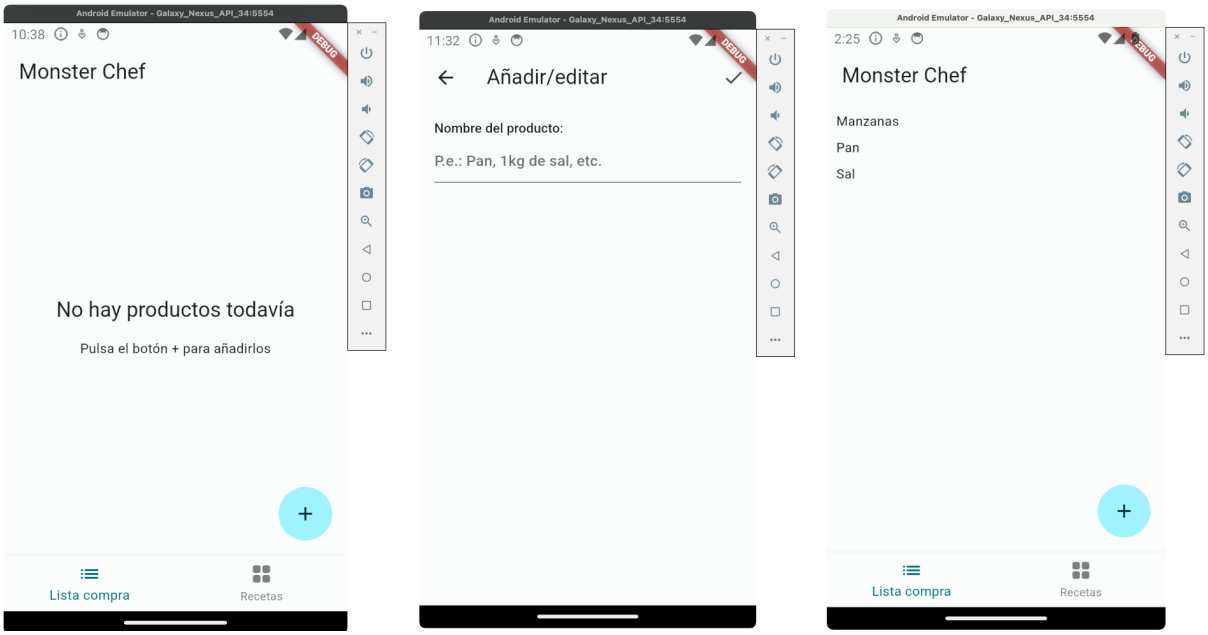
Flutter 2: Lista de la compra

Parte 1

Contenidos

¿Qué voy a hacer en esta sesión?

En esta sesión comenzaremos a escribir una aplicación para gestionar una lista de la compra. Es parte de una aplicación más grande llamada **Monster Chef,** en la que también se podrán consultar recetas de cocina. En la siguiente sesión de prácticas terminaremos de escribir la lista de la compra y en la siguiente terminaremos la aplicación Monster Chef, con la capacidad de visualizar recetas. El aspecto de la aplicación al terminar esta sesión es este:



Como vemos aún no estará completa, pero mientras la escribimos paso a paso habremos aprendido:

- A usar el patrón de interacción llamado **Barra de navegación** que en Flutter se programa mediante una vista `BottomNavigationBar` gestionada por la vista `Scaffold` que permite organizar los elementos principales de la interfaz de usuario de una aplicación móvil.
- A usar el patrón de interacción llamado **Información anidada** que en Flutter se programa mediante las clases `Navigator` y `MaterialPageRoute`
- A usar el patrón de diseño de software llamado **Observador** y la forma en que se implementa en Flutter: **Provider – Consumer.**
- A utilizar los **modos claro y oscuro del sistema operativo** en la aplicación de forma que se adapte al modo seleccionado por el usuario en su dispositivo móvil.
- A gestionar el contenido de una vista `TextField` de forma más flexible que la aprendida en la sesión anterior, mediante un `TextEditingController`
- A añadir paquetes a una aplicación para aumentar su funcionalidad. En flutter se gestionan como **dependencias.**
- A utilizar el paquete `Uuid` para asignar identificadores únicos para los distintos objetos de la aplicación.
- A utilizar entre otros los widgets de Flutter: `FloatingActionButton` y `ListView`.

Pasos previos

Crea una nueva aplicación Flutter llamada `flutter_sesion_2_4` dentro de la carpeta `flutter_projects`

> **Note**
>
> El nombre `flutter_session_2_4` hace referencia a que en esta segunda sesión vamos a comenzar a escribir una aplicación que continuaremos en la sesión 3 y terminaremos en la sesión 4.

Comenzaremos a trabajar a partir de una aplicación básica funcional pero vacía. Para ello:

1. Arranca Android Studio, abre la recién creada aplicación `flutter_session_2_4` y borra el contenido del archivo `main.dart` para dejar solo una función `main` vacía. Como sabemos, Flutter rellena el archivo `main.dart` con algunas clases. Eso no es una buena idea, cada clase debería estar en su propio archivo.
2. A continuación escribe una clase nueva llamada `MonsterChefApp` que sea subclase de `StatelessWidget` en su propio archivo, al que llamaremos `monster_chef_app.dart`, siguiendo las reglas para nombrar clases y archivos de Flutter. En esa clase hay que redeclarar el método `build`. Comienza devolviendo una instancia de la clase `Placeholder`
3. A continuación, invoca la ejecución de `MonsterChefApp` desde la función `main` del programa.
4. Termina modificando el método `build` de la clase `MonsterChefApp` para que devuelva una instancia de `MaterialApp`. Usa el parámetro con nombre `title:` del constructor de esta clase para pasar la cadena de caracteres `'Monster Chef Application'` como título de la aplicación y el parámetro `home:` para asignar una instancia de la clase `Placeholder`

   Usa también el parámetro `theme:` para asignar como tema visual de la aplicación uno basado en un `seedColor` de color blanco `white`

> **Tip**
>
> Si no te acuerdas de cómo hacer algo de esto, puedes consultar la documentación de la sesión anterior (parte 1 y 2) El proceso que necesitamos aquí es exactamente el mismo que el que se describe en las primeras secciones de esa documentación.

Escribir la aplicación lista de la compra

1 – Esqueleto de la página principal de la aplicación

Añade un nuevo archivo llamado `monster_chef_pagina_principal.dart` a la carpeta `lib` del proyecto y en él declara una clase llamada `MonsterChefPaginaPrincipal` que sea subclase de `StatefulWidget`.

> **Note**
>
> Recuerda que la forma más sencilla de escribir este tipo de clase es escribir `stful` al comienzo del archivo y pulsar `Enter`, a continuación escribir `MonsterChefPaginaPrincipal` y volver a pulsar `Enter`

> **Warning**
>
> Recuerda también escribir el `import` correspondiente para eliminar los errores de compilación.

Por defecto, el método `build` de `_MonsterChefPaginaPrincipalState` así construido, devuelve una vista de clase `Placeholder`. Cámbialo para que devuelva una vista de clase `Scaffold` básica como esta:

```
return Scaffold(
  appBar: AppBar(
    title: Text(
      widget.titulo,
    ),
  ),
  body: const SafeArea(
    // 2
    child: Center()
  ),
  // 1
);
```

> **Note**
>
> En esta ocasión hemos modificado la clase `MonsterChefPaginaPrincipal` para que tenga una propiedad llamada `titulo` que se le pasa al construir la clase en el constructor, y a la que se accede desde `_MonsterChefPaginaPrincipalState` a través de la propiedad `widget` de la clase `State`. El código completo no se muestra, intenta escribirlo tú mismo aplicando lo que sabes de propiedades, constructores y parámetros en Dart.
>
> Si no se te ocurre cómo, siempre puedes escribir la cadena `'MonsterChef'` como argumento de la vista `Text`

Una vez escrito el esqueleto de la página principal de la aplicación Monster Chef, es el momento de conectarla con la vista `MonsterChefApp`. Una de las líneas de código del método `build` que construye esta vista, es:

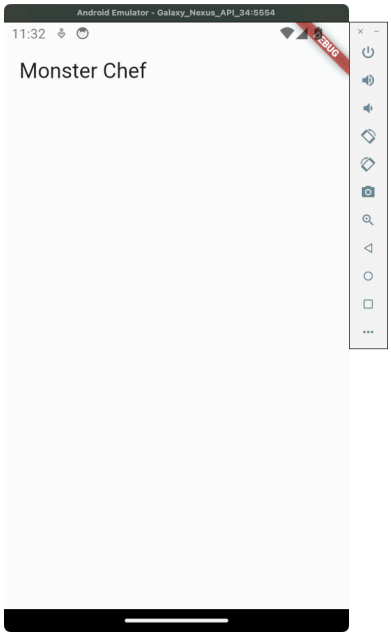
```
home: const Placeholder(),
```

Cámbiala por:

```
// 5
home: MonsterChefPaginaPrincipal(titulo: "Monster Chef",),
```

> **Note**
>
> Si no has modificado el constructor de la clase `` `MonsterChefPaginaPrincipal `` como se indica en la nota anterior, usa el constructor por defecto.

Si se compila y ejecuta la aplicación en un emulador, se obtiene una ventana como esta:



2 – Añadir la barra de navegación

En esta aplicación usaremos dos categorías, una donde consultar recetas y otra donde gestionar una lista de la compra. Por eso usaremos el patrón de interacción llamado **barra de navegación** para gestionar la navegación por esas dos categorías.

> **Note**
>
> El patrón de interacción llamado: **barra de navegación,** consiste en un rectángulo de pequeña altura situado en la parte inferior de la pantalla que incluye botones para acceder a las categorías principales de la aplicación. Es un **patrón de navegación primaria persistente.** Lo normal es que en los botones se muestre un icono representativo de cada categoría y además debajo se incluya un texto con un nombre explicativo.
>
> No se deben usar iconos conocidos por los usuarios, para denotar otra funcionalidad que no sea la que se asocia habitualmente a esos iconos. Eso dificultaría la comprensión del funcionamiento de la aplicación y empobrecería la experiencia de uso.

