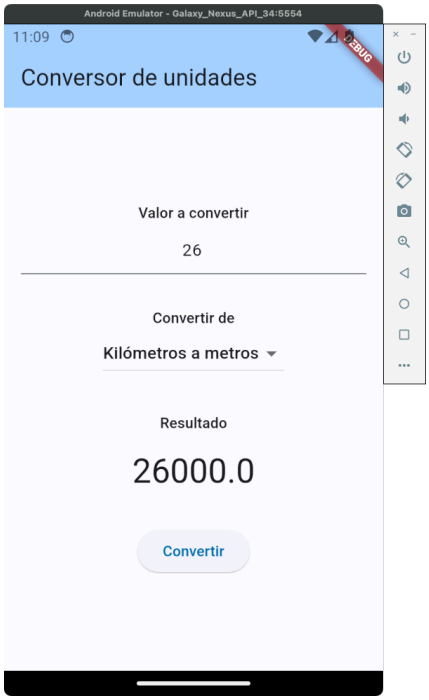
Flutter 1: Conversor de unidades

Parte II: Escribir la aplicación conversor

Contenidos

¿Qué voy a hacer en esta sesión?

En esta sesión escribiremos una aplicación para convertir medidas en diferentes unidades, por ejemplo: metros a kilómetros o viceversa. El aspecto de la aplicación terminada es este:



Es una aplicación sencilla, pero eso no debe engañarnos, mientras la escribimos aprenderemos:

- Cómo es la estructura de una aplicación Flutter y que todo en Flutter son vistas (Widgets).
- A utilizar, entre otros, los widgets de Flutter: `Placeholder`, `StatelessWidget`, `StatefulWidget`, `State`, `SafeArea`, `Scaffold`, `AppBar`, `Center`, `Padding`, `Column`, `Text`, `TextField`, `DropdownButton`, `SingleChildScrollView`, `SizedBox` y `ElevatedButton`
- Cómo conseguir que las vistas se vean con tamaños similares aunque la aplicación se ejecute en dispositivos con características físicas (tamaño y resolución) muy distintas.
- Cómo se gestiona el **estado mutable** de una aplicación con una sola pantalla, y cómo se refrescan los cambios de estado en la interfaz de usuario.

Abrir la aplicación

Arranca Android Studio y abre la aplicación `flutter_sesion_1` que has hecho en la primera parte de esta sesión. Contiene el código de la aplicación de prueba que Flutter genera automáticamente.

Escribir la aplicación conversor de unidades

Podríamos intentar reutilizar el código de la aplicación de prueba para escribir el código de nuestra aplicación conversor, pero vamos a escribirlo desde cero para poder aprender mejor algunas cosas fundamentales de las aplicaciones Flutter.

1 – Borrar el código de la aplicación de prueba

Comenzaremos borrando todo el contenido del archivo `main.dart` y sustituyéndolo por una función `main` vacía:

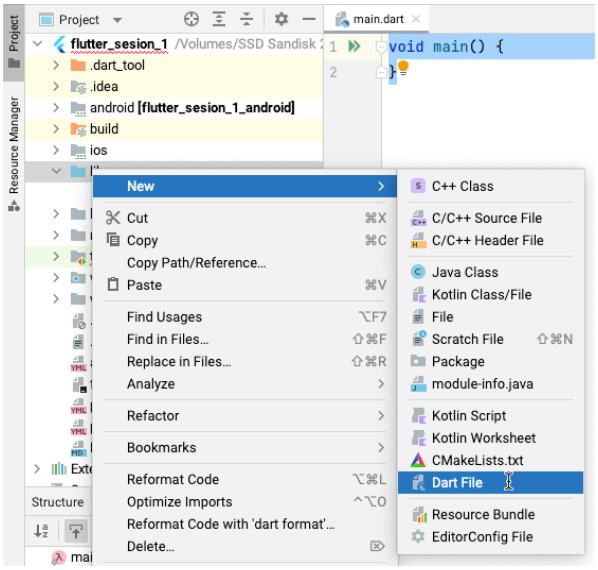
```
import 'package:flutter/material.dart';

void main() {
  // 1
}
```

> **Importante:** La función `main` es muy importante porque, como en otros lenguajes de programación, es el **punto de partida de la aplicación.** Cuando se ejecuta la aplicación se busca esta función y se ejecuta.

2 – Escribir el esqueleto de la clase aplicación

El siguiente paso es comenzar a escribir la clase aplicación. Será ésta la clase que se llamará dentro de la función `main` para iniciar la ejecución de la aplicación.

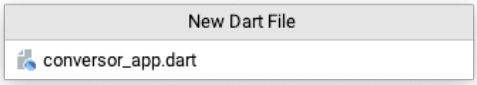
Como sabemos, cada clase en Flutter debe ir en su propio archivo, así que empezaremos añadiendo un nuevo archivo llamado `conversor_app.dart` Para eso, en la pestaña `Project` de Android Studio, haremos clic derecho sobre la carpeta `lib` para que aparezca el menú de contexto y ejecutaremos el menú `New › Dart File`.



Aparece un pequeño cuadro de diálogo en el que escribiremos el nombre del archivo:

```
conversor_app.dart
```

y pulsaremos `Enter`



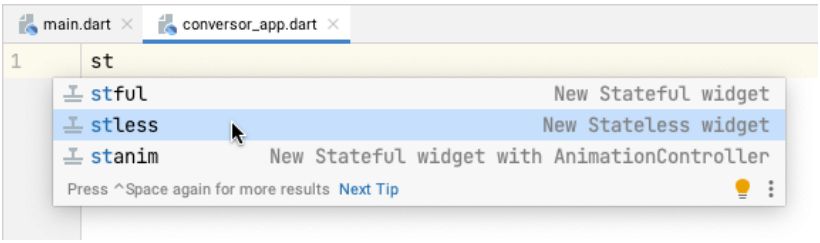
El nuevo archivo aparece dentro de la carpeta `lib` junto al archivo existente `main.dart`

> **Nota:** No es obligatorio seguir, para los nombres de archivo, el mismo convenio que se sigue para el nombre de la aplicación, pero Google lo recomienda, así que es el criterio que seguiremos en estas prácticas para nombrar los archivos de código.

Dentro de ese archivo declararemos la clase `ConversorApp` La forma más sencilla de hacerlo en Android Studio es comenzar a escribir:

```
stless
```

y cuando aparezca la sugerencia de código pulsar `Enter`. A continuación teclear `ConversorApp` y pulsar de nuevo `Enter`



Aparece un fragmento de código con la declaración de una nueva clase en Dart. Ese código tiene algunos errores de compilación que se deben a que falta **importar un paquete,** así que en la primera línea del archivo escribiremos:

```
import 'package:flutter/material.dart';
```

En este punto, el contenido del archivo `conversor_app.dart` debería ser:

```
import 'package:flutter/material.dart';

class ConversorApp extends StatelessWidget {
  const ConversorApp({super.key});

  @override
  Widget build(BuildContext context) {
    // 2
    return const Placeholder();
  }
}
```

Antes de seguir, veamos algunos aspectos importantes:

- Un **paquete** es una biblioteca que contiene código reutilizable. Para poder usarlo hay que importarlo mediante la instrucción `import` seguida del nombre del paquete entre comillas simples `' '`

  El paquete `flutter/material.dart` define widgets con los que construir la interfaz de usuario. En concreto proporciona *widgets de tipo material* que implementan **Material Design,** el lenguaje visual de descripción de interfaces desarrollado por Google.

  Flutter proporciona cientos de paquetes que nos permiten construir interfaces de usuario muy avanzadas con poco esfuerzo.

- La clase `ConversorApp` es una subclase de `StatelessWidget` que como su nombre indica **¡es un widget!**

  > **Importante: Todo en Flutter son vistas,** hasta la propia aplicación es una vista.

  > **Importante:** Estamos hablando de **vistas** y **widgets** como si fueran lo mismo, pero no lo son aunque están muy relacionadas. Un widget es una descripción de un componente de la interfaz de usuario. Esta descripción se convierte en una vista física que se dibuja en la pantalla, cuando se construye la interfaz, al arrancar la aplicación.
  >
  > La interfaz de una aplicación Flutter se describe mediante un **árbol de widgets,** en el que unos widgets contienen a otros. Ese árbol de widgets se transformará en un **árbol de vistas** al ejecutar la aplicación.

- Todas las subclases de `StatelessWidget` tienen que redefinir el método:

  ```
  Widget build(BuildContext context)
  ```

  que como vemos devuelve un objeto de clase `Widget` que es la superclase de todas las vistas que se pueden usar en Flutter. Este método se llama cuando el `StatelessWidget` se inserta en el árbol de widgets. Su misión es describir mediante código cómo es la parte de la interfaz de usuario que está contenida dentro de la vista.

  El código que nos ha proporcionado Flutter devuelve una instancia de `Placeholder` que es un widget que dibuja un cuadro que representa dónde se situarán otros widgets en el futuro.

  `Placeholder` es un widget útil durante el desarrollo, para indicar que la interfaz de usuario aún no está completa.

