

Layout Widgets

1. Что такое **Layout Widgets** и для чего применяются?

Layout widgets — виджеты-контейнеры компоновки (layout widgets), которые управляют компоновкой виджетов (в том числе и других контейнеров).

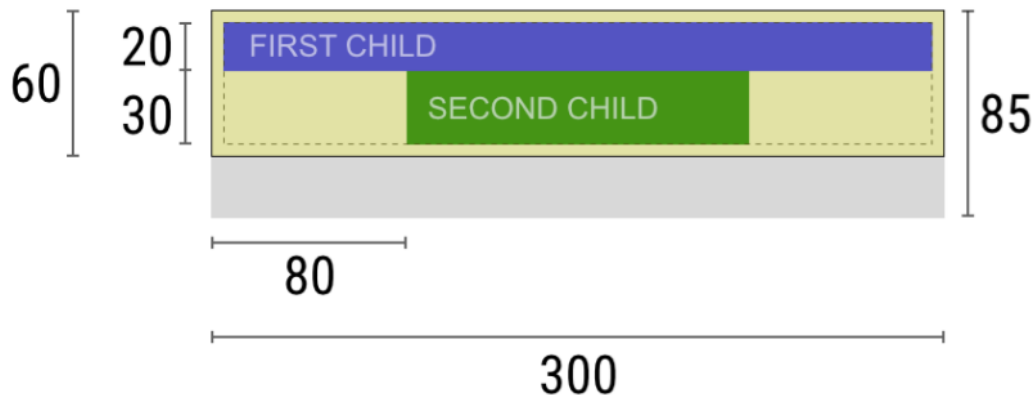
Эти специальные виджеты применяются, чтобы организовать и расположить определенным образом один виджет или наборы виджетов.

Во Flutter эта группа виджетов очень широко представлена.

Компоновка виджетов

Ограничения для виджетов объявляются в родителях. Размеры (желаемые) задаются в самом виджете. Позиция виджета на экране устанавливается родителем.

- Виджет получает свои ограничения от своего родителя. "Ограничение" — это всего 4 значения: минимальная и максимальная ширина, минимальная и максимальная высота.
- Затем виджет проходит по своему списку детей. Виджет сообщает своим дочерним элементам, каковы их ограничения (которые могут быть разными для каждого ребенка), а затем спрашивает каждого ребенка, какого размера он "хочет" быть.
- Затем виджет размещает свои дочерние элементы (горизонтально по оси x и вертикально по оси y) один за другим.
- И, наконец, виджет сообщает своему родителю о собственном размере (в пределах исходных ограничений, конечно).



Типы layout виджетов

☐ Single-child layout widgets

- ☐ Align
- ☐ AspectRatio
- ☐ Baseline Center
- ☐ ConstrainedBox
- ☐ Container
- ☐ Padding
- ☐ SizedBox
- ☐ Transform
- ☐ Expanded
- ☐ FittedBox
- ☐ LimitedBox
- ☐ Offstage
- ☐ OverflowBox
- ☐ SizedOverflowBox

☐ Multi-child layout widgets

- ☐ CustomMultiChildLayout
- ☐ Column
- ☐ Flow
- ☐ GridView
- ☐ IndexedStack
- ☐ LayoutBuilder
- ☐ ListBody
- ☐ ListView
- ☐ Row
- ☐ Stack
- ☐ Table
- ☐ Wrap

Single-child layout widgets

— виджеты-контейнеры компоновки, которые могут иметь только один дочерний виджет внутри родительского виджета макета.



Single-child layout widgets

В Flutter доступно много виджетов с одним дочерним макетом, все они облегчат вашу жизнь, если вы выберете правильный виджет в соответствии с вашими требованиями.

Рассмотрим самые часто встречающиеся single-layout виджеты:

☐ Container

☐ Center

☐ Align

☐ Padding

☐ SizedBox

☐ Expanded

Align

Align Widget — это виджет, который используется для выравнивания своего дочернего элемента внутри себя.

Align Widget довольно гибкий и может изменять свой размер в соответствии с размером своего дочернего элемента.

С Align Class у вас будет больше контроля над точным положением дочернего виджета. Это позволяет разместить дочерний виджет именно там, где вам нужно.

Конструктор класса Align

Синтаксис:

```
Align({Key key,           //ключ виджета
AlignmentGeometry alignment: Alignment.center, //задает выравнивание
double widthFactor,      //если не равно нулю,устанавливает свою ширину на ширину дочернего элемента, умноженную на этот
widthFactor
double heightFactor,    //если не равно нулю,устанавливает свою высоту на высоту дочернего элемента,умноженную на этот heightFactor
Widget child})          // дочерний виджет в дереве
```

Настройка выравнивания

Если вы не передадите аргумент `alignment`, по умолчанию дочерний элемент будет выровнен по центру. Чтобы изменить выравнивание дочернего элемента, вам нужно передать `alignment` аргумент.

Align

Есть некоторые определенные константы, которые позволяют легко выравнивать дочерний элемент в определенных позициях.

 <p>Woolha.com</p>	 <p>Woolha.com</p>	 <p>Woolha.com</p>	 <p>Woolha.com</p>	
<i>Выравнивание.topLeft</i>	<i>Выравнивание.topCenter</i>	<i>Выравнивание.topRight</i>	<i>Выравнивание.bottomLeft</i>	
 <p>Woolha.com</p>	 <p>Woolha.com</p>	 <p>Woolha.com</p>	 <p>Woolha.com</p>	 <p>Woolha.com</p>
<i>Alignment.centerLeft</i>	<i>Alignment.center</i>	<i>Alignment.centerRight</i>	<i>Выравнивание.bottomCenter</i>	<i>Выравнивание.bottomRight</i>

Пример: выравнивание изображения по правому верхнему углу контейнера.

```
class _MyHomePageState extends State<MyHomePage> {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('GeeksforGeeks Align Widget'),  
        backgroundColor: Colors.green),  
      body: Center(  
        child: Container(  
          height: 240.0,  
          width: 240.0,  
          color: Colors.green,  
          child: Align(  
            alignment: Alignment.topRight,  
            child: Image.network(  
              "https://www.geeksforgeeks.org/wp-content/uploads/gfg\_200X200-  
1.png",  
              width: 100,  
            )),  
        )),  
    );  
  }  
}
```

GeeksforGeeks Align W...



