Comprensión del árbol de widgets

El árbol de widgets es cómo creas tu interfaz de usuario; colocas widgets entre sí para crear diseños simples y complejos. Dado que casi todo en el framework de Flutter es un widget, y cuando comienzas a anidarlos, el código puede volverse más difícil de seguir. Una buena práctica es tratar de mantener el árbol de widgets lo más superficial posible. Para comprender todos los efectos de un árbol profundo, verás un árbol completo de widgets y luego lo refactorizarás en un árbol de widgets poco profundo, haciendo que el código sea más manejable. Aprenderás tres formas de crear un árbol de widgets poco profundo mediante la refactorización: con una constante, con un método y con una clase de widget.

Introducción a los widgets

Antes de analizar el árbol de widgets, veamos la breve lista de widgets que utilizarás para las aplicaciones de ejemplo de este tema (práctica). En este punto, no te preocupes por comprender la funcionalidad de cada widget; solo concéntrate en lo que sucede cuando anidas los widgets y en cómo puedes separarlos en secciones más pequeñas. En el siguiente tema, "Uso de widgets comunes", analizarás en profundidad el uso de los widgets más comunes por funcionalidad.

Como se mencionó en el tema anterior, utilizamos Material Design para todos los ejemplos. Los siguientes son los widgets (utilizables solo con Material Design) que usarás para crear los proyectos de árbol de widgets completos y poco profundos para esta práctica:

- *Scaffold*: Implementa el diseño visual de Material Design, lo que permite el uso de los widgets Material Components de Flutter.

- *AppBar*: Implementa la barra de herramientas en la parte superior de la pantalla.

- *CircleAvatar*: Generalmente se usa para mostrar una foto de perfil de usuario redondeada, pero puedes usarla para cualquier imagen.

- Divider: Dibuja una línea horizontal con relleno arriba y abajo.

Si la aplicación que estás creando usa Cupertino, puedes usar los siguientes widgets. Ten en cuenta que con Cupertino puedes utilizar dos andamios (scaffolds) diferentes, un andamio de página o un andamio de pestañas (tab).

- *CupertinoPageScaffold*: Implementa el diseño visual de iOS para una página. Funciona con CupertinoNavigationBar para proporcionar el uso de los widgets estilo iOS de Cupertino de Flutter.

- *CupertinoTabScaffold*: Implementa el diseño visual de iOS. Esto se usa para navegar por varias páginas, con las pestañas en la parte inferior de la pantalla que le permiten usar los widgets estilo iOS de Cupertino de Flutter.

- *CupertinoNavigationBar*: Implementa la barra de herramientas de diseño visual de iOS en la parte superior de la pantalla.

La tabla 5.1 resume una breve lista de los diferentes widgets que se pueden usar según la plataforma.

Tabla 5.1: Material Design vs. Cupertino Widgets.

Material Design	Cupertino
Scaffold	CupertinoPageScaffold CupertinoTabScaffold
AppBar	CupertinoNavigationBar
CircleAvatar	n/a
Divider	n/a

Los siguientes widgets se pueden utilizar tanto con Material Design como con Cupertino:

- *SingleChildScrollview*: Eesto agrega la capacidad de desplazamiento vertical u horizontal a un solo widget secundario.

- *Padding*: Añade relleno izquierdo, superior, derecho e inferior.

- *Column*: Muestra una lista vertical de widgets secundarios.

- *Row*: Muestra una lista horizontal de widgets secundarios.

- *Container*: Este widget se puede usar como un marcador de posición vacío (invisible) o puede especificar la altura, el ancho, el color, la transformación (rotar, mover, sesgar) y muchas más propiedades.

- *Expanded*: Expande y llena el espacio disponible para el widget secundario que pertenece a un widget de columna o fila.

- *Text*: El widget de texto es una excelente manera de mostrar etiquetas en la pantalla. Puede configurarse para ser una sola línea o varias líneas. Se puede aplicar un argumento de estilo opcional para cambiar el color, la fuente, el tamaño y muchas otras propiedades.

- *Stack*: ¡qué widget tan poderoso! Stack te permite apilar widgets uno encima del otro y usar un widget posicionado (opcional) para alinear cada elemento secundario de la pila para el diseño necesario. Un gran ejemplo es el ícono de un carrito de compras con un pequeño círculo rojo en la parte superior derecha para mostrar la cantidad de artículos que comprar.

- *Positioned*: El widget Positioned funciona con el widget Pila para controlar la posición y el tamaño de los hijos. Un widget posicionado te permite establecer la altura y el ancho. También puedes especificar la distancia de ubicación de la posición desde los lados superior, inferior, izquierdo y derecho del widget Pila.

Has aprendido acerca de cada widget que implementarás durante el resto de este tema. Ahora crearás un árbol de widgets completo y luego aprenderás a refactorizarlo en un árbol de widgets poco profundo.

Construyendo el árbol completo de widgets

Para mostrar cómo un árbol de widgets puede comenzar a expandirse rápidamente, usarás una combinación de widgets Columna (Column), Fila (Row), Contenedor (Container), CircleAvatar, Divider, Padding y Text.
Examinarás más de cerca estos widgets en el siguiente tema. El código que escribirá es un ejemplo simple y podrás ver inmediatamente cómo el árbol de widgets puede crecer rápidamente (figura 5.1).