  El método `build` siempre recibe como único argumento el contexto de la aplicación. Este contexto es compartido por todos los widgets del árbol de widgets. A través de él se puede acceder, por ejemplo, al tema general de la aplicación (define los colores, tipos de letra, etc. que se usan en toda la aplicación) y a otra información útil que veremos en próximas sesiones.

- El constructor de `ConversorApp` se marca como `const` lo que como sabemos, significa que se construye una **instancia canónica de la clase.** Es decir, aunque se construyan muchas instancias de la clase `ConversorApp` todas ellas son la misma instancia en memoria.

  Eso es así ya que **una `StatelessWidget` es inmutable.** Una vez que se crea, no puede cambiarse, o lo que es igual, se crea solo una vez, y no se puede volver a reconstruir durante la ejecución de la aplicación. El método `build()` de una subclase de `StatelessWidget` se llama solo una vez.

Esto es muy importante, ya que cada vez que se refresque la interfaz de usuario de la aplicación se debe reconstruir el árbol de vistas. Al tener una instancia canónica, solo se construye la instancia de `ConversorApp` la primera vez, el resto de veces que se refresca la interfaz se reutiliza la misma instancia, con el consiguiente ahorro de recursos.

> **Importante:** Para que nuestra aplicación móvil sea realmente eficiente, todas aquellas vistas que no cambien durante la ejecución de una aplicación como por ejemplo, títulos o botones, deberían declararse como `const` De esta forma no se generarán nuevas instancias de esas vistas que no cambian al refrescar la interfaz de usuario.

3 – Iniciar la ejecución de la aplicación

Hasta ahora, tenemos una función `main` vacía en el archivo `main.dart` y una clase aplicación `ConversorApp` en el archivo `conversor_app.dart` Es el momento de conectar ambas cosas para que al iniciar la aplicación se cargue la clase `ConversorApp` que como sabemos es el nodo raíz del árbol de widgets de la aplicación.

Para ello, en el archivo `main.dart` , dentro del cuerpo de la función `main` busca el comentario:

```
  // 1
```

y sustitúyelo por:

```
  // 1
  runApp(const ConversorApp());
```

y al comienzo del archivo `main.dart` escribe el `import` :

```
  import 'package:flutter_sesion_1/conversor_app.dart';
```

que es necesario para que no haya errores de compilación. Con todo esto, el contenido del archivo `main.dart` completo debería ser:

```
  import 'package:flutter/material.dart';
  import 'package:flutter_1_conversor/conversor_app.dart';

  void main() {
    // 1
    runApp(const ConversorApp());
  }
```

Veamos algunos aspectos importantes de este código:

- El código de la función `main` es muy simple: se encarga de arrancar la ejecución de la aplicación mediante el método `runApp` que recibe un objeto de clase `ConversorApp`

  El método `runApp` recibe como argumento un widget que contiene otros widgets y los transforma en las distintas vistas de la interfaz de usuario.
- El argumento que se pasa a `runApp` está marcado como `const` ya que la clase `ConversorApp` tiene un constructor también marcado como `const`

Podríamos no escribir la palabra `const` delante de la llamada al constructor de `ConversorApp` pero en la pestaña `! Problems` podremos ver el siguiente warning:

```
Use 'const' with the constructor to improve performance.
```

donde se nos avisa que si se puede invocar un constructor como `const` para producir una instancia canónica, es preferible hacerlo, ya que de esta forma la aplicación es mucho más eficiente.

- Dado que la función `main` solo tiene una línea de código, también podría haberse escrito usando notación flecha así:

```
void main() => runApp(const ConversorApp());
```

4 – Escribir el contenido real de la clase aplicación

La clase `Placeholder` es útil para poder compilar la aplicación cuando no está completa la interfaz, pero no es el contenido que queremos para la clase `ConversorApp`. Lo vamos a cambiar. En el archivo `conversor_app.dart` buscaremos las líneas:

```
// 2
return const Placeholder();
```

y las sustituiremos por:

```
// 2
return MaterialApp(
  title: 'Conversor',
  theme: ThemeData(
    colorScheme: ColorScheme.fromSeed(
      seedColor: Colors.blue
    ),
    useMaterial3: true,
  ),
  // 3
  home: const Placeholder(),
);
```

Veamos algunos aspectos importantes de ese código antes de seguir:

- Todas las aplicaciones en Flutter devuelven un objeto de clase `MaterialApp` que es el contenedor que usaremos para construir interfaces de usuario basadas en **Material Design.** Se usa el constructor generativo personalizado de `MaterialApp` que tiene muchos parámetros con nombre (algunos opcionales y otros no) que permiten personalizar su apariencia y comportamiento.

En este ejemplo se han usado tres:

1. `title:` que es el título de la aplicación. En este ejemplo el título de la aplicación es:

   ```
   title: 'Conversor',
   ```

2. `theme:` que permite elegir el **tema general de la aplicación,** que define la apariencia global de las vistas que la forman. Define cómo se van a visualizar las demás vistas de la aplicación. En este ejemplo, utiliza el color azul para la barra de título.

   ```
   theme: ThemeData(
     colorScheme: ColorScheme.fromSeed(
       seedColor: Colors.blue
     ),
     useMaterial3: true,
   ),
   ```

   No se crea el tema desde cero, sino que se configura a partir de un `ThemeData` basado en un `ColorScheme` predeterminado. `Colors` es una clase abstracta que contiene propiedades de clase (static) para representar los colores del sistema. En este ejemplo se ha usado la propiedad `blue` que representa al color azul.

3. `home:` que recibe como argumento otro `Widget` que es la **página principal de la aplicación.** En este caso, dado que aún no hemos escrito el código de esa página principal, de nuevo se usa la clase `Placeholder`

   La instancia de `Placeholder` que se pasa al parámetro `home:` se ha declarado como `const` de esta forma no hay que reconstruirla cada vez que se deba refrescar la interfaz de usuario con el consiguiente ahorro de recursos.

5 – Escribir el esqueleto de la página principal de la aplicación

Como hemos visto, la propiedad `home:` de la clase `MaterialApp` espera que se le pase un widget que es la página principal de la aplicación. En este paso vamos a construirlo. Para ello añadiremos un nuevo archivo a la carpeta `lib` llamado `conversor_pagina_principal.dart`

En él añadiremos la clase `ConversorPaginaPrincipal` que en este caso es de tipo `StatefulWidget`. La forma más sencilla de escribir una subclase de `StatefulWidget` es escribiendo `stful` y pulsar `Enter` y justo a continuación, escribir el nombre de la clase: `ConversorPaginaPrincipal` y pulsar `Enter` otra vez.

Hay algunos errores de compilación que desaparecen con el import:

```
import 'package:flutter/material.dart';
```

El archivo `conversor_pagina_principal.dart` ahora contiene el siguiente código:

```
import 'package:flutter/material.dart';

class ConversorPaginaPrincipal extends StatefulWidget {
  const ConversorPaginaPrincipal({super.key});

  @override
  State<ConversorPaginaPrincipal> createState() =>
                                  _ConversorPaginaPrincipalState();
}

class _ConversorPaginaPrincipalState extends State<ConversorPaginaPrincipal> {
  // 8
  @override
  Widget build(BuildContext context) {
    // 4
    return const Placeholder();
  }

  // 16
}
```

Veamos algunos aspectos interesantes de este código:

- No se ha creado una clase, como en el caso de `StatelessWidget` sino que se han creado dos clases, la clase `ConversorPaginaPrincipal` y la clase `_ConversorPaginaPrincipalState` que como su nombre indica es **privada.**

  Eso es así, porque una subclase de `StatefulWidget` tiene asociado un estado mutable que cambia durante la ejecución de la aplicación. En el ejemplo del conversor, el estado mutable permitirá almacenar y cambiar los datos de la última conversión realizada. Esto permite a la aplicación responder a las acciones del usuario.

  La mejor forma de gestionar esto es separar la gestión del estado en una clase aparte que es subclase de la clase `State`. Dado que las subclases de `State` se declaran para gestionar el trabajo de una subclase de la clase `StatefulWidget` concreta y está muy acoplada con ésta, no tiene sentido que sea una clase pública. Lo mejor es hacerla privada y escribir las dos clases juntas en el mismo archivo.

- La clase `State` es genérica. Eso significa que para poder utilizarla, hay que escribir entre ángulos `< >` el nombre de la clase con la que está asociado ese estado. En esta aplicación el estado está asociado a la clase `ConversorPaginaPrincipal` por lo que se declara así:

  ```
  State<ConversorPaginaPrincipal>
  ```

- La subclase de `StatefulWidget`, `ConversorPaginaPrincipal`, no redefine el método `build`, sino que redefine el método `createState` que no tiene argumentos y que devuelve una subclase de la clase `State<ConversorPaginaPrincipal>` La subclase que implementa la gestión del estado de la clase `ConversorPaginaPrincipal`
- Quien redefine el método `build` es la clase `_ConversorPaginaPrincipalState` que es la clase que gestiona el estado asociado a la clase `ConversorPaginaPrincipal`.