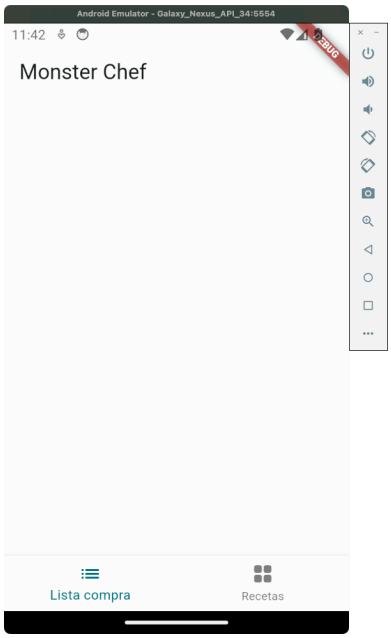
En Flutter, el patrón barra de navegación se programa añadiendo la propiedad `bottomNavigationBar:` a la vista `Scaffold`. A esta propiedad hay que pasarle una vista de clase `BottomNavigationBar`. Para ello, en el archivo `monster_chef_pagina_principal.dart` busca el comentario:

```
// 1
```

Y sustitúyelo por las líneas de código:

```
// 1
bottomNavigationBar: BottomNavigationBar(
  // 3
  items: const <BottomNavigationBarItem>[
    BottomNavigationBarItem(
      icon: Icon(Icons.list),
      label: 'Lista compra',
    ),
    BottomNavigationBarItem(
      icon: Icon(Icons.grid_view_rounded),
      label: 'Recetas',
    ),
  ],
),
```

Si hacemos hot reload, veremos que en la parte inferior de nuestra aplicación ha aparecido una barra de navegación con dos botones con sus correspondientes iconos y etiquetas:



Los botones no son funcionales todavía, si los pulsamos, no hacen nada, pero antes de seguir completando la aplicación veamos algunos aspectos interesantes:

1. La vista `Scaffold` es la que organiza en Flutter las zonas principales de la interfaz de usuario. En este caso, la barra de navegación. Para añadir una barra de navegación a la aplicación solo hay que usar la propiedad `bottomNavigationBar:` de la vista `Scaffold`

2. Una de las propiedades de la clase barra de navegación es `items:` que como su nombre indica espera una lista de vistas de clase `BottomNavigationBarItem`. Hay que añadir uno por cada botón que se quiera colocar en la barra de navegación (uno por cada categoría principal de la aplicación).
3. Dentro de cada `BottomNavigationBarItem` hay dos propiedades: `icon:` donde se define el icono que se muestra en el botón, y `label:` donde se muestra la etiqueta asociada al botón.

   > **Important**
   >
   > Las guías de estilo, tanto de Android, como de iOS, recomiendan que se usen iconos y etiquetas de texto para los botones de las barras de navegación, por tanto **siempre deberías usar las dos propiedades** en cada `BottomNavigationBarItem`

4. La lista de `BottomNavigationBarItem` se hace constante ya que los botones de esa barra siempre se van a ver igual, no cambian, por tanto si se hacen constantes, Dart los va a gestionar como **instancias canónicas** y no se van a reconstruir en memoria cada vez que se refresque la interfaz.

   > **Note**
   >
   > Cada vez que un usuario toque un botón de la barra de navegación se tiene que refrescar la interfaz. Son muchas operaciones de refresco. Todos los recursos que se puedan ahorrar en creación/destrucción de objetos en memoria ayudan a hacer mas eficiente la aplicación.

3 – Conectar botones y pantallas

La aplicación tiene una barra de navegación, pero todavía no funciona. Si se pulsa en la categoría `Recetas` no ocurre nada. Es normal ya que aún no hemos construido las pantallas asociadas a la lista de la compra y a las recetas y no las hemos conectado con la barra de navegación.

Escribir la clase con el esqueleto de la pantalla de la lista de la compra

Añade un nuevo archivo a la carpeta `lib` del proyecto llamado: `lista_compra_pantalla.dart` y dentro de él escribe la clase `ListaCompraPantalla` como una subclase de `StatelessWidget`. Debería quedar así:

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

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

  @override
  Widget build(BuildContext context) {
    // 4
    return const Center(
      child: Text('Lista de la compra 🛒'),
    );
  }
}
```

Escribir la clase con el esqueleto de la pantalla de la lista de recetas

De forma similar, añade un nuevo archivo a la carpeta `lib` del proyecto llamado: `lista_recetas_pantalla.dart` y dentro de él escribe la clase `ListaRecetasPantalla` como una subclase de `StatelessWidget`. Debería quedar así:

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

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

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Text('Galería de recetas 🥩'),
    );
  }
}
```

Establecer la conexión

Queremos que cada vez que el usuario pulse en uno de los botones de la barra de navegación, se muestre la pantalla adecuada a la categoría pulsada. Para ello asociaremos las pantallas de la lista de la compra y de la lista de recetas, a la pantalla principal de la aplicación. De esta forma, si se pulsa una categoría, la aplicación podrá mostrar una pantalla y si se pulsa la otra categoría, la aplicación mostrará la otra pantalla.

En la clase `_MonsterChefPaginaPrincipalState` añade la siguiente propiedad:

```
int _categoriaActiva = 0;
```

A continuación, justo debajo, añade una propiedad de clase (`static`) con la lista de páginas asociadas a cada categoría y que se han escrito en la sección anterior:

```
static var paginas = <Widget>[
  ListaCompraPantalla(),
  ListaRecetasPantalla(),
];
```

> ⚠ **Warning**
>
> Acuérdate de añadir los imports de los archivos `lista_compra_pantalla.dart` y `lista_recetas_pantalla.dart` o de lo contrario tendrás errores de compilación.

Ahora busca las líneas de código:

```
// 2
child: Center()
```

Y sustitúyelas por:

```
// 2
child: paginas[_categoriaActiva],
```

> ⚠ **Warning**
>
> Ahora, ya no puedes seguir usando la palabra reservada `const` delante de `SafeArea` así que tienes que eliminarla o tendrás errores de compilación.

Lo siguiente es añadir el siguiente método privado a la clase `_MonsterChefPaginaPrincipalState` para gestionar el cambio de estado que provoca el cambio de pantalla:

```
void _alPulsar(int indice) {
  setState(() => _categoriaActiva = indice);
}
```

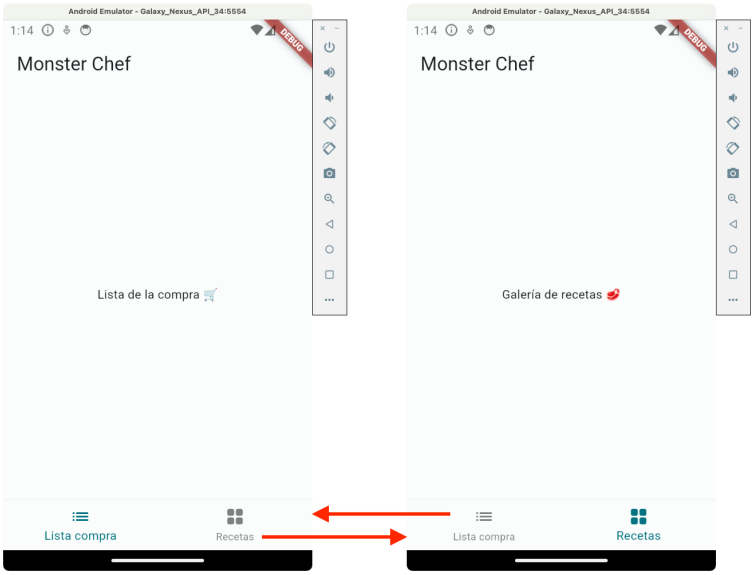
Y por último hay que conseguir que se llame a ese método cada vez que se pulse uno de los botones de las categorías. Para ello buscaremos el comentario;

```
// 3
```

Y sustitúyelo por las siguientes líneas de código:

```
// 3
currentIndex: _categoriaActiva,
onTap: _alPulsar,
```

Si hacemos Hot restart, vemos que ahora la barra de navegación es completamente funcional. Al pulsar en las categorías, se muestra la pantalla asociada a cada una de ellas. Por ahora es una pantalla que solo contiene un mensaje, pero eso lo arreglaremos en próximas secciones.



Antes de seguir veamos algunos aspectos interesantes del código que acabamos de escribir:

1. Hemos declarado una propiedad privada `_categoriaActiva` que como su nombre indica, contendrá el índice de la categoría de navegación que se haya seleccionado en ese momento. Como valor inicial le daremos el valor `0`.

   Forma parte del estado mutable de la clase `MonsterChefPaginaPrincipal` por tanto, cualquier cambio que se le haga deberá estar dentro de una llamada al método `setState` que hereda la clase `_MonsterChefPaginaPrincipalState`

2. También hemos declarado una lista llamada `paginas`, con todas las vistas que definen el aspecto de las pantallas asociadas a cada categoría. En este aplicación son dos, una pantalla para la lista de la compra y otra para la galería de recetas. Las páginas se han colocado en la lista ordenadas en el mismo orden que los botones que le dan acceso en la barra de navegación. En la posición `0` de la lista está la pantalla asociada al primer botón, etc.

   > **Important**
   >
   > La lista `paginas` se podría haber declarado como una propiedad de instancia en lugar de como una propiedad de clase (`static`) ya que en todo momento solo va a haber en memoria una instancia de la página principal. ¿Adivinas por qué se ha optado por hacerla de clase? ¿Qué líneas de código deberías escribir y dónde las escribirías si quisieras demostrar la razón por la que se ha declarado así?

3. En la propiedad `body:` de la vista `Scaffold` pasamos `paginas[_categoriaActiva]`. Esto hará que cada vez que se redibuje la vista `MonsterChefPaginaPrincipal`, su estado asociado `_MonsterChefPaginaPrincipalState` dibujará como cuerpo en la vista `Scaffold`, bien una pantalla de tipo `ListaCompraPantalla`, bien una pantalla de tipo `Lista`
4. `RecetasPantalla`, en función del valor de la propiedad `_categoriaActiva`.
5. El método `_alPulsar` será llamado cada vez que se pulse uno de los botones de la barra de navegación. Se le pasa como argumento un `indice`, que indica cuál de los botones es el que se ha pulsado.

   En el interior del método, hay que cambiar el valor de la variable de estado `_categoriaActiva` asignándole el valor del parámetro `indice` pero como se trata del estado mutable de la vista, debe hacerse dentro de una llamada a `setState` o de lo contrario Flutter no se enterará de que se ha modificado el estado y no refrescará la interfaz para mostrar el cambio.
6. Usamos la propiedad `currentIndex:` de la clase `BottomNavigationBar` para indicarle a la barra de navegación que la categoría activa en un momento dado viene determinada por el valor de la propiedad `_categoriaActiva`
7. Usamos la propiedad `onTap:` de la clase `BottomNavigationBar` para indicarle a la barra de navegación qué función es la que debe ejecutar cada vez que el usuario pulse alguno de los botones de esa barra. Pasaremos el método privado `_alPulsar`.

4 – Temas claro y oscuro

Ahora, antes de seguir completando la pantalla de lista de la compra, vamos a hacer un pequeño inciso para aprender a gestionar **los temas claro y oscuro en nuestra aplicación.**

A partir de la versión 12 de iOS y de la versión 10 de Android, se puede escoger entre un tema claro y uno oscuro a nivel del Sistema Operativo. El tema claro es más adecuado cuando se usa el móvil de día con más luz natural y el tema oscuro lo es cuando se usa de noche con poca iluminación. Hacer que los colores de la interfaz de nuestra aplicación Flutter se adapten a esos cambios de tema en el Sistema Operativo es muy sencillo.

