Flutter 3: Lista de la compra

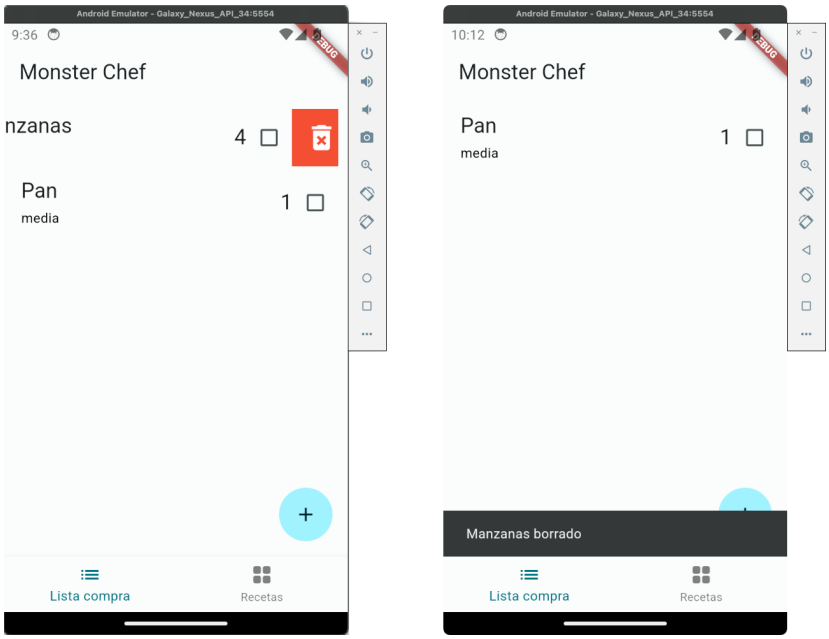
Parte 2

Contenidos

¿Qué voy a hacer en esta sesión?

En esta sesión vamos a terminar completamente la parte de la lista de la compra en la aplicación **Monster Chef.**

El aspecto de la aplicación al terminar esta sesión será este:



Al escribirla paso a paso habremos aprendido:

- A implementar gestos avanzados como deslizar para eliminar un producto de la lista de la compra.
- A utilizar el patrón de notificación transitorio conocido como **snackbar** para informar al usuario de situaciones que requieran su atención.
- A salvar y leer datos en formato JSON de un archivo local almacenado en el directorio **documentos** propio de la aplicación, en el que sólo esta aplicación puede leer.
- A utilizar entre otros los siguientes widgets de Flutter: `Row`, `Container`, `Wrap`, `ChoiceChip`, `Slider`, `InkWell`, `Dismissible` y `SnackBar`.

Pasos previos

Abre la aplicación: `flutter_sesion_2_4` que se encuentra en tu carpeta `flutter_projects` que comenzaste a escribir en la sesión anterior para continuar el código por donde lo dejamos.

Terminar la aplicación lista de la compra

Todavía faltan muchos elementos de interacción para completar la lista de la compra. Vamos a ir construyéndolos paso a paso.

1 – Completar la pantalla para añadir productos

Comenzaremos editando el archivo: `lista_compra_anadir_producto.dart`. En él se declara la pantalla que usamos tanto para añadir nuevos productos como para editar las propiedades de los productos ya añadidos. Por ahora sólo permite escribir el nombre del producto. Vamos ahora a añadir elementos de interacción para asignarle la importancia y la cantidad.

Empieza añadiendo una nueva propiedad a la clase `_ListaCompraAnadirProductoState`. Para ello busca el comentario:

```
// 12
```

y sustitúyelo por las líneas de código:

```
// 12
Importancia _importancia = Importancia.baja;
// 15
```

> **Important**
>
> La propiedad `_importancia` 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`

Al final de la clase, justo debajo del método de utilidad `construyeCampoNombre` que se escribió la sesión anterior, añade el siguiente método de utilidad:

```
Widget construyeCampoImportancia() {
  return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      Text(
        'Importancia',
        style: Theme.of(context).textTheme.titleSmall,
      ),
      Wrap(
        spacing: 10.0,
        children: Importancia.values.map((elemento) {
          return ChoiceChip(
            selected: _importancia == elemento,
            shape: StadiumBorder(),
            label: Text(
              elemento.name,
            ),
```

```
            onSelected: (selecion) {
              setState(() { _importancia = elemento; });
            },
          );
        }).toList(),
      )
    ],
  );
}
```

Ahora debemos añadir, a la lista de propiedades del producto, los elementos de interacción para elegir la importancia del producto. Para ello busca el comentario:

```
// 13
```

y sustitúyelo por las líneas de código:

```
// 13
const SizedBox(height: 16,),
construyeCampoImportancia(),
// 16
```

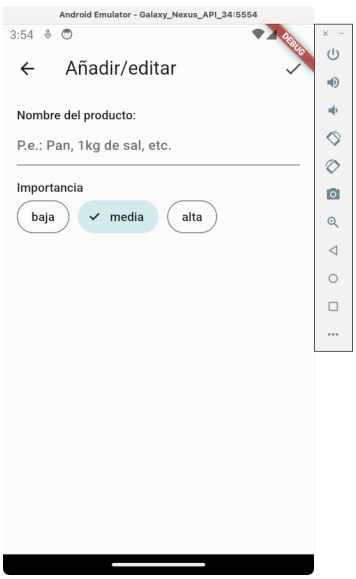
Por último, para que se grabe correctamente el valor de la importancia en el producto, busca las líneas:

```
// 14
importancia: Importancia.media,
```

y sustitúyelas por las líneas:

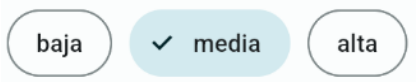
```
// 14
importancia: _importancia,
```

Si compilamos y ejecutamos la aplicación y pulsamos el botón `+` para añadir nuevos productos veremos una pantalla similar a esta:



Antes de añadir los controles para asignar la cantidad al producto, veamos algunas consideraciones interesantes sobre el código que acabamos de escribir:

1. Se usa la vista `ChoiceChip` de Flutter para codificar la importancia de un producto. Se ha optado por usar la clase `ChoiceChip` de Flutter, porque tiene el mismo comportamiento que un radio button, pero con mejor apariencia visual.



2. Para distribuir los `ChoiceChip` en la pantalla se usa la vista `Wrap` que recibe en su propiedad `children:` una lista de vistas y las muestra en una fila mientras haya sitio. Si se termina el espacio horizontal disponible, las demás vistas se muestran en filas sucesivas.

> **Note**
>
> Es una vista muy útil cuando se deben mostrar organizadas por filas una lista de vistas y queremos asegurar que se ven bien con independencia del tamaño de la pantalla del smartphone concreto que se use.

3. En cada `ChoiceChip` hay que pasar una función anónima a la propiedad `onSelected:` Esa función anónima indica qué debe hacerse cuando el usuario selecciona ese chip. En este caso hay que actualizar la propiedad `_importancia` de la clase `_ListaCompraAnadirProductoState` con el valor adecuado.

> **Important**
>
> Recuerda que las funciones anónimas son clausuras. Por eso se puede usar la variable `elemento` dentro de la función anónima que se usa dentro del método `setState`. En cada constante del tipo enumerado, la variable `elemento` contiene justamente ese valor. Por ejemplo, cuendo el higher-order method `map` esté procesando el valor `Importancia.baja`, la asignación `_importancia = elemento;` equivale a `_importancia = Importancia.baja` y así con todas las demás.

4. Los chips se rellenan automáticamente. Para eso se usa el higher-order method `map` aplicado al iterable que devuelve la propiedad `values` que contiene todos los valores del tipo enumerado `Importancia`.

   Recuerda que `map` espera un único argumento (en este ejemplo lo he llamado `elemento`) cuyo tipo se infiere del tipo de los elementos del iterable sobre el que se aplica. En este caso son las constantes de tipo `Importancia`

   Para cada elemento se construye un `ChoiceChip` usando el getter `name` del tipo enumerado `Importancia` para obtener una cadena de caracteres con el nombre de la constante del tipo enumerado que se tenga en `elemento`, y pasar ese valor como etiqueta del chip en su propiedad `label:`

> **Note**
>
> Desde la versión 2.15 de Dart, para cualquier tipo enumerado, se tiene la propiedad `name` que cuando se aplica a una constante del enumerado, devuelve un `String` con el nombre de esa constante pero en formato cadena de caracteres. Cuando la constante del tipo enumerado tiene un nombre que también tiene sentido para el usuario, puede usarse esta propiedad `name` sin tener que declarar una propia.

Por último, dado que `map` devuelve un `Iterable` y `children:` espera una `List<Widget>` hay que llamar al método `toList()` de los iterables para hacer la conversión.