Creación del árbol completo de widgets

Crea un nuevo proyecto de Flutter llamado widget_tree. Puedse seguir las instrucciones del tema anterior.

Para este proyecto, solo necesitas crear la carpeta de páginas (pages). Puedes ver el árbol de widgets completo al final de los pasos.

1. Abre el archivo home.dart.

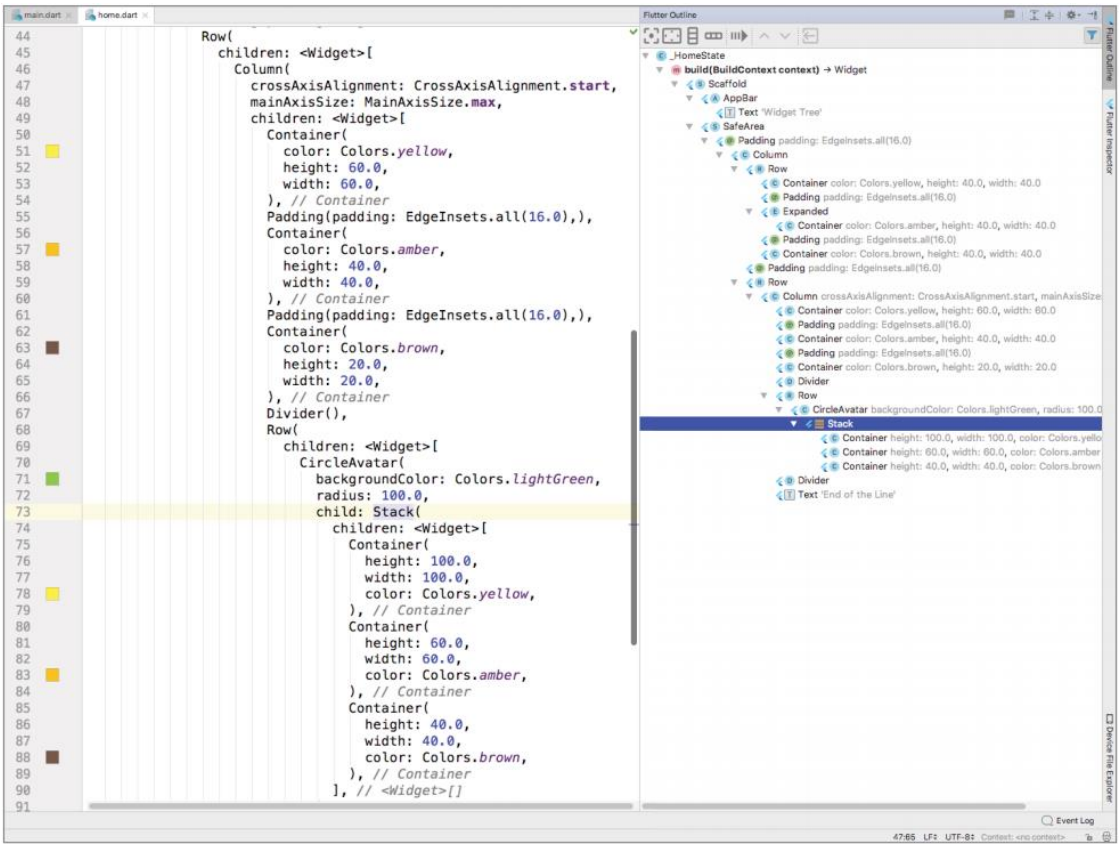


Figura 5.1: Vista de árbol completa de widgets.

2. Agrega a la propiedad del cuerpo de Scaffold un widget SafeArea con la propiedad secundaria (child) establecida en SingleChildScrollview. Agrega un PAdding widget como elemento secundario (child) de SingleChildScrollView. Establece la propiedad de relleno (padding) en EdgeInsets.all (16.0).

```
body: SafeArea(
   child: SingleChildScrollView(
      child: Padding(
         padding: EdgeInsets.all(16.0),
      ),
   ),
),
```

3. Agrega a la propiedad secundaria (child) Padding un widget Column con la propiedad secundaria (children) establecida en Row.

```
body: SafeArea(
   child: SingleChildScrollView(
      child: Padding(
         padding: EdgeInsets.all(16.0),
```

```
            child: Column(
              children: <Widget>[
                Row(
                  children: <Widget>[
                  ],
                ),
              ],
            ),
          ),
        ),
      ),
```

4. Agrega los children widgets de Row en este orden: Container, Padding, Expanded, Padding, Container y Padding. No has terminado de agregar widgets; en el siguiente paso, agregarás un widget de fila con varios widgets anidados.

```
Row(
  children: <Widget>[
    Container(
      color: Colors.yellow,
      height: 40.0,
      width: 40.0,
    ),
    Padding(padding: EdgeInsets.all(16.0),),
    Expanded(
      child: Container(
        color: Colors.amber,
        height: 40.0,
        width: 40.0,
      ),
    ),
    Padding(padding: EdgeInsets.all(16.0),),
    Container(
      color: Colors.brown,
      height: 40.0,
      width: 40.0,
    ),
  ],
)
```

5. Agrega un Padding widget para crear un espacio antes del siguiente Row widget.

```
Padding(padding: EdgeInsets.all(16.0),),
```