Container

Контейнер во Flutter — это **родительский виджет, который может содержать несколько дочерних виджетов** и эффективно управлять ими с помощью ширины, высоты, заполнения, цвета фона и т. д.

- виджет, который сочетает в себе общую отрисовку, расположение и размеры дочерних виджетов.
- класс для хранения одного или нескольких виджетов и их размещения на экране в соответствии

В общем, это похоже на коробку для хранения содержимого.

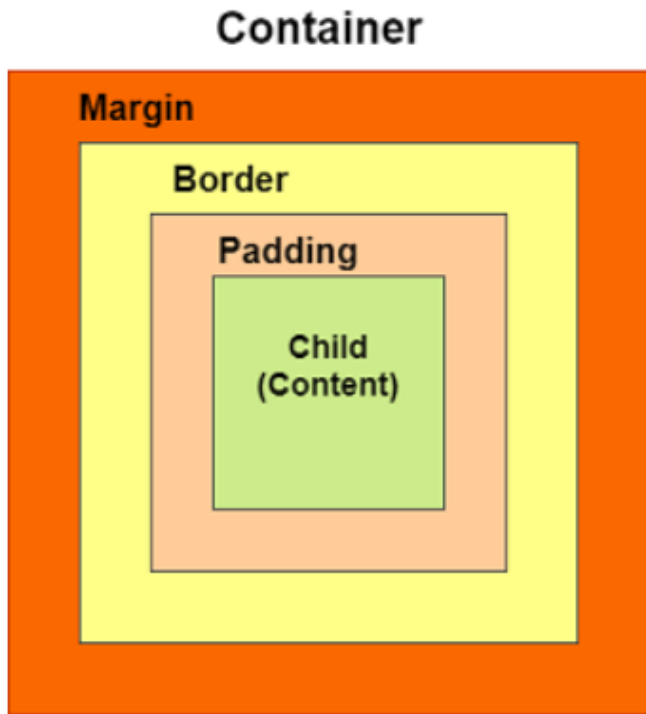
Зачем нам нужен контейнерный виджет во Flutter?

Если у нас есть виджет, для которого требуется какой-либо стиль фона, например, ограничения цвета, формы или размера, мы можем попытаться обернуть его в виджет-контейнер . Этот виджет помогает нам создавать, украшать и размещать его дочерние виджеты.

Если бы мы заворачивали наши виджеты в контейнер, то без использования каких-либо параметров мы не заметили бы никакой разницы в его внешнем виде. Но если мы добавим в контейнер какие-либо свойства, такие как цвет, поля, отступы и т. д., мы сможем стилизовать наши виджеты на экране в соответствии с нашими потребностями.

Container

Базовый контейнер имеет свойства `margin`, `border` и `padding`, окружающие его дочерний виджет, как показано на изображении ниже:



Конструктор класса Container

Синтаксис:

```
Container({Key key,           //ключ виджета
           AlignmentGeometry alignment, //для установки положения дочернего элемента в контейнере
           EdgeInsetsGeometry padding, //для установки расстояния между границей контейнера и его дочерним
виджетом
           EdgeInsetsGeometry margin,           // для выделения пустого пространства вокруг контейнера
           Color color,                        // для установки цвета фона текста
           double width,                      // для установки ширины контейнера в соответствии с нашими
потребностями
           double height,                    // для установки высоты контейнера в соответствии с нашими
потребностями
           Decoration decoration,             //украшает или раскрашивает виджет позади child
           Decoration foregroundDecoration,    //украшает или раскрашивает виджет перед child
           BoxConstraints constraints, //используется, когда хотим добавить дополнительные ограничения дочернему
элементу
           Matrix4 transform,                //возможность вращать контейнер
           Widget child,                    // для хранения дочернего виджета контейнера
           Clip clipBehavior: Clip.none     //свойство принимает в качестве объекта Clip Enum,будет ли содержимое внутри
контейнера обрезано или нет.
```

Свойства класса Container: Child

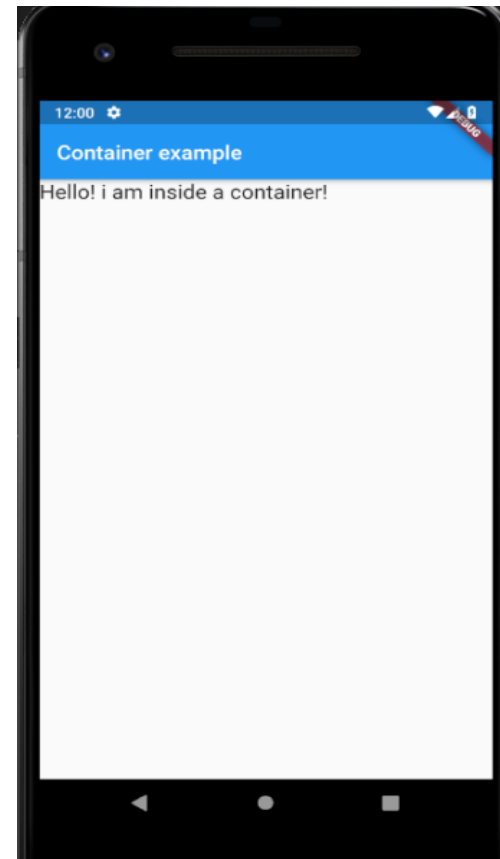
Дочерним классом может быть любой виджет. Давайте возьмем пример, взяв текстовый виджет в качестве дочернего элемента.