> **Important**
>
> Con esta forma de rellenar los `ChoiceChip`, si se quiere añadir una nueva constante para la importancia, por ejemplo `muy_importante` los únicos cambios que hay que hacer se realizan en el tipo enumerado. La interfaz funcionará correctamente sin tener que modificar nada.

5. La propiedad `selected:` toma un valor de tipo `bool`. Si tiene valor `true` el chip se dibuja seleccionado (color más oscuro y con un tick mark ✔) y si tiene valor `false` se dibuja no seleccionado (color más claro).

   En el chip que corresponda con el valor inicial, se debe pasar el valor `true` a la propiedad `selected:` y `false` en los demás. Para desacoplar el código no se escribe directamente el valor `true` o `false` sino que se hace las comparación:

   ```
   _importancia == elemento,
   ```

   que devuelve `true` sólo cuando el valor inicial que tenga la variable `_importancia` sea igual al `elemento` que se esté procesando. En los demás casos se devolverá `false`

   De esta forma si se quiere cambiar el valor inicial para la propiedad `_importancia` de la clase `_ListaCompraAnadirProductoState`, no hay que reescribir el código que inicializa los chips.

Vamos ahora a terminar la pantalla añadir productos añadiendo los elementos de interacción para que el usuario indique la cantidad de producto a comprar. Comienza añadiendo una nueva propiedad privada `_valorActualSlider` a la clase `_ListaCompraAnadirProductoState`. Para ello busca el comentario:

```
// 15
```

y sustitúyelo por el siguiente código:

```
// 15
int _valorActualSlider = 0;
```

Al final de la clase `_ListaCompraAnadirProductoState`, justo debajo del método de utilidad `construyeCampoImportancia` que acabas de escribir, añade el siguiente método de utilidad:

```
Widget construyeCampoCantidad() {
  return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: <Widget>[
      Row(
        crossAxisAlignment: CrossAxisAlignment.baseline,
        textBaseline: TextBaseline.alphabetic,
        children: <Widget>[
          Text(
            'Cantidad',
            style: Theme.of(context).textTheme.titleSmall,
          ),
          const SizedBox(width: 16.0),
          Text(
            _valorActualSlider.toInt().toString(),
            style: Theme.of(context).textTheme.headline6,
          ),
        ],
      ),
      Slider(
        value: _valorActualSlider.toDouble(),
        min: 0.0,
        max: 100.0,
        divisions: 100,
        label: _valorActualSlider.toInt().toString(),
        onChanged: (double value) {
          setState(
            () {
              _valorActualSlider = value.toInt();
            },
          );
        },
      ),
    ],
  );
}
```

Para que se muestren los elementos de interacción relacionados con la cantidad, en la pantalla de añadir producto, busca el comentario:

```
// 16
```

y sustitúyelo por el siguiente código:

```
// 16
const SizedBox(
  height: 16,
),
construyeCampoCantidad(),
```

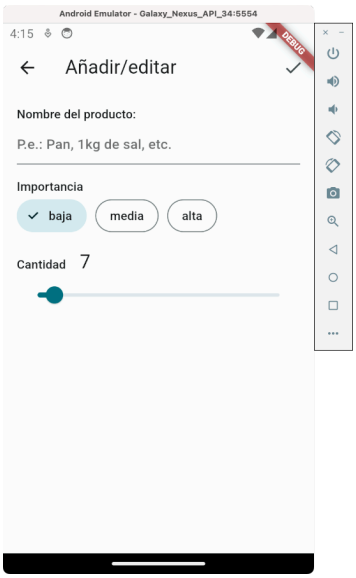
Por último, para que se grabe correctamente el valor de cantidad en el producto, busca las líneas:

```
// 17
cantidad: 1,
```

y sustitúyelas por las siguientes líneas de código:

```
// 17
cantidad: _valorActualSlider,
```

Si haces Hot Reload y pulsas el botón `+` para añadir un producto, verás una pantalla similar a esta:



Veamos algunas consideraciones interesantes sobre el nuevo código:

1. La propiedad `_valorActualSlider` es de tipo `int`, pero la vista `Slider` usa valores de tipo `double`, por tanto es necesario hacer las siguientes conversiones:

   ```
   value: _valorActualSlider.toDouble(),
   ```

   ```
   _valorActualSlider = value.toInt();
   ```

2. Una de las propiedades del `Slider` es `onChanged:` A esta propiedad se le pasa una función anónima que es el callback que se invoca cuando el usuario desliza el `Slider` y cambia su valor.

   En esa función anónima se debe cambiar el valor de `_valorActualSlider` pero como forma parte del estado mutable de la pantalla para añadir/editar productos, se debe hacer la asignación dentro de una llamada al método `setState` de la clase `State`

3. Además del `Slider` para cambiar el valor de la cantidad de producto, se usa una vista de tipo `Text` para que el usuario pueda ver claramente la cantidad seleccionada.

   Esa vista muestra el valor de la propiedad `_valorActualSlider` que es parte del estado mutable de la vista `_ListaCompraAnadirProductoState`, por tanto, cada vez que se cambia por medio del `Slider`, se reconstruye la pantalla para mostrar el nuevo valor.

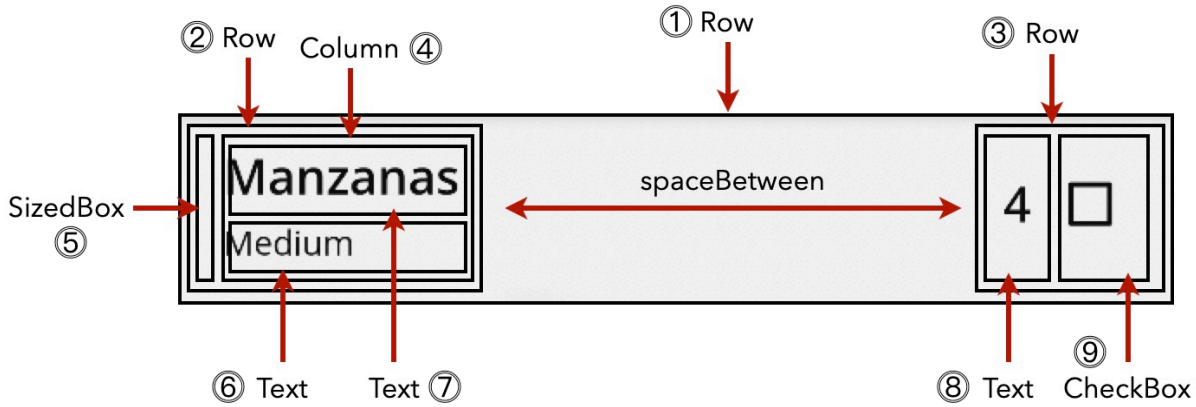
4. Junto a la vista `Text` para mostrar el valor del `Slider` se muestra otra vista `Text` con una etiqueta que indica que el valor que visualiza a continuación es la cantidad de producto.

   Ambas vistas deben figurar una al lado de la otra, por lo que se ha usado una vista de clase `Row` para agruparlas. Es idéntica a la vista de clase `Column` que ya hemos usado, pero coloca sus vistas hijas en fila en lugar de en columna.

2 – Detallar la información de los productos

Hasta ahora, cada vez que se añade un producto, se muestra en una lista donde sólo se muestra su nombre y no permite interacción. Vamos a completar esta parte ahora.