6. Agrega un Row widget con la propiedad children establecida en una Column. Agrega a los elementos secundarios (children) de la columna un Container, Padding, Container, Padding, Container, Divider, Row, Divider y Text. Aún no has terminado de agregar widgets y, en el siguiente paso, agregarás otro widget de fila con varios widgets anidados.

```
Row(
    children: <Widget>[
      Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        mainAxisSize: MainAxisSize.max,
        children: <Widget>[
          Container(
            color: Colors.yellow,
            height: 60.0,
            width: 60.0,
          ),
          Padding(padding: EdgeInsets.all(16.0),),
          Container(
            color: Colors.amber,
            height: 40.0,
            width: 40.0,
          ),
          Padding(padding: EdgeInsets.all(16.0),),
          Container(
            color: Colors.brown,
            height: 20.0,
            width: 20.0,
          ),
          Divider(),
          Row(
            children: <Widget>[
              // Next step we'll add more widgets
            ],
          ),
          Divider(),
          Text('End of the Line'),
        ],
      ),
    ],
),
```

7. Modifica el último Row widget (del paso 6) y establece la propiedad children en un CircleAvatar con un child como una Stack. Agrega a la propiedad Stack children tres Container widgets.

```
Row(
    children: <Widget>[
      CircleAvatar(
        backgroundColor: Colors.lightGreen,
        radius: 100.0,
        child: Stack(
          children: <Widget>[
            Container(
              height: 100.0,
              width: 100.0,
              color: Colors.yellow,
            ),
            Container(
              height: 60.0,
              width: 60.0,
              color: Colors.amber,
            ),
            Container(
              height: 40.0,
              width: 40.0,
              color: Colors.brown,
            ),
          ],
        ),
      ),
    ],
),
```

8. Después del widget Stack (del paso 7), agrega un widget Divider y luego un widget Text con una cadena de 'End of the Line'.

```
Divider(),
Text('End of the Line'),
```

Agregaste muchos widgets anidados para crear un diseño complejo. A continuación, se muestra el código completo. En una aplicación del mundo real, esto es común. Inmediatamente comenzarás a ver cómo puedes crecer el árbol de widgets, haciendo que el código sea ilegible e inmanejable. Para mantener el ejemplo centrado en la rapidez con que puedes crecer el árbol de widgets, aquí usamos widgets básicos. En una aplicación de nivel

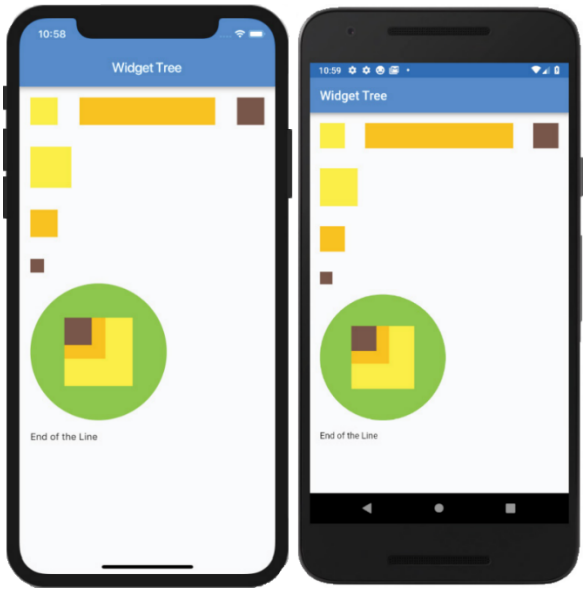
de producción, tendrías aún más widgets, como campos de texto, para permitir que el usuario ingrese texto.

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

class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Widget Tree'),
      ),
      body: SafeArea(
        child: SingleChildScrollView(
          child: Padding(
            padding: EdgeInsets.all(16.0),
            child: Column(
              children: <Widget>[
                Row(
                  children: <Widget>[
                    Container(
                      color: Colors.yellow,
                      height: 40.0,
                      width: 40.0,
                    ),
                    Padding(padding: EdgeInsets.all(16.0),),
                    Expanded(
                      child: Container(
                        color: Colors.amber,
                        height: 40.0,
                        width: 40.0,
                      ),
                    ),
                    Padding(padding: EdgeInsets.all(16.0),),
                    Container(
                      color: Colors.brown,
                      height: 40.0,
                      width: 40.0,
                    ),
                  ],
                ),
                Padding(padding: EdgeInsets.all(16.0),),
                Row(
                  children: <Widget>[
                    Column(
                      crossAxisAlignment: CrossAxisAlignment.start,
                      mainAxisSize: MainAxisSize.max,
                      children: <Widget>[
                        Container(
                          color: Colors.yellow,
                          height: 60.0,
                          width: 60.0,
                        ),
                        Padding(padding: EdgeInsets.all(16.0),),
                        Container(
                          color: Colors.amber,
                          height: 40.0,
                          width: 40.0,
                        ),
```