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text("Container example"),
        ),
        body: Container(
          child: const Text("Hello! i am inside a container!",
            style: TextStyle(fontSize: 20)),
        ),
      ),
    );
  }
}
```



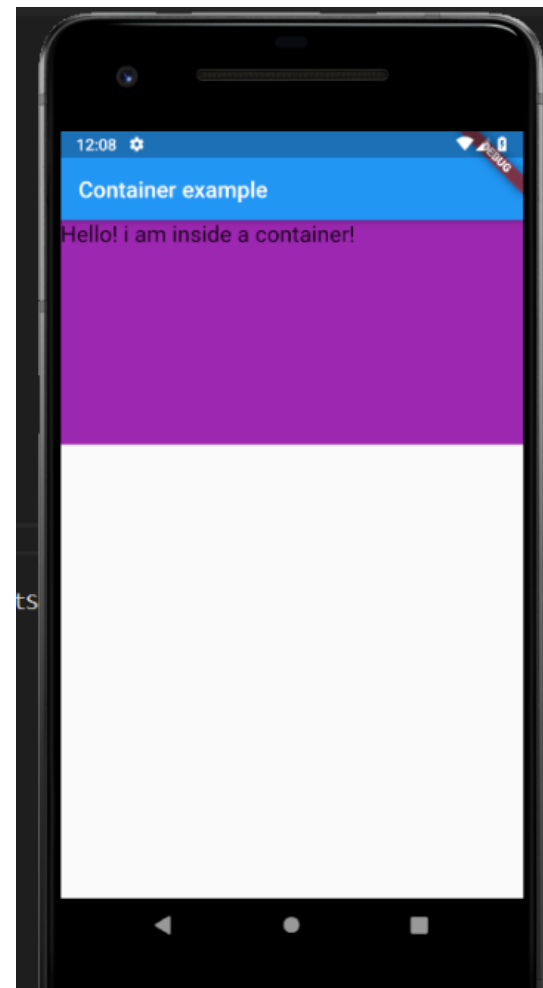
Свойства класса Container: Color, width and heigh

Свойство **color** устанавливает цвет фона всего контейнера.

```
body: Container(  
  color: Colors.purple,  
  child: const Text("Hello! i am inside a container!",  
    style: TextStyle(fontSize: 20)),  
)
```

Высота и ширина: по умолчанию класс-контейнер занимает пространство, необходимое дочернему элементу. Мы также можем указать высоту и ширину контейнера в соответствии с нашими требованиями.

```
body: Container(  
  height: 200,  
  width: double.infinity,  
  color: Colors.purple,  
  child: const Text("Hello! i am inside a container!",  
    style: TextStyle(fontSize: 20)),  
),
```



Свойства класса Container: Margin, padding

Margin: поле используется для создания пустого пространства вокруг контейнера. Обратите внимание на белое пространство вокруг контейнера. Здесь `EdgeInsets.geometry` используется для установки поля. `all()` указывает, что поле одинаково присутствует во всех четырех направлениях.

Padding : заполнение используется для создания пространства от границы контейнера от его дочерних элементов. Обратите внимание на пространство между границей и текстом.

```
body: Container(  
  height: 200,  
  width: double.infinity,  
  color: Colors.purple,  
  margin: const EdgeInsets.all(20),  
  padding: const EdgeInsets.all(30),  
  child: const Text("Hello! i am inside a container!",  
    style: TextStyle(fontSize: 20)),  
),
```



Свойства класса Container: Decoration

Свойство украшения используется для украшения коробки (например, для создания границы). Это рисует позади дочернего виджета. Дадим контейнеру границу. Но нельзя указать и цвет, и цвет границы.

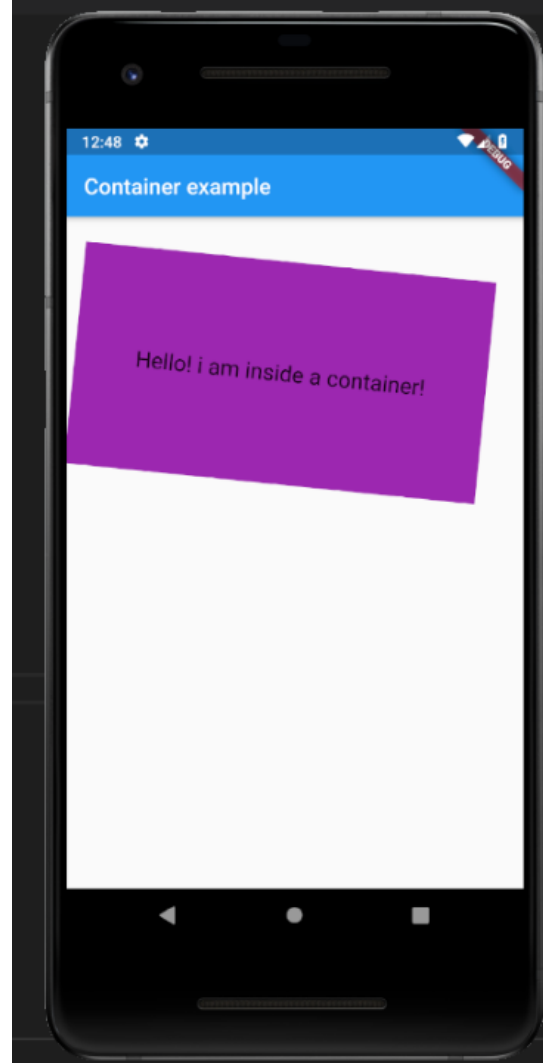
```
body: Container(  
  height: 200,  
  width: double.infinity,  
  //color: Colors.purple,  
  alignment: Alignment.center,  
  margin: const EdgeInsets.all(20),  
  padding: const EdgeInsets.all(30),  
  decoration: BoxDecoration(  
    border: Border.all(color: Colors.black, width: 3),  
  ),  
  child: const Text("Hello! i am inside a container!",  
    style: TextStyle(fontSize: 20)),  
),
```



Свойства класса Container: Transform

Transform: это свойство контейнера помогает нам вращать контейнер. Мы можем вращать контейнер по любой оси, здесь мы вращаемся по оси Z.