Para mostrar adecuadamente cada producto de la lista de la compra, usaremos la siguiente estructura de vistas:



Tiene muchas vistas dentro de vistas para permitir una correcta alineación y espaciado, aprovechando al máximo el tamaño de la pantalla del dispositivo. Es una estructura compleja de escribir, por eso, en lugar de pasar todas las vistas directamente a la función anónima de la propiedad `itemBuilder:` crearemos una clase personalizada para encapsular toda esa estructura y simplificar el constructor del `ListView`

Para ello, añadiremos un nuevo archivo al proyecto al que llamaremos: `linea_producto.dart` y en él escribiremos la clase `LineaProducto` que representa a las vistas que usaremos para mostrar en pantalla un producto concreto de la lista:

```
class LineaProducto extends StatelessWidget {
  final Producto producto;
  final TextDecoration textDecoration;
  // 19

  // 20
  LineaProducto({Key? key, required this.producto})
      : textDecoration = producto.completado
            ? TextDecoration.lineThrough
            : TextDecoration.none,
        super(key: key);
```

```dart
@override
 Widget build(BuildContext context) {
   var titleLarge = Theme.of(context).textTheme.titleLarge;
   var decortitleLarge = titleLarge?.copyWith(decoration: textDecoration);

   return Container(
     height: 60.0,
     child: Row(
       // - Número ① en el diagrama
       mainAxisAlignment: MainAxisAlignment.spaceBetween,
       children: <Widget>[
         Row(
           // - Número ② en el diagrama
           children: <Widget>[
             const SizedBox(
               width: 8,
             ), // - Número ⑤ en el diagrama
             Column(
               // - Número ④ en el diagrama
               mainAxisAlignment: MainAxisAlignment.center,
               crossAxisAlignment: CrossAxisAlignment.start,
               children: <Widget>[
                 Text(
                   // - Número ⑦ en el diagrama
                   producto.nombre,
                   style: decortitleLarge!,
                 ),
                 const SizedBox(height: 4.0),
                 construirImportancia(
                     context, producto), // - Número ⑥ en el diagrama
               ],
             ),
           ],
         ),
         Row(
           // - Número ③ en el diagrama
           children: <Widget>[
             Text(
               // - Número ⑧ en el diagrama
               producto.cantidad.toString(),
               style: titleLarge,
             ),
             Checkbox(
               // - Número ⑨ en el diagrama
               value: producto.completado,
               // 21
               onChanged: (cambiado) {},
             ),
           ],
         ),
       ],
     ),
   );
 }
```

```
  Widget construirImportancia(BuildContext context, Producto producto) {
    var temaTexto = Theme.of(context).textTheme.bodyMedium;
    var temaTextoColoreado = temaTexto?.copyWith(
        color: producto.importancia == Importancia.alta
            ? Colors.red
            : temaTexto?.color);
    return Text(
      producto.importancia.name,
      style: temaTextoColoreado!,
    );
  }
}
```

> ⚠ **Warning**
>
> Acuérdate también de añadir los imports correspondientes o de lo contrario tendrás errores de compilación.

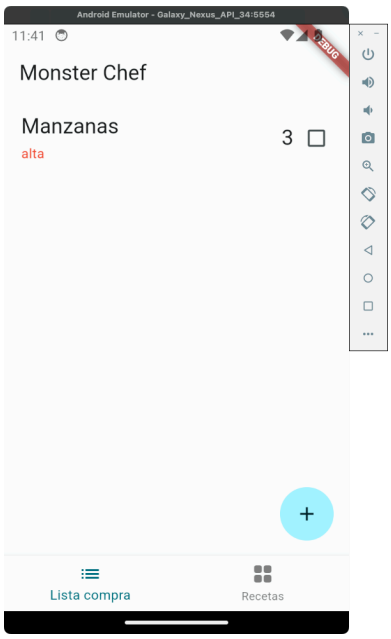
Ya que está escrita la clase `LineaProducto` sólo hay que usarla. Para ello busca, en la clase `ListaCompraPantallaLlena` las líneas de código:

```
// 18
return Text(
  producto.nombre,
);
```

y sustitúyelas por las siguientes líneas de código:

```
// 18
return LineaProducto(
  producto: producto,
);
```

Si haces Hot Restart, pulsas el botón `+` para añadir producto y a continuación pulsas el botón ☑ para aceptar el nuevo producto, verás una pantalla similar a esta:



Que es mucho más informativa e interesante, aunque todavía no responde a las acciones del usuario. Lo siguiente será añadirle capacidad de responder a gestos, pero antes vamos a estudiar algunos aspectos interesantes del código que acabamos de añadir:

1. El código anterior se ejecuta una vez por cada producto que haya en la lista de la compra. Se vuelve a ejecutar cada vez que hay algún cambio en la lista de la compra (se añade o borra un producto) o bien se edita alguno de los que ya hay.
2. En la clase `LineaProducto` se ha añadido una propiedad llamada `textDecoration` que se incializa con el valor `TextDecoration.lineThrough` (tachado) si el producto se ha marcado como completado (su propiedad `completado` es `true`) y con el valor `TextDecoration.none` si no está marcado como completado.

   De esta forma, los productos de la lista de la compra ya comprados se podrán distinguir fácilmente de los pendientes, ya que además de tener el tick mark en el `CheckBox`, su nombre estará tachado.

   La propiedad `textDecoration` se inicializa en la lista de inicialización del constructor de `LineaProducto`.

   ```
   textDecoration = producto.completado
       ? TextDecoration.lineThrough
       : TextDecoration.none
   ```

3. En la primera vista de clase `Row` (marcada con el número ① en el diagrama) que es la más externa y contiene a todas las demás, se ha usado el modo de alineación `MainAxisAlignment.spaceBetween`. Eso hace que el espacio que sobre después de construir las vistas hijas (en este caso otras dos vistas `Row`) se sitúe entre ellas, de forma que una quede completamente pegada a la izquierda y la otra a la derecha. Ese espacio intermedio actuaría como un muelle que empuja a ambas vistas hacia los extremos.

   De esta forma siempre se van a ver bien alineadas sea cual sea el tamaño y la resolución de la pantalla del teléfono en el que se ejecute.

4. En la vista `Row` situada mas a la izquierda (Marcada con ② en el diagrama), la que contiene el texto para el nombre del producto y su importancia, se ha añadido un `SizedBox` con una anchura de 8 dp para que los textos no queden demasiado pegados al borde izquierdo de la pantalla.
5. Las vistas que representan al campo `importancia` son un poco más complejas de construir, por eso se han encapsulado en un método de utilidad `construirImportancia` dentro de la propia clase `LineaProducto`

   A este método se le pasa el contexto, para que se pueda acceder al tema común definido para la aplicación y el producto para el que se está construyendo su vista, para que pueda acceder al valor de la propiedad `importancia.
6. Se quiere que si un producto tiene valor `Importancia.alta` para la propiedad `importancia` se muestre en color rojo y en cualquier otro caso que se use el color del tema activo.

Para aplicarlo al texto se construye una variable `temaTextoColoreado` basada en el tema `bodyMedium` del contexto, al que se le aplica un color u otro en función de si `producto.importancia` es igual a `Importancia.alta`:

```
var temaTextoColoreado = temaTexto?.copyWith(
        color: producto.importancia == Importancia.alta
            ? Colors.red
            : temaTexto?.color);
```

7. El objeto de la clase `CheckBox` que muestra el valor de la propiedad `completado` de los productos, recibe una función anónima en su propiedad `onChanged:` Esta función anónima es la que se ejecuta cuando se pulsa en el `CheckBox` para cambiar su estado. Por ahora está vacía. Se completará en la próxima sección.
8. Por ahora, la vista no responde a ninguna acción del usuario. En próximas secciones se añadirá el código necesario para marcar como completada, editar y borrar.

3 – Marcar una tarea como completada

En este paso vamos a hacer que el `CheckBox` que muestra la propiedad `completado` de un `Producto` de la lista de la compra sea funcional, es decir, que cuando se pulse, su estado cambie. Si estaba desmarcado que se marque y viceversa.

Para ello añadiremos una nueva propiedad llamada `completar` a la clase `LineaProducto`. Busca el comentario:

```
// 19
```

y sustitúyelo por las líneas de código:

```
// 19
final Function(bool?)? completar;
```

A continuación en el constructor de `LineaProducto` busca las líneas de código:

```
// 20
LineaProducto({Key? key, required this.producto})
```

Y sustitúyelas por las siguientes líneas de código:

```
// 20
LineaProducto({Key? key, required this.producto, required this.completar})
```

Ahora en el `CheckBox` busca las líneas de código:

```
// 21
onChanged: (cambiado) {},
```

y sustitúyelas por estas otras líneas de código:

```
// 21
onChanged: completar,
```

Hemos terminado con el archivo `linea_producto.dart`.