En la clase `MonsterChefApp` tienes las siguientes líneas de código:

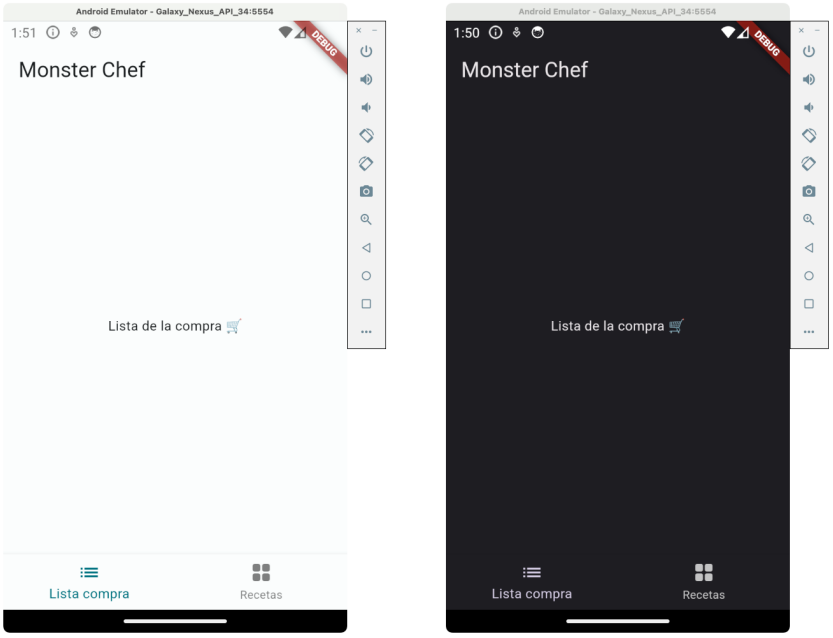
```
theme: ThemeData(
  colorScheme: ColorScheme.fromSeed(seedColor: Colors.white,),
  useMaterial3: true,
),
```

Definen el aspecto visual general de la aplicación, en este caso usando el color blanco para fondos, etc.

Sustitúyelas por las siguientes líneas de código:

```
theme: ThemeData(
  colorScheme: ColorScheme.fromSeed(seedColor: Colors.white,),
  brightness: Brightness.light,
  useMaterial3: true,
),
darkTheme: ThemeData.dark(),
themeMode: ThemeMode.system,
```

Ahora, si haces Hot Reload, verás que al cambiar el tema en el sistema operativo del dispositivo `Settings → Display → Appearance` el aspecto de la aplicación cambia adaptándose al modo elegido:



Veamos cómo funciona:

1. Declaramos el tema general de la aplicación, usando la propiedad `theme:` a la que se le pasa una instancia de la clase `ThemeData` con las características que se quieran para el tema. En esta aplicación sólo hemos usado la propiedad `colorScheme:` para indicar que el color general de la aplicación es `white`.

   Además de las características visuales del tema, hay que usar otras dos propiedades de `ThemeData`:

   - `brightness:` que indican a Flutter si el tema que se está usando corresponde al modo claro u oscuro. En este caso, estamos definiendo las propiedades del tema claro, por lo que se pasa la constante `light` del tipo enumerado `Brightness`.
   - `useMaterial3:` con valor `true` para indicar a Flutter que se quiere usar la versión 3 de **Material Design.**

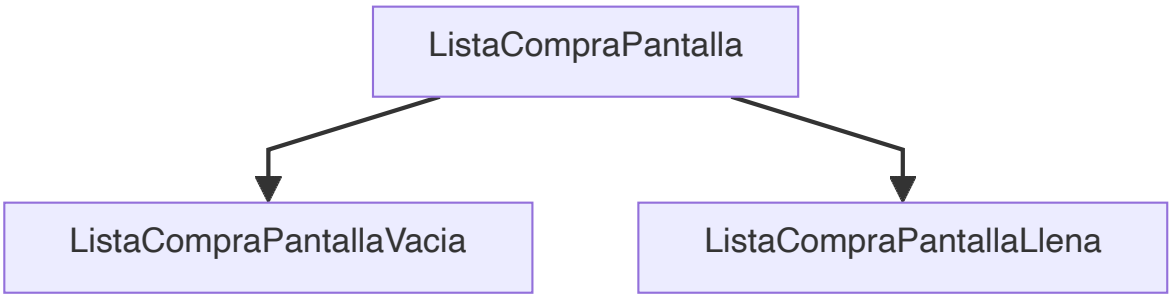
   > **Note**
   >
   > La clase `ThemeData` contiene muchas otras propiedades que permiten personalizar todos y cada uno de los aspectos de la apariencia visual de tu aplicación. Pídele a Android Studio que te muestre la ayuda de esa clase y experimenta con otras configuraciones.

2. Además, se ha usado la propiedad `darkTheme:` con la que se indica el tema a utilizar cuando se está en modo oscuro. Podríamos haber usado la clase `ThemeData` para definirlo, de forma similar a como se ha declarado el tema claro, para ello deberíamos haber usado `Brightness.dark`. Sin embargo en este ejemplo hemos usado otro enfoque: llamar al método `dark()` de la clase `ThemeData` que devuelve automáticamente la versión oscura del tema actualmente definido.
3. Por último, para que el tema de la aplicación refleje el tema seleccionado en el dispositivo, se ha usado la propiedad: `themeMode:` pasándole como valor la constante `system` del tipo enumerado `ThemeMode` que indica a Flutter que el modo lo tome del que haya definido en el sistema operativo del teléfono. Así, cuando el usuario en su móvil seleccione el tema oscuro, la interfaz de la aplicación Monster Chef cambiará automáticamente a este tema y viceversa.

5 – Estructura de las pantallas

La pantalla para la lista de la compra es especial. Debe mostrar una pantalla vacía con indicaciones para el usuario cuando no haya productos en la lista de la compra y una pantalla en la que se muestren los productos de la lista de la compra cuando la lista no esté vacía.

El siguiente diagrama ilustra esta situación:



Veamos un resumen de los pasos que deberemos dar para implementarlo en nuestra aplicación:

- Además de la clase `ListaCompraPantalla` deberemos escribir otras dos clases, `ListaCompraPantallaVacia` que se mostrará cuando no haya elementos en la lista y `ListaCompraPantallaLlena` que se mostrará cuando la lista tenga productos.
- La decisión de cuál de las dos pantalla se muestra depende del estado de la lista de la compra y la toma la clase `ListaCompraPantalla`.

  El problema es distinto al que hemos gestionado hasta ahora, donde el estado mutable y la pantalla que refleja los cambios en ese estado estaban dentro de la misma clase.

  Ahora, el estado mutable está dentro de la lista de la compra, que es una clase del **modelo de la aplicación** (la parte de la aplicación que gestiona los datos y algoritmos que los procesan). El modelo se usa por varias pantallas por tanto no puede formar parte de `ListaCompraPantalla` ya que si lo hiciéramos así, las demás estarían acopladas con ella. Este caso requiere otra forma para gestionar el estado.

  > **Important**
  >
  > En Flutter se usa el patrón de diseño de software llamado **estado** para gestionar esta situación. Lo programaremos un poco más adelante en esta sesión de prácticas.

- También hará falta un botón para añadir nuevos productos a la lista de la compra. Este botón se debe mostrar tanto en la pantalla `ListaCompraPantallaVacia` como en la pantalla `ListaCompraPantallaLlena`, y en ambas pantallas se comporta exactamente de la misma forma. No podemos repetirlo o estaríamos incumpliendo el principio de diseño de software DRY (Don't Repeat Yourself), por eso, será un elemento de la clase `ListaCompraPantalla` y las otras dos pantallas se mostrarán dentro de ella, como subelementos de `ListaCompraPantalla`.
- También se necesitará una pantalla para añadir los nuevos productos a la lista de la compra. A esta pantalla se accederá tanto desde la pantalla `ListaCompraPantallaVacia` como desde la pantalla `ListaCompraPantallaLlena` mediante el patrón de navegación: **Información anidada.**

  > **Note**
  >
  > El patrón de interacción llamado: **información anidada** o nested doll, lleva a los usuarios de forma lineal desde un contenido más general hasta un contenido más detallado. Para ello sólo hay que pulsar en el contenido general. Es un **patrón de navegación secundaria.** Se suele utilizar en combinación con otros patrones de navegación primarios como las barras de navegación o las galerías.
  >
  > Debe usarse junto con el patrón de interacción **botón atrás** o back button, que es un botón con un icono en forma de flecha situado en la esquina superior izquierda de la pantalla, que permite al usuario volver a la pantalla anterior.
  >
  > Se usa cuando un usuario visita los detalles de algún elemento que se encuentra en una pantalla anterior (información anidada).

Ahora que tenemos claro el diseño que hay que implementar, es el momento de comenzar a implementarlo.

> **Tip**
>
> **Nunca** comiences a programar una parte de una aplicación sin saber qué es lo que vas a programar, cómo lo vas a programar y por qué.

6 – Pantalla para la lista de la compra vacía

Añade un nuevo archivo llamado: `lista_compra_pantalla_vacia.dart`, a la carpeta `lib` del proyecto Flutter. Dentro, declara una clase llamada `ListaCompraPantallaVacia` y haz que sea subclase de `StatelessWidget`. Es la vista que se mostrará cuando no haya productos en la lista de la compra. Es una pantalla que muestra un texto explicativo en el centro, con indicaciones para el usuario sobre cómo añadir productos. Es una subclase de `StatelessWidget` ya que no va a cambiar en toda la ejecución de la aplicación. El texto explicativo será siempre el mismo.

Modifica el método `buid` de esa clase para que devuelva lo siguiente:

```
return Center(
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text(
        'No hay productos todavía',
        style: Theme.of(context).textTheme.titleLarge,
      ),
      const SizedBox(height: 16.0,),
      Text(
        'Pulsa el botón + para añadirlos',
        textAlign: TextAlign.center,
        style: Theme.of(context).textTheme.bodyMedium,
      ),
    ],
  ),
);
```

Veamos algunas consideraciones sobre este código:

1. Se usan diferentes tamaños para los textos de esta pantalla, pero no se especifica el tamaño o grosor del tipo de letra numéricamente, sino que se utilizan las constantes definidas en el tema general de al aplicación. De esta forma, si en una próxima versión de la aplicación decidimos definir nuestros propios estilos de texto, no habrá que ir cambiando los ajustes en todos los textos que se hayan usado en la aplicación.

   Como sabemos, el tema general de la aplicación es una propiedad de la clase `MaterialApp` que se usa dentro de la clase de la aplicación `MonsterChefApp`. Esa clase es el origen del árbol de vistas. La clase `ListaCompraPantallaVacia` es un nodo hoja dentro de ese árbol.

   Para acceder al tema de la aplicación hay que usar:

   ```
   Theme.of(context)
   ```

   que accede al tema a través del **contexto común compartido** por todas las vistas del árbol de vistas. Ese contexto se pasa a cada vista hija al ejecutar su método `build(BuildContext: context)`. A través de ese tema de la aplicación se puede acceder a todos los ajustes. En este caso se han usado las propiedades del tema de texto `titleLarge` para la primera línea de texto y `bodyMedium` para la segunda.