```
                Padding(padding: EdgeInsets.all(16.0),),
                Container(
                  color: Colors.brown,
                  height: 20.0,
                  width: 20.0,
                ),
                Divider(),
                Row(
                  children: <Widget>[
                    CircleAvatar(
                      backgroundColor: Colors.lightGreen,
                      radius: 100.0,
                      child: Stack(
                        children: <Widget>[
                          Container(
                            height: 100.0,
                            width: 100.0,
                            color: Colors.yellow,
                          ),
                          Container(
                            height: 60.0,
                            width: 60.0,
                            color: Colors.amber,
                          ),
                          Container(
                            height: 40.0,
                            width: 40.0,
                            color: Colors.brown,
                          ),
                        ],
                      ),
                    ),
                  ],
                ),
                Divider(),
                Text('End of the Line'),
              ],
            ),
          ],
        ),
      ],
    ),
  ),
);
  }
}
```

La siguiente imagen muestra el diseño de la página resultante del árbol de widgets.

Para crear un diseño de página, anida los widgets para crear una interfaz de usuario personalizada. El resultado de agregar widgets juntos se llama árbol de widgets. A medida que aumenta el número de widgets, el árbol de widgets comienza a expandirse rápidamente y hace que el código sea difícil de leer y administrar.



Construyendo un árbol de widgets poco profundo

Para que el código de ejemplo sea más legible y fácil de mantener, refactorizarás las secciones principales del código en entidades separadas. Tienes varias opciones de refactorización y las técnicas más comunes son las constantes, los métodos y las clases de widgets.

Refactorizar con una constante

La refactorización con una constante inicializa el widget a una variable final. Este enfoque te permite separar los widgets en secciones, lo que mejora la legibilidad del código. Cuando los widgets se inicializan con una constante, se basan en el objeto BuildContext del widget principal.

¿Qué significa esto? Cada vez que se vuelve a dibujar el widget principal, todas las constantes también volverán a dibujar sus widgets, por lo que no puedes realizar ninguna optimización del rendimiento. En la siguiente sección, analizarás detalladamente la refactorización con un método en lugar de una constante. Los beneficios de hacer que el árbol de widgets sea menos profundo son similares con ambas técnicas.

El siguiente código de muestra detalla cómo usar una constante para inicializar la variable de contenedor como final con el widget de contenedor. Inserta la variable de contenedor en el árbol de widgets donde sea necesario.

```
final container = Container(
   color: Colors.yellow,
   height: 40.0,
```

```
    width: 40.0,
);
```

Refactorizar con un método

La refactorización con un método devuelve el widget llamando al nombre del método. El método puede devolver un valor mediante un widget general (widget) o un widget específico (contenedor, fila y otros).

Los widgets inicializados por un método se basan en el objeto BuildContext del widget principal (parent). Podría haber efectos secundarios no deseados si este tipo de métodos están anidados y llaman a otros métodos/funciones anidados. Dado que cada situación es diferente, no asumas que usar métodos no es una buena opción.

Este enfoque te permite separar los widgets en secciones, lo que mejora la legibilidad del código. Sin embargo, al igual que cuando se refactoriza con una constante, cada vez que se vuelve a dibujar el widget principal, todos los métodos también volverán a dibujar sus widgets. Eso significa que el árbol de widgets no se puede optimizar para el rendimiento.

El siguiente código de muestra detalla cómo utilizar un método para devolver un widget Container. Este primer método devuelve el widget Container como un widget general y el segundo método devuelve el widget Container como un widget Container. Ambos enfoques son aceptables. Inserta el nombre del método _buildContainer() en el árbol de widgets donde sea necesario.

```
// Return by general Widget Name
Widget _buildContainer() {
    return Container(
      color: Colors.yellow,
      height: 40.0,
      width: 40.0,
    );
}