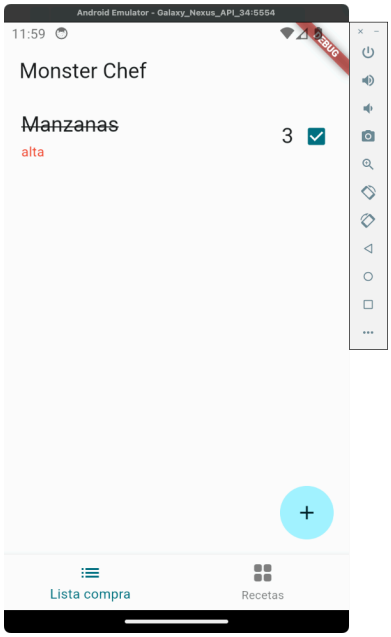
Ahora abre el archivo `lista_compra_pantalla_llena.dart` y dentro de él busca las líneas:

```
// 18
return LineaProducto( producto: producto, );
```

para sustituirlas por estas otras líneas de código:

```
// 18
return LineaProducto(
  producto: producto,
  completar: (valor) {
    if (valor != null) {
      listaCompra.marcaCompletado(index, valor);
    }
  },
);
```

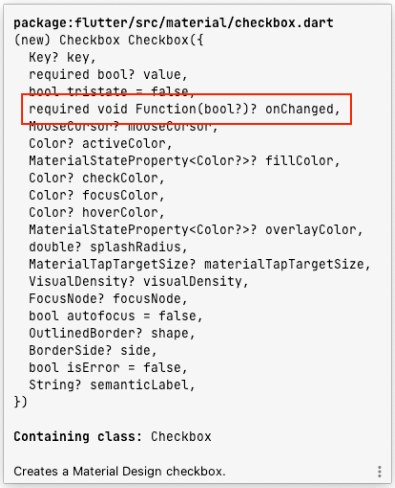
Si haces Hot Restart, pulsas el botón `+` para añadir algún producto, rellenas sus campos, y pulsas el botón ✔ para aceptar el nuevo producto, verás como al pulsar el `CheckBox` situado más a la derecha de cada producto, puedes cambiar el estado completado de cada producto. El aspecto de la aplicación cuando hay un producto marcado es el siguiente:



Como se puede ver, no sólo aparece el tick mark en `CheckBox`, sino que además se tacha el texto con el nombre del producto, para dar una mayor sensación de que ese producto ya está comprado.

Veamos algunas consideraciones interesantes sobre este código:

1. Como puedes ver en la siguiente captura de pantalla de la documentación que sobre el `Widget` de tipo `CheckBox` ofrece Android Studio, la propiedad `onChanged:` de la vista `CheckBox` espera una función anónima de tipo `void Function(bool?)?`. Por tanto la variable `completar` **debe** definirse de ese tipo. Esta función anónima se llama cuando el valor del `CheckBox` deba ser cambiado. La propia clase `CheckBox` pasa el nuevo valor a la función anónima, pero en realidad no cambia su estado interno hasta que no se reconstruya el `CheckBox` con el nuevo valor.



   Por eso, el contenido de esa función invoca al método `marcaCompletado` de `ListaCompra` para que actualice la lista de productos y notifique que se debe reconstruir el árbol de vistas completo. Eso se consigue con la orden:

   ```
   listaCompra.marcaCompletado(index, valor);
   ```

   > **Note**
   >
   > Fíjate que la propia propiedad `onChanged:` puede ser nula (lo indica el segundo signo `?` en su declaración). Si en lugar de una función, se pasa el valor `null`, el `CheckBox` se dibuja deshabilitado y no responde a los toques del usuario.

2. Como consecuencia de lo anterior, `valor` es de tipo `bool?` por tanto podría ser `null`, para que compile y ejecute correctamente es necesario el `if` que comprueba que no se tenga el valor `null`

4 – Editar un producto ya añadido

Ahora vamos a añadir el comportamiento para que al pulsar sobre un producto en la lista de la compra, se muestre de nuevo la pantalla `ListaCompraAnadirProducto` esta vez para poder modificarlo. En este ocasión usaremos la clase `InkWell` para detectar el gesto de pulsar y realizar la animación.

En primer lugar editaremos el archivo `lista_compra_anadir_producto.dart`. Buscaremos el comentario:

```
// 22
```

y lo sustituiremos por las siguientes líneas de código:

```
// 22
final productoOriginal = widget.productoOriginal;
if (productoOriginal != null) {
  _controladorNombre.text = productoOriginal.nombre;
  _nombre = productoOriginal.nombre;
  _valorActualSlider = productoOriginal.cantidad;
  _importancia = productoOriginal.importancia;
  _completado = productoOriginal.completado;
}
```

A continuación abriremos el archivo `lista_compra_pantalla_llena.dart` y en él buscaremos dentro del método `build` las líneas:

```
// 18
return LineaProducto(
  producto: producto,
  completar: (valor) {
    if (valor != null) {
      listaCompra.marcaCompletado(index, valor);
    }
  },
);
```

Y las sustituiremos por estas otras líneas de código:

```
// 18
return InkWell(
  child: LineaProducto(
    producto: producto,
    completar: (valor) {
      if (valor != null) {
        listaCompra.marcaCompletado(index, valor);
      }
    },
  ),
  onTap: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) => ListaCompraAnadirProducto(
          productoOriginal: producto,
          editarProducto: (producto) {
            listaCompra.actualizaProducto(producto, index);
            Navigator.pop(context);
          },
          crearProducto: (producto) {},
        ),
      ),
    );
  },
);
```

> ⚠ **Warning**
>
> Acuérdate de añadir los imports necesarios ya que de lo contrario el programa tendrá errores de compilación.

Como puede apreciarse se ha envuelto la vista `LineaProducto` dentro de una vista `InkWell`. Esta vista responde a los toques del usuario y realimenta la acción mediante una animación que simula el efecto de una gota de tinta que cae sobre la vista.

Si haces Hot Restart, añades algún producto a la lista de la compra y tocas sobre algún punto de la vista `Row` que encierra a un producto (siempre y cuando no sea el `CheckBox` para marcarla como completada), veras la siguiente animación:

y se mostrará de nuevo la pantalla para añadir productos. Esta vez con los datos del producto sobre el que se ha tocado cargados. Si en esa vista se pulsa sobre el botón para volver atrás `←`, los cambios que se hayan podido hacer se descartan, y si se pulsa sobre el botón `✔` se aceptan los cambios.

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

1. Dentro del método `initState` de la vista `_ListaCompraAnadirProductoState` se comprueba si el producto original es `null`. Si es así es porque se está añadiendo un producto nuevo, mientras que si no lo es, es porque se está editando uno ya creado. De esta forma se pueden distinguir los dos casos y se puede utilizar la misma pantalla para añadir nuevos productos y editar productos ya añadidos.
2. En caso de estar en modo edición (`productoOriginal` no es `null`) se rellenan las propiedades de `_ListaCompraAnadirProductoState` de forma que al construir las vistas que muestran los datos del producto se usen los valores de las propiedades del producto que esté seleccionado y no los valores por defecto.
3. Se ha envuelto la vista `LineaProducto` dentro de una vista `InkWell`. De las muchas propiedades que incluye la vista `InkWell` en este caso se han usado dos:
    - `child:` en la que se ha incluído la `LineaProducto`. Es lo que se muestra dentro del `InkWell`.
    - `onTap:` que acepta una función anónima sin parámetros, en la que se indica cómo debe comportarse la vista cuando se pulsa sobre ella. El comportamiento es muy similar al ya estudiado en la sesión anterior para crear nuevos objetos. La diferencia es que en este caso el callback que se pasa vacío es el de la propiedad `crearProducto` ya que no se están creando nuevos productos, sino editando uno ya añadido previamente a la lista de la compra.

5 – Eliminar un producto de la lista de la compra

La última acción que se va a implementar es la de eliminar un producto de la lista, deslizando hacia la izquierda sobre ese producto. Para ello usaremos una vista de tipo `Dismissible` y dentro de ella envolveremos a la vista `InkWell` construída en el paso anterior.

Lo único que hay que hacer para programar la acción de borrado mediante deslizamiento sobre una línea de producto es buscar dentro del método `build` de la clase `ListaCompraPantallaLlena`, las líneas de código siguientes (algunas líneas se han ocultado para mayor claridad);

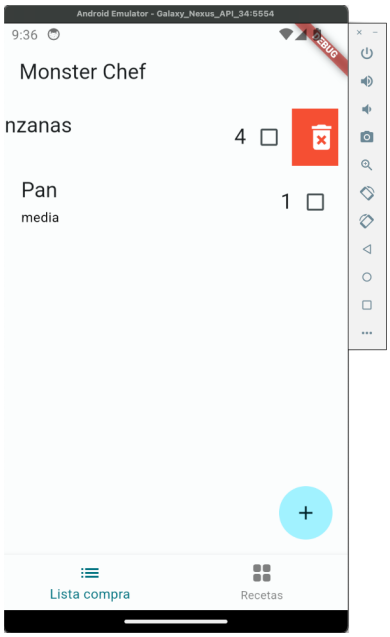
```
// 18
return InkWell(
  child: LineaProducto(
    // - El resto del contenido se ha ocultado
  ),
  onTap: () {
    Navigator.push(
      context,
      // - El resto del contenido se ha ocultado
  },
);
```

