDialog(
        this@MainActivity,
        timeSetListener,
        cal.get(Calendar.HOUR_OF_DAY),
        cal.get(Calendar.MINUTE),
        true
    ).show()
}
```

En primer lugar, deberá crearse un *listener* para el *Time Picker*, éste permitirá recoger los valores una vez cerrado el cuadro diálogo utilizando el botón aceptar. 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 se utilizarán son, el contexto, la hora y el minuto con los valores que contendrá al abrirse, en este caso se utilizará la hora actual del sistema, además se indica a *true* el formato 24 horas.

Figura 8

9.4. Date Picker

Un cuadro de diálogo ***Date Picker*** funciona exactamente igual que el *Time Picker*. A continuación, puedes ver el código para crearlo y asignar el valor seleccionado por el usuario a un *TextView* como se ha hecho en el ejemplo anterior.

```
btnDatePicker.setOnClickListener {
    val cal = Calendar.getInstance()
    val dateSetListener = DatePickerDialog.OnDateSetListener { dp, y, m, d ->
        cal.set(Calendar.YEAR, y)
        cal.set(Calendar.MONTH, m + 1) // 0→Enero, 1→Febrero, etc.
        cal.set(Calendar.DAY_OF_MONTH, d)

```

```
        tvDatePicker.setText(
            getString(
                R.string.txtDate,
                cal.get(Calendar.DAY_OF_MONTH),
                cal.get(Calendar.MONTH),
                cal.get(Calendar.YEAR)
            )
        )
    }

    DatePickerDialog(
        this@MainActivity,
        dateSetListener,
        cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH),
        cal.get(Calendar.DAY_OF_MONTH)
    ).show()
}
```

Si observas detenidamente el código, verás que los argumentos creados en el *listener* se corresponden con el año, mes y día, en este orden. El resto funciona exactamente igual que en el *Time Picker* visto en el punto anterior.

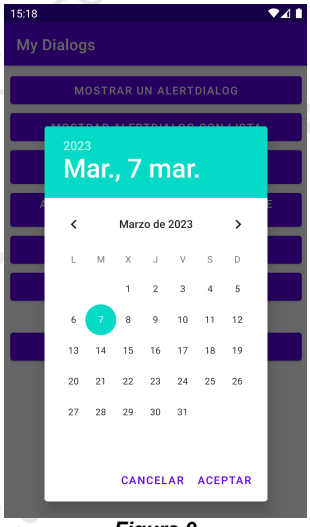

Figura 9

Por último, fíjate en el uso que hace de los meses la clase `Calendar`, el mes de enero se encuentra en la posición cero, de ahí que haya que sumarle uno a la variable *m*, que es la que contiene en este caso los meses.

9.5. Utilizando DialogFragment

Una opción para reutilizar el código de un diálogo es crear una clase que extienda de `DialogFragment`, además otra ventaja que tiene es que respeta el ciclo de vida de la actividad, por lo que, si giras la pantalla, no desaparecerá.

Empieza creando una clase que extienda de `DialogFragment` para construir dentro un `AlertDialog` básico. Crea la clase `CommonDialog`. A continuación, puedes ver el código que contendrá la clase `CommonDialog.kt` creada. También deberá crearse un *companion object* en el que se creará el método `newInstance`, similar al visto con los *Fragments*, para instanciar el cuadro de diálogo.

```
class CommonDialog : DialogFragment() {
    private lateinit var message: String

    companion object {
        fun newInstance(message: String): CommonDialog {
            return CommonDialog().apply {
```

```
                arguments = Bundle().apply {
                    putString("message", message)
                }
            }
        }
    }

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        return activity?.let {
            message = arguments?.getString("message").toString()

            AlertDialog.Builder(it).run {
                setMessage(message)
                setPositiveButton(android.R.string.ok) { dialog, id ->
                    Toast.makeText(it, "Aceptado", Toast.LENGTH_SHORT).show()
                }
                setNegativeButton(android.R.string.cancel) { dialog, id ->
                    Toast.makeText(it, "Cancelado", Toast.LENGTH_SHORT).show()
                }
            }.create()
        } ?: throw IllegalStateException("Activity cannot be null")
    }
}
```

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.

Seguidamente se define el *AlertDialog*. Muy básico, se asigna el mensaje (`.setMessage`), un botón para la acción positiva (`.setPositiveButton()`) y otro para la acción de cancelación (`.setNeutralButton()`). Fíjate que se hace uso de los *resources*, tanto de la aplicación como de Android, además, se pasa por parámetro el texto que se quiera mostrar. Por último, se muestra el diálogo.

```
binding.button.setOnClickListener {
    CommonDialog.newInstance("This is my first AlertDialog")
        .show(supportFragmentManager, "dialog")
}
```

Al mostrar el *dialog* se usa el método `show()`, el segundo argumento, *"dialogFragment"*, 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 (`CommonDialog`) la que debe resolver las acciones para las respuestas dadas por el usuario.

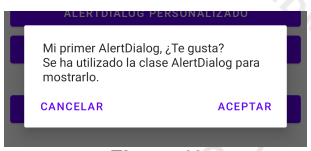

Figura 10