Como en el caso de las subclases de `StatelessWidget` el método `build` recibe como argumento un contexto y devuelve un `Widget`

Ese método `build` **se llama cada vez que el estado mutable cambia,** para reflejar el nuevo estado en la interfaz de usuario de la aplicación. Por ejemplo, cada vez que se realiza una conversión, el estado cambia para poder mostrar en la interfaz de usuario el valor calculado.

- El constructor de `ConversorPaginaPrincipal` se marca como `const` ya que lo que cambia en una vista `StatefulWidget` no es la vista en sí, sino su estado. Por tanto, para aprovechar las ventajas de construir instancias canónicas, el constructor se declara `const`

Ya que tenemos una página principal para la aplicación, es el momento de conectarlo con ésta. Para ello, en el archivo: `conversor_app.dart` buscaremos las líneas:

```
// 3
home: const Placeholder(),
```

y las sustituiremos por las siguientes:

```
// 3
home: const ConversorPaginaPrincipal(),
```

6 – Completar la página principal de la aplicación

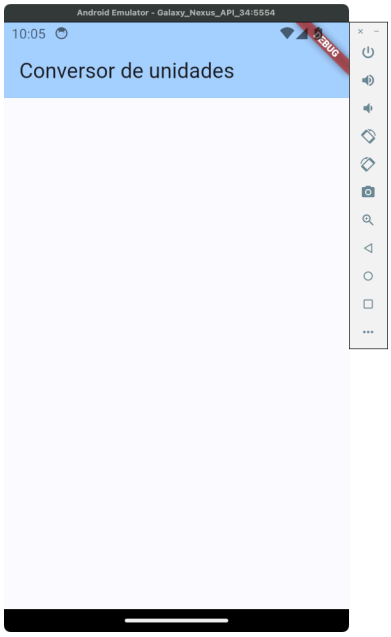
Comenzaremos completando el método `build` de la clase `_ConversorPaginaPrincipalState`. Para ello busca las líneas:

```
// 4
return const Placeholder();
```

y sustitúyelas por esta otra versión:

```
// 4
return Scaffold(
  appBar: AppBar(
    title: const Text('Conversor de unidades'),
    backgroundColor: Theme.of(context).colorScheme.inversePrimary,
  ),
  // 5
  body: const SafeArea(
    child: Center(),
  ),
);
```

Ahora, si se ejecuta la aplicación, se obtiene una salida que no es una pantalla en negro como hasta ahora:



La aplicación todavía no está completa, pero antes de seguir veamos algunos aspectos interesantes del código anterior:

- En el corazón de toda página principal de una aplicación hay un widget de tipo `Scaffold`. En Inglés, scaffold significa andamio. Es un nombre muy apropiado ya que este widget permite definir fácilmente varias zonas que pueden mostrarse en la ventana. Cada una de esas zonas se define mediante una propiedad con nombre del constructor de `Scaffold`

  En esta aplicación hemos usado dos de esas zonas. En próximas sesiones aprenderemos a usar otras zonas definidas por el widget `Scaffold`

- En esta aplicación se usa la propiedad `appBar:` de la vista `Scaffold` para define una **barra de aplicación.**

  > **Nota:** El patrón de interacción llamado: **barra de aplicación,** consiste en un rectángulo de pequeña altura situado en la parte superior de la pantalla, debajo de la barra de estado. Puede incluir además de un título, botones para acceder a las **herramientas más frecuentes de una determinada pantalla.**
  >
  > A diferencia de los botones de la barra de pestañas, en los botones de la barra de aplicación solo se usan iconos sin texto explicativo. Por lo general, las herramientas se colocan alineadas a la derecha.

  En esta aplicación no hay herramientas, por tanto en la barra de aplicación sólo se define un título, mediante la propiedad `title:` de la clase `AppBar` a la que se pasa un widget de la clase `Text`. Los widgets de la clase `Text` muestran texto de salida, es decir, el usuario solo puede leer su contenido, pero no escribir en ellos.

  También se usa la propiedad `backgroundColor` que permite indicar el color de fondo para esa barra de aplicación. En este caso, se ha optado por tomar el color del tema general que se asoció al contexto de la aplicación al crear la vista de tipo `MaterialApp`

- `body:` Como su nombre indica, es el cuerpo de la página principal. Recibe como argumento el widget que queramos que ocupe esa zona.

  En este ejemplo se ha usado un objeto de la clase `SafeArea` que añade el sangrado necesario para asegurar que las vistas hijas que se coloquen en ese cuerpo no se vean tapadas por elementos del sistema operativo o por las formas de la pantalla del dispositivo como por ejemplo esquinas redondeadas, etc.

  La clase `SafeArea` es un **contenedor,** un widget que puede contener a otro widget. Para eso tiene la propiedad `child:` que nos permite indicar qué widget queremos colocar dentro del área segura.

  En este caso hemos usado una instancia de la clase `Center` que es un **widget posicional.** También **es un contenedor que centra su contenido en la pantalla.** El widget que se coloque dentro de un widget de tipo `Center` estará centrado horizontal y verticalmente en la pantalla del dispositivo. Por ahora el widget contenedor `Center` está vacío. Pronto lo llenaremos de contenido.

Vamos a rellenar el cuerpo de la pantalla principal con widgets que permitan realizar la conversión de unidades. Se necesitará un campo en el que escribir el valor a convertir, un desplegable para escoger la conversión que se le quiere aplicar (por ejemplo kilómetros a metros), un botón para realizar la conversión y un texto donde se muestre el resultado. Son varios widgets y lo mejor es situarlos unos sobre otros formando una columna.

En Flutter existe un widget contenedor llamado `Column` cuya misión es justamente esa. Contiene en su interior más de un widget hijo, y los coloca en la pantalla, uno debajo del otro formando una columna. Para ello la clase `Column`, a diferencia de la clase `Center` que tiene la propiedad `child:` y que sólo acepta un único widget hijo, tiene una propiedad llamada `children:` que recibe una lista de widgets.

Dentro del método `build` de la clase `_ConversorPaginaPrincipalState` buscaremos las líneas:

```
// 5
body: const SafeArea(
  child: Center(),
),
```

y las sustituiremos por:

```
// 5
body: SafeArea(
  child: Center(
    child: Column(
      // 7
      children: <Widget>[
        // 6
      ],
    ),
  ),
),
```

Por ahora la lista de `Widget` está vacía, la vamos a rellenar a continuación.

> **Cuidado:** La palabra `const` que había justo delante de `SafeArea` ha desaparecido.

Etiqueta para el valor a convertir

Es una buena idea colocar antes del campo donde escribir el valor a convertir, una etiqueta que indique al usuario cuál es el contenido de ese campo. Esto le ayudará a entender y usar la interfaz. Usaremos el texto `'Valor a convertir'`

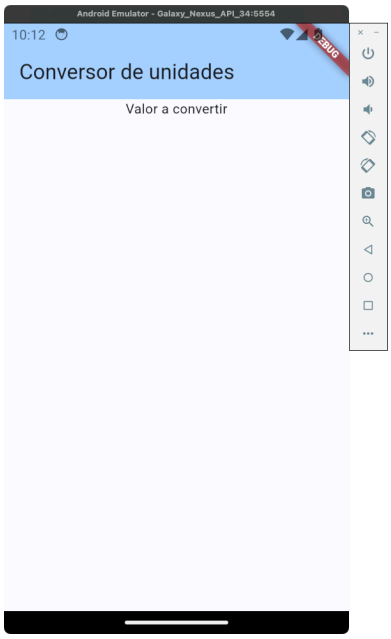
Para ello usaremos el widget `Text` ya conocido. Dentro de la lista de widgets que se pasa en la propiedad `children:` de la clase `Column` buscaremos la línea:

```
// 6
```

y la sustituiremos por lo siguiente:

```
// 6
Text('Valor a convertir',),
```

Si hacemos Hot Reload veremos que ahora la interfaz tiene este aspecto:



No es exactamente lo que teníamos en mente. Hay dos problemas:

1. La etiqueta está centrada horizontalmente, pero no verticalmente, sino que aparece alineada en la parte superior. El widget `Center` alinea en el centro a su widget hijo, tanto horizontalmente como verticalmente. Entonces, ¿qué es lo que ha fallado?

   No ha fallado nada. Recuerda que el widget hijo es `Column` y ese sí que está alineado tanto horizontalmente como verticalmente. El problema es que, por defecto, **la columna ocupa todo el espacio vertical disponible,** así que es tan alta como el cuerpo de la página principal.

   Dentro de la columna, por defecto, sus elementos se colocan alineados en la parte superior, así que por eso el texto `Valor a convertir` aparece arriba.

   La solución es sencilla. Sólo hay que usar la propiedad `mainAxisAlignment:` del widget `Column` para indicar que los elementos de la columna también estén alineados verticalmente.