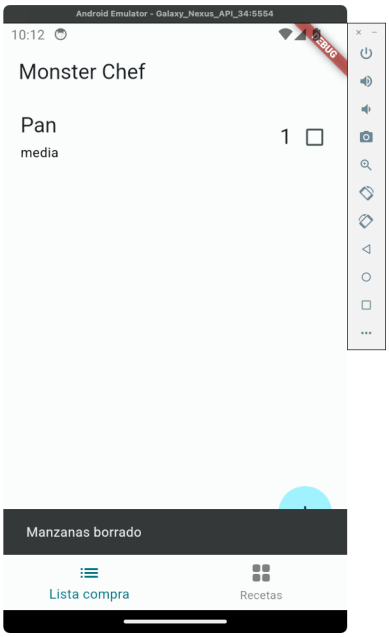
y sustituirlas por estas otras:

```
// 18
return Dismissible(
  key: Key(producto.id),
  direction: DismissDirection.endToStart,
  background: Container(
    color: Colors.red,
    alignment: Alignment.centerRight,
    child: const Icon(
      Icons.delete_forever,
      color: Colors.white,
      size: 35.0
    ),
  ),
  onDismissed: (direction) {
    listaCompra.borraProducto(index);
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text('${producto.nombre} borrado'),
      ),
    );
  },
```

```
    child: InkWell(
      key: Key(producto.id),
      child: LineaProducto(
        producto: producto,
        completar: (valor) {
          if (valor != null) {
            listaCompra.marcaCompletado(index, valor);
          }
        },
      ),
      onTap: () {
        Navigator.push(
          context,
          MaterialPageRoute(
            builder: (context) => ListaCompraAnadirProducto(
              productoOriginal: producto,
              editarProducto: (producto) {
                listaCompra.actualizaProducto(producto, index);
                Navigator.pop(context);
              },
              crearProducto: (producto) {},
            ),
          ),
        );
      },
    ),
  );
```

Si hacemos Hot Restart, añadimos algunos productos a la lista de la compra y sobre alguno de ellos deslizamos a la izquierda, veremos que bajo él se muestra una barra de color rojo con el icono de una papelera marcada con una X. Si completamos el deslizamiento, veremos que el producto correspondiente se elimina de la lista de la compra y aparece en la parte inferior de la pantalla una vista de tipo `SnackBar` informando de la acción:





Veamos algunos aspectos interesantes del código que acabamos de añadir:

1. Hemos envuelto la vista `InkWell` dentro de una vista `Dismissible`. Esta vista acepta acciones de deslizar y permite descartar la vista con una animación. Es muy apropiado para implementar operaciones de borrado, porque la animación permite al usuario percibir claramente que ese elemento del `ListView` se ha eliminado de la lista.
2. Un cambio que llama la atención es que se usa el parámetro `key:` tanto en la vista `Dismissible` como en la vista `InkWell`:

   ```
   key: Key(producto.id),
   ```

   Como valor para `key:` se usa el `id` del producto, que como sabemos se inicializa a una constante única para cada producto, y es de tipo `String`. Para generarlos usamos el paquete `Uuid` en la sesión anterior. Hasta ahora no habíamos usado la propiedad `key:` de las vistas, sin embargo, si ahora no se usa, se producen errores.

   > **Caution**
   >
   > En las vistas de tipo `Dismissible` **el parámetro** `key:` **es obligatorio,** no puede ser nulo. La razón es que las vistas `Dismissible` se usan por lo general en listas y se eliminan de la lista cuando se descartan.
   >
   > El comportamiento predeterminado de `ListView` es sincronizar las vistas que representan a cada fila en función de su índice en la lista. Si no se eliminan filas no hay problema, pero si se elimina una fila, la fila posterior a la fila descartada se sincronizaría con el estado del producto que ocupaba la fila descartada, lo que es incorrecto. El uso del parámetro `key:` hace que los widgets se sincronicen de acuerdo con sus valores de la propiedad `key:` y no por su índice, por lo que no hay ningún problema.

3. La propiedad `direction:` de la clase `Dismissible` es de tipo `DismissDirection`. Permite limitar el tipo de los deslizamientos que se permiten sobre la vista. Si no se indica nada, se puede deslizar hacia arriba, hacia abajo, hacia la derecha y hacia la izquierda.

   En esta aplicación solo se permiten deslizamientos hacia la izquierda, por eso, se le ha dado el valor `DismissDirection.endToStart`

   > **Note**
   >
   > Si se quiere deslizar hacia la izquierda y hacia la derecha, pero no hacia arriba y hacia abajo, se puede dar el valor `DismissDirection.horizontal` a esta propiedad.

4. La propiedad `background` de la vista `Dismissible` permite cambiar el color del fondo, colocar un icono, o cualquier otra vista que se quiera para que al deslizar se indique visualmente la acción que se está llevando a cabo. En este caso se usa un `Container` con fondo de color rojo y un icono blanco de una papelera.

   > **Note**
   >
   > También existe la propiedad `secondaryBackground` para poder diseñar otro fondo distinto cuando se desliza hacia la derecha.

5. En la propiedad `onDismissed:` se pasa la función anónima que describe el comportamiento que se va a tener cuando la vista ha sido descartada.

   > **Important**
   >
   > El comportamiento por defecto de la vista `Dismissible` es que siempre que se completa el deslizamiento, la vista `Dismissible` se descarta (se elimina del árbol de vistas) y se llama a la función anónima que se pasa en la propiedad `onDismissed:`.

   La función tiene un parámetro llamado `direction:` que es rellenado por la propia vista `Dismissible`. Si se han permitido varios deslizamientos en la vista, se debe usar este parámetro para comprobar cual se ha producido y realizar la acción correspondiente.

   La función anónima llama al método `borraProducto` de la `ListaCompra` pasándole como argumento `index` que indica el índice del elemento de la lista de productos que hay que eliminar.

   > **Caution**
   >
   > Si se quisiera implementar algún otro comportamiento que no implique eliminar la vista del árbol de vistas, como por ejemplo, poner otro valor de importancia al deslizar hacia la derecha, hay que pasar otra función anónima a la propiedad `confirmDismiss:` de la clase `Dismissible`. Esa función anónima debe devolver `false` en aquellas situaciones en las que la vista no se descarta, sino que se conserva y `true` en las situaciones en las que deba descartarse.
   >
   > Si no se implementa esta función, el comportamiento por defecto de la vista `Dismissible` es considerar que todos los deslizamientos implican que la vista se elimina del árbol de vistas.
   >
   > La propiedad `confirmDismiss:` da a la aplicación la oportunidad de confirmar o vetar la eliminación de la vista.

6. Se ha añadido un snack en la `SnackBar` (la barra temporal que se muestra justo por encima de la `BottomNavigationBar`) para informar al usuario de la acción de borrado. Para acceder a la `SnackBar` se usa `ScaffoldMessenger.of(context)` que permite acceder desde la vistas inferiores del árbol de vistas a la vista `Scaffold` en la que están contenidas.

   La `SnackBar` desaparece automáticamente al cabo de un tiempo predeterminado que puede modificarse con su propiedad `duration:`

   > **Note**
   >
   > En este caso la `SnackBar` solo tiene una vista `Text` indicando que se acaba de eliminar un elemento. Se pueden añadir también otras vistas, como por ejemplo un botón para deshacer el borrado si se ha producido por error.
   >
   > Para ello se usa la propiedad `action:` de la clase `SnackBar` a la que se le pasa una vista de tipo `SnackBarAction`

Almacenar la lista de la compra en el dispositivo

La aplicación lista de la compra, tal y como está en este punto del desarrollo es bastante potente, pero tiene un grave problema: los datos no se almacenan de forma permanente en el dispositivo y por tanto se pierden cada vez que se sale de la aplicación. En este apartado vamos a aprender a leer / salvar datos a un archivo local en nuestro dispositivo.

> **Note**
>
> Los dos principales sistemas operativos para dispositivos móviles (Android e iOS) implementan una técnica conocida como **sandboxing.** Esta técnica se utiliza para aislar las aplicaciones, permitiendo que cada una opere en su propio entorno seguro o **sandbox.** Este aislamiento ayuda a proteger los datos del usuario y al sistema operativo de actividades maliciosas y errores de software.
>
> Cada aplicación se ejecuta en su propio espacio de memoria y no puede acceder directamente a los recursos o datos de otras aplicaciones sin permisos explícitos. Esto significa que, por defecto, una aplicación no puede leer ni escribir archivos pertenecientes a otra aplicación. En la práctica, si quieres que tu aplicación móvil escriba en su sandbox, debes pedir al sistema operativo que lo gestione.
>
> Afortunadamente Flutter simplifica todo ese proceso y aisla a nuestro programa de las peculiaridades de cada sistema operativo.

Dart/Flutter proporcionan el plug-in `path_provider` que permite el acceso de forma independiente del dispositivo a las ubicaciones comunes (sandbox) donde las aplicaciones pueden almacenar sus propios archivos. En esta aplicación grabaremos nuestro archivo en el directorio **documentos** que es un directorio en el que la aplicación almacena archivos a los que solo ella puede acceder.