```
body: Container(  
  height: 200,  
  width: double.infinity,  
  color: Colors.purple,  
  alignment: Alignment.center,  
  margin: const EdgeInsets.all(20),  
  padding: const EdgeInsets.all(30),  
  transform: Matrix4.rotationZ(0.1),  
  child: const Text("Hello! i am inside a container!",  
    style: TextStyle(fontSize: 20)),  
),
```



Center

Центральный виджет поставляется со встроенным флаттером, он выравнивает дочерний виджет по центру доступного пространства на экране.

Синтаксис:

```
Center({Key key, //ключ виджета  
double widthFactor, //устанавливает ширину виджета Center  
double heightFactor, //устанавливает высоту виджета Center  
Widget child}) //для хранения дочернего виджета контейнера
```

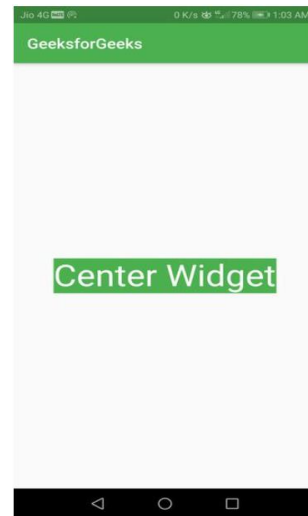
Размер этого виджета будет максимально большим, если для свойств `widthFactor` и `heightFactor` задано значение `null`, а размеры ограничены. И в случае, если размеры не ограничены, а для `widthFactor` и `HeightFactor` задано значение `null`, виджет `Center` принимает размер своего дочернего виджета. Давайте разберемся в этом с помощью примеров.

Свойства центрального виджета:

- **WidthFactor:** это свойство принимает двойное значение в качестве параметра и устанавливает ширину виджета Center равной ширине дочернего элемента, умноженной на этот коэффициент. Например, если установлено значение 3.0, виджет Center будет занимать в три раза больше размера своего дочернего виджета.
- **HeightFactor:** это свойство также принимает двойное значение в качестве параметра и устанавливает высоту виджета Center равной высоте дочернего элемента, умноженной на этот коэффициент.

Давайте разберемся в этом с помощью примеров.

```
body: Center(`  
  // heightFactor: 3,  
  // widthFactor: 0.8,  
  child: Container(  
    color: Colors.green,  
    child: Text(  
      'Center Widget',  
      textScaleFactor: 3,  
      style: TextStyle(color: Colors.white),  
    ),  
  ),  
),
```



Примеры

1)

```
body: Center(  
  heightFactor: 3,  
  child: Container(  
    color: Colors.green,  
    child: Text(  
      'Center Widget',  
      textScaleFactor: 3,  
      style: TextStyle(color: Colors.white),  
    ),  
  ),  
),
```



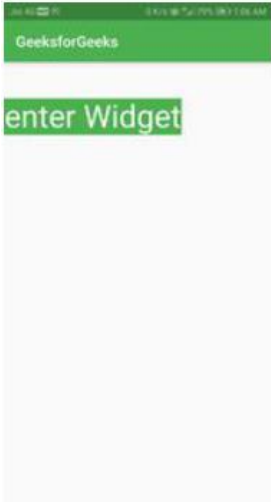
2)

```
body: Center(  
  widthFactor: 1,  
  child: Container(  
    color: Colors.green,  
    child: Text(  
      'Center Widget',  
      textScaleFactor: 3,  
      style: TextStyle(color: Colors.white),  
    ),  
  ),  
),
```



3)

```
body: Center(  
  heightFactor: 3,  
  widthFactor: 0.8,  
  child: Container(  
    color: Colors.green,  
    child: Text(  
      'Center Widget',  
      textScaleFactor: 3,  
      style: TextStyle(color: Colors.white),  
    ),  
  ),  
),
```



Padding

Виджет заполнения во флаттере делает именно то, о чем говорит его название, он добавляет отступы или пустое пространство вокруг виджета или группы виджетов. Мы можем применить отступ вокруг любого виджета, поместив его в качестве дочернего элемента виджета `Padding`. Размер дочернего виджета внутри заполнения ограничен тем, сколько места осталось после добавления пустого пространства вокруг.

Виджет `Padding` добавляет пустое пространство вокруг любого виджета с помощью абстрактного класса `EdgeInsetsGeometry`.

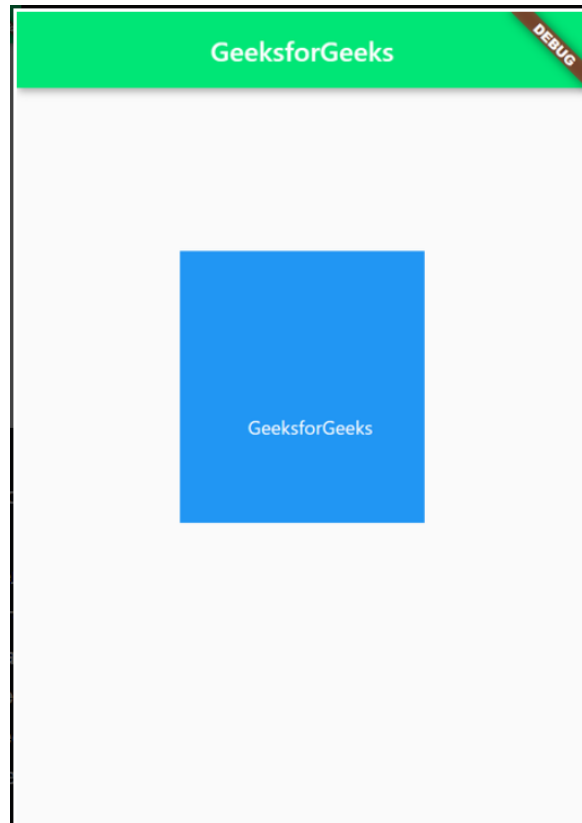
Синтаксис:

```
const Padding(  
  {Key key,                               //ключ виджета  
  @required EdgeInsetsGeometry padding,    //для добавления пустого пространства вокруг виджета.  
  Widget child}                           //для хранения дочернего виджета контейнера  
)
```

Пример

```
body: Padding(  
  padding: EdgeInsets.all(120),  
  child: Container(  
    padding: const EdgeInsets.fromLTRB(50, 120, 10, 10),  
    color: Colors.blue,  
    width: 200.0,  
    height: 200.0,  
    child: Text(  
      'GeeksforGeeks',  
      style: TextStyle(color: Colors.white),
```

Объяснение: здесь, в этом приложении, все тело приложения содержит виджет `Padding`. Свойство `Padding` внутри использует `EdgeInsets.all(120)`, которое добавляет пустое пространство в 120 пикселей вокруг во всех направлениях. Свойство дочернего элемента содержит `Container` синего цвета. Контейнер имеет высоту и ширину по 200 пикселей каждый, а его свойство заполнения содержит `const EdgeInsets.fromLTRB(50, 120, 10, 10)`, что обеспечивает пустое пространство 50 пикселей, 120 пикселей, 10 пикселей и 10 пикселей в левое, верхнее, правое и нижнее направления соответственно.



Expanded

Расширенный виджет во флаттере удобен, когда мы хотим, чтобы дочерний виджет или дочерние виджеты занимали все доступное пространство вдоль главной оси (для строки основная ось горизонтальна, а для столбца вертикальна). И в случае, если мы не хотим давать равные места нашим дочерним виджетам, мы можем распределить доступное пространство по своему усмотрению, используя фактор гибкости .

Контейнер Expanded позволяет своему вложенному виджету child заполнить доступное пространство (или его часть) других контейнеров - Row и Column.

Синтаксис:

```
const Expanded(  
  {Key? key,           //ключ виджета  
  int flex: 1,         //указывает на часть пространства контейнера, которая будет отдаваться виджету Expanded  
  required Widget child} //вложенный виджет  
)
```


Сначала рассмотрим проблему, с которой мы можем столкнуться. Допустим, в контейнере Row мы хотим вывести относительно длинный текст:

```
void main() {  
  runApp(Container(  
    padding: EdgeInsets.all(30),  
    color: Colors.teal,  
    child: Row(  
      textDirection: TextDirection.ltr,  
      crossAxisAlignment: CrossAxisAlignment.start,  
      verticalDirection: VerticalDirection.down,  
      children: <Widget>[  
        Text('Через несколько дней после отъезда Анатоля, Пьер получил  
записку от князя Андрея, извещавшего '  
          'его о своем приезде и просившего Пьера заехать к нему.',  
          textDirection: TextDirection.ltr)      ])
```

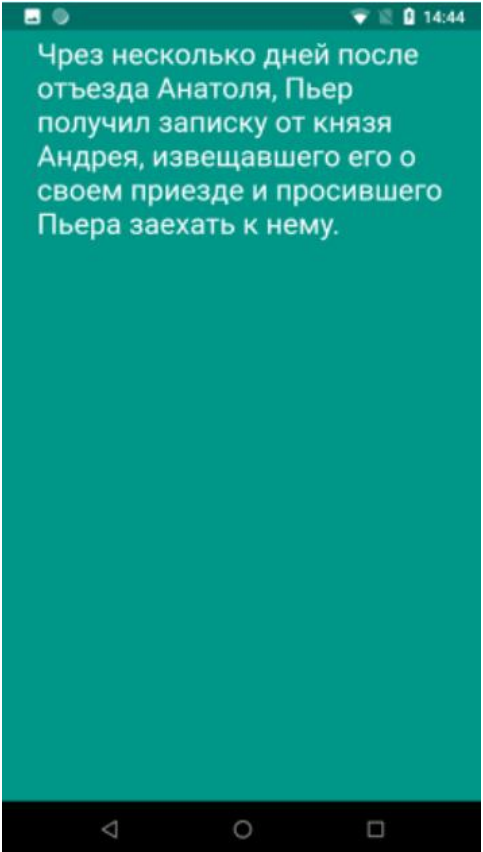
Поскольку ширина контейнера Row недостаточна, чтобы вместить строку текста, то мы увидим желто-черную полосу и сообщение о превышении размера на столько пикселей. Тем не менее все остальное пространство Row под строкой текста у нас пусто. И было бы неплохо, чтобы вместо этой полосы мы видели бы перенос текста ниже, чтобы он заполнял все пространство Row.



Пример

Воспользуемся виджетом Expanded:

```
void main() {
  runApp(Container(
    padding: EdgeInsets.all(30),
    color: Colors.teal,
    child: Row(
      textDirection: TextDirection.ltr,
      crossAxisAlignment: CrossAxisAlignment.start,
      verticalDirection: VerticalDirection.down,
      children: <Widget>[
        Expanded(
          child: Text('Чрез несколько дней после отъезда Анатоля, Пьер
получил записку от князя Андрея, извещавшего '
            'его о своем приезде и просившего Пьера заехать к нему.',
            textDirection: TextDirection.ltr)
        )
      ]
    )
  );
}
```



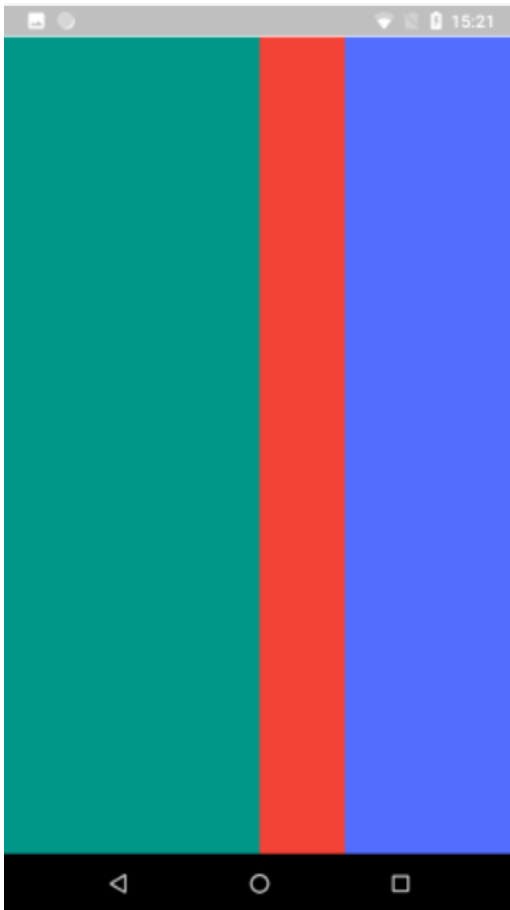
Свойство flex

Флекс-фактор указывает на часть пространства контейнера, которая будет отдаваться виджету Expanded. При вычислении данной части пространства флекс-фактор данного виджета Expanded делиться на сумму флекс-факторов всех элементов. Полученное значение умножается на общее доступное пространство контейнера.

Например, пусть у нас есть следующий контейнер Row:

```
void main() {  
  runApp(Container(  
    padding: EdgeInsets.only(top:25),  
    color: Colors.white,  
    child:Row(  
      textDirection: TextDirection.ltr,  
      crossAxisAlignment: CrossAxisAlignment.start,  
      verticalDirection: VerticalDirection.down,  
      children: <Widget>[  
        Expanded(  
          child: Container(color: Colors.teal),  
          flex: 3,  
        ),  
        Expanded(  
          child: Container(color: Colors.red),  
          flex:1  
        ),  
        Expanded(  
          child: Container(color: Colors.indigoAccent),  
          flex: 2,  
        ),  
      ],  
    ),  
  ),  
)
```

Результат:



Таким образом, у нас три виджета Expanded, каждый из которых содержит виджет Container с определенной цветовой окраской. Первый виджет Expanded имеет флекс-фактор 3, второй - 1, третий - 2. Сумма всех флекс-факторов равна $3 + 1 + 2 = 6$. Поэтому первый виджет Expanded получить $3/6$ или $1/2$ пространства Row, второй виджет Expanded - $1/6$ пространства Row, а третий виджет Expanded - $2/6$ или $1/3$ пространства Row.

Если flex-factor явным образом не указан, то по умолчанию он равен 1.

SizedBox

SizedBox — это встроенный виджет в Flutter SDK. Это простая коробка определенного размера. Его можно использовать для установки ограничений размера дочернего виджета, размещения пустого **SizedBox** между двумя виджетами, чтобы между ними было пространство и т.п. Он чем-то похож на виджет **Container** с меньшим количеством свойств.

Конструктор класса **SizedBox**:

Он рисует простой блок с указанными высотой и шириной или дочерний виджет внутри.

```
const SizedBox(  
  {Key key,  
  double width,  
  double height,  
  Widget child}  
)
```

Конструктор `SizeBox.expand`:

Эта реализация виджета `SizeBox` позволяет ему быть настолько большим, насколько позволяет родительский виджет.

```
const SizeBox.expand(  
  {Key,  
  Widget child}  
)
```

Конструктор `SizeBox.fromSize`:

Это позволяет создать `SizeBox` заданного размера.

```
SizeBox.fromSize(  
  {Key, key,  
  Widget child,  
  Size, size}  
)
```

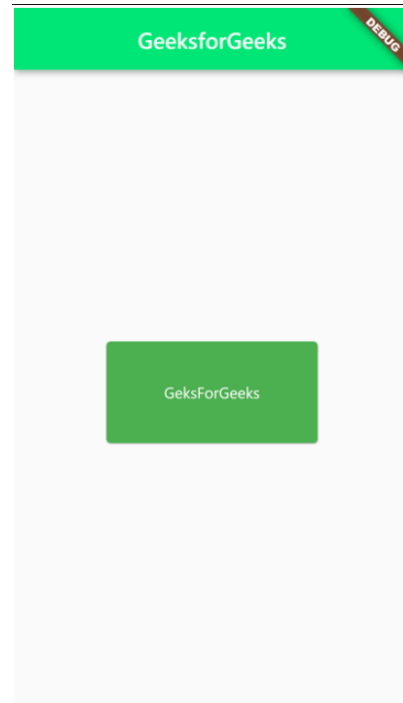
Конструктор `SizeBox.shrink`:

Эта реализация виджета `SizeBox` позволяет ему быть настолько маленьким, насколько это позволяет дочерний виджет.