Busca el comentario:

```
// 7
```

y sustitúyelo por:

```
// 7
mainAxisAlignment: MainAxisAlignment.center,
```

2. El texto es demasiado pequeño. No es legible y puede que algunos usuarios no puedan leerlo con claridad. La solución también es sencilla. El widget `Text` tiene la propiedad `style:` que permite personalizar la forma en que se muestra el texto en la pantalla. Para usarlo busca las líneas de código:

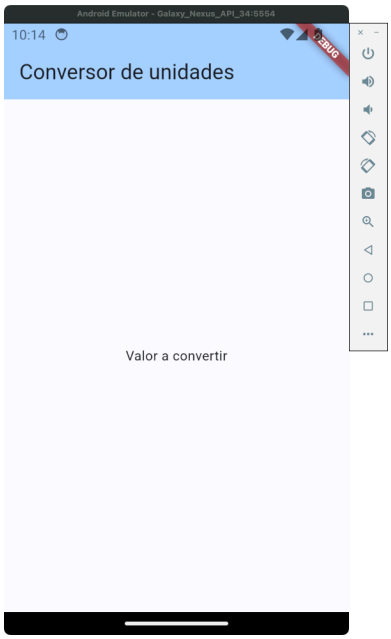
```
// 6
Text('Valor a convertir',),
```

y sustitúyelas por:

```
// 6
Text(
  'Valor a convertir',
  style: Theme.of(context).textTheme.labelLarge,
),
// 9
```

De nuevo se ha usado el tema general de la aplicación asociado a su contexto, pero en esta ocasión se han usado las propiedades relativas a cómo se muestra el texto en la pantalla. `labelLarge` es una variable que contiene los ajustes de texto (tamaño, etc.) para una etiqueta de texto grande.

Si hacemos Hot Reload, la interfaz se ve así:



que ahora sí es lo que queríamos.

Campo de texto donde escribir el valor a convertir

El valor a convertir forma parte del estado mutable de la pantalla principal, es decir, debe haber alguna propiedad en la clase `_ConversorPaginaPrincipalState` donde poder guardar ese valor. Por ello, al comienzo de la clase `_ConversorPaginaPrincipalState` buscaremos el comentario:

```
// 8
```

y lo sustituiremos por las siguientes líneas de código:

```
// 8
late double _valorAConvertir;
// 10
```

A continuación añadiremos un nuevo widget a la lista que se pasa a la clase `Column` en la propiedad `children:` Busca el comentario:

```
// 9
```

Que está justo debajo del widget `Text` introducido en el paso anterior y sustitúyelo por:

```
// 9
TextField(
  textAlign: TextAlign.center,
  keyboardType: const TextInputType.numberWithOptions(
    signed: true,
    decimal: true
  ),
  onChanged: (texto) {
    var valorIntroducido = double.tryParse(texto);
    setState(() { _valorAConvertir = valorIntroducido ?? 0.0; });
  },
),
// 11
```

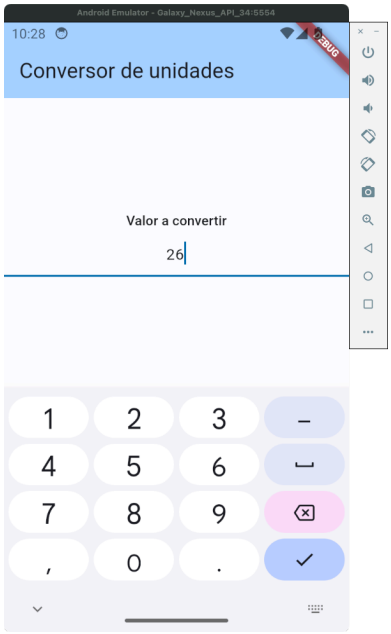
Tal y como está la aplicación en este momento, la propiedad `_valorAConvertir` no se ha inicializado y en la clase no hay constructor. El código tal y como está no funciona. Antes de poder ejecutar la aplicación hay que redeclarar el método `initState()` en `_ConversorPaginaPrincipalState`. Busca:

```
// 10
```

y sustitúyelo por:

```
// 10
@override
void initState() {
  _valorAConvertir = 0.0;
  // 12
  super.initState();
}
```

Si hacemos Hot Restart el aspecto de la interfaz será parecido a este:



Como vemos la interfaz de usuario es mejor. El texto que se introduce aparece centrado para una mayor legibilidad y el teclado en pantalla aparece ya preparado para introducir números, lo que simplifica la tarea del usuario.

Veamos algunos aspectos interesantes de ese código:

- La propiedad `_valorAConvertir` se ha marcado como `late` para que pueda ser inicializada posteriormente en el método `initState`. De otra forma se produce un error de compilación.
- El método `initState` se llama una sola vez cuando se construye el estado. Aquí es donde generalmente se asignan los valores iniciales para las propiedades. En este caso, estamos estableciendo el valor `0.0` como valor inicial para la propiedad `_valorAConvertir`.

  > **Importante:** Si se redefine el método `initState`, siempre se debe llamar a `super.initState();` al final del mismo.

- Se usa un widget de clase `TextField` que es el widget que se usa en Flutter cuando se quiere proporcionar al usuario un campo donde pueda escribir alguna entrada. Dentro del constructor de `TextField` se han usado las propiedades:
  1. Para que el texto que el usuario escribe en el `TextField` aparezca centrado, se usa la propiedad `textAlign:` con el valor `TextAlign.center`. El argumento es de tipo `TextAlign` que es un `enum` que contiene los posibles modos de alinear un texto en un campo de texto.
  2. A continuación se usa la propiedad `keyboardType:` de la clase `TextField` para elegir el teclado que se muestra al usuario por defecto. Se ha elegido un teclado numérico con las opciones `signed:` y `decimal:` con valor `true`. Si se hubiese elegido un teclado de tipo `TextInputType.number` solo se podrían introducir enteros.
  3. Existen varias formas en Flutter de actuar cuando un usuario escribe algo en un widget `TextField`. La que usamos aquí es mediante la propiedad `onChanged:` que recibe como argumento una función anónima que tiene un único parámetro de tipo `String`

Cada vez que el usuario cambie el valor del campo de texto (escriba, borre, etc.) se ejecuta la función anónima que se pasa a `onChanged:` Esa función tiene un argumento de tipo `String` que aquí se ha llamado `texto`. A través de él se tiene acceso al texto tecleado por el usuario en el `TextField`

El usuario puede escribir, por equivocación, algo que no sea un número, por eso antes de usar el valor contenido en la variable `texto` hay que parsearlo para convertirlo en un `double`. Eso se hace con el método `tryParse` de la clase `double` que devuelve un valor de tipo `double?` Si el valor introducido por el usuario no corresponde con un número válido, entonces se devuelve `null`

En la variable `valorIntroducido` de clase `double?` se tiene, o bien `null` si el valor no es correcto o bien un `double`. Con él se actualizará el estado de la clase `_ConversorPaginaPrincipalState` representado por la propiedad `_valorAConvertir`

> **Nota:** Para simplificar el código se ha usado el operador **if-null** (`??`). Si no recuerdas còmo funciona, repasa el tema dedicado a valores nulos de la sesión 2 de Dart.

Para actualizar el estado, debe llamar al método `setState` de la clase `State` que se hereda en `_ConversorPaginaPrincipalState`.

El método `setState` le dice a Flutter que el estado de un objeto ha cambiado y que la interfaz de usuario debe actualizarse. Dentro del método `setState`, se cambian las propiedades de la clase que almacenen el estado mutable (en nuestro caso, `_valorAConvertir`).

> **Muy importante:** Si se quiere modificar alguna propiedad que forma parte del estado mutable de una subclase de la clase `State` hay que hacerlo **siempre** dentro de una función anónima que se pasa al método `setState` de la clase `State`. De lo contrario Flutter no sabrá que se ha cambiado el estado y no refrescará la interfaz.

Añadir un desplegable para elegir el tipo de conversión

Es el momento de añadir los componentes de la interfaz para mostrar al usuario las distintas conversiones que se pueden realizar. Comenzaremos añadiendo nuevas propiedades a la clase `_ConversorPaginaPrincipalState`. Busca el comentario:

```
// 10
```

y sustitútyelo por:

```
final _conversiones = ['Kilómetros a metros', 'Metros a kilómetros'];
late int _idConversionActual;
// 10
```

A continuación busca dentro del método `initState`, justo antes de la llamada a `super.initState();` el comentario:

```
// 12
```

y sustitúyelo por:

```
// 12
_idConversionActual = 0;
// 13
```

y añade dos nuevos widgets en la lista de widgets que se pasa al widget de clase `Column`. Busca el comentario:

```
// 11
```

justo detrás del widget `TextField` que añadimos en el paso anterior, y sustitúyelo por:

```
// 11
Text(
    'Convertir de',
    style: Theme.of(context).textTheme.labelLarge,
),
DropdownButton<String>(
    value: _conversiones[_idConversionActual],
    items: _conversiones.map((String value) {
        return DropdownMenuItem<String>(
            value: value,
            child: Text(value),
        );
    }).toList(),
    onChanged: (nuevoValor) {
        setState(() {
            _idConversionActual = _conversiones.indexOf(nuevoValor!);
        });
    },
),
// 14
```

Si hacemos Hot Restart vemos que ahora la interfaz de usuario tiene este aspecto:

La interfaz es más completa, pero los componentes aparecen amontonados unos sobre otros. Visualmente no queda muy bien. Lo cambiaremos para que tenga un aspecto más cuidado y profesional, pero antes veamos algunos aspectos interesantes del código que acabamos de añadir:

- Las conversiones posibles se tienen predefinidas en una lista privada de `String` llamada `_conversiones`. Por ahora solo se contemplan dos posibles conversiones para nuestra aplicación, convertir de kilómetros a metros y de metros a kilómetros.