// Or Return by specific Widget like Container in this case
Container _buildContainer() {
     return Container(
       color: Colors.yellow,
       height: 40.0,
       width: 40.0,
     );
}
```

Veamos un ejemplo que refactoriza utilizando métodos. Este enfoque mejora la legibilidad del código al separar las partes principales del árbol de widgets en métodos separados. Se podría adoptar el mismo enfoque refactorizando con una constante.

¿Cuál es el beneficio de utilizar el enfoque de método? El beneficio es la legibilidad del código pura y simple, pero pierde los beneficios de la reconstrucción del subárbol de Flutter: rendimiento.

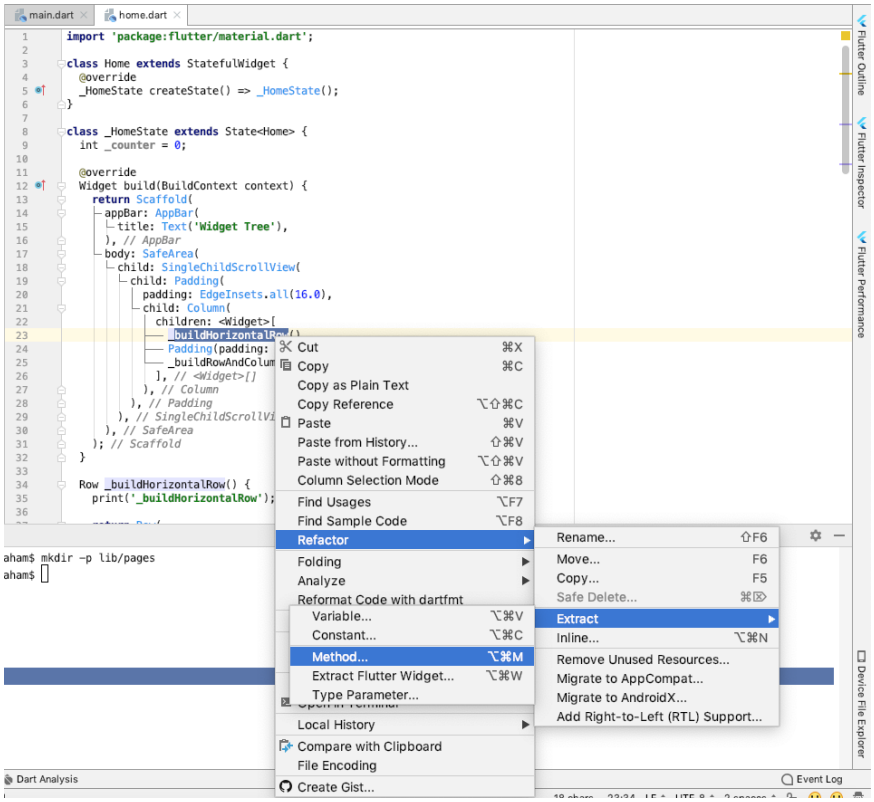
Refactorización con un método para crear un árbol de widgets poco profundo

Para refactorizar los widgets, usa el patrón de método para aplanar el árbol de widgets.

1. Abre el archivo home.dart.

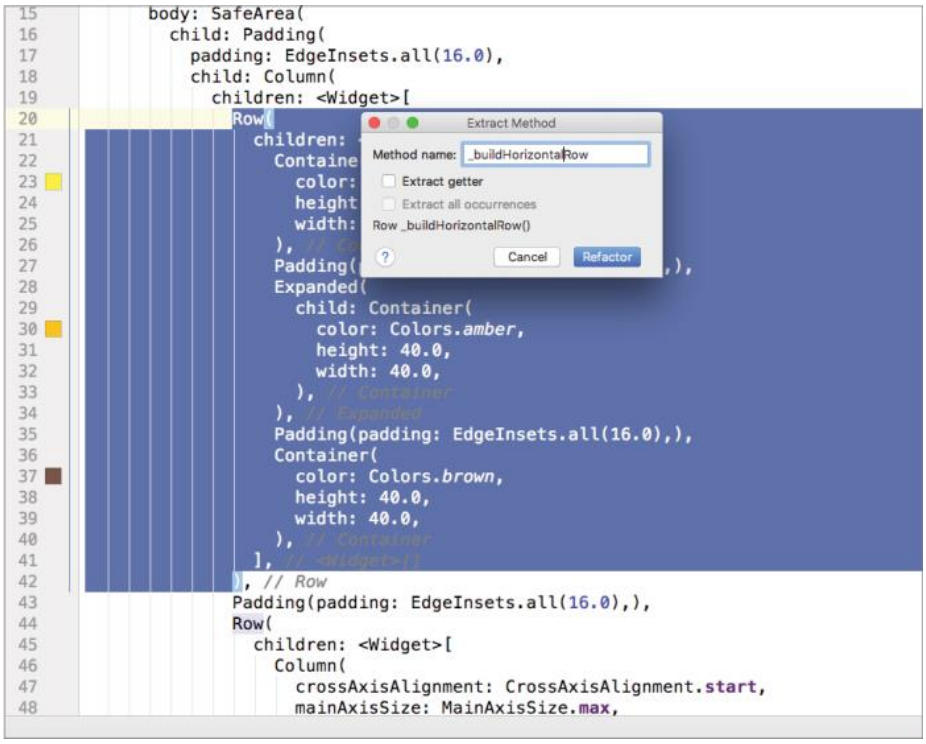
2. Coloca el cursor en el primer widget Row y haz clic con el botón derecho.

3. Selecciona Refactor ➪ Extractr ➪ Method y haz clic en Method.



4. En el cuadro de diálogo Extraer método (Extract Method), introduce _buildHorizontalRow para el nombre del método. Nota el subrayado antes de construir; esto le permite a Dart saber que es un método privado. Observa que todo el

widget Row y los elementos secundarios están resaltados para facilitar la visualización del código afectado.



5. El widget Row se reemplaza por el método _buildHorizontalRow(). Desplázate hasta la parte inferior del código y el método y los widgets se refactorizaron muy bien.

```
Row _buildHorizontalRow() {
  return Row(
    children: <Widget>[
      Container(
        color: Colors.yellow,
        height: 40.0,
        width: 40.0,
      ),
      Padding(padding: EdgeInsets.all(16.0),),
      Expanded(
        child: Container(
          color: Colors.amber,
          height: 40.0,
          width: 40.0,
        ),
      ),
      Padding(padding: EdgeInsets.all(16.0),),
      Container(
```