```
const SizeBox.shrink(  
  {Key key,  
  Widget child}  
)
```

Пример

```
body: Center(  
  //SizedBox Widget  
  child: SizedBox(  
    width: 200.0,  
    height: 100.0,  
    child: Card(  
      color: Colors.green,  
      child: Center(  
        child: Text(  
          'GeeksForGeeks',  
          style: TextStyle(color: Colors.white),
```

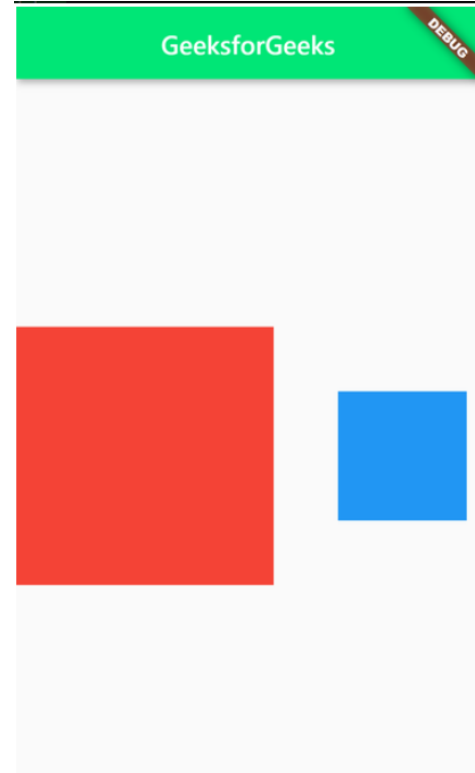


Объяснение: взглянув на код, мы видим, что тело этого флаттер-приложения содержит виджет `Center` в качестве родителя для всех. Внутри виджета `Center` у нас есть `SizedBox`, высота которого указана как 100, а ширина — 200. Виджет `SizedBox` содержит карту как дочерний элемент и ограничивает ее размер.

Пример

```
body: Center(  
  child: Row(  
    children: <Widget>[  
      Container(  
        width: 200,  
        height: 200,  
        color: Colors.red,  
      ), //Container  
      //SizedBox Widget  
      SizedBox(  
        width: 50,  
      ),  
      Container(  
        width : 100,  
        высота: 100,  
        цвет: Colors.blue,  
      ) //Container  
    ], //<Widget>[]  
  ), //Row  

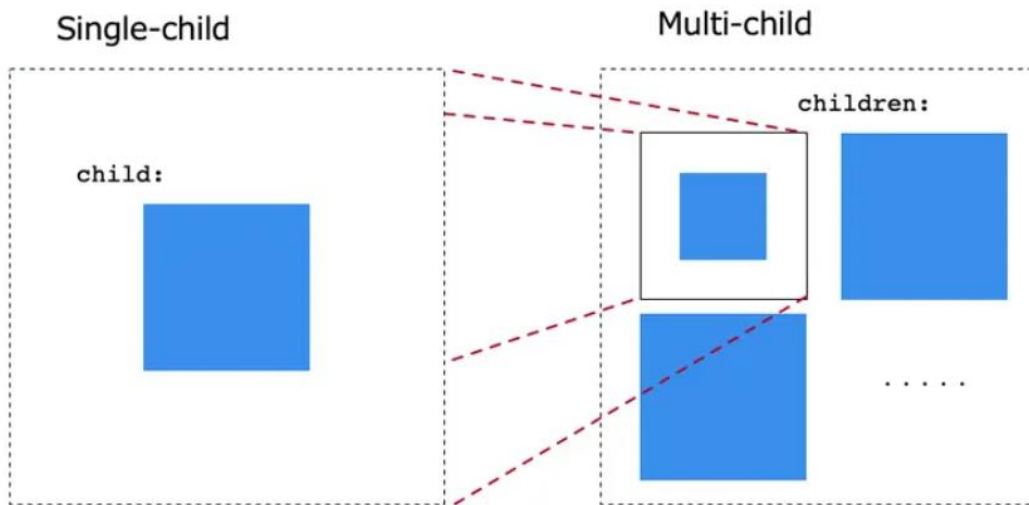
```



Объяснение: В этом примере мы использовали виджет `SizedBox` , чтобы добавить промежуток между двумя виджетами `Container` . Здесь у нас есть два виджета `Container` внутри строки. Первый контейнер имеет красный цвет и высоту и ширину по 200 пикселей каждый. Второй синий контейнер имеет высоту и ширину по 100 пикселей каждый. А между этими двумя контейнерами у нас есть виджет `SizedBox` шириной 50 пикселей и без высоты.


Multi-child layout widgets

— контейнеры компоновки, которые могут содержать более одного дочернего элемента.



Multi-child layout виджеты поддерживают свойство *children*, в отличие от single-child layout виджетов со свойством *child*.

Мы разберем следующие multi-child layout виджеты, как самые часто встречающиеся:

- 
- Row
 - Column
 - ListView
 - Wrap
 - Flow
 - Stack
 - GridView

Row

Виджет **Row** располагает свои дочерние элементы в горизонтальном направлении, в виде строки.

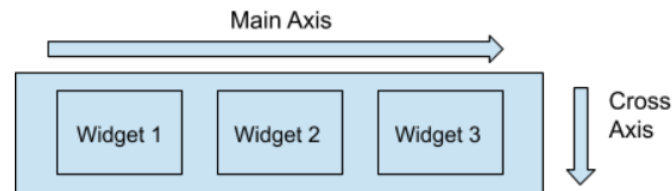
Виджет Row *не имеет прокрутки* (и вообще считается ошибкой наличие в строке большего числа дочерних элементов, чем может в неё поместиться). Если вы хотите, чтобы виджеты могли прокручиваться, если места недостаточно, рассмотрите возможность использования ListView.

Виджеты Row и Column позволяют *создавать гибкие макеты*. Их функции основаны на веб-модели flexbox.

Если у вас только один дочерний элемент, рассмотрите возможность использования Align или Center для его позиционирования.

Конструктор:

```
Row({  
  
    Key key, // ключ виджета  
  
    MainAxisAlignment mainAxisAlignment: MainAxisAlignment.start, // направление по горизонтали  
  
    MainAxisSize mainAxisSize: MainAxisSize.max, // пространство, занимаемое виджетом по горизонтали  
  
    CrossAxisAlignment crossAxisAlignment: CrossAxisAlignment.center, // выравнивание по вертикали  
  
    TextDirection textDirection, // порядок расположения вложенных элементов по горизонтали  
  