  > **Atención spoiler:** Esta no es la mejor forma de codificar esto, pero nos dará juego para aprender técnicas de diseño de software.

- La conversión seleccionada en un momento determinado se almacena en la propiedad privada `_idConversionActual` Es de tipo `int` y representa al índice de la conversión seleccionada. Posteriormente en el método `initState` se le asigna el valor inicial `0`
- Se ha añadido un widget de clase `Text` con el texto `Convertir de` que sirve de título y ayuda a comprender al desplegable. Las opciones utilizadas son las mismas que se han usado para la etiqueta `Valor a convertir`, si es necesario se pueden consultar ahí.
- A continuación se usa el widget `DropdownButton` de Flutter para construir el desplegable. Es un widget que permite a los usuarios seleccionar un valor de una lista de valores. En todo momento muestra el valor seleccionado actualmente, así como un pequeño triángulo que despliega una lista para seleccionar otro valor.

  En este widget llaman la atención algunas cosas:

  1. Es genérico, lo que permite que el desplegable contenga valores de varias clases. En este caso se ha usado el tipo `String`. Eso indica que cuando el usuario seleccione un nuevo valor en el desplegable el tipo asociado a ese valor seleccionado será `String`
  2. El valor asociado, el que se muestra seleccionado en el desplegable, se indica mediante la propiedad `value:` En este caso se pasa el valor `_conversiones[_idConversionActual]`

     `_idConversionActual` forma parte del estado mutable de la clase, por lo que cuando cambie, se refrescará la interfaz y en consecuencia el valor mostrado en el desplegable también cambiará.
  3. La lista que muestra los valores del desplegable es de tipo `List<DropdownMenuItem<String>>?` Es decir, es una lista anulable de widgets de tipo `DropdownMenuItem<String>`. Eso es lo que hay que pasarle a la propiedad `items:` de la clase `DropdownButton`
  4. En nuestro caso, las posibles conversiones están en una lista, pero de `String`, así que hay que transformarla en una lista de `DropdownMenuItem<String>` antes de poder

pasarla a la propiedad `items:`

Esto puede hacerse con la ayuda del higher-order method `map` de la clase `List` aplicado a la lista `_conversiones`. Como sabemos `map` recibe como argumento una función anónima con un parámetro del mismo tipo que los elementos de la lista a la que se aplica, en este caso la lista `_conversiones`, cuyos argumentos son de tipo `String`. La función anónima se ejecuta una vez para cada elemento y devuelve un `Iterable` formado por el resultado de aplicar la función anónima a cada elemento de la lista.

En este caso lo único que hay que hacer en el interior de la función anónima es construir un `DropdownMenuItem` con el `String` que se recibe como argumento y devolverlo.

5. El higher-order method `map` devuelve un `Iterable`, pero lo que se debe pasar a la propiedad `items:` de la clase `DropdownButton` es una instancia de la clase `List`.

   Recordemos que la clase `Iterable` tiene un método llamado `toList` que permite hacer la conversión de forma sencilla. Solo hay que invocarlo detrás del paréntesis de cierre del método `map`

6. Al igual que en el caso del widget `TextField`, se usa la propiedad `onChanged:` para responder cuando el usuario seleccione un nuevo valor en el desplegable. Esta propiedad recibe una función anónima con un único argumento de tipo `String?` que contiene el nuevo valor seleccionado por el usuario.

   Dentro de la función anónima hay que actualizar el valor de `_idConversionActual` que es parte del estado mutable, así que. hay que hacerlo dentro del método `setState` que de nuevo recibe una función anónima sin argumentos.

   En el cuerpo de esa función anónima se busca si en la lista `_conversiones` hay algún valor que coincida con el `nuevoValor` y se usa su índice para actualizar el valor de `_idConversionActual`

   Al escribirla hay que tener en cuenta que el tipo de `nuevoValor` es `String?` por lo que debe usarse así: `nuevoValor!` (con signo de admiración `!` al final)

Ahora veamos algunas de las alternativas que tenemos para que los elementos no aparezcan tan juntos y haya un poco de espacio alrededor de la pantalla principal.

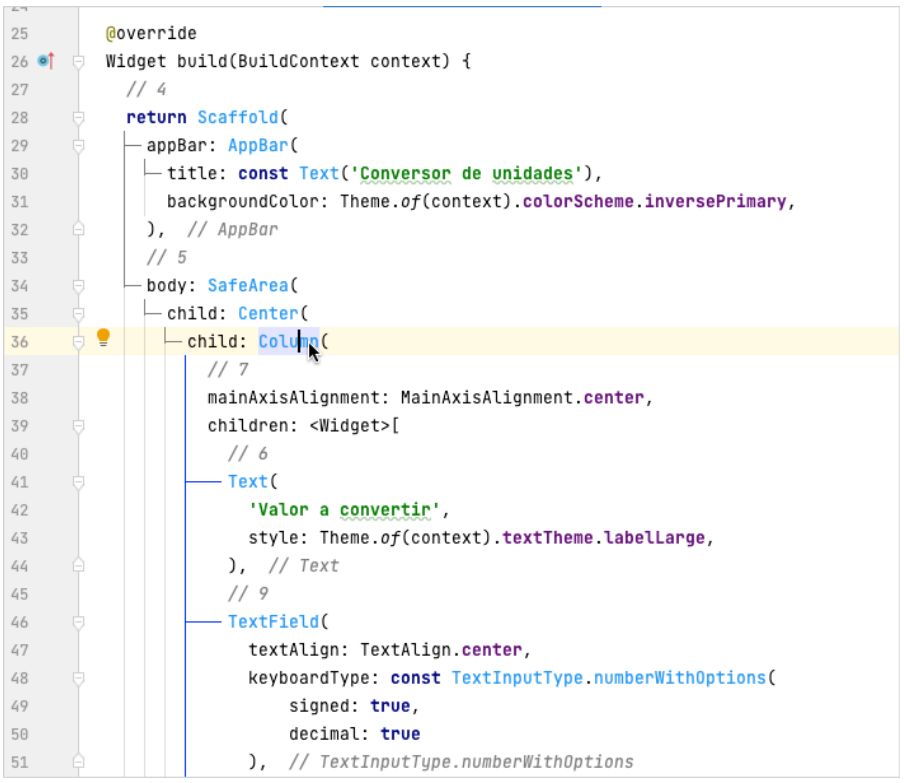
- Se puede mejorar el aspecto de una vista controlando la separación entre sus bordes y los del contenedor en el que está situada. Para ello Flutter proporciona el widget contenedor `Padding` que añade un espacio alrededor de su widget hijo. El relleno puede aplicarse a todos los lados por igual o solo a algunos de ellos.

  Vamos a colocar el widget `Column` dentro de un widget `Padding`. Este widget tiene un solo widget hijo, y lo dibuja en la pantalla dejando un poco de espacio alrededor.

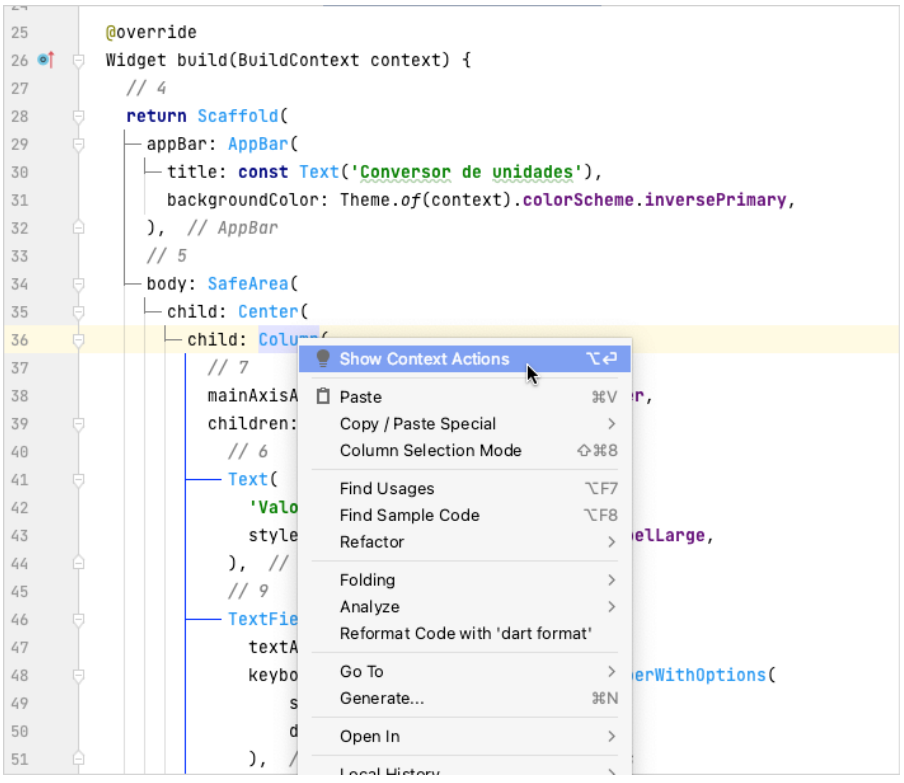
  El árbol de vistas de nuestra aplicación, en este momento, es complejo. Así que no es sencillo insertar manualmente un nuevo elemento alrededor de otro ya construido.