Lo bueno de usar el plug-in `path_provider` es que no nos tenemos que preocupar de en qué plataforma se esté ejecutando la aplicación, iOS o Android, ya que el propio plug-in se encarga de averiguarlo y usar la clase correcta que en iOS es `NSDocumentDirectory` y en Android `AppData`.

> **Important**
>
> El sistema operativo borra automáticamente el directorio cuando se desinstala la aplicación.

6 – Decidir el formato del archivo JSON

Usaremos el formato JSON para formatear los productos de la lista de la compra en el archivo local.

> **Note**
>
> El formato JSON se usa muchísimo para serializar e intercambiar información dado que es muy sencillo de leer e interpretar para humanos y muy sencillo de escribir y leer en código.
>
> Se basa en dos estructuras: mapa y lista, que se implementan en casi todos los lenguajes de programación, por tanto su lectura y escritura es simple ya que lo que se lee se puede volcar casi directamente a estructuras de datos en memoria.

> **Note**
>
> La explicación del formato JSON va más allá de esta asignatura, pero puede encontrarse una buena descripción aquí: https://www.json.org/json-es.html

La lista de productos de la lista de la compra puede representarse en formato JSON de varias formas, por tanto, es necesario decidir qué formato se le va a dar en esta aplicación.

Lo más sencillo es representar directamente la lista de objetos de la clase `Producto`, por tanto el archivo deberá comenzar y terminar con corchetes `[ ]`. Cada `Producto` se representará como un objeto, es decir sus propiedades encerradas entre llaves `{ }`. Cada objeto `Producto` en el archivo JSON se separa del siguiente mediante una coma. Las propiedades de los objetos se representan mediante pares nombre de propiedad, valor de la propiedad, separados por dos puntos.

Un posible ejemplo de una lista de la compra en formato JSON, tendrá este aspecto:

```
[
  {
    "id": "e8e1ab20-b746-11ec-a635-6f01405b6be7",
    "nombre": "Manzanas",
    "importancia": "media",
    "cantidad": 4,
    "completado": false
  },
  {
    "id": "fd299fc0-b746-11ec-a635-6f01405b6be7",
    "nombre": "Chocolate",
    "importancia": "low",
    "cantidad": 1,
    "completado": true
  }
]
```

Los corchetes `[ ]` externos indican que se trata de una lista de objetos. Cada objeto va encerrado entre pares de llaves `{ }` y separado de los demás por comas `,`

Dentro de cada objeto se tienen todas sus propiedades mediante pares, en los que el primer elemento del par es una **cadena** con el nombre de la propiedad y el segundo es el valor de esa propiedad. Los dos elementos van separados por punto y coma `:` Por ejemplo: `"cantidad": 4`

Cada propiedad de un objeto `Producto` va separada de las demás propiedades del mismo producto por comas `,`

> **Important**
>
> No se pueden poner comas ni detrás del último objeto de la lista, ni detrás de la última propiedad de un objeto. Si se hace, se tiene un error en el formato JSON.

Ahora que tenemos claro el formato JSON que aplicaremos a nuestra lista de productos, vamos a implementar la escritura / lectura.

7 – Preparar el tipo enumerado Importancia

Comenzaremos preparando al tipo enumerado `Importancia` para que sea capaz de generar una constante del tipo enumerado a partir de su nombre en formato `String`. Necesitaremos esta funcionalidad para poder leer de un archivo JSON valores de importancia para un producto.

A la hora de leer el archivo JSON, para cada producto de la lista de la compra, se va a obtener un `String` con el nombre de una constante del tipo enumerado `Importancia`. Ese `String` debe transformarse en una constante del tipo enumerado `Importancia` antes de pasarla al constructor de `Producto`. Para ello añade el siguiente método de clase al final del tipo enumerado.

```
static Importancia getImportanciaDesde({required String nombre}) {
  return Importancia.values.byName(nombre);
}
```

> **Important**
>
> El método anterior es un método del tipo `Importancia` por tanto debes escribirlo **dentro** de la declaración del tipo enumerado (justo antes de la llave de cierre). También debes quitar la coma `,` que hay al final de la constante `alta` del tipo enumerado y en su lugar poner un punto y coma `;`

Veamos cómo funciona este método de clase:

1. El método `getImportanciaDesde` hace uso de la propiedad `values` de cualquier tipo enumerado, que en este caso devuelve una lista de tipo `List<Importancia>` que contiene todos los valores del tipo enumerado.

   Una vez que se tiene esa lista, se usa el método `byName` que encuentra dentro de ella el valor del tipo enumerado `Importance` cuyo nombre simbólico coincide con el parámetro `nombre`.

> **Note**
>
> Como recordarás, para obtener un `String` que represente a una constante de un tipo enumerado, se puede usar la propiedad `name` que tienen todos los enumerados. Por tanto, para guardar un valor de `Importancia` en un archivo JSON no hay que modificar nada en el código actual.

8 – Preparar la clase Producto

Para simplificar el proceso de escribir un `Producto` a un archivo en formato JSON, añadiremos un método `aJson` a la clase `Producto`:

```
String aJson() {
  var json = '''
{
  "id": "$id",
  "nombre": "$nombre",
  "importancia": "${importancia.name}",
  "cantidad": $cantidad,
  "completado": $completado
}''';
  return json;
}
```

El funcionamiento de este método es sencillo:

1. El método construye una cadena multilínea de Dart, para ello comienza y termina la cadena con `'''`

   > **Important**
   >
   > Recuerda en cuando se usan cadenas multilínea, el sangrado es muy importante, por eso el contenido de la cadena no respeta las normas de sangrado estándar de Dart.

2. Dentro de la cadena, la primera línea es una llave abierta `{`
3. En las líneas interiores, se escribe el nombre de cada propiedad encerrado entre comillas dobles `"` seguido de dos puntos `:` y el valor de la propiedad. Es un proceso sencillo pero hay que tener en cuenta algunos aspectos:
   - **Los nombres de propiedad en JSON son siempre cadenas de caracteres,** por eso al pasarlos a formato JSON se encierran entre comillas dobles `"`. Por ejemplo: `"id"`
   - JSON solo permite usar como valores de propiedad: cadenas de caracteres, números, true, false, null, arrays (de JSON) y objetos (de JSON). Por eso, si una clase tiene propiedades que no sean de alguno de esos tipos, debe transformarse a uno que pueda transformarse fácilmente. En este caso, la propiedad `"importancia"` se almacena como una cadena con el nombre de la constante del tipo enumerado: `"${importancia.name}"`
   - Tras todas las propiedades, **menos la última,** se debe escribir una coma `,` ya que es el carácter que se usa en JSON como delimitador.

     > **Caution**
     >
     > Si no se ponen comas entre propiedades o bien se pone una coma tras la última propiedad de un `Producto` se tiene un error en el formato JSON.

4. La última línea es una llave cerrada `}`

Usaremos las capacidades de Dart para parsear un archivo JSON y encontrar los distintos elementos que contiene. Cuando encuentre un objeto describiendo a un producto de la lista de la compra, tendremos que transformar la estructura parseada por Dart en una instancia de la clase `Producto`. Para ello usaremos un nuevo método llamado `desdeJson` que añadiremos al final de la clase `Producto` justo detrás del método `aJson`:

```
factory Producto.desdeJson(Map<String, dynamic> json) {
  return Producto(
    id: json['id'],
    nombre: json['nombre'],
    importancia: Importancia.getImportanciaDesde(nombre: json['importancia']),
    cantidad: json['cantidad'],
    completado: json['completado'],
  );
}
```

Veamos su funcionamiento:

1. La estructura parseada por Dart para cualquier objeto JSON de un archivo JSON es: `Map<String, dynamic> json`, por eso el parámetro `json` del método `desdeJson` tiene este tipo. Veamos por qué:
    - Las propiedades de cada objeto en formato JSON se dan en forma de grupo de pares nombre-valor separados por `:`. Eso en Dart se implementa como un mapa, de ahí que la estructura sea de tipo `Map`
    - Los nombres de propiedad en un objeto JSON **siempre son cadenas de caracteres,** por eso el tipo de la llave del mapa es `String`
    - Los valores pueden ser de cualquiera de los tipos vistos anteriormente. El único tipo en Dart que los representa a todos es `dynamic`. Por eso el tipo del valor del mapa es `dynamic`. Es un tipo peligroso, pero en este caso su uso no tiene problema.
2. El método está marcado como `factory` ¿Recuerdas cuáles eran las ventajas de esta clase de métodos? ¿Por qué crees que se usa aquí?
3. Devuelve un `Producto`. Para ello se usa el constructor generativo personalizado de la clase `Producto` y a cada propiedad le pasa el valor asociado a la llave con el mismo nombre en el mapa. Así por ejemplo, a la propiedad `completado:` le pasa el valor en el mapa para esa propiedad: `json['completado']`

    > **Important**
    >
    > El tipo de cada valor es `dynamic` por lo que el compilador compila ese código sin problemas. No es problema usar `dynamic` porque conocemos perfectamente la estructura de las propiedades del objeto, así que con un poco de cuidado, asignaremos el valor correcto a cada una. Aunque este valor es de tipo `dynamic` corresponde perfectamente con la propiedad a la que se asigna, por lo que Dart lo puede convertir sin problemas.

