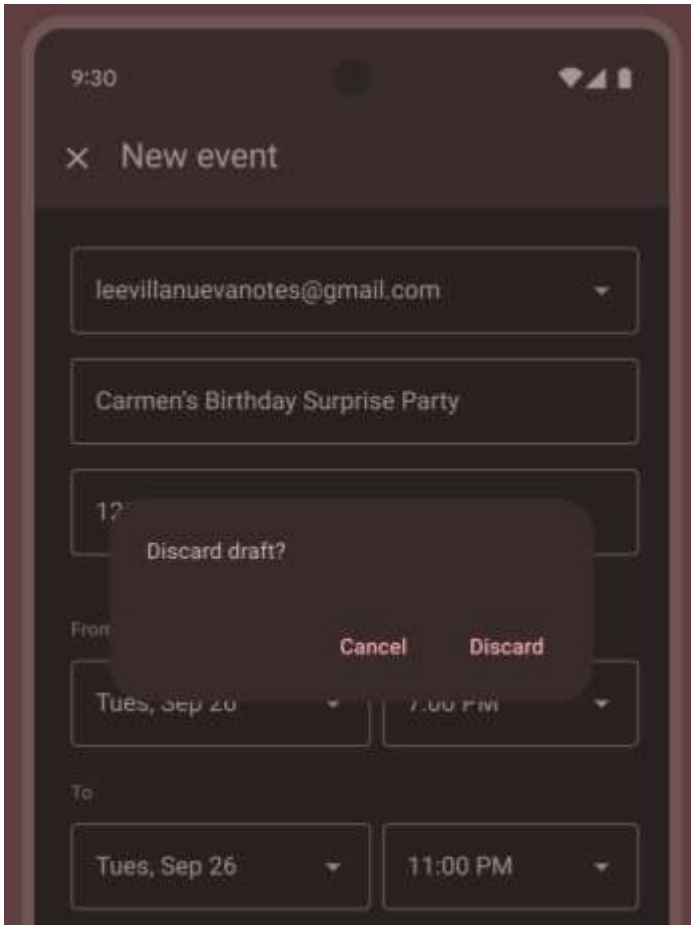
Dialog, Toast and Snackbar Widgets

ONE LOVE. ONE FUTURE.

Android Dialogs

A ***dialog*** is a small window that prompts the user to make a decision or enter additional information. A dialog doesn't fill the screen and is normally used for modal events that require users to take an action before they can proceed.