Sin embargo, Android Studio y su plugin de Flutter nos ayudan en esta tarea:

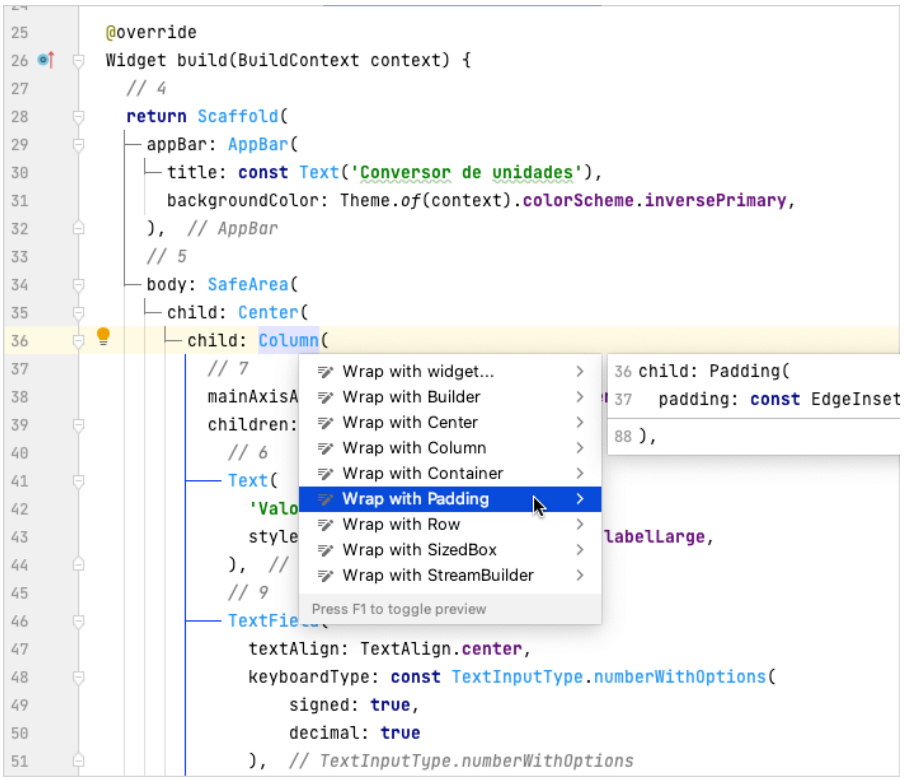
1. En primer lugar, haremos clic con el ratón sobre la llamada al constructor de la clase `Column`:



2. A continuación, haremos clic derecho para que se muestre el menú de contexto, y dentro de éste elegiremos la primera opción `Show Context Actions`:



3. En esa lista de acciones elegir `Wrap with padding`:



4. El código se reescribe y el widget `Column` ahora aparece como widget hijo de un widget de clase `Padding`. Solo hay que escoger el valor para la propiedad `padding:` que se quiera. En este caso se ha elegido un valor de `EdgeInsets.all(16.0)` que añade 16 pixels de espacio alrededor del widget `Column` por los cuatro lados:

- Para dejar espacio entre dos widgets en una lista, una opción muy útil es usar un widget `SizedBox` vacío. Este widget se coloca entre los dos widgets que se quieren separar. La ventaja de este contenedor es que se puede asignar un tamaño concreto, tanto en anchura, como en altura, lo que permite controlar de forma muy precisa la separación entre las vistas.

  Buscaremos en el método `build` de la clase `_ConversorPaginaPrincipalState` el comentario:

  ```
  // 11
  ```

  y lo sustituiremos por:

  ```
  // 11
  const SizedBox(
      height: 32.0,
  ),
  ```

  Tras hacer Hot Restart, ahora el aspecto de la aplicación es más atractivo visualmente:

Medidas independientes del dispositivo

En Flutter, cuando hablamos de píxeles, en realidad estamos hablando de **píxeles independientes de la densidad o dp** (density-independent pixels), y no de píxeles físicos. Los píxeles físicos son el número real de píxeles que tiene un dispositivo. Pero hay varios factores de forma, y la resolución de una pantalla puede variar sustancialmente. Por ejemplo, el Sony Xperia E4 tiene un tamaño de pantalla de 5'' y una resolución de 960 * 540 píxeles. El Xperia X tiene el mismo tamaño de pantalla de 5'', pero una resolución de 1920 * 1080. Por lo tanto, si quisieramos dibujar un cuadrado de 540 píxeles por lado, sería mucho más pequeño en el segundo dispositivo.

Es por eso que las medidas se expresan en píxeles independientes de la densidad, dp, lo que permite que los widgets de la interfaz de usuario se vean de igual tamaño en todos los dispositivos, con independencia de su resolución.

Veamos como funciona. Para ello necesitamos estudiar algunos conceptos básicos.

Densidad de píxeles de una pantalla o Screen density

El número de píxeles que caben en una pulgada se conoce como densidad de píxeles.

Las pantallas de alta densidad tienen más píxeles por pulgada que las de baja densidad. Como resultado, los elementos de la interfaz de usuario de las mismas dimensiones de píxeles aparecen más grandes en pantallas de baja densidad y más pequeños en pantallas de alta densidad.

Para calcular la densidad de píxeles de una pantalla se usa esta ecuación:

$$screen\ density = \frac{\text{Anchura de pantalla en pixeles}}{\text{Anchura de pantalla en pulgadas}} \quad (1)$$

La independencia de la densidad se refiere a la visualización uniforme de los elementos de la interfaz de usuario en pantallas con diferentes densidades.

Píxeles independientes de la densidad o Density-independent pixels

Los píxeles independientes de la densidad, escritos como dp o dip, son unidades flexibles que se escalan para tener dimensiones uniformes en cualquier pantalla. Proporcionan una forma flexible de mostrar un mismo diseño en todas las plataformas.

Las interfaces de usuario Material Design utilizan píxeles independientes de la densidad para mostrar elementos de manera consistente en pantallas con diferentes densidades.

Cuando se desarrolla una aplicación usando Material Design, se usan dp para mostrar elementos de manera uniforme en pantallas con diferentes densidades.

Un dp es igual a un píxel físico en una pantalla con una densidad de 160.

Para calcular los dp en pantallas con otras densidades de píxeles se usa esta fórmula:

$$dp = \frac{(\text{dimensión en pixeles} * 160)}{screen\ density} \quad (2)$$

Anchura pantalla (pulgadas)	Screen density	Anchura pantalla (pixeles)	Anchura pantalla (dps)
1.5 in	120	180 px	240 dp
1.5 in	160	240 px	240 dp
1.5 in	240	360 px	240 dp

En la tabla anterior se muestran algunos ejemplos de la relación entre tamaño físico de la pantalla, densidad de píxeles, tamaño en píxeles de la pantalla y tamaño en dps

Los **píxeles escalables o sp,** cumplen la misma función que los píxeles independientes de la densidad, dp, pero para los tipos de letra. El valor predeterminado de un sp es el mismo que el valor predeterminado de un dp.

La principal diferencia entre sp y dp es que los sp preservan la configuración del tipo de letra que haya establecido el usuario. Los usuarios que tengan una configuración de texto más grande para una mejor accesibilidad verán que los tamaños de fuente coinciden con sus preferencias de tamaño de texto.

Añadir el botón para iniciar los cálculos

En la clase `_ConversorPaginaPrincipalState` se necesita una nueva propiedad donde se almacene el resultado de la conversión. Busca el comentario:

```
// 10
```

y sustitúyelo por:

```
late double _valorConvertido;
// 10
```

Ahora hay que inicializarlo en el método `initState`. Busca el comentario:

```
// 13
```

que hay justo delante de la llamada a `super.initState()` y sustitúyelo por lo siguiente:

```
// 13
_valorConvertido = 0.0;
```

`_valorConvertido` es parte del estado mutable de la aplicación, es decir, cada vez que se cambie el valor de la propiedad `_valorConvertido` se debe refrescar la interfaz para que se muestre ese resultado al usuario.