4. El caso de la propiedad `importancia:` es especial, porque los valores de `Importancia` no son valores que el formato JSON gestione de forma correcta. El valor de `Importancia` se convirtió a `String` al guardarlo en el archivo y ahora hay que transformarlo a la inversa, de `String` a una constante del tipo enumerado `Importancia`. Para eso se usa el método de clase `getImportanciaDesde` que se añadió al tipo enumerado en el paso anterior.

```
importancia: Importancia.getImportanciaDesde(nombre: json['importancia']),
```

9 – Añadir las nuevas dependencias al proyecto

Como ya se ha dicho, se necesita el plug-in `path_provider` para acceder, de forma independiente del dispositivo, al sandbox de la aplicación. Por eso, antes de continuar añadiremos la dependencia `path_provider` a nuestro proyecto.

Para ello como sabemos, la forma mas sencilla es abrir un terminal o símbolo del sistema, navegar hasta la carpeta raíz del proyecto flutter `flutter_session_2_4` y ahí ejecutar:

```
flutter pub add path_provider
```

A partir de este momento, en nuestro proyecto tendremos capacidad para usar las clases y métodos que permiten el acceso al sandbox de la aplicación.

10 – Encontrar la ruta correcta al directorio de la aplicación

Cada plataforma, iOS y Android, usan un directorio diferente para los documentos propios de cada aplicación, por eso, el primer paso siempre es averiguar cuál es la ruta al directorio **documentos.** Para ello se usa la orden:

```
getApplicationDocumentsDirectory()
```

que pertenece al plug-in `path_provider` por lo que para poder usarla se debe añadir a la clase `ListaCompra` el import:

```
import 'package:path_provider/path_provider.dart';
```

A continuación añadiremos al final de la clase `ListaCompra` el siguiente getter privado calculado:

```
Future<String> get _localPath async {
  final directory = await getApplicationDocumentsDirectory();
  return directory.path;
}
```

> **Important**
>
> Si no se ha añadido el plug-in `path_provider` como una dependencia del proyecto flutter, lo anterior no funcionará (véase el paso 9 – Añadir las nuevas dependencias al proyecto).

Veamos como funciona:

1. La petición de acceso al directorio **documentos** de la aplicación se resuelve por el sistema operativo (en una hebra distinta), por lo que debe ejecutarse en modo asíncrono. Así, si hay algún retraso (por ejemplo si el SSOO está ocupado con otras tareas), no se va a intentar crear un archivo dentro de ese directorio antes de tener acceso al propio directorio.
2. La signatura de: `getApplicationDocumentsDirectory()` es:

   ```
   Future<Directory> getApplicationDocumentsDirectory()
   ```

   por tanto debe usarse en un método marcado como `async`. Estos métodos siempre deben devolver un `Future`.

   > **Note**
   >
   > En la siguiente sesión haremos un uso más profundo de las llamadas asíncronas y los `Future`. Por eso en la documentación de esta no se explica a fondo.

3. Para construir un archivo en el directorio (se hace en el siguiente paso) hace falta la ruta hasta ese directorio. Esta ruta se almacena en forma de cadena de caracteres `String`, por tanto el valor devuelto es `directory.path` que devuelve la cadena de caracteres que contiene la ruta hasta el directorio.

11 – Construir una referencia al archivo

Una vez que se tiene la ruta del directorio **documentos** (averiguada en el paso anterior) hay que construir una referencia al archivo en el que se va a almacenar la lista de la compra. Esto se hace mediante la clase `File` de la biblioteca `dart:io` (para usarla hay que añadir el correspondiente `import`).

Al final de la clase `ListaCompra` añade otro getter privado calculado de esta forma:

```
Future<File> get _localFile async {
  final path = await _localPath;
  return File('$path/productos.json');
}
```

Veamos cómo funciona:

1. El getter `_localFile` devuelve un objeto de la clase `File` (un archivo local) que se encuentra en la ruta compuesta por el directorio **documentos** de la aplicación, seguido del nombre del archivo, en este ejemplo se le ha llamado **productos.json**
2. Para ello primero llama de forma asíncrona al getter `_localPath` escrito en el paso anterior.

Una vez que se tienen los dos getters privados `_localFile` y `_localPath`, ya podemos grabar la lista de la compra.

12 – Salvar la lista de la compra

Para salvar la lista de la compra añade a la clase `ListaCompra` el siguiente método privado:

```
Future<void> _salvaProductos() async {
  final file = await _localFile;

  var cadena = '[\n';
  for (int i=0; i<_productos.length; i++) {
    cadena += _productos[i].aJson();
    if (i<_productos.length-1) {
      cadena += ',\n';
    } else {
      cadena += '\n';
    }
  }
  cadena += ']';
  file.writeAsString(cadena);
}
```

Veamos algunas consideraciones sobre este método:

1. El método usa el getter asíncrono `_localFile`, por tanto debe marcarse como `async` y por tanto debe devolver `Future<void>`.

   > **Note**
   >
   > En este caso, se podría devolver solo `void` pero de esta forma (devolviendo `Future<void>`) queda más claro que es una llamada asíncrona.

2. Una vez obtenida la referencia al archivo de la lista de la compra, hay que construir una cadena de caracteres con el archivo JSON y después grabarla en el disco con la orden `writeAsString` de la clase `File.`
3. Hemos decidido grabar en el archivo JSON directamente la lista de la compra que en JSON debe ir entre corchetes `[ ]`, por tanto, los primeros caracteres que se guardan en la cedena son `[\n` un corchete y un salto de renglón, y el último es `]` un corchete.
4. Para cada elemento de la lista de la compra almacenada en `_productos` se construye un `String` con los datos de ese elemento en formato JSON. Para ello se usa el método `aJson` que añadimos anteriormente a la clase `Producto` y el resultado se añade al `String` llamado `cadena`.
5. En el archivo JSON, cada elemento de la lista va encerrado entre llaves `{ }` y separado de los demás por `,` salvo el último que no debe llevar coma porque si no, la función `jsonDecode` puede fallar. Por eso, en cada paso se añade al `String` `cadena` `,\n` (una coma y un salto de renglón) excepto en el último donde solo se añade el salto de renglón `\n`

   > **Note**
   >
   > Podría haberse usado el operador condicional ternario `(condición) ? valorSiVerdadero : valorSiFalso;` pero en este ejemplo se ha usado un `if-else` tradicional para mayor claridad.

> **Important**
>
> Esta forma de construir la cadena que representa a la lista de la compra en formato JSON **no es la mejor.** ¿Adivinas por qué? ¿Cómo rediseñarías la construcción de la cadena para que el proceso fuese mucho más eficiente? Refactoriza tu código siguiendo ese nuevo diseño.

13 – ¿Dónde y cuándo se realiza la escritura?

Cabe preguntarse ¿Cuándo se debe grabar la lista de la compra? En este caso lo mejor es grabarla cada vez que se modifique. Es una estructura simple, con pocos elementos, que se escribe muy rápidamente, por lo que su grabación en el dispositivo cada vez que haya un cambio no ralentiza la ejecución de la aplicación. De esta forma, si hay algún problema y la aplicación se cierra inesperadamente, la lista de la compra está completamente actualizada.

Por ello, en **todos** los métodos que modifican la lista de la compra justo antes de `notifyListeners()` hay que llamar a `_salvaProductos();` Por ejemplo, el método `anadeProducto` de la clase `ListaCompra` debe reescribirse así:

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

> **Important**
>
> No te olvides de reescribir los demás métodos que modifican la lista de la compra, o no se salvará correctamente en el dispositivo.

14 – Leer la lista de la compra

Para leer la lista de la compra añade a la clase `ListaCompra` el siguiente método privado:

```
Future<void> _leeProductos() async {
  try {
    final file = await _localFile;
    final productosString = await file.readAsString();
    final List<dynamic> productosJson = jsonDecode(productosString);
    for (var prodJson in productosJson) {
      _productos.add(Producto.desdeJson(prodJson));
    }
    notifyListeners();
  } on FileSystemException catch (e) {
    return;
  }
}
```

No olvides escribir el siguiente import:

```
import 'dart:convert';
```

Necesario para acceder a la función: `jsonDecode`, proporcionada por Dart.