```
            color: Colors.brown,
            height: 40.0,
            width: 40.0,
          ),
        ],
      );
  }
```

6. Continúe y refactoriza las otras filas (Rows) y los widgets de fila (Row) y pila (Stack). A continuación se muestra el código fuente completo de home.dart. Observa cómo se aplana el árbol de widgets, lo que facilita su lectura. Decidir qué tan superficial hacer el árbol de widgets depende de cada circunstancia y de tus preferencias personales. Por ejemplo, digamos que estás trabajando en tu código y comienzas a notar que está desplazándose mucho vertical u horizontalmente para realizar cambios. Ésta es una buena indicación de que puedes refactorizar partes del código en secciones separadas.

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

class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Widget Tree'),
      ),
      body: SafeArea(
        child: SingleChildScrollView(
          child: Padding(
            padding: EdgeInsets.all(16.0),
            child: Column(
              children: <Widget>[
                _buildHorizontalRow(),
                Padding(padding: EdgeInsets.all(16.0),),
                _buildRowAndColumn(),
              ],
            ),
          ),
        ),
      ),
    );
  }

  Row _buildHorizontalRow() {
    return Row(
      children: <Widget>[
        Container(
          color: Colors.yellow,
          height: 40.0,
          width: 40.0,
        ),
```

```
      Padding(padding: EdgeInsets.all(16.0),),
      Expanded(
        child: Container(
          color: Colors.amber,
          height: 40.0,
          width: 40.0,
        ),
      ),
      Padding(padding: EdgeInsets.all(16.0),),
      Container(
        color: Colors.brown,
        height: 40.0,
        width: 40.0,
      ),
    ],
  );
}

Row _buildRowAndColumn() {
  return Row(
    children: <Widget>[
      Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        mainAxisSize: MainAxisSize.max,
        children: <Widget>[
          Container(
            color: Colors.yellow,
            height: 60.0,
            width: 60.0,
          ),
          Padding(padding: EdgeInsets.all(16.0),),
          Container(
            color: Colors.amber,
            height: 40.0,
            width: 40.0,
          ),
          Padding(padding: EdgeInsets.all(16.0),),
          Container(
            color: Colors.brown,
            height: 20.0,
            width: 20.0,
          ),
          Divider(),
          _buildRowAndStack(),
          Divider(),
          Text('End of the Line'),
        ],
      ),
    ],
  );
}

Row _buildRowAndStack() {
  return Row(
    children: <Widget>[
```

```
        CircleAvatar(
          backgroundColor: Colors.lightGreen,
          radius: 100.0,
          child: Stack(
            children: <Widget>[
              Container(
                height: 100.0,
                width: 100.0,
                color: Colors.yellow,
              ),
              Container(
                height: 60.0,
                width: 60.0,
                color: Colors.amber,
              ),
              Container(
                height: 40.0,
                width: 40.0,
                color: Colors.brown,
              ),
            ],
          ),
        ),
      ],
    );
  }

}
```

La creación de un árbol de widgets poco profundo significa que cada widget está separado en su propio método por funcionalidad. Ten en cuenta que la forma en que separes los widgets será diferente según la funcionalidad necesaria. La separación de los widgets por método mejora la legibilidad del código, pero pierde los beneficios de rendimiento de la reconstrucción del subárbol de Flutter. Todos los widgets del método dependen del BuildContext del padre, lo que significa que cada vez que se vuelve a dibujar el padre, el método también se vuelve a dibujar.

En este ejemplo, creaste el método _buildHorizontalRow() para crear el widget Row horizontal con widgets secundarios (child). El método _buildRowAndColumn() es un excelente ejemplo de cómo aplanarlo aún más llamando al método _buildRowAndStack() para uno de los widgets secundarios (children) de Column. La separación de _buildRowAndStack() se realiza para mantener plano el árbol de widgets porque el método _buildRownAndStack() crea un widget con varios widgets secundarios (children).

Refactorización con una clase de widget

Refactorizar con una clase de widget te permite crear el widget subclasificando la clase StatelessWidget. Puedes crear widgets reutilizables dentro del archivo Dart actual o separado e iniciarlos en cualquier lugar de la aplicación. Observa que el constructor comienza con una palabra clave const, que te permite almacenar en caché y reutilizar el widget. Cuando llames al constructor para iniciar el widget, usa la palabra clave const. Al llamar con la palabra clave

const, el widget no se reconstruye cuando otros widgets cambian su estado en el árbol. Si omites la palabra clave const, se llamará al widget cada vez que se vuelva a dibujar el widget principal.

La clase de widget se basa en su propio BuildContext, no en el padre, como los enfoques de método y constante. BuildContext es responsable de manejar la ubicación de un widget en el árbol de widgets.

En el tema 7, "Adición de animación a una aplicación", crearás un ejemplo que refactoriza y separa widgets con varios StatefulWidgets en lugar de la clase StatelessWidget.
¿Qué significa esto? Cada vez que se redibuja el widget principal, no se redibujarán todas las clases de widget. Se crean solo una vez, lo que es excelente para optimizar el rendimiento.

El siguiente código de muestra detalla cómo usar una clase de widget para devolver un widget Container. Insertas el widget const ContainerLeft() en el árbol de widgets donde sea necesario. Ten en cuenta el uso de la palabra clave const para aprovechar el almacenamiento en caché.

```
class ContainerLeft extends StatelessWidget {
  const ContainerLeft({
     Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
      return Container(
        color: Colors.yellow,
        height: 40.0,
        width: 40.0,
      );
  }
}

// Call to initialize the widget and note the const keyword
const ContainerLeft(),
```

Veamos un ejemplo que refactoriza usando clases de widgets (un widget Flutter). Este enfoque mejora la legibilidad y el rendimiento del código al separar las partes principales del árbol de widgets en clases de widgets independientes.
¿Cuál es el beneficio de usar las clases de widgets? Es un rendimiento puro y simple durante las actualizaciones de pantalla. Al llamar a una clase de widget, debes utilizar la declaración const; de lo contrario, se reconstruirá cada vez, sin almacenamiento en caché. Un ejemplo de

refactorización con una clase de widget es cuando tienes un diseño de interfaz de usuario en el que solo los widgets específicos cambian de estado y otros permanecen igual.