A continuación escribiremos los siguientes widgets en la lista de widgets para la columna, justo debajo del `DropdownButton` que escribimos en el paso anterior. Para ello buscaremos el comentario:

```
// 14
```

y lo sustituieremos por el siguiente código:

```
// 14
const SizedBox(
  height: 32.0,
),
Text(
  'Resultado',
  style: Theme.of(context).textTheme.labelLarge,
),
const SizedBox(
  height: 16.0,
),
Text(
  '$_valorConvertido',
  style: Theme.of(context).textTheme.headlineLarge,
),
const SizedBox(
  height: 32.0,
),
```

```dart
ElevatedButton(
  child: const Text(
    'Convertir',
  ),
  onPressed: () {
    // 17
  },
),
```

si hacemos Hot Restart vemos que se han añadido los widgets que faltaban a la interface.

De todos los widgets que se han añadido, el único que es nuevo es `ElevatedButton` Veamos algunos aspectos interesantes de este widget.

- Un `ElevatedButton` es un widget que facilita la interacción con el usuario. Se muestra resaltado para atraer la atención del usuario y cuando éste lo toca, se redibuja con una animación que indica que realmente se ha pulsado.

  Implementa de forma muy sencilla para el programador, la doble realimentación que debe usarse en todas las interfaces de usuario para aplicaciones móviles:

  - El primer nivel de realimentación consiste en informar claramente al usuario de que realmente ha pulsado el elemento de interacción correspondiente (en este caso el botón)
  - El segundo nivel de realimentación consiste en mostrar claramente al usuario el resultado de la acción desencadenada por el elemento de interacción correspondiente.

- Entre sus propiedades destacan dos:

  1. `child:` que permite indicar cuál es el contenido del botón. Lo normal es que se use como hija una vista de tipo `Text` para escribir un texto dentro del botón, pero también se podían usar otras vistas para añadir iconos u otra información.

     En este caso se ha usado una etiqueta con el texto: `Convertir`.

2. `onPressed:` En esta propiedad se pasa una función anónima. La que queremos que se ejecute cuando el usuario pulse el botón. Por ahora se ha dejado en blanco, porque todavía no se ha programado el modelo de la aplicación (las clases que implementan la lógica de negocio de la aplicación). Volveremos sobre este punto un poco más adelante.

   > **Nota:** También existe la propiedad `onLongPress` que permite distinguir cuándo el usuario hace una pulsación prolongada sobre el botón.

- Si no se indica un estilo concreto para el botón, su color de fondo, etc. se toman del tema de la aplicación.

7 – Probando la interfaz de usuario

Ya está terminada la interfaz de usuario de la aplicación. Todavía no se ha escrito el modelo de la aplicación, por lo que no hace ninguna tarea, pero antes de seguir, conviene probar a fondo esa interfaz para detectar de forma temprana posibles problemas.

Cuando hablo de probar a fondo la interfaz me refiero a probarla en varios dispositivos con diferentes características, por ejemplo, diferentes tamaños de pantalla. Para eso habremos generado varios emuladores con características físicas distintas.

En la imagen anterior puede verse, como, en un dispositivo con pantalla pequeña hay elementos de la interfaz de usuario a los que no se pueden acceder cuando se usa el teclado.

Sin embargo, si se prueba en un dispositivo con pantalla grande, sí cabe correctamente:

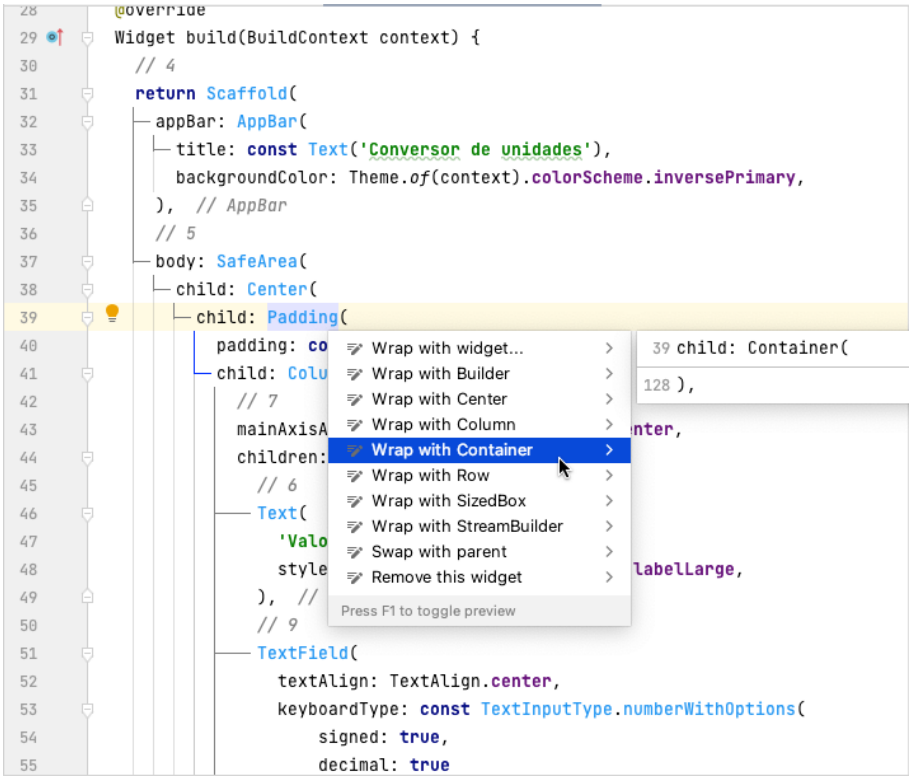
Si hay algún problema y no todos los elementos de la interfaz caben en la pantalla actual, Flutter lo muestra mediante un patrón visual de rayas amarillas y negras sobre las que se muestra un mensaje. En este ejemplo:

```
Bottom overflowed by 83 pixels
```

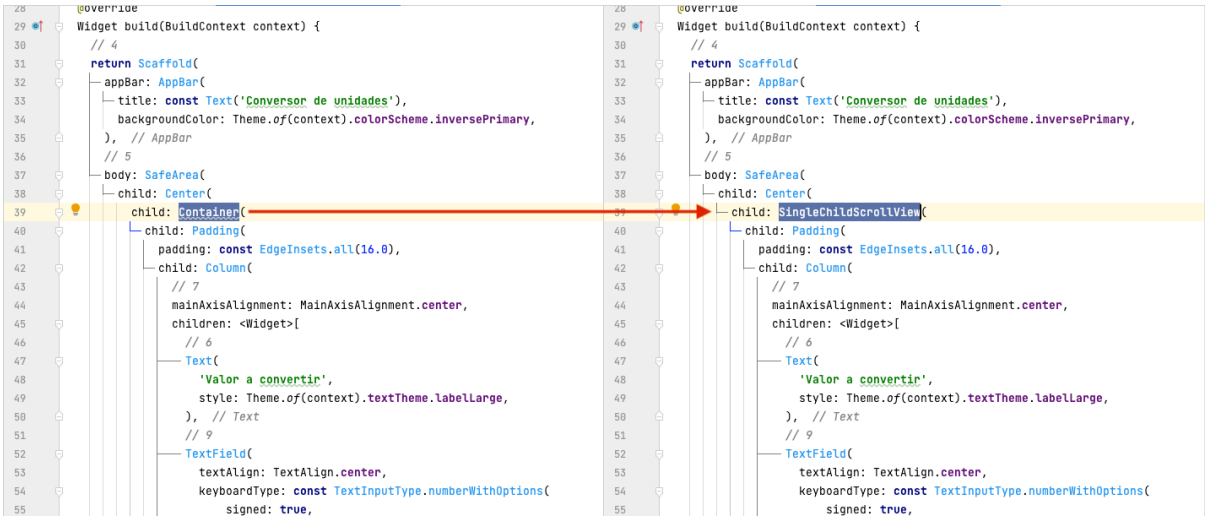
Flutter proporciona una solución sencilla para este problema. Consiste en incluir las vistas cuyo tamaño excede el tamaño de la pantalla, dentro de un widget de tipo `SingleChildScrollView`. Como su nombre indica es un contenedor que tiene un único widget hijo y que permite hacer scroll para acceder a todos los componentes de ese widget hijo, aunque no quepan todas a la vez en la pantalla. En nuestro ejemplo, incluiremos el widget `Padding` dentro de un widget `SingleChildScrollView`

Anteriormente, tuvimos que incluir la vista `Column` dentro de una vista `Padding` para dejar espacio alrededor, y vimos que la forma más sencilla de hacerlo era mediante las **acciones de contexto** `Context Actions` que proporciona el plugin de Flutter para Android Studio.

Sin embargo, si hacemos clic sobre la llamada al constructor de la clase `Padding`, a continuación hacemos clic derecho para mostrar el menú de contexto y ahí elegimos el menú `Show Context Actions`, vemos como no existe ninguna acción de contexto para envolver con una vista `SingleChildScrollView`



Pero no es problema ya que podemos envolver la vista `Padding` con una vista de tipo `Container` y después sustituir a mano la palabra `Container` por `SingleChildScrollView`



> **Consejo:** Siempre debes probar tus interfaces en diferentes tamaños de pantalla y orientaciones para comprobar que en todos los casos se ve correctamente.