2. Se ha usado una instancia de la clase `SizedBox` para añadir algo de separación entre los dos textos. Aprendimos a utilizarlo en la sesión anterior. Como siempre, es una vista que no va a cambiar durante toda la ejecución de la aplicación, además sus ajustes se conocen en tiempo de compilación, por eso se ha declarado `const` lo que la hace una instancia canónica que solo se construye una vez en memoria.

7 – Mostrar la pantalla para la lista de la compra vacía

Ya se ha programado la clase `ListaCompraPantallaVacia` ahora es el momento de mostrarla al usuario como un subelemento dentro de la pantalla `ListaCompraPantalla`.

En la clase `ListaCompraPantalla` busca el comentario:

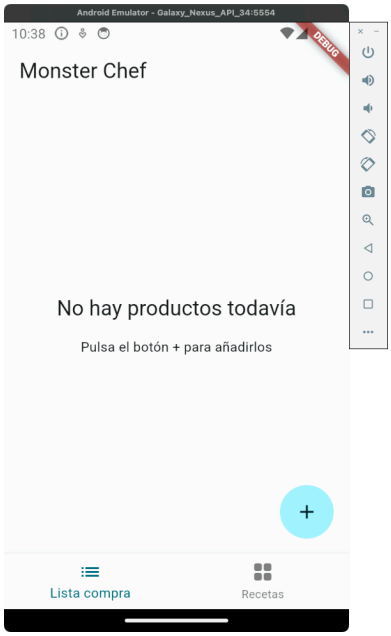
```
// 4
return const Center(
  child: Text('Lista de la compra 🛒'),
);
```

Y sustitúyelo por las siguientes líneas de código:

```
// 4
return Scaffold(
  // 6
  body: const ListaCompraPantallaVacia(),
  floatingActionButton: FloatingActionButton(
    elevation: 0,
    shape: const CircleBorder(),
    // 7
    onPressed: () {
    },
    child: const Icon(Icons.add),
  ),
);
```

> ⚠ **Warning**
>
> No olvides incluir los imports necesarios o tendrás errores de compilación.

Si hacemos Hot Restart, en el emulador se mostrará esta pantalla:



Veamos algunas consideraciones interesantes sobre el código que acabamos de escribir:

1. Dentro del método `build` de la vista `ListaCompraPantalla` se devuelve una vista de tipo `Scaffold`. Además, la vista `ListaCompraPantalla` está dentro de la vista `MonsterChefPaginaPrincipal` cuyo método `build` también devuelve una vista de tipo `Scaffold`.

   Se tiene una vista `Scaffold` dentro de otra. Flutter lo permite, de forma que se pueda tener una barra de aplicación y una barra de navegación comunes a todas las pantallas, junto con FAB (Floating Action Button) u otros elementos personalizados para algunas de las pantallas. De esta forma se puede construir una interfaz muy flexible sin tener que repetir elementos en varias vistas, lo que ayuda a cumplir el principio de diseño de software DRY (Don't Repeat Yourself).

2. Para programar el botón para añadir nuevos productos a la lista de la compra se ha usado un **FAB** (Floating Action Button). En Flutter se programa a través de una vista `Scaffold` mediante la propiedad `floatingActionButton:` Esta propiedad recibe un objeto de la clase `FloatingActionButton` que tiene multitud de propiedades. Aquí se han usado tres:

   - `child:` que permite definir el contenido del FAB. Lo normal es que se pase un icono por medio de una vista de clase `Icon`
   - `elevation:` que controla la sombra del FAB sobre la pantalla. El valor por defecto es `6` lo que proyecta una pequeña sombra. En este caso no se quiere sombra para no recargar la interfaz y se ha usado el valor `0`
   - `onPressed:` que recibe como valor una función anónima. La que realiza la tarea asignada al botón. En este caso está vacía, por eso, el botón todavía no es funcional.

> **Note**
>
> Un **botón flotante** o Floating Action Button (FAB), también llamados botones de llamada a la acción o Call To Action buttons (CTA), es un botón que destaca de los demás por su forma, color, tamaño y posición, de forma que permite resaltar una funcionalidad sobre las demás funcionalidades de una pantalla. Es un patrón de interacción de tipo **herramienta.**

> Es recomendable utilizar este patrón para tareas que realmente se destaquen de las demás en una pantalla, bien por su importancia, bien porque son las que se realizan con más frecuencia.
>
> Los FAB se suelen colocar en la parte inferior derecha de la pantalla donde son muy sencillos de pulsar con el pulgar incluso sujetando el smartphone con una sola mano.

8 – El patrón estado en Flutter

Antes de continuar con la definición de la pantalla que se usa cuando la lista de la compra tiene productos, vamos a aprender cómo gestionar el patrón estado en Flutter. En la sección 5 hablamos brevemente de este patrón de diseño de software y por qué es necesario en esta aplicación. Ahora vamos a estudiarlo en profundidad.

Para programar la lista de la compra, que es el modelo de datos de la aplicación, es necesario conocer cómo funciona el patrón estado en Flutter. Esto es así, ya que la lista de la compra es el estado mutable de las vistas `ListaCompraPantalla` y `ListaCompraPantallaLlena` y el paradigma estudiado en la sesión anterior no sirve para modelar este caso.

La lista de la compra es el **estado mutable** de la aplicación, pero a diferencia de lo que ocurría en la sesión anterior de prácticas, en la que los cambios **se producían y consumían** en la misma vista, en esta sesión los cambios se producen en la vista para añadir/modificar productos de la lista y se consumen en las vistas `ListaCompraPantalla` y `ListaCompraPantallaLlena`. Se necesita algún mecanismo para que varias vistas distintas accedan al estado mutable y para que cuando haya cambios, todas sean notificadas para poder refrescarse adecuadamente. Todo ello sin que las distintas vistas estén acopladas entre sí.

Este mecanismo es el **patrón estado,** que en Flutter se implementa mediante el mecanismo conocido como **provider - consumer.**

Hay cuatro conceptos importantes relacionados con el enfoque provider - consumer:

- `ChangeNotifier` Es la clase que en el patrón estado juega el papel de **Observable**. Es la clase que contiene el estado mutable cuyos cambios se observan desde otras clases que deben refrescarse cuando se produzcan cambios. Cuando en una aplicación se necesite un Observable, se debe construir una subclase de `ChangeNotifier`, esta subclase contendrá el estado mutable y métodos para cambiarlo. En esos métodos se debe incluir la llamada al método `notifyListeners()` heredado de la superclase `ChangeNotifier` que notifica a los Observadores que se han producido cambios en el Observable.

  > **Important**
  >
  > Los observables no son widgets. Son clases del modelo de la aplicación.

- `ChangeNotifierProvider` Es el **proveedor.** Es quien da acceso al estado mutable a las vistas que lo necesitan. Cuando se crea, en su constructor se se debe pasar una función anónima a la propiedad `create:` cuya misión es construir el `ChangeNotifier` que va a contener el estado mutable a observar y asociarlo con el contexto de la aplicación. Esto se hace en la vista principal de la aplicación (en este caso `MonsterChefApp`), de forma que el observable estará disponible para cualquier vista de la aplicación que necesite observar los cambios que se produzcan.

- `Consumer` Es la clase que en el patrón estado juega el papel de **Observador.** Su tarea es escuchar los cambios que se producen en un `ChangeNotifier`. Si se quiere que un subárbol del árbol de vistas observe los cambios en un Observable que encapsula a un estado mutable concreto, hay que incluir el subárbol de vistas dentro de una vista de clase `Consumer`. Cada vez que el estado mutable cambie, el subárbol asociado al consumer será notificado y se refrescará.

  > **Important**
  >
  > Los observadores en Flutter sí son widgets, que deben ser subclases de la clase genérica `Consumer`.

- `Provider.of` Permite acceder al estado mutable, pero sin escuchar los cambios en ese estado. Se usa en las vistas **que originan los cambios,** por ejemplo los botones de aceptar cambios. Esas vistas no reflejan el estado mutable (no se refrescan cuando cambia), por tanto no son Observadores o `Consumer`, pero sí deben tener acceso al observable porque son las que lo modifican, para, por ejemplo: añadir o eliminar elementos a la lista de la compra.

Puede parecer complejo, pero en realidad no lo es. La mejor forma de terminar de entenderlo es verlo en acción. Para ello necesitamos una biblioteca llamada `provider` a la que no se tiene acceso por defecto. Es una **dependencia** que debe añadirse al proyecto Flutter antes de poder usarla.

> **Note**
>
> En Flutter, una dependencia es un paquete o plugin que se añade al proyecto para proporcionar funcionalidades adicionales o reutilizar código. Estas dependencias se gestionan a través del archivo `pubspec.yaml`, que se encuentra en la raíz del proyecto Flutter. Para añadir una dependencia hay que escribirla en el archivo `pubspec.yaml` y a continuación ejecutar la orden:
>
> ```
> flutter pub get
> ```
>
> Que le dice a Flutter que descargue e instale la nueva dependencia dentro del proyecto.

> **Tip**
>
> Sin embargo hay otra forma más sencilla de conseguir ese mismo efecto. Para ello se usa la orden `flutter pub add [nombre_dependencia]` que busca si hay alguna dependencia con el nombre que se indica e instala la versión más moderna en el proyecto, además incluye la llamada a `flutter pub get` para descargar la dependencia y resolver las dependencias anidadas que deban ser instaladas también para que funcione la nueva dependencia. Esta forma simple será la que usaremos en estas prácticas.

Abre un terminal o símbolo del sistema en tu sistema operativo y navega hasta la **carpeta raíz de tu proyecto Flutter.** Una vez en ella ejecuta:

```
flutter pub add provider
```

Esta orden busca si existe un paquete o plugin llamado `provider`, averigua la versión más reciente, resuelve incompatibilidades con otros paquetes, edita el archivo `pubspec.yaml` y por último ejecuta `flutter pub get` para descargar la dependencia e instalarla en el proyecto.

> **Caution**
>
> Si se produce algún error, asegúrate de haber escrito correctamente el nombre de la dependencia y de que has ejecutado `flutter pub add` en el directorio raíz del proyecto.

> **Note**
>
> No es obligatorio, pero sí muy recomendable, ejecutar `flutter upgrade` antes de ejecutar `flutter pub add` para asegurarnos de que tenemos instalada la última versión de Flutter en nuestro equipo.

9 – Estado mutable para la lista de la compra

Escribir la clase para representar un producto de la lista

Una lista de la compra es básicamente una lista de productos, por eso, necesitamos una clase `Producto` para encapsular el comportamiento de un producto individual de la lista de la compra. Como siempre la escribiremos en su propio archivo.