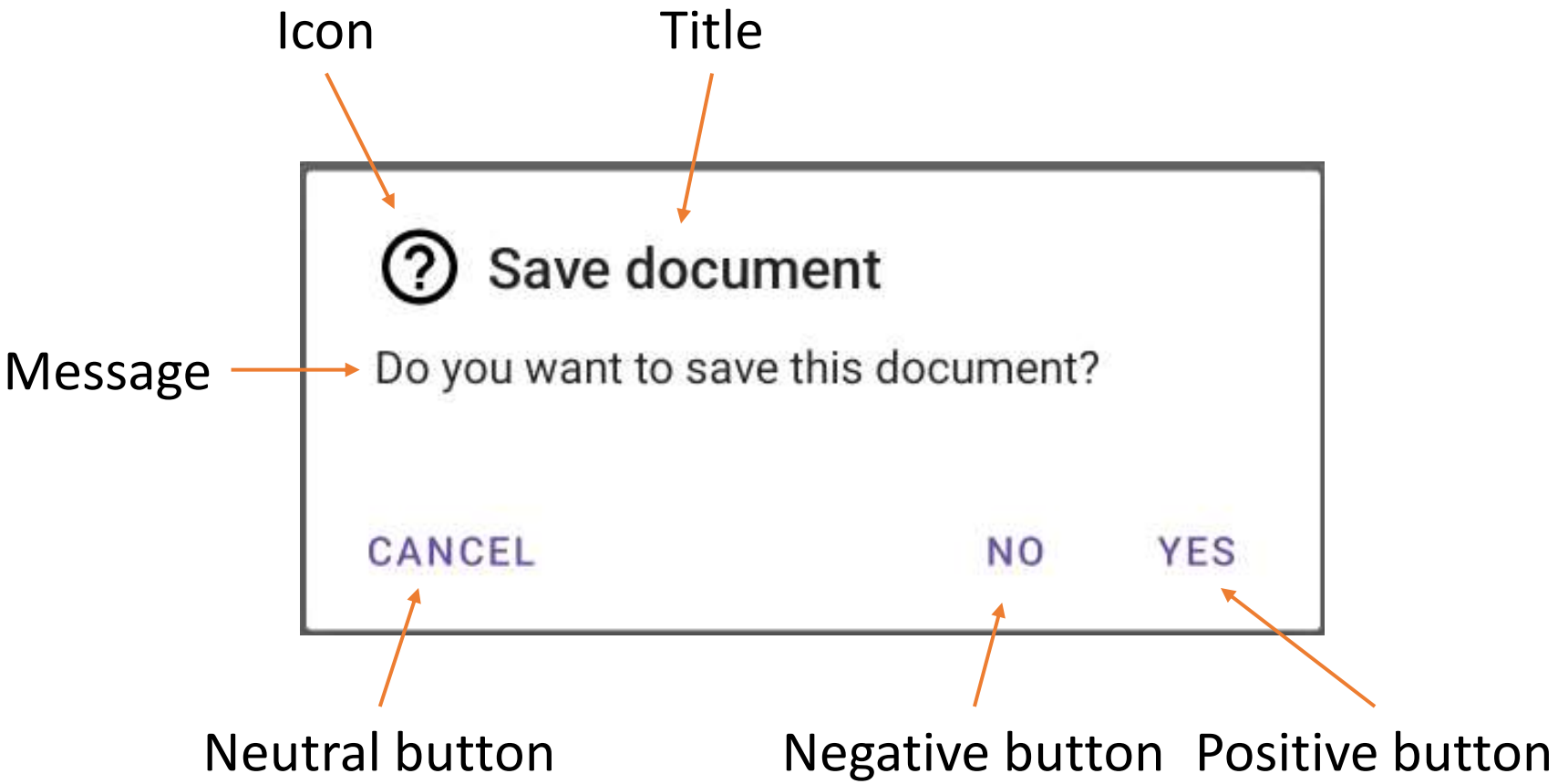


The **Dialog** class is the base class for dialogs, but don't instantiate Dialog directly. Instead, use one of the following subclasses:

- **AlertDialog** A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.
- **DatePickerDialog** or **TimePickerDialog** A dialog with a predefined UI that lets the user select a date or time.

The AlertDialog

Dissecting an AlertDialog



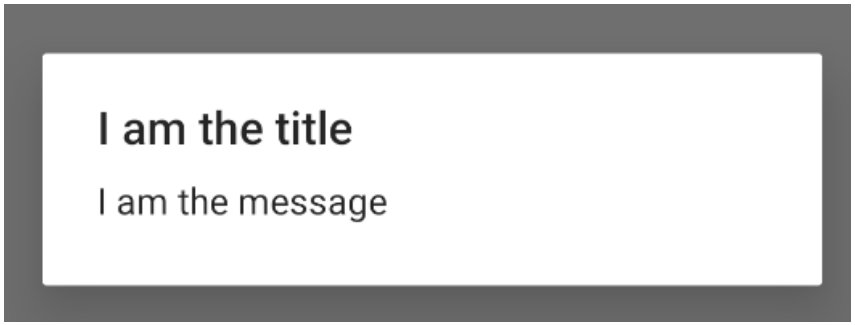
Build an alert dialog

The **AlertDialog** class lets you build a variety of dialog designs and is often the only dialog class you need. As shown in the following figure, there are three regions of an alert dialog:

- Title
- Content area
- Action buttons

The **AlertDialog.Builder** class provides APIs that let you create an AlertDialog with these kinds of content, including a custom layout.

```
val builder: AlertDialog.Builder = AlertDialog.Builder(context)
builder
    .setMessage("I am the message")
    .setTitle("I am the title")

val dialog: AlertDialog = builder.create()
dialog.show()
```