Añádele también el siguiente método de utilidad:

```
void init() {
  _leeProductos();
}
```

Veamos algunas consideraciones sobre el código que acabamos de añadir:

- En primer lugar el método `_leeProductos` usa el getter asíncrono `_localFile`, por tanto debe marcarse como `async` y por tanto debe devolver `Future<void>`.
- En este método hay varias cosas que podrían salir mal, por eso el cuerpo del método se encierra en un bloque `try-catch`. Si no existe el archivo (esto ocurre la primera vez que se arranca la aplicación) o no tiene un formato JSON parseable por `jsonDecode` se lanza una excepción y se sale, por lo que la lista de la compra no se rellena con ningún elemento leído. Se inicializa a una lista de productos vacía.

  > **Important**
  >
  > En esta situación no es necesario tratar de ninguna forma especial la excepción, por eso se usa la instrucción `return` en el cuerpo del `catch`. Al salir directamente del método se tiene una lista de la compra inicializada por su constructor, es decir una lista vacía, que es la situación correcta en ese caso.
  >
  > Esta es la razón por la que se separa la creación de la lista, de su inicialización. Consulta el método de utilidad: `init` y la forma en que se usa en la sección 15 - ¿dónde y cuándo se realiza la lectura?

- Para leer el archivo que contiene la lista de la compra en formato JSON a una cadena de caracteres se usa el método de la clase `File` `readAsString`
- A continuación se parsea mediante la función `jsonDecode`. Esta función recibe una cadena de caracteres, la analiza y devuelve el objeto JSON que contiene, que como sabemos solo puede ser: cadenas de caracteres, números, true, false, null, arrays (de JSON) y objetos (de JSON). En este caso, lo que se guarda en el archivo es la lista de productos, es decir se trata de un array de JSON, por tanto el tipo leído es `List<dynamic>`. Cada uno de los elementos de la lista es un producto, por lo tanto su tipo es `Map<String,dynamic>`.

  > **Note**
  >
  > Recuerda que los objetos JSON se codifican mediante pares nombre : valor, donde el nombre es siempre un `String` y el valor puede ser algún elemento de entre los siguientes: cadenas de caracteres, números, true, false, null, arrays (de JSON) y objetos (de JSON).

- Cada elemento de `productosJson` que es la representación en JSON de un elemento de la lista de la compra, se convierte en un objeto de la clase `Producto` mediante su método `desdeJson` que le añadimos en pasos anteriores.

- Los distintos objetos `Producto` se van añadiendo a la lista de la compra `_productos` a medida que se decodifican. Al final del proceso se tiene reconstruída la lista de la compra que se guardó en disco.
- La clase `ListaCompra` es un observable, que como sabes, en Flutter se implementa mediante una subclase de `ChangeNotifier`. Siempre que se cambia el estado del observable, hay que notificar a los observadores `Consumer` que haya registrados. Por eso, justo tras decodificar el archivo JSON hay que llamar al método `notifyListeners`.
- También se ha añadido un método de utilidad `init` para facilitar la separación de la creación de la lista de la compra, de su inicialización.

15 – ¿Dónde y cuándo se realiza la lectura?

La lectura de la lista de la compra debe realizarse una sola vez, justo al principio de la ejecución de la aplicación por eso cabe preguntarse ¿dónde y cuándo se debe realizar esa lectura?

Lo mejor es hacerlo justo al arrancar la aplicación, justo antes de comenzar la construcción de las vistas que están contenidas en la clase `MonsterChefApp`.

Abre el archivo: `monster_chef_app.dart` y justo antes de la línea `return MaterialApp(` añade las siguientes líneas de código:

```
final listaCompra = ListaCompra();
listaCompra.init();
```

A continuación busca las siguientes líneas de código:

```
// 24
ChangeNotifierProvider(create: (context) => ListaCompra(),),
```

Y sustitúyelas por estas otras:

```
// 24
ChangeNotifierProvider(create: (context) => listaCompra,),
```

Veamos cómo funcionan estos cambios:

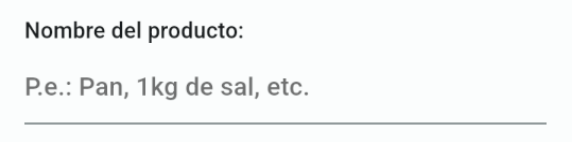
- Como puede verse, ahora no se crea la `ListaCompra` justo al pasárselo al `ChangeNotifierProvider`, sino que se construye antes, justo al principio del método `build`
- Se separa la construcción de la `ListaCompra` de su inicialización mediante el método de utilidad `init` que añadimos en el paso anterior, esta es siempre una buena forma de proceder.

  > **Important**
  >
  > Primero se construye una nueva instancia de la clase `ListaCompra` lo que hace que se tenga una lista de la compra vacía. A continuación se llama al método `init` de la clase `ListaCompra` sobre esa nueva instancia.
  >
  > Si hay un archivo en el sandbox de la aplicación que contiene una lista de la compra, se lee y con los productos que contiene se rellena la propiedad `_productos`.
  >
  > Si no hay archivo (es la primera vez que se ejecuta la aplicación) o bien el archivo no puede leerse o el formato no es correcto, se produce una excepción que se trata saliendo del método sin hacer nada, con lo que la lista sigue vacía.

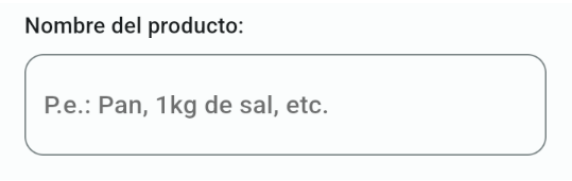
En este punto ya tenemos terminada la aplicación lista de la compra, en la que se pueden añadir, modificar y borrar los productos, al mismo tiempo que se guarda en el dispositivo para futuros usos.

Mini ejercicios

1. ➲ Los usuarios esperan ver los elementos de interacción con un determinado aspecto visual. Si se les cambia ese aspecto, es posible que no los reconozcan como elementos de interacción. Por ejemplo, en esta versión de la aplicación lista de la compra, el cuadro de texto para escribir el nombre del producto se muestra así:

   Mientras que la mayoría de usuarios espera ver los cuadros de entrada de texto, con este aspecto:

   Cambia el cuadro de entrada de texto para que tenga un aspecto más familiar para los usuarios de la aplicación.

   > **Tip**
   >
   > Entra en api.flutter.dev y busca documentación sobre `OutlineInputBorder` y `TextField`. Con eso podrás completar el ejercicio.

2. ➲ La lista de la compra tal y como está no proporciona la mejor experiencia de uso. Hay pasos que se podrían evitar al usuario. Identifica esos pasos superfluos e indica qué se podría hacer para resolverlos.

3. ➲ En la lista de la compra tal y como está, también hay algunas acciones de interacción que son difíciles de realizar, lo que hace que la experiencia de uso sea menor. Identifica esas acciones e indica qué se podría hacer para mejorar la interacción.
4. ➲ Refactoriza el programa para que incorpore los nuevos diseños que has realizado en los dos ejercicios anteriores.
5. ➲ Añade a la aplicación la capacidad de observar un nuevo gesto. Cada vez que se deslice hacia la derecha sobre un producto de la lista de la compra, se cambiará su propiedad `_importancia` al valor `Importancia.alta`.

   > **Tip**
   >
   > Lee bien los comentarios que hay en secciones anteriores de esta documentación relativos a la vista `Dismissible` porque ahí tienes todas las pistas que necesitas para completar el ejercicio.

6. ➲ Añade a la aplicación la capacidad de mostrar mediante una insignia, el número de productos sin comprar que haya en la lista de la compra en un momento dado. Si no hay ningún producto sin comprar, no debe mostrar ningún badge.

   > **Tip**
   >
   > Piensa primero en la información que necesitas observar de la clase `ListaCompra` para poder completar este ejercicio, a continuación refactoriza la `ListaCompra` y por último, con esa información refactoriza la barra de navegación para que incluya la insignia.

7. ➲ Añade a la aplicación la capacidad de eliminar completamente todos los productos de una lista de la compra.

   > **Tip**
   >
   > Piensa en qué patrón de interacción es el más apropiado para añadir esta funcionalidad. A continuación piensa qué funcionalidad necesitas en la clase `ListaCompra` para realizar la tarea, a continuación refactoriza la `ListaCompra` y por último refactoriza la interfaz para añadir el nuevo patrón de interacción.