Añade un nuevo archivo llamado `producto.dart` a la carpeta `lib` de tu proyecto Flutter, y dentro escribe el siguiente código:

```
class Producto {
  final String id;
  final String nombre;
  final Importancia importancia;
  final int cantidad;
  final bool completado;

  Producto({
    required this.id,
    required this.nombre,
    required this.importancia,
    required this.cantidad,
    this.completado = false,
  });

  Producto copiaSiNulo({
    String? id,
    String? nombre,
    Importancia? importancia,
    int? cantidad,
    bool? completado,
  }) {
    return Producto(
      id: id ?? this.id,
      nombre: nombre ?? this.nombre,
      importancia: importancia ?? this.importancia,
      cantidad: cantidad ?? this.cantidad,
      completado: completado ?? this.completado,
    );
  }
}
```

```
enum Importancia {
  baja,
  media,
  alta,
}
```

Antes de seguir veamos algunas consideraciones importantes sobre esta clase:

1. Uno de los atributos de la clase `Producto` es la importancia o prioridad que tiene ese producto dentro de la lista de la compra. Puede tener tres valores: baja, media y alta, por eso se ha optado por encapsularlos dentro de un tipo enumerado al que se ha llamado `Importancia`.

   > **Note**
   >
   > Este tipo enumerado se podría haber escrito en su propio archivo separado, pero dado que solo se usa junto con productos, se ha preferido declararlo en el archivo donde se declara la clase `Producto`.

2. Cada objeto de la clase `Producto` tiene una propiedad llamada `id` para diferenciar unos productos de otros. Es necesario para poder implementar correctamente algunas operaciones en la vista de tipo `ListView` que usaremos en la próxima sesión para mostrar la lista de productos. Poder diferenciar unos productos de otros es fundamental para mostrarlos correctamente en la interfaz de usuario.

   Más adelante usaremos el paquete `Uuid` para generar identificadores únicos de tipo `String` para cada producto.

3. Se ha añadido un constructor generativo personalizado para poder construir instancias de la clase `Producto` a partir de los datos que se introduzcan en la interfaz de usuario.

   Todos los parámetros del constructor son parámetros con nombre y requeridos, a excepción del parámetro `completado` que se ha dejado opcional. Esto es así porque cuando se da de alta un producto, no está completado y su valor siempre va a ser `false` (no tiene mucho sentido añadir un producto ya completado a una lista de la compra)

   Es posible que te preguntes, ¿por qué añadirlo como parámetro al constructor? ¿no sería mejor declarar la propiedad `completado` así?

   ```
   final bool completado = false;
   ```

   ¿y de esta forma eliminar el parámetro en el constructor?

   > **Tip**
   >
   > Mira el resto de código y enseguida descubrirás que no es posible.

4. Todas las propiedades de la clase `Producto` se han marcado como `final`, lo que significa en la práctica, que no se pueden modificar una vez que se ha construido una instancia de la clase. Cuando en la lista de productos se quiera cambiar alguna propiedad de alguno de los productos, lo que se hace es construir un nuevo producto y sustituir el anterior por éste.

5. El método de instancia `copiaSiNulo` facilita mucho la tarea de construir nuevas instancias de `Producto` basadas en otras instancias pero con algunas propiedades cambiadas. Funciona como un constructor de copia, pero mucho más potente y flexible.

   Recibe un parámetro opcional, con nombre y con capacidad de tener valores nulos, para cada propiedad de la clase.

   Es un método de instancia, por tanto, solo se puede llamar a través de un objeto de la clase `Producto`. Cuando se llama al método `copiaSiNulo` sobre un objeto de la clase `Producto`, si no se pasa nada en algún parámetro (se tiene un valor nulo) se toma el valor de propiedad del objeto, y si se pasa algún valor, se toma como valor de la propiedad ese valor. Con todos esos valores se construye un nuevo objeto `Producto` y se devuelve.

   Este método es muy útil para construir copias de productos modificando los valores de algunas de sus propiedades dejando las demás intactas. Lo usaremos para modificar productos de la lista de la compra.

   Ejemplo:

   Veamos un ejemplo. Si se hubiera construido el siguiente objeto:

   ```
   final producto = Producto(
     id: 'abc',
     nombre: 'Manzanas',
     importancia: Importancia.media,
     cantidad: 4
   );
   ```

   Y a continuación se ejecutase:

   ```
   final productoCopiado = producto.copiaSiNulo(importancia: Importancia.baja);
   ```

   el objeto resultado de la copia: `productoCopiado`, tendrá como valores de propiedad:

   ```
   id: 'abc'
   nombre: 'Manzanas'
   importancia: Importancia.baja
   cantidad: 4
   ```

   Es decir, es una copia del objeto `producto` original, que tiene los mismos valores de propiedad, excepto en la propiedad `importancia` para la que se ha pasado un valor distinto como argumento del método `copiaSiNulo`

Escribir la clase para representar la propia lista de la compra

Vamos a escribir ahora la lista de la compra, que será una lista de instancias de la clase `Producto`. Esta lista de la compra es el estado mutable de la aplicación, que como sabemos, es un **Observable** que en Flutter se programa como una subclase de `ChangeNotifier`.

Añade un nuevo archivo llamado `lista_compra.dart` a la carpeta `lib` de tu proyecto Flutter, y dentro escribe el siguiente código:

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

class ListaCompra extends ChangeNotifier {
  final _productos = <Producto>[];
  List<Producto> get productos => List.unmodifiable(_productos);

  void borraProducto(int indice) {
    _productos.removeAt(indice);
    notifyListeners();
  }

  void anadeProducto(Producto item) {
    _productos.add(item);
    notifyListeners();
  }

  void actualizaProducto(Producto item, int indice) {
    _productos[indice] = item;
    notifyListeners();
  }

  void marcaCompletado(int indice, bool completado) {
    final producto = _productos[indice];
    _productos[indice] = producto.copiaSiNulo(completado: completado);
    notifyListeners();
  }
}
```

Veamos algunas consideraciones importantes sobre esta clase:

1. `ListaCompra` es un observable, encapsula el estado mutable cuyos cambios se quieren observar desde las vistas de la aplicación, por tanto, debe ser una subclase de `ChangeNotifier`.
2. Tiene una propiedad privada llamada `_productos` que contiene la lista de la compra. Esa propiedad es el estado mutable encapsulado en este `ChangeNotifier`. Cada vez que se añada, elimine, o modifique un producto se informará a los observadores para que se refresquen y reflejen el nuevo estado.
3. Tiene un getter llamado `productos` para permitir que la lista pueda ser consultada desde otras vistas que deban refrescarse en función del contenido de la lista de la compra. Para que no se puedan hacer cambios no deseados o no controlados en la lista, el getter proporciona una copia inmutable mediante `List.unmodifiable(_productos);` De esta forma desde otras clases no se puede cambiar su longitud (añadir o borrar elementos), ni modificar los elementos existentes.
4. Se incluyen cuatro métodos para modificar la lista de la compra: `borraProducto(int indice)`, `anadeProducto(Producto producto)`, `actualizaProducto(Producto item, int indice)` y `marcaCompletado(int indice, bool completado)`. En todos ellos se tiene como última instrucción `notifyListeners();` que notifica a los observadores registrados que se han producido los cambios y por tanto deben refrescarse para mostrarlos.

> **Important**
>
> En todos los métodos que modifican el estado mutable almacenado en un Observable `ChangeNotifier` **la última línea** debe ser `notifyListeners();`. De lo contrario, los Observadores no sabrán que se han producido cambios y no se refrescarán.

10 – Proveedor de acceso a la lista de la compra

Ya que se tiene el observable que encapsula la lista de la compra, la clase `ListaCompra`, es el momento de hacerlo disponible para todas las vistas de la aplicación que quieran registrarse para ser notificadas cuando haya cambios. Eso se hace mediante un provider.

Para proporcionar acceso al `ChangeNotifier` abre el archivo: `monster_chef_app.dart`, busca las líneas de código:

```
// 5
home: MonsterChefPaginaPrincipal(titulo: "Monster Chef",),
```

Y sustitúyelas por:

```
// 5
home: MultiProvider(
  providers: [
    // 24
    ChangeNotifierProvider(create: (context) => ListaCompra(),),
  ],
  child: MonsterChefPaginaPrincipal(titulo: 'Monster Chef',),
),
```

> **Warning**
>
> Recuerda incluir los imports:
>
> ```
> import 'package:provider/provider.dart';
> import 'lista_compra.dart';
> ```
>
> O de lo contrario tendrás errores de compilación.

Veamos algunas consideraciones sobre el código anterior:

1. En la propiedad `home:` de la clase `MaterialApp` que contiene la aplicación, se añade el provider que encapsula al observable `ListaCompra` y lo hace disponible para todo el árbol de vistas de la aplicación.

   Para ello se usa la vista `Multiprovider` que tiene como propiedades una lista de `ChangeNotifierProvider` llamada `providers:` y el árbol de vistas de la aplicación `child:`

Se usa `Multiprovider` porque lo normal es que en una aplicación se observe a más de un observable, cada uno de ellos subclase de `ChangeNotifier` y encapsulado dentro de una clase `ChangeNotifierProvider`. En esta aplicación solo se tiene un observable y por eso la lista solo contiene un elemento.

2. La función anónima que se pasa en la propiedad `create:` del constructor de la clase `ChangeNotifierProvider` tiene como argumento el contexto de la aplicación, ya que los observables están disponibles para los observadores a través del contexto.
3. Los `ChangeNotifier` se crean justo en el momento de hacerlos disponibles para el árbol de vistas a través de un `ChangeNotifierProvider` aunque también se podrían haber creado antes y aquí pasar la variable que contiene al `ChangeNotifier`

> **Note**
>
> En la siguiente sesión aprenderemos a guardar la lista de la compra en el dispositivo, para poder seguir utilizándola la siguiente vez que se ejecute la aplicación. Entonces crearemos el `ChangeNotifier` antes de construir la vista y aquí solo se pasará al árbol de vistas.

Con todo esto, cada vista del árbol de vistas de la aplicación tiene acceso al observable `ListaCompra` que encapsula a la lista de la compra, y si lo necesita se puede registrar como observador `Consumer` de ese observable.

11 – Consumir los cambios en la lista de la compra

Los cambios que se producen en un Observable se consumen en un **Observador** que en el caso de Flutter se implementa mediante la clase `Consumer`. Si en una determinada vista o subárbol de vistas, se necesita estar informado de los cambios en un Observable, esa vista o subárbol debe incluirse dentro de una vista de la clase `Consumer`.

En la aplicación Monster Chef, los cambios en la lista de la compra se consumen en diferentes parte de la aplicación. Una de ellas es en la vista `ListaCompraPantalla` que debe mostrar una pantalla de tipo `ListaCompraPantallaVacia` o de tipo `ListaCompraPantallaLlena` en función de si la lista de la compra está vacía o tiene productos

Para hacer que la vista `ListaCompraPantalla` sea un Observador, abre el archivo `lista_compra_pantalla.dart` y añade el siguiente método al final de la clase `ListaCompraPantalla`:

```
Widget construirPantallaListaCompra() {
  return Consumer<ListaCompra>(
    builder: (context, manager, child) {
      if (manager.productos.isNotEmpty) {
        // 11
        return Container( color: Colors.blueGrey, );
      } else {
        return const ListaCompraPantallaVacia();
      }
    },
  );
}
```

> ⚠ **Warning**
>
> No te olvides de los imports:
>
> ```
> import 'package:provider/provider.dart';
> import 'lista_compra.dart';
> ```
>
> Ya que de lo contrario tendrás errores de compilación.

Luego busca las líneas de código:

```
// 6
body: const ListaCompraPantallaVacia(),
```

Y sustitúyelas por las siguientes líneas de código:

```
// 6
body: construirPantallaListaCompra(),
```

Veamos los aspectos relevantes de este código:

1. La vista devuelta por el método de utilidad `construirPantallaListaCompra` es de clase `Consumer` que es una clase genérica. Debe estar asociada a un `ChangeNotifier` concreto de los que se hayan puesto a disposición de la aplicación mediante la vista `MultiProvider`. En este caso es un Observador de `ListaCompra`, por eso la vista que se devuelve es de tipo: `Consumer<ListaCompra>`.
2. El efecto que tiene ese método de utilidad es que que cada vez que se realiza un cambio en la lista de la compra, se reconstruyan las vistas que se construyen dentro de la propiedad `builder:` del `Consumer`. En este caso el contenido de la pantalla para la lista de la compra.

   > **Important**
   >
   > Para ganar eficiencia, solo se deberían incluir dentro de un `Consumer` las vistas que lo necesiten. Si se incluyen vistas que no reflejan de ninguna forma los cambios en un Observable se van a reconstruir en vano, ya que su apariencia o contenido no va a cambiar.

3. La lista de parámetros que hay que escribir en la función anónima que se pasa a la propiedad `builder:` de una vista `Consumer` (un Observador) son tres:
   - `context` que es el contexto de la aplicación.
   - `manager` que es el `ChangeNotifier` al que se quiere observar.
   - `child` permite ahorrar tiempo si alguna de las vistas del subárbol de vistas no debe reconstruirse por los cambios del Observable. En ese caso, en a la propiedad `builder:` del `Consumer` se usa la propiedad `child:` en la que se construye la parte que no cambia.

     El argumento `child` en la función anónima de la propiedad `builder:` hace referencia a la vista que no cambia. En este ejemplo no hay ninguna parte que no cambie.

4. En `ListaCompraPantalla` se debe mostrar la pantalla correspondiente a la situación en la que la lista de la compra está vacía `ListaCompraPantallaVacia` o la pantalla que muestra la lista de productos a comprar cuando la lista no está vacía. Esta última pantalla aún no se ha escrito. Para comprobar si la lista está vacía o no se usa el siguiente `if`:

   ```
   if (manager.productos.isNotEmpty)
   ```

   `manager` hace referencia al Observable, que en este caso es de clase `ListaCompra`. El tipo se infiere del hecho de que el consumer está asociado a esta clase mediante `Consumer<ListaCompra>`. Una de las propiedades de `ListaCompra` es el getter `productos` que da acceso a una copia inmutable de la lista de la compra. Cuando no está vacía, `isNotEmpty` devuelve `true`, en ese caso se devuelve la pantalla para cuando hay elementos en la lista (por ahora un `Container` vacío en color gris azulado), y cuando devuelve `false` se devuelve la pantalla `ListaCompraPantallaVacia`.

En este punto ya se tiene el estado mutable (la lista de la compra), encapsulado dentro de un `ChangeNotifier` (Observable), puesto a disposición del árbol de vistas mediante un `ChangeNotifierProvider` y consumido (observado) por la vista `ListaCompraPantalla` que es un `Consumer` (Observador). Es decir, está todo listo para mostrar pantallas distintas en función de si la lista de la compra está vacía o contiene productos, pero para poder probarlo primero es necesario poder añadir productos a la lista de la compra.

12 – Pantalla para añadir productos a la lista de la compra

Lo siguiente es construir la pantalla que aparece cuando se pulsa el FAB `+` para añadir elementos. Es la pantalla para añadir productos a la lista de la compra.

Añade un nuevo archivo llamado `lista_compra_anadir_producto.dart` a la carpeta `lib` de tu proyecto Flutter, y dentro escribe el siguiente código:

```
class ListaCompraAnadirProducto extends StatefulWidget {
  final Function(Producto) crearProducto;
  final Function(Producto) editarProducto;
  final Producto? productoOriginal;
  final bool actualizando;

  const ListaCompraAnadirProducto({
    Key? key,
    required this.crearProducto,
    required this.editarProducto,
    this.productoOriginal,
  }) : actualizando = (productoOriginal != null),
       super(key: key);

  @override
  _ListaCompraAnadirProductoState createState() =>
_ListaCompraAnadirProductoState();
}
```

> **Note**
>
> También reutilizaremos esa misma pantalla para editar los productos que ya se hayan añadido a la lista de la compra (cambiar los valores de sus propiedades).

> **Warning**
>
> No te olvides de añadir los imports necesarios o tu código tendrá errores de compilación.

> **Note**
>
> El archivo no está completo. Falta declarar la clase `_ListaCompraAnadirProductoState`, por eso sigue saliendo un error de compilación en la última línea. No te preocupes. Lo añadiremos justo después de comentar el código que acabamos de escribir.

El archivo no está terminado, pero antes de seguir veamos algunos aspectos interesantes de este código:

1. La pantalla `ListaCompraAnadirProducto` se emplea tanto para añadir nuevos productos a la lista de la compra, como para editar los ya existentes. La interfaz es la misma, pero el proceso varía por lo que hay que poder indicar a la clase si se está añadiendo o editando un producto.

   Si se está editando, se le pasa como parámetro el producto que se quiere editar. Para ello se usa la propiedad `productoOriginal`. Si se está añadiendo, se pasa el valor `null`

   Por comodidad se usa una propiedad de tipo `bool` llamada `actualizando` que se inicializa automáticamente en la lista de inicialización del constructor mediante la expresión:

   ```
   actualizando = (productoOriginal != null)
   ```

2. Hay dos propiedades `crearProducto` y `editarProducto` que almacenan funciones anónimas con un único parámetro de tipo `Producto` definidas como:

   ```
   final Function(Producto) crearProducto;
   final Function(Producto) editarProducto;
   ```

   Son los callback que se ejecutarán cuando el usuario pulse el botón de aceptar ✔️ (todavía no se ha construído, eso se hace en el estado asociado).

   > **Note**
   >
   > Estas funciones se pasan desde fuera en vez de escribir su código directamente en esta clase, ya que eso permite una mayor flexibilidad. Se puede cambiar los algoritmos sin tener que tocar esta clase.

Como sabemos, toda `StatefulWidget` debe tener su correspondiente clase privada de tipo `State` asociada, para gestionar su estado mutable. Es en esta clase donde se construye la interfaz de usuario y sin ella no estaría completa.

Por esa razón, justo al final del archivo: `lista_compra_anadir_producto.dart` añade el siguiente código:

```
class _ListaCompraAnadirProductoState extends State<ListaCompraAnadirProducto> {

  final _controladorNombre = TextEditingController();
  String _nombre = '';
  bool _completado = false;
  // 12

  @override
  void initState() {
    super.initState();

    _controladorNombre.addListener(() {
      setState(() { _nombre = _controladorNombre.text; });
    });

    // 22
  }

  @override
  void dispose() {
    _controladorNombre.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        actions: [
          IconButton(
            icon: const Icon(Icons.check),
            // 8
            onPressed: () {
              Navigator.pop(context);
            },
          ),
        ],
        elevation: 0.0,
        title: const Text( 'Añadir/editar', ),
      ),
      body: Container(
        padding: const EdgeInsets.all(16.0),
        child: ListView(
          children: <Widget>[
            construyeCampoNombre(),
            // 13
          ],
        ),
      ),
    );
  }
```