Add buttons

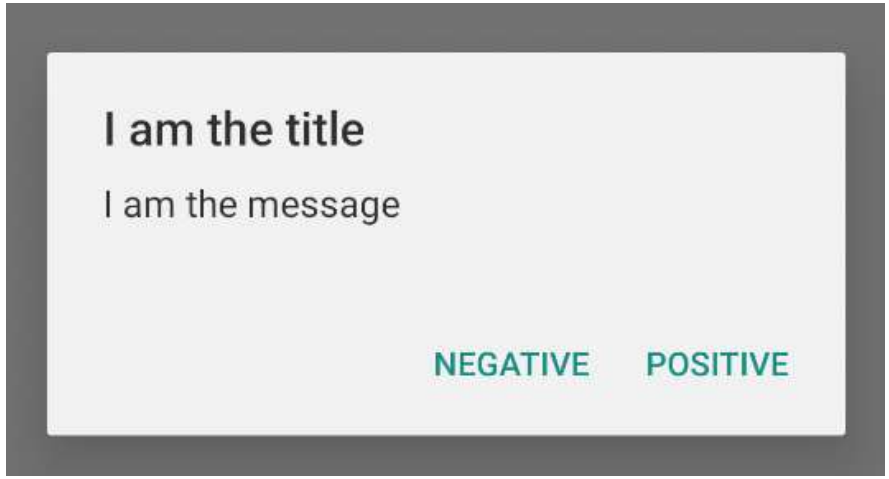
There are up to three action buttons you can add:

- **Positive:** use this to accept and continue with the action.
- **Negative:** use this to cancel the action.
- **Neutral:** use this when the user might not want to proceed with the action but doesn't necessarily want to cancel. It appears between the positive and negative buttons.

You can add only one of each button type to an **AlertDialog**.
The **set...Button()** methods require a title for the button (supplied by a string resource) and a **DialogInterface.OnClickListener** that defines the action to take when the user taps the button.

Add buttons - Example

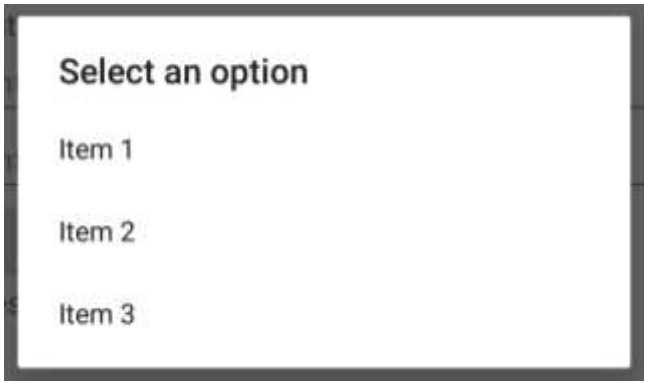
```
val builder: AlertDialog.Builder = AlertDialog.Builder(context)
builder
    .setMessage("I am the message")
    .setTitle("I am the title")
    .setPositiveButton("Positive") { dialog, which ->
        // Do something.
    }
    .setNegativeButton("Negative") { dialog, which ->
        // Do something else.
    }

val dialog: AlertDialog = builder.create()
dialog.show()
```



Adding a list

There are three kinds of lists available with the AlertDialog APIs:

- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)



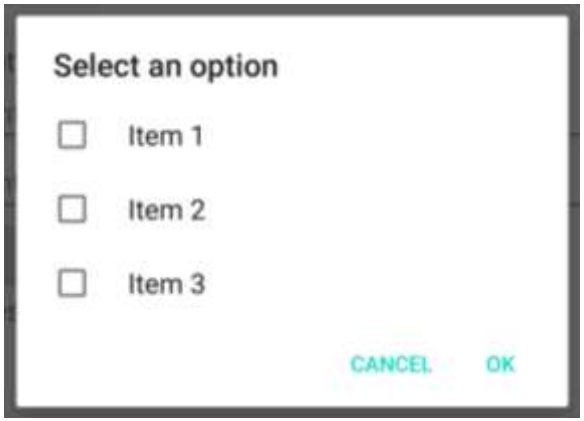
To create a single-choice list like the one in figure 3, use the **setItems()** method:

```
AlertDialog.Builder(this)
    .setTitle("Select an option")
    .setItems(R.array.items) { dialog, which ->
        println("Item selected - $which")
    }
    .create().show()
```

Adding a list

Adding a persistent multiple-choice or single-choice list:

To add a list of multiple-choice items (checkboxes) or single-choice items (radio buttons), use the **setMultiChoiceItems()** or **setSingleChoiceItems()** methods, respectively.




```
val items = resources.getStringArray(R.array.items)
val checked = BooleanArray(items.size)

AlertDialog.Builder(this)
    .setTitle("Select an option")
    .setMultiChoiceItems(items, checked) { dialog, which, isChecked ->
        println("Item selected - $which - $isChecked")
    }
    .setPositiveButton("OK") { dialog, which ->
        println("OK")
    }
    .setNegativeButton("Cancel", null)
    .create().show()
```

Adding a list

Adding a persistent multiple-choice or single-choice list:

To add a list of multiple-choice items (checkboxes) or single-choice items (radio buttons), use the **setMultiChoiceItems()** or **setSingleChoiceItems()** methods, respectively.


```
AlertDialog.Builder(this)
    .setTitle("Select an option")
    .setSingleChoiceItems(items, 0) { dialog, which ->
        println("Item selected - $which")
    }
    .setPositiveButton("OK") { dialog, which ->
        println("OK")
    }
    .setNegativeButton("Cancel", null)
    .create().show()
```

Create a custom layout

If you want a custom layout in a dialog, create a layout and add it to an **AlertDialog** by calling **setView()** on your AlertDialog.Builder object, or add the layout to a **Dialog** by using **setContentView()**.

When the user taps an action button created with an **AlertDialog.Builder**, the system dismisses the dialog for you.
The system also dismisses the dialog when the user taps an item in a dialog list, except when the list uses radio buttons or checkboxes. Otherwise, you can manually dismiss your dialog by calling **dismiss()**.

Custom layout example

XML Layout – custom_dialog_layout.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="8dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Enter your information:"
        android:textSize="18sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/edit_fullname"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="text"
        android:hint="Enter your full name"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />
```

Custom layout example

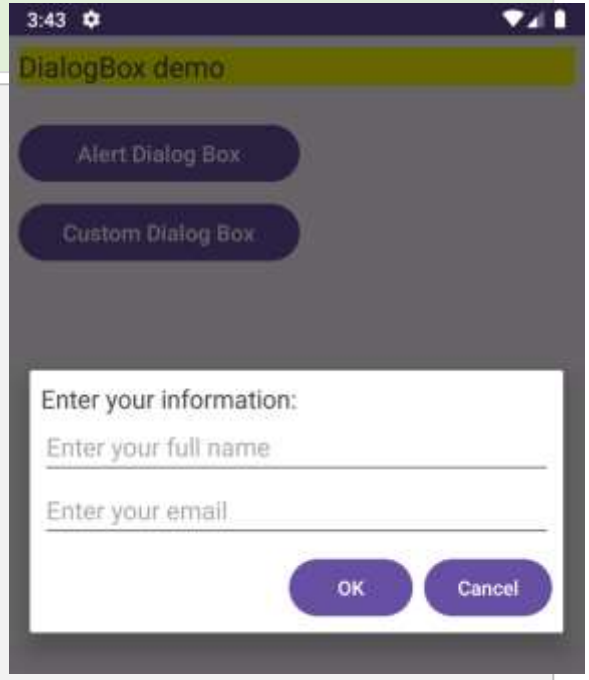
XML Layout – custom_dialog_layout.xml

```
    <Button
        android:id="@+id/button_cancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:text="Cancel"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/edit_email" />

    <Button
        android:id="@+id/button_ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:text="OK"
        app:layout_constraintEnd_toStartOf="@+id/button_cancel"
        app:layout_constraintTop_toBottomOf="@+id/edit_email" />

    <EditText
        android:id="@+id/edit_email"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textEmailAddress"
        android:hint="Enter your email"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/edit_fullname" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Custom layout example

DialogActivity.kt

```
    fun showCustomDialog() {
        val dialog = Dialog(this)
        dialog.setContentView(R.layout.dialog_custom_view)
        val editFullName = dialog.findViewById<EditText>(R.id.edit_fullname)
        val editEmail = dialog.findViewById<EditText>(R.id.edit_email)
        dialog.findViewById<Button>(R.id.button_ok).setOnClickListener {
            val fullName = editFullName.text.toString()
            val email = editEmail.text.toString()

            // TODO: Do something with name and email

            dialog.dismiss()
        }
        dialog.findViewById<Button>(R.id.button_cancel).setOnClickListener {
            dialog.dismiss()
        }
        dialog.window?.setLayout(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT)
        dialog.show()
    }
}
```



Create a dialog fragment

You also need a **DialogFragment** as a container for your dialog. The **DialogFragment** class provides all the controls you need to create your dialog and manage its appearance, instead of calling methods on the **Dialog** object.

Using **DialogFragment** to manage the dialog makes it correctly handle lifecycle events such as when the user taps the Back button or rotates the screen. The **DialogFragment** class also lets you reuse the dialog's UI as an embeddable component in a larger UI — just like a traditional Fragment — such as when you want the dialog UI to appear differently on large and small screens.