Refactorización con una clase de widget para crear un árbol de widget poco profundo

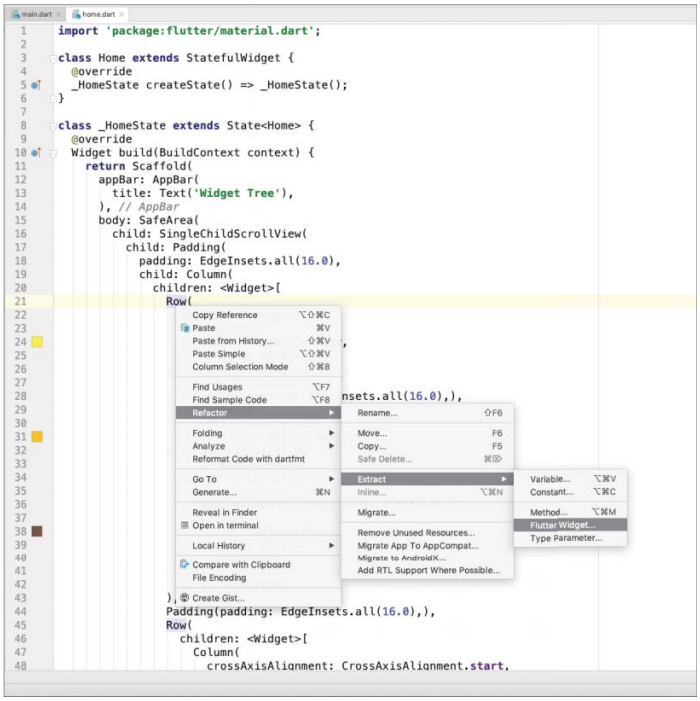
Para refactorizar los widgets, usa el patrón de clase de widget para aplanar el árbol de widgets.

Crea un nuevo proyecto de Flutter llamado widget_tree_performance. Puedes seguir las instrucciones del tema 4. Para este proyecto, solo necesitas crear la carpeta de páginas (pages). Para mantener este ejemplo simple, crearás las clases de widget en el archivo home.dart, pero en el tema 7 aprenderás a separarlas en archivos separados.

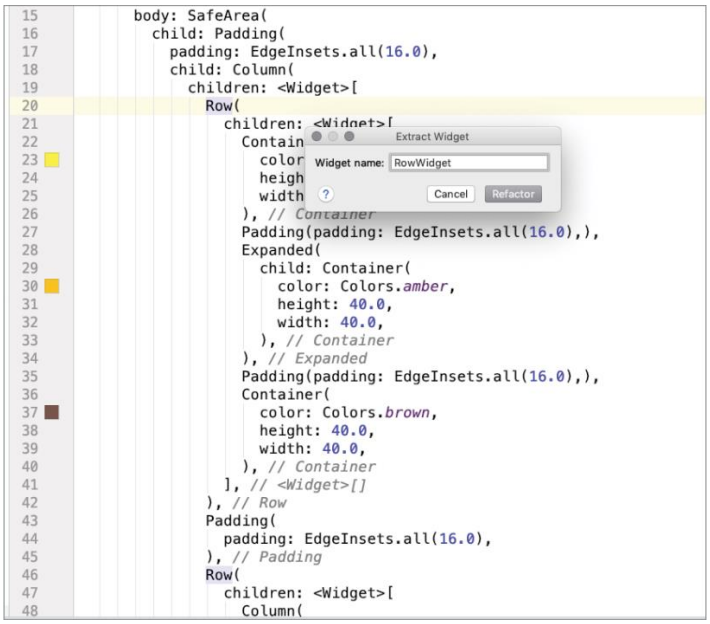
1. Abre el archivo home.dart. Copia el árbol de widgets completo original en home.dart (de la sección “Creación del árbol de widgets completo” de este documento) en el archivo home.dart de este proyecto.

2. Coloca el cursor en el primer widget Row y haz clic con el botón derecho.

3. Selecciona Refactor ➪ Extract ➪ Flutter Widget.



4. En el cuadro de diálogo Extract Widget, introduce RowWidget como nombre del widget.



5. El Row Widget se reemplaza con la clase de widget RowWidget(). Dado que el widget Row no cambiará de estado, agrega la palabra clave const antes de llamar a la clase RowWidget(). Desplázate hasta la parte inferior del código y los widgets se refactorizaron en la clase RowWidget (StatelessWidget).

```
class RowWidget extends StatelessWidget {
  const RowWidget({
     Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
     print('RowWidget');

     return Row(
       children: <Widget>[
          Container(
             color: Colors.yellow,
             height: 40.0,
            width: 40.0,
          ),
          Padding(
              padding: EdgeInsets.all(16.0),
          ),
          Expanded(
             child: Container(
                color: Colors.amber,
```

```
            height: 40.0,
            width: 40.0,
          ),
        ),
        Padding(
          padding: EdgeInsets.all(16.0),
        ),
        Container(
          color: Colors.brown,
          height: 40.0,
          width: 40.0,
        ),
      ],
    );
  }
}
```

6. Continua y refactoriza las otras filas (clase RowAndColumnWidget) y los widgets Row y Stack (clase RowAndStackWidget).