```
  Widget construyeCampoNombre() {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        Text(
          'Nombre del producto:',
          style: Theme.of(context).textTheme.titleSmall,
        ),
        TextField(
          controller: _controladorNombre,
          decoration: const InputDecoration(
            hintText: 'P.e.: Pan, 1kg de sal, etc.',
          ),
        ),
      ],
    );
  }
}
```

> **Note**
>
> Como puede verse, todavía no se ha terminado esta pantalla. Por ahora, solo se puede escribir el nombre del producto. Otras propiedades de los productos como su importancia o cantidad aún no se pueden añadir. Por ahora es suficiente para probar el funcionamiento de la pantalla para añadir o editar productos.
>
> En la siguiente sesión de prácticas se terminará.

Veamos algunas consideraciones sobre este código, antes de conectar la pantalla para añadir/modificar productos con el FAB para añadir productos:

1. En la vista `AppBar` de la vista `Scaffold` de esta pantalla, se ha usado la propiedad `actions:` y no solo la propiedad `title:` como en otras aplicaciones. Esa propiedad permite añadir botones a la barra de la aplicación.

   En esta aplicación sólo se necesita un botón cuyo icono es un tick mark ✔ que permitirá aceptar, bien el nuevo producto para la lista, o bien los cambios en un producto ya existente.

   > **Important**
   >
   > Esta pantalla no se va a insertar en el árbol de vistas de la aplicación, sino que cuando se active, lo va a **sustituir.** Por eso, hay que escribir la vista `Scaffold` completa. Se va a usar el patrón **Información anidada.**
   >
   > En este patrón, la vista de detalles (la pantalla para editar productos) sustituye a la vista general (la lista de la compra), por tanto **no es un subelemento de la lista de la compra.**

2. Los botones que se pueden añadir a una `AppBar` son de clase `IconButton`. Tienen dos propiedades importantes:
   - `icon:` donde se especifica el icono asociado a este botón.

     > **Note**
     >
     > Recuerda que las guías de estilo, tanto de Android, como de iOS, recomiendan no usar texto (solo el icono) en los botones de la barra de aplicación porque permiten ejecutar **herramientas.**

- `onPressed:` a la que se pasa una función anónima. En ella se encapsula el comportamiento del botón. Es la función que se va a ejecutar cuando el usuario de la aplicación pulse ese botón.

3. En este caso concreto, la propiedad `onPressed:` del botón, llama al método `pop` de la clase `Navigator` que quita de la pantalla del dispositivo la pantalla para editar productos y vuelve a poner en su lugar la pantalla para mostrar la lista de la compra.

   Además de volver a la pantalla para mostrar la lista de la compra, hay que editar o añadir el producto. Por tanto, la programación de este botón todavía no está completa. La completaremos un poco más adelante, pero por ahora nos permitirá probar el patrón de interacción **información anidada.**

   > **Note**
   >
   > En Flutter, el patrón de interacción **Información anidada** se implementa mediante las clases `Navigator` y `MaterialPageRoute` .

4. Se ha añadido un método de utilidad llamado `construyeCampoNombre` a la clase `_ListaCompraAnadirProductoState`, cuya misión es controlar cómo se añade el campo de texto donde se escribe el nombre del producto. Realiza varias tareas:

   - Añade dos vistas, una con un texto explicativo y otra con el campo de texto editable. El proceso es similar al seguido en la sesión anterior cuando se construyó el conversor. Se usa una vista de clase `Column` para situar ambos elementos uno sobre otro.

     En esta ocasión se ha usado el valor `CrossAxisAlignment.start` para la propiedad `crossAxisAlignment:` de la clase `Column` para que los textos se alineen a la izquierda.
   - En la propiedad `controller:` del `TextField` se asigna el `TextEditingController` que queremos que gestione los cambios en el texto, en este caso `_controladorNombre`
   - En la propiedad `decoration:` de tipo `InputDecoration` hay una propiedad llamada `hintText:` en la que se puede escribir el texto que queramos que se muestre en el campo cuando esté vacío.

     > **Note**
     >
     > Lo normal es que se escriba alguna pista sobre lo que hay que escribir en el campo. Eso mejora mucho la experiencia de uso.

5. La única vista para escribir los datos del producto que hemos añadido por ahora, es un campo de texto `TextField` donde el usuario pueda escribir el nombre del producto que quiere añadir a la lista de la compra.

   > **Note**
   >
   > En la sesión anterior ya usamos una vista de esta clase, sin embargo, en esta sesión la gestionamos con una clase de tipo `TextEditingController` que es la forma más flexible de gestionar una vista de clase `TextField` en Flutter. Este método es un poco más complejo de escribir que el que se usó en la sesión anterior pero ofrece una mayor flexibilidad.

   La propiedad `_controladorNombre` contiene el `TextEditingController` que se usa para gestionar los cambios que el usuario haga al contenido del campo de texto para el nombre del producto.

La propiedad `_nombre` almacena el valor que el usuario da al nombre del producto. Por ahora su valor inicial es la cadena vacía.

> **Important**
>
> La propiedad `_nombre` forma parte del estado mutable de la clase, por tanto, siempre que se actualice debe hacerse dentro de una función anónima que se pase al método `setState` de la clase `State`.

> **Note**
>
> En la siguiente sesión de prácticas, cuando se termine la pantalla para mostrar la lista de la compra, y se pueda editar alguno de los productos que contiene, se cambiará el valor inicial de la propiedad `_nombre`. En ese caso, el valor inicial de `_nombre` no será la cadena vacía sino el nombre del producto que se esté editando.

6. Todas las vistas que son subclases de la clase `State`, heredan el método `initState` que se ejecuta antes de que se construya la vista. Se puede utilizar para inicializar algunos aspectos de las propiedades de las vistas antes de mostrarlas en la pantalla.

   En este caso se usa para añadir a la clase `_ListaCompraAnadirProductoState` como observadora del `TextEditingController` llamado `_controladorNombre` que se usa para controlar los cambios en el `TextField`. Siempre que el usuario cambie el contenido del campo de texto para el nombre del producto, se llamará a la función anónima que se pasa a `addListener`. En este caso se actualiza el valor de la propiedad `_nombre` con el valor que se tenga en ese momento en el `TextEditingController`.

7. Las vistas también tienen le método `dispose` que se ejecuta **cuando la vista se elimina del árbol de vistas.** Se usa para liberar recursos, etc. En este caso, se usa para eliminar el `TextEditingController` llamado `_controladorNombre` que ya no será necesario cuando la vista se elimine del árbol de vistas.

8. En la propiedad. `body:` de la vista `Scaffold` se usa una vista de tipo `ListView` para organizar las propiedades de un `Producto` de forma que se pueda hacer scroll. Esto permite seguir viendo todas las propiedades aunque el tamaño de pantalla de un dispositivo concreto sea pequeño y no quepan todas a la vez.

   > **Note**
   >
   > La vista `ListView` de Flutter permite el desplazamiento de su contenido (scroll). Contiene una lista de widgets organizados linealmente. Es ideal para representar listas de objetos en la interfaz de usuario.

9. Dado que, en este caso, no todos los elementos del `ListView` son del mismo tipo, se usará el constructor parametrizado de `ListView` que espera la propiedad `children:` que es una lista con las vistas que se quieren mostrar dentro del `ListView`

   Por ahora, solo se tiene la vista que devuelve el método de utilidad `construyeCampoNombre` pero en la próxima sesión de prácticas se completará con las demás.

10. Para mejorar la apariencia, se ha encerrado el `ListView` dentro de una vista de tipo `Padding` que añade espacio alrededor. De esta forma los datos del producto que se añade/edita no llegan justo hasta el borde, lo que añade legibilidad.

13 – Mostrar la pantalla para añadir/editar productos

Lo único que falta para poder probar la aplicación es conectar el FAB `+` con la pantalla para añadir y/o editar productos que hemos escrito en la sección anterior.

Para ello, abre el archivo: `lista_compra_pantalla.dart`, busca las líneas de código:

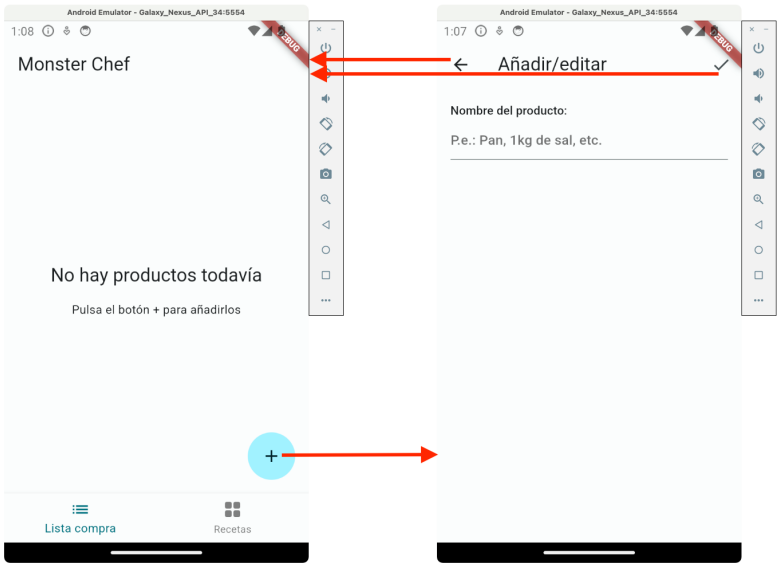
```
// 7
onPressed: () {
},
```

Y en su lugar escribe:

```
// 7
onPressed: () {
  // 9
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) {
        return ListaCompraAnadirProducto(
          // 10
          crearProducto: (producto) { Navigator.pop(context); },
          editarProducto: (producto) { },
        );
      },
    ),
  );
},
```

> ⚠ **Warning**
>
> No te olvides de añadir los imports necesarios o tu código tendrá errores de compilación.

Si haces Hot Reload, verás que ahora el FAB `+` es completamente funcional. Si lo pulsas, aparecerá la pantalla para añadir y/o editar productos, y si en esta pantalla pulsas el botón de vuelta atrás `←` o el botón `✓` verás que se muestra de nuevo la pantalla de la lista de la compra.



Todavía no se añaden productos. Programaremos esa funcionalidad en la siguiente sección, pero antes veamos algunos aspectos interesantes del nuevo código:

1. La propiedad `onPressed:` de la vista `FloatingActionButton` permite asignar la función anónima que se ejecutará como **callback** cuando el usuario pulse el FAB etiquetado como `+`. La función anónima no debe tener parámetros. Mediante esta función indicamos a Flutter cómo queremos que se comporte la aplicación al pulsar el FAB.

   Se debe pasar de la pantalla de lista de la compra a la pantalla de añadir/editar productos, pero **para que el usuario comprenda mejor el cambio,** programaremos la transición entre pantallas con una **animación.**

   Para que la transición entre pantallas se realice con una animación se usan dos clases:

   - `Navigator` que es un widget que gestiona un conjunto de widgets secundarios que forman una pila. En la pila puede haber varios niveles, aunque en este caso sólo tenemos dos: la pantalla con la lista de la compra y la pantalla para añadir/editar un producto.

     Implementa el patrón de interacción llamado **Información anidada,** permite al usuario hacer la transición visual de una pantalla a otra, por lo que es muy intuitivo y sencillo de utilizar.

   - `MaterialPageRoute` que permite implementar una transición entre pantallas, con una animación adaptable a la plataforma en la que se va a ejecutar.

     - En el caso de Android, la transición de entrada consiste en deslizar hacia arriba la pantalla nueva y desvanecer la pantalla actual. La transición de salida es la misma, pero a la inversa.
     - En el caso de iOS, la transición de entrada desliza los elementos de la pantalla nueva desde la derecha, al mismo tiempo que se deslizan los de la pantalla actual. La transición de salida es la misma pero a la inversa.

     > **Important**
     >
     > Si usamos Flutter no nos tenemos que preocupar por estos detalles. La propia plataforma elegirá la animación correcta en función del sistema operativo del terminal en el que se ejecute la aplicación.

2. Por ahora, la función anónima que se pasa en la propiedad `crearProducto:` al constructor de `ListaCompraAnadirProducto` solo contiene la orden `Navigator.pop(context);` que produce que se vuelva a la pantalla `ListaCompraPantalla`.

   > **Note**
   >
   > En la siguiente sección se completará añadiendo el código que realmente añade un nuevo producto a la lista de la compra.

3. La función anónima que se pasa en la propiedad `editarProducto:` al constructor de `ListaCompraAnadirProducto` está vacío ya que si se muestra la pantalla `ListaCompraAnadirProducto` a consecuencia de haber pulsado el botón `+` de la interfaz, es porque se está **creando un nuevo producto,** no editando uno existente, así que el callback `editarProducto:` nunca se va a ejecutar en esta situación.

14 – Añadir los productos a la lista de la compra

La pantalla `ListaCompraAnadirProducto` donde se añaden/editan los productos de la lista de la compra es completamente funcional, pero todavía no desencadena que se añadan productos a la lista. Tal y como está en este punto del desarrollo, al pulsar el botón ☑️ simplemente se vuelve a la pantalla `ListaCompraPantalla` sin añadir nada.

En esta sección vamos a conectar el botón ☑️ con el `ChangeNotifier` al que hemos llamado `ListaCompra` para que cuando se pulse el botón se añada un nuevo producto a la lista.

Pero antes necesitamos añadir una nueva dependencia a nuestro proyecto, para poder usar el paquete `Uuid` que simplificará la tarea de asignar identificadores únicos a los productos.

Como sabemos, la forma más sencilla de añadir una dependencia a un proyecto es abrir un terminal o símbolo del sistema en tu sistema operativo y navegar hasta la carpeta raíz de tu proyecto Flutter. Una vez en ella solo hay que ejecutar:

```
flutter pub add uuid
```

Una vez hecho esto, modificaremos la función anónima asociada a la propiedad `onPressed:` del botón de la pantalla `ListaCompraAnadirProducto` (el que está marcado con un tick mark ☑️), que por ahora solo hace que se vuelva a la pantalla anterior.

Para ello busca dentro del método `build` de la clase `_ListaCompraAnadirProductoState` las líneas de código:

```
// 8
onPressed: () {
  Navigator.pop(context);
},
```

Y sustitúyela por las líneas de código:

```
// 8
onPressed: () {
  final producto = Producto(
    id: widget.productoOriginal?.id ?? const Uuid().v1(),
    nombre: _controladorNombre.text,
    completado: _completado,
    // 14
    importancia: Importancia.media,
    // 17
    cantidad: 1,
  );

  if (widget.actualizando) {
    widget.editarProducto(producto);
  } else {
    widget.crearProducto(producto);
  }
},
```

> ⚠ **Warning**
>
> No te olvides de añadir los imports necesarios o tu código tendrá errores de compilación.

A continuación busca dentro del método `build` de la clase `ListaCompraPantalla` el comentario:

```
// 9
```

Y sustitúyelo por las siguientes líneas de código:

```
// 9
final manager = Provider.of<ListaCompra>(context, listen: false);
```

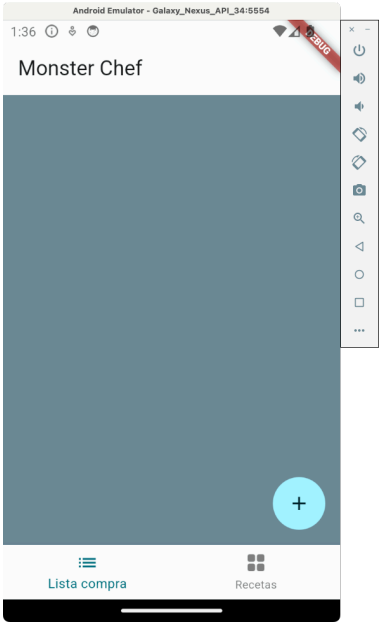
Busca las siguientes líneas de código:

```
// 10
crearProducto: (producto) { Navigator.pop(context); },
```

Y sustitúyelas por las líneas de código:

```
// 10
crearProducto: (producto) {
  manager.anadeProducto(producto);
  Navigator.pop(context);
},
```

Si haces Hot Reload, puedes probar cómo ahora al pulsar el botón ☑ en la pantalla de añadir productos, ya no se muestra la pantalla que avisa de que la lista de productos está vacía, sino que se muestra esta otra pantalla, lo que indica que el producto se ha añadido realmente a la lista.



En la última sección de esta sesión aprenderemos a mostrar los productos de la lista, aunque todavía no tendrán su formato definitivo, pero antes veamos algunos aspectos interesantes de este último código que se ha añadido:

1. En la función anónima asociada a la propiedad `onPressed:` del `IconButton` etiquetado como ✔️ se construye un nuevo objeto de la clase `Producto` que contendrá los valores dados por el usuario y que están en las propiedades privadas de `_ListaCompraAnadirProductoState`.

   > **Note**
   >
   > Por ahora sólo se inicializan las propiedades `id` y `nombre`, ya que aún no se tienen widgets en la pantalla para añadir productos, donde dar valor a la importancia y la cantidad del producto. Por eso a estas últimas propiedades se les pasa un valor arbitrario. En la siguiente sesión de prácticas se cambiará.

2. La propiedad `id` requiere una explicación mas profunda. Hay que distinguir dos situaciones: se está añadiendo un nuevo `Producto` o se está modificando uno ya creado.

   Para ello se usa la propiedad `productoOriginal` de la clase `ListaCompraAnadirProducto`. Si es `null` es porque se está construyendo un nuevo producto y en ese caso hay que asignarle un nuevo identificador único, mientras que si se está editando uno ya existente, se debe conservar el identificador que ya tenga.

   Para generar identificadores únicos se usa el paquete `Uuid` mediante la llamada:

   ```
   Uuid().v1()
   ```

   que genera una cadena de caracteres que es un identificador único de tipo `v1`, es decir, basado en la hora a la que se genera. El resultado obtenido es una cadena similar a esta:

   ```
   '6c84fb90-12c4-11e1-840d-7b25c5ee775a'
   ```

   > **Note**
   >
   > El paquete UUID tiene otros modos para generar identificadores únicos. Para una mayor información se puede consultar la documentación en el siguiente enlace: Documentación paquete UUID

   > **Note**
   >
   > La propiedad `productoOriginal` pertenece a la clase `ListaCompraAnadirProducto` pero se usa en la clase `State` asociada `_ListaCompraAnadirProductoState`, por eso se accede a través de la propiedad `widget` de la clase `State`

3. Una vez construido el objeto `Producto` con los valores correctos de las propiedades, se llama al callback apropiado `editarProducto` o `crearProducto` según sea el valor de la propiedad `actualizando` de la clase `ListaCompraAnadirProducto`.
4. El callback `crearProducto` necesita acceder a la lista de la compra para poder añadirle el nuevo producto creado, por eso, la primera instrucción del método `onPressed:` del `FloatingActionButton` es:

   ```
   final manager = Provider.of<ListaCompra>(context, listen: false);
   ```

Donde se accede al `ChangeNotifierProvider` que encapsula al `ChangeNotifier` al que hemos llamado `ListaCompra`, solo para acceder a sus métodos y ejecutar las acciones para añadir un nuevo producto a la lista, no para observar sus cambios. Por eso no se usa un `Consumer` que es un Observador de los cambios, sino `Provider.of<ListaCompra>`.

> **Note**
>
> También es por eso que se pasa el valor `false` al parámetro `listen:`.

5. El callback que se pasa a la propiedad `crearProducto:` será ejecutado cuando el usuario pulse el botón ✔️ de la pantalla `ListaCompraAnadirProducto` por tanto deberá hacer dos cosas:
    - Añadir un nuevo producto a la lista de la compra. Esto se hace a través de la variable `manager` que hace referencia a la lista de la compra y que se ha inicializado en el paso que se describe en el punto anterior.
    - Hacer que se vuelva a la pantalla `ListaCompraPantalla`. Esto se hace mediante el método `pop` de la clase `Navigator`.

15 – Pantalla para la lista de la compra con productos

El siguiente paso es añadir una pantalla para mostrar los productos que haya en la lista de la compra. Esta pantalla se mostrará en lugar de la pantalla `ListaCompraPantallaVacia` cuando la lista de la compra tenga productos.

Añade un nuevo archivo llamado `lista_compra_pantalla_llena.dart` a la carpeta `lib` de tu proyecto Flutter, y dentro escribe el siguiente código:

```
class ListaCompraPantallaLlena extends StatelessWidget {
  const ListaCompraPantallaLlena({Key? key, required this.listaCompra}) :
super(key: key);

  final ListaCompra listaCompra;

  @override
  Widget build(BuildContext context) {
    final productos = listaCompra.productos;
    return Padding(
      padding: const EdgeInsets.all(10.0),
      child: ListView.separated(
        itemCount: productos.length,
        separatorBuilder: (context, index) {
          return const SizedBox(height: 8.0);
        },
        itemBuilder: (context, index) {
          final producto = productos[index];
          // 18
          return Text(
            producto.nombre,
          );
        },
      ),
    );
  }
}
```

> ⚠ **Warning**
>
> No olvides incluir los imports necesarios o tendrás errores de compilación.

A continuación, vamos a usar esta nueva vista. Para ello abre el archivo: `lista_compra_pantalla.dart`, busca las líneas de código:

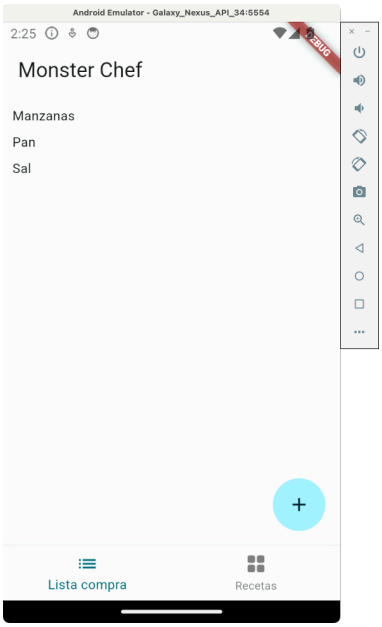
```
// 11
return Container( color: Colors.blueGrey, );
```

Y sustitúyelas por una llamada a la nueva vista:

```
// 11
return ListaCompraPantallaLlena(listaCompra: manager);
```

> ⚠ **Warning**
>
> No te olvides de añadir los imports necesarios o tu código tendrá errores de compilación.

Si haces Hot restart, verás que al pulsar el botón ✔ en la pantalla de añadir productos, no se vuelve a la pantalla que avisa de que la lista de productos está vacía sino que se muestra la lista de la compra:



> ℹ **Note**
>
> Por ahora, el formato es muy sencillo y no permite editar o borrar los productos añadidos. Tampoco se guarda la lista de la compra entre dos ejecuciones de la aplicación. Todo eso lo arreglaremos en la siguiente sesión de prácticas.

Veamos algunas consideraciones importantes del código que acabamos de escribir:

1. Para visualizar la lista de la compra se usa un objeto de la clase `ListView`. Para instanciarlo usaremos el constructor con nombre llamado `separated` que construye un array de vistas que tiene un tamaño conocido, separadas entre sí por otras vistas que actúan como separadores. La vista permite que se puede hacer scroll para deslizarse al comienzo o al final de la lista.

   > **Important**
   >
   > Este constructor es apropiado para listas de vistas con un gran número de elementos y separadores, porque los constructores solo se llaman para los hijos que son realmente visibles.

   Se usan tres parámetros del constructor `separated` de la clase `ListView`:

   - `itemCount:` que indica el número de elementos que se muestran en el `ListView`. En este caso se usa el número de productos de la lista de la compra `productos.length`
   - `itemBuilder:` recibe una función anónima que se ejecuta tantas veces como productos haya en la `ListaCompra`. Esa función anónima construye la vista que va a representar en la pantalla los datos de cada producto de la lista.

     La función anónima que se pasa en `itemBuilder` tiene dos parámetros: `context` e `index` que indica el número del elemento de la lista de la compra para el que que se está construyendo su vista en un momento dado.

     Por ahora, para cada producto de la lista de la compra se construye una vista de clase `Text` en la que se muestra su nombre `producto.nombre`

     > **Note**
     >
     > En la próxima sesión se cambiará para mostrar el resto de propiedades de los productos de la lista de la compra, como por ejemplo su importancia o cantidad.

   - `separatorBuilder` es un builder que indica cómo se construyen las vistas que actúan como separadores entre dos ítems consecutivos del `ListView`. En este caso, simplemente se construye un `SizedBox` con altura `8`

Mini ejercicios

1. ➲ En el apartado Establecer conexión de la sección 3–Conectar botones y pantallas, se declara una lista llamada `paginas` con las vistas asociadas a las categorías. La correspondencia entre unas y otras se realiza manualmente y obliga a tener cuidado de asociar tantas pantallas como categorías y en el mismo orden.

   ¿Se te ocurre alguna forma mejor de organizar eso, encapsulando las categorías, sus iconos y mensajes, y las pantallas asociadas a cada una, en algún objeto para que sea mucho más sencillo añadir, eliminar o reordenar categorías? Refactoriza tu aplicación para que use esta forma de organizar el código.

   > **Tip**
   >
   > Necesitas refrescar lo que ya has estudiado sobre los tipos enumerados mejorados y los higher-order methods de las colecciones.