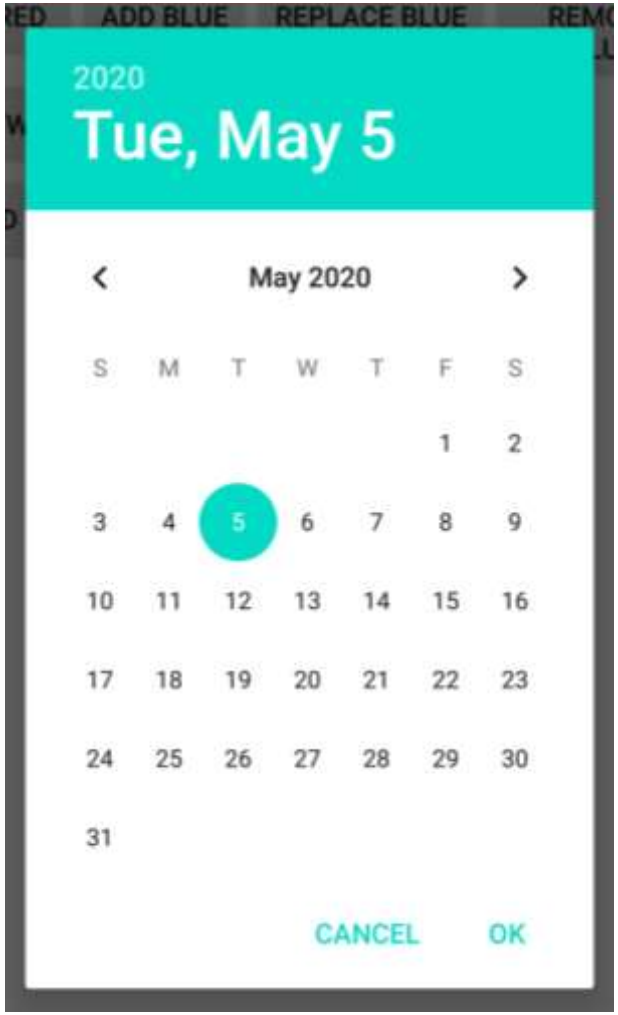
Create a dialog fragment – Example

```
class StartGameDialogFragment : DialogFragment() {
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        return activity?.let {
            // Use the Builder class for convenient dialog construction.
            val builder = AlertDialog.Builder(it)
            builder.setMessage("Start game")
                .setPositiveButton("Start") { dialog, id ->
                    // START THE GAME!
                }
                .setNegativeButton("Cancel") { dialog, id ->
                    // User cancelled the dialog.
                }
            // Create the AlertDialog object and return it.
            builder.create()
        } ?: throw IllegalStateException("Activity cannot be null")
    }
}

class OldXmlActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_old_xml)

        StartGameDialogFragment().show(supportFragmentManager, "GAME_DIALOG")
    }
}
```

DatePickerDialog

Android provides controls for the user to pick a time or date as ready-to-use dialogs.
These *pickers* provide controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year).

```
// Get current date
val c = Calendar.getInstance()
val mYear = c.get(Calendar.YEAR)
val mMonth = c.get(Calendar.MONTH)
val mDay = c.get(Calendar.DAY_OF_MONTH)

// Show dialog
val datePickerDialog = DatePickerDialog(this, {
    view: DatePicker, year: Int, month: Int, day: Int ->
    println("$year-${month + 1}-$day")
}, mYear, mMonth, mDay)
datePickerDialog.show()
```



TimePickerDialog

```
// Get current time
val c = Calendar.getInstance()
val mHour = c.get(Calendar.HOUR)
val mMinute = c.get(Calendar.MINUTE)

// Show dialog
val timePickerDialog = TimePickerDialog(this, {
        view: TimePicker, hourOfDay: Int, minute: Int ->
    println("$hourOfDay:$minute")
}, mHour, mMinute, false)
timePickerDialog.show()
```

Toast widget

A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. Toasts automatically disappear after a timeout.
For example, clicking **Send** on an email triggers a "Sending message..." toast, as shown in the following screen capture:

If your app targets Android 12 (API level 31) or higher, its toast is limited to two lines of text and shows the application icon next to the text. Be aware that the line length of this text varies by screen size, so it's good to make the text as short as possible.