El código fuente completo de home.dart se detalla a continuación. Observa cómo se aplana el árbol de widgets, lo que facilita su lectura. Decidir qué tan superficial hacer el árbol de widgets depende de cada circunstancia y de tus preferencias personales.

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

class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}

class _HomeState extends State<Home> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Widget Tree'),
      ),
      body: SafeArea(
        child: SingleChildScrollView(
          child: Padding(
            padding: EdgeInsets.all(16.0),
            child: Column(
              children: <Widget>[
                const RowWidget(),
                Padding(
                  padding: EdgeInsets.all(16.0),
                ),
                const RowAndColumnWidget(),
              ],
            ),
```

```
          ),
        ),
      ),
    );
  }
}

class RowWidget extends StatelessWidget {
  const RowWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Container(
          color: Colors.yellow,
          height: 40.0,
          width: 40.0,
        ),
        Padding(
          padding: EdgeInsets.all(16.0),
        ),
        Expanded(
          child: Container(
            color: Colors.amber,
            height: 40.0,
            width: 40.0,
          ),
        ),
        Padding(
          padding: EdgeInsets.all(16.0),
        ),
        Container(
          color: Colors.brown,
          height: 40.0,
          width: 40.0,
        ),
      ],
    );
  }
}

class RowAndColumnWidget extends StatelessWidget {
  const RowAndColumnWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        Column(
          crossAxisAlignment: CrossAxisAlignment.start,
```

```
          mainAxisSize: MainAxisSize.max,
          children: <Widget>[
            Container(
              color: Colors.yellow,
              height: 60.0,
              width: 60.0,
            ),
            Padding(
              padding: EdgeInsets.all(16.0),
            ),
            Container(
              color: Colors.amber,
              height: 40.0,
              width: 40.0,
            ),
            Padding(
              padding: EdgeInsets.all(16.0),
            ),
            Container(
              color: Colors.brown,
              height: 20.0,
              width: 20.0,
            ),
            Divider(),
            const RowAndStackWidget(),
            Divider(),
            Text('End of the Line. Date: ${DateTime.now()}'),
          ],
        ),
      ],
    );
  }
}

class RowAndStackWidget extends StatelessWidget {
  const RowAndStackWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Row(
      children: <Widget>[
        CircleAvatar(
          backgroundColor: Colors.lightGreen,
          radius: 100.0,
          child: Stack(
            children: <Widget>[
              Container(
                height: 100.0,
                width: 100.0,
                color: Colors.yellow,
              ),
              Container(
                height: 60.0,
```

```
                width: 60.0,
                color: Colors.amber,
              ),
              Container(
                height: 40.0,
                width: 40.0,
                color: Colors.brown,
              ),
            ],
          ),
        ),
      ],
    );
  }
}
```

Crear un árbol de widgets poco profundo significa que cada widget está separado en su propia clase de widget por funcionalidad. Ten en cuenta que la forma en que separes los widgets será diferente según la funcionalidad necesaria.

En este ejemplo, creaste la clase de widget RowWidget() para construir el widget Row horizontal con widgets secundarios. La clase de widget RowAndColumnWidget() es un excelente ejemplo de cómo aplanarla aún más llamando a la clase de widget RowAndStackWidget() para uno de los widgets secundarios de Column. La separación mediante la adición de RowAndStackWidget() adicional se realiza para mantener plano el árbol de widgets porque la clase RowAndStackWidget() crea un widget con varios elementos secundarios.

En el código fuente del proyecto, agregaste para tu conveniencia un botón que aumenta el valor de un contador, y cada clase de widget usa una declaración de impresión para mostrar cada vez que se llama a cada uno cuando cambia el estado del contador.

El siguiente es el archivo de registro que muestra cada vez que se llama a un widget. Cuando se presiona el botón, el widget CounterTextWidget se vuelve a dibujar para mostrar el nuevo valor del contador, pero observa que los widgets RowWidget, RowAndColumnWidget y RowAndStackWidget se llaman solo una vez y no se vuelven a dibujar cuando cambia el estado. Al utilizar la técnica de la clase de widget, solo se llaman los widgets que necesitan volver a dibujar, lo que mejora el rendimiento general.

```
// App first loaded
flutter: RowWidget
flutter: RowAndColumnWidget
flutter: RowAndStackWidget
flutter: CounterTextWidget 0

// Increase value button is called and notice the row widgets are not redrawn
flutter: CounterTextWidget 1
flutter: CounterTextWidget 2
flutter: CounterTextWidget 3
```

Resumen

En este tema, aprendiste que el árbol de widgets es el resultado de widgets anidados. A medida que aumenta el número de widgets, el árbol de widgets se expande rápidamente y reduce la legibilidad y la capacidad de administración del código.

A esto le llamamos el árbol *completo* de widgets. Para mejorar la legibilidad y la capacidad de administración del código, puedes separar los widgets en su propia clase de widgets, creando un árbol de widgets menos profundo. En cada aplicación, debes esforzarte por mantener el árbol de widgets poco profundo.

Al refactorizar con una clase de widget, puedes aprovechar la reconstrucción del subárbol de Flutter, que mejora el rendimiento.

En el próximo tema, analizarás el uso de widgets básicos. Aprenderás a implementar diferentes tipos de botones, imágenes, íconos, decoradores, formularios con validación y orientación de campos de texto.