Con esto el código del método `build` de la clase `_ConversorPaginaPrincipalState` queda así:

```
@override
Widget build(BuildContext context) {
  // 4
  return Scaffold(
    appBar: AppBar(
      title: const Text('Conversor de unidades'),
      backgroundColor: Theme.of(context).colorScheme.inversePrimary,
    ),
    // 5
    body: SafeArea(
      child: Center(
        child: SingleChildScrollView(
          child: Padding(
            padding: const EdgeInsets.all(16.0),
            child: Column(
              // 7
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                // 6
                Text(
                  'Valor a convertir',
                  style: Theme.of(context).textTheme.labelLarge,
                ),
                // 9
                TextField(
                  textAlign: TextAlign.center,
                  keyboardType: const TextInputType.numberWithOptions(
                    signed: true,
                    decimal: true
                  ),
                  onChanged: (texto) {
                    var valorIntroducido = double.tryParse(texto);
                    setState(() {
                      _valorAConvertir = valorIntroducido ?? 0.0;
                    });
                  },
                ),
                // 11
                const SizedBox(
                  height: 32.0,
                ),
                Text(
                  'Convertir de',
                  style: Theme.of(context).textTheme.labelLarge,
                ),
                DropdownButton<String>(
                  value: _conversiones[_idConversionActual],
                  items: _conversiones.map((String value) {
                    return DropdownMenuItem<String>(
                      value: value,
                      child: Text(value),
                    );
                  }).toList(),
```

```
                  onChanged: (nuevoValor) {
                    setState(() {
                      _idConversionActual = _conversiones.indexOf(nuevoValor!);
                    });
                  },
                ),
                // 14
                const SizedBox(
                  height: 32.0,
                ),
                Text(
                  'Resultado',
                  style: Theme.of(context).textTheme.labelLarge,
                ),
                const SizedBox(
                  height: 16.0,
                ),
                Text(
                  '$_valorConvertido',
                  style: Theme.of(context).textTheme.headlineLarge,
                ),
                const SizedBox(
                  height: 32.0,
                ),
                // 15
                ElevatedButton(
                  child: const Text(
                    'Convertir',
                  ),
                  onPressed: () {
                    // 17
                  },
                ),
              ],
            ),
          ),
        ),
      ),
    ),
  );
}

// 16
```

8 – Escribir el modelo de la aplicación

Ya que se ha probado que la interfaz de la aplicación está completa y se ve bien en diferentes tamaños de pantalla, es el momento de terminar la aplicación móvil. Para ello escribiremos el modelo de la aplicación.

Esta aplicación es tan simple, que no escribiremos una clase específica para gestionar el modelo, sino que añadiremos esta responsabilidad a la clase `_ConversorPaginaPrincipalState`.

Al final de la clase `_ConversorPaginaPrincipalState` añadiremos el siguiente método privado:

```
// 16
void _convertir() {
  double constanteConversion = 1.0;
  switch(_idConversionActual) {
    case 0: // Kilómetros a metros
      constanteConversion = 1000.0;
      break;
    case 1: // Metros a kilómetros
      constanteConversion = 0.001;
      break;
    default:
      break;
  }
  setState(() {
    _valorConvertido = _valorAConvertir * constanteConversion;
  });
}
```

Lo más importante de este método es acordarse de modificar el valor de la propiedad `_valorConvertido` dentro de una función anónima que se pasa al método `setState` de la clase `State`. Si no se hace así, aunque se modifique el valor de la propiedad `_valorConvertido`, la interfaz no se refrescará y por tanto se seguirá mostrando el valor antiguo.

Por último, hay que conectar el método `_convertir` con la interfaz de usuario. En concreto hay que conseguir que al pulsar el botón `Convertir` se llame a este método.

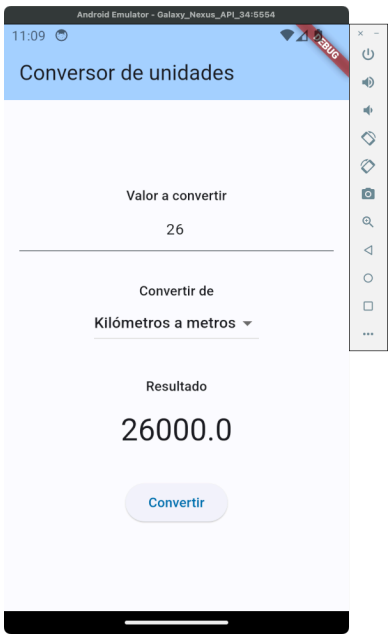
El proceso es sencillo. Solo hay que buscar la línea:

```
// 17
```

dentro de `ElevatedButton` y sustituirla por la línea:

```
// 17
_convertir();
```

Con ello la aplicación está completamente terminada y es funcional. Podemos probarla haciendo algunas conversiones.



¿Es realmente una buena aplicación?

Tal y como está actualmente, la aplicación es completamente funcional y hace lo que el usuario necesita, pero ¿es realmente una buena aplicación?

Para responder a esta pregunta hay que fijarse en dos aspectos importantes:

- ¿Proporciona a su usuario la mejor experiencia de uso (UX)? ¿Hay formas mejores que las que ofrece la aplicación, de organizar la interacción con el usuario? ¿Hay pasos que son superfluos y que por tanto podrían eliminarse para facilitar el número de acciones que debe hacer el usuario para completar una tarea?
- ¿El código de la aplicación está bien diseñado? Para responder a esta pregunta lo mejor es fijarse en si acepta bien los cambios. Si para añadir alguna nueva funcionalidad a la aplicación hay que realizar cambios en muchos módulos distintos ya programados, es una mala aplicación.

> **Atención, spoiler:** Esta aplicación no proporciona la mejor experiencia de uso y su código tampoco está bien diseñado. Has los ejercicios que se proponen a continuación y aprenderás mejores formas de organizar la interacción y el código.

Mini ejercicios

1. ¿Cuál es la diferencia entre widgets con estado `StatefulWidget` y widgets sin estado `StatelessWidget`? Justifica tu respuesta.
2. ¿Por qué la clase privada, subclase de `State`, asociada a una vista `StatefulWidget` se escribe en el mismo archivo? Justifica tu respuesta.

3. ➲ Analiza la aplicación de conversión. ¿Cómo es la experiencia de uso que proporciona al usuario?

   > **Pista:** ¿Crees que el botón `Convertir` es realmente necesario? ¿La sería mejor o peor sin él? Justifica tu respuesta.

4. ➲ Rediseña la aplicación para que no sea necesario el botón `Convertir`. Programa la aplicación rediseñada sin ese botón. ¿Qué problemas has encontrado? ¿Cómo los has resuelto?
5. ➲ Añade otra conversión de tu elección a la aplicación (por ejemplo metros a centímetros). Explica los pasos que has tenido que seguir. ¿En cuántos módulos, métodos, etc. distintos has tenido que hacer cambios?
6. ➲ Basándote en la respuesta al ejercicio anterior, ¿es una buena aplicación? ¿su código está bien diseñado?
7. ➲ Refactoriza la aplicación para que tenga un código mejor diseñado.

   > **Pista:** Los tipos enumerados en Dart son muy potentes. Úsalos para rediseñar la aplicación.

8. ➲ Usa la versión refactorizada de la aplicación para añadir una nueva conversión (por ejemplo, pulgadas a centímetros) ¿En cuántos módulos, métodos, etc. distintos has tenido que hacer cambios? ¿El código está bien diseñado ahora?

Bonus: Técnica de programación: Refactorizar

Cuando descubramos que nuestro código no está estructurado de una manera conveniente para añadirle nueva funcionalidad. Antes de comenzar a añadir la nueva funcionalidad hay que **refactorizarlo** para que sea fácil añadirla. Solo entonces la añadiremos. Cualquier otro enfoque nos traerá problemas en el futuro.

> **Importante:** La técnica de refactorización es muy importante ya que ninguno de nosotros, por mucha experiencia que tengamos, vamos a diseñar código 100% adaptado a todos los cambios futuros. Siempre vamos a descubrir pequeñas cosas que hacen que las nuevas funcionalidades no se integren del todo bien con las actuales.

Es una técnica cuyo estudio profundo se sale del ámbito de una asignatura de desarrollo de aplicaciones móviles, pero en esta sección voy a dar unas pinceladas sobre cómo funciona. En clase de prácticas enseñaré una solución al problema. No faltes.

> **Definición de refactorización**
> Cambio realizado en la estructura interna del software para que sea más fácil de entender y más sencillo de modificar **sin cambiar su comportamiento.**

Lo importante de la refactorización es que se cambia la estructura interna del código, pero no se cambia su funcionalidad o comportamiento. Se trata de que siga haciendo lo mismo, pero estructurado de forma que las nuevas funcionalidades se integren con las actuales sin que haya que modificar estas últimas.

> **Muy importante:** La forma correcta de refactorizar un programa es cambiarlo en pequeños pasos. Nunca hay que intentar cambiarlo todo de golpe. Así, si se comete un error, es fácil encontrar dónde está.