Toast widget

```
Toast.makeText ( context,  message,  duration ).show()
```

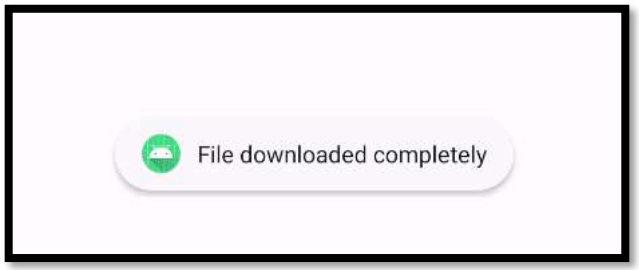
Context: A reference to the view's environment (where am I, what is around me…)

Message: The message you want to show

Duration: Toast.*LENGTH_SHORT* (0) about 2 sec
Toast.*LENGTH_* LONG (1) about 3.5 sec

Toast widget

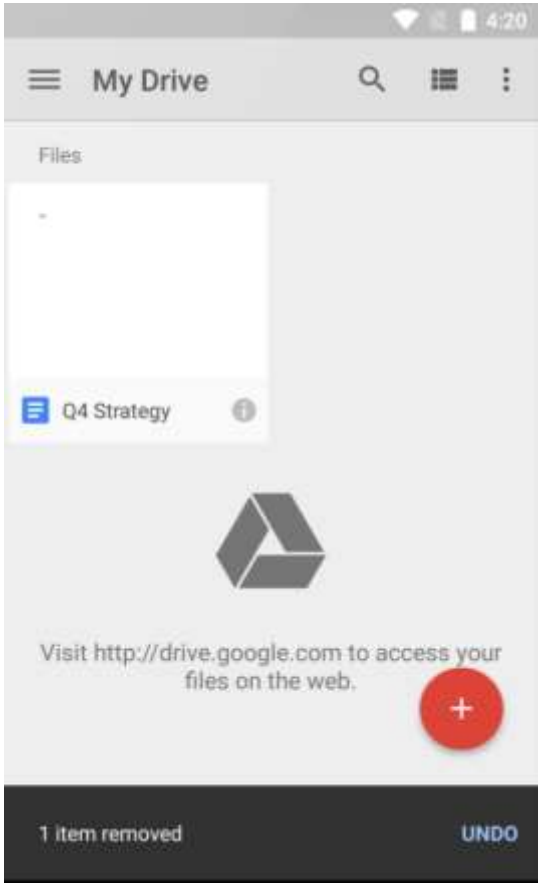
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        Toast.makeText(applicationContext, "File downloaded completely",
            Toast.LENGTH_LONG).show()
    }
}
```



In this simple application, passing the **context** variable could be done using: `applicationContext`, or **`this`**

Snackbar widget

- Snackbars provide brief messages about app processes at the bottom of the screen.
- Snackbars inform users of a process that an app has performed or will perform. They appear temporarily, towards the bottom of the screen. They shouldn't interrupt the user experience, and they don't require user input to disappear.
- Only one snackbar may be displayed at a time.
- A snackbar can contain a single action. "Dismiss" or "cancel" actions are optional.



Using snackbar

The `Snackbar` class provides static `make` methods to produce a snackbar configured in the desired way.
These methods take a `View`, which will be used to find a suitable ancestor `ViewGroup` to display the snackbar in, a text string to display, and a duration to display the snackbar for.
Available duration presets are:

- `LENGTH_INDEFINITE` (Show the snackbar until it's either dismissed or another snackbar is shown)
- `LENGTH_LONG` (Show the snackbar for a long period of time)
- `LENGTH_SHORT` (Show the snackbar for a short period of time)

Showing a snackbar

Calling `make` creates the snackbar, but doesn't cause it to be visible on the screen. To show it, use the show method on the returned `Snackbar` instance. Note that only one snackbar will be shown at a time. Showing a new snackbar will dismiss any previous ones first.
To show a snackbar with a message and no action:

```
// The view used to make the snackbar.
// This should be contained within the view hierarchy you want to display the
// snackbar. Generally it can be the view that was interacted with to trigger
// the snackbar, such as a button that was clicked, or a card that was swiped.
val contextView = findViewById<View>(R.id.context_view)

Snackbar.make(contextView, R.string.text_label, Snackbar.LENGTH_SHORT)
    .show()
```

Adding an action

To add an action, use the `setAction` method on the object returned from `make`. Snackbars are automatically dismissed when the action is clicked.
To show a snackbar with a message and an action:

```
Snackbar.make(contextView, R.string.text_label, Snackbar.LENGTH_LONG)
    .setAction(R.string.action_text) {
        // Responds to click on the action
    }
    .show()
```