    VerticalDirection verticalDirection: VerticalDirection.down, // порядок расположения вложенных элементов по вертикали  
  
    TextBaseline textBaseline, // базовая линия для выравнивания элементов  
  
    List<Widget> children: const [] // набор вложенных элементов  
  
})
```



Пример.

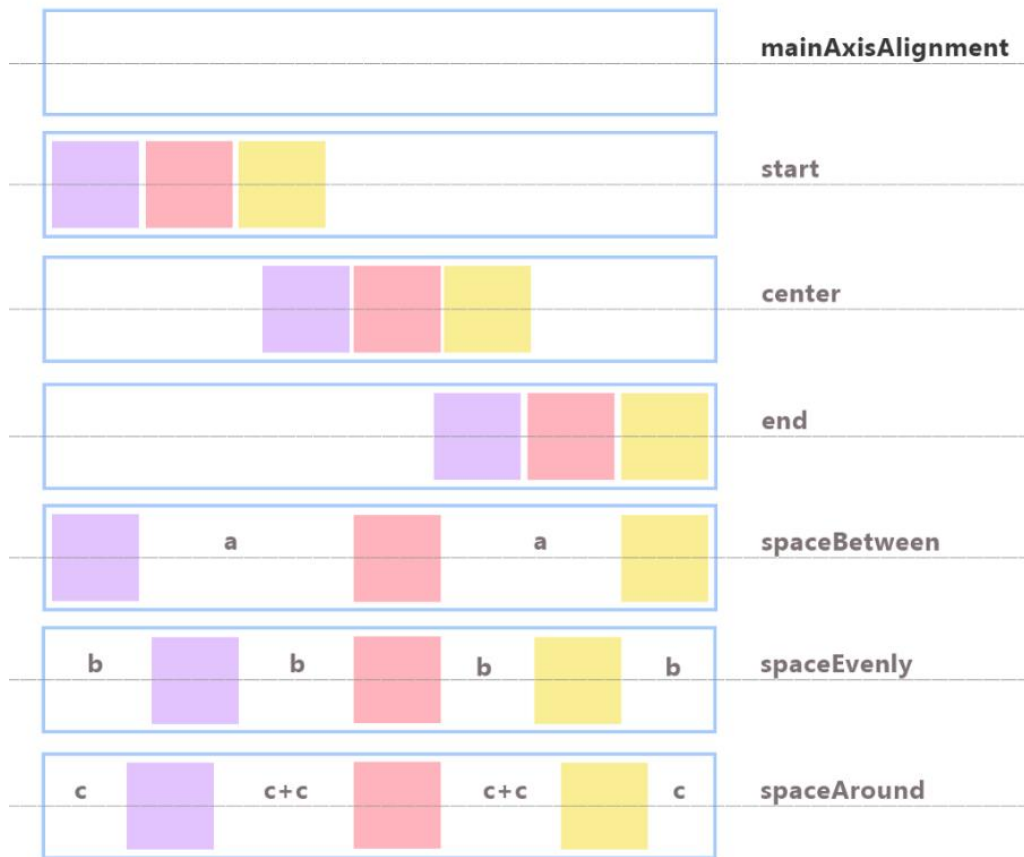
```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    debugShowCheckedModeBanner: false,
    home: Scaffold(
      body: Row(
        mainAxisAlignment: MainAxisAlignment.end,
        children: [
          Text('1 ☆'),
          Text('2 ☀'),
          Text('3 ☺'),
        ],
      ),
    ),
  );
}
```



Параметр *mainAxisAlignment*

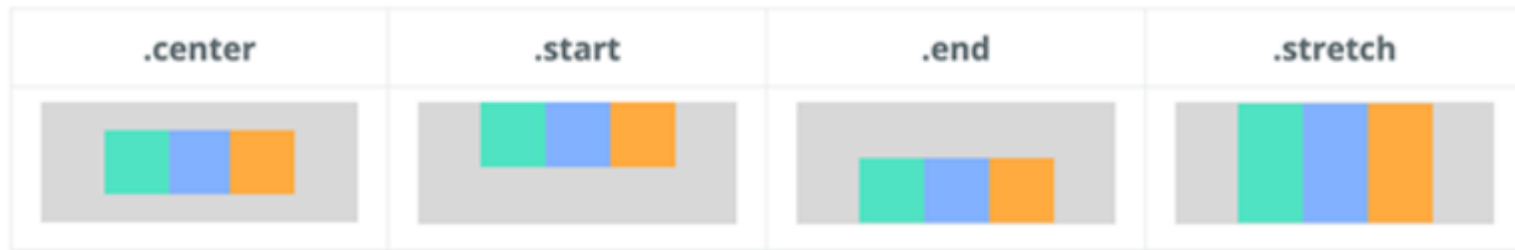
конструктора позволяет настроить позиционирование вложенных элементов по горизонтали. Он представляет тип `MainAxisAlignment` и может принимать следующие значения:

- `MainAxisAlignment.center`
- `MainAxisAlignment.end`
- `MainAxisAlignment.start`
- `MainAxisAlignment.spaceBetween`
- `MainAxisAlignment.spaceEvenly`
- `MainAxisAlignment.spaceAround`

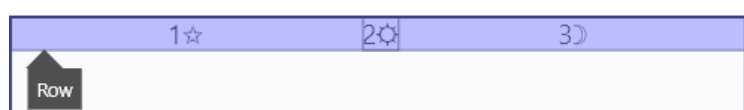


Параметр *crossAxisAlignment* в конструкторе Row указывает, как вложенные виджеты будут располагаться по вертикали. Этот параметр представляет класс `CrossAxisAlignment` и может принимать следующие значения:

- `CrossAxisAlignment.center`
- `CrossAxisAlignment.end`
- `CrossAxisAlignment.start`
- `CrossAxisAlignment.stretch`
- `CrossAxisAlignment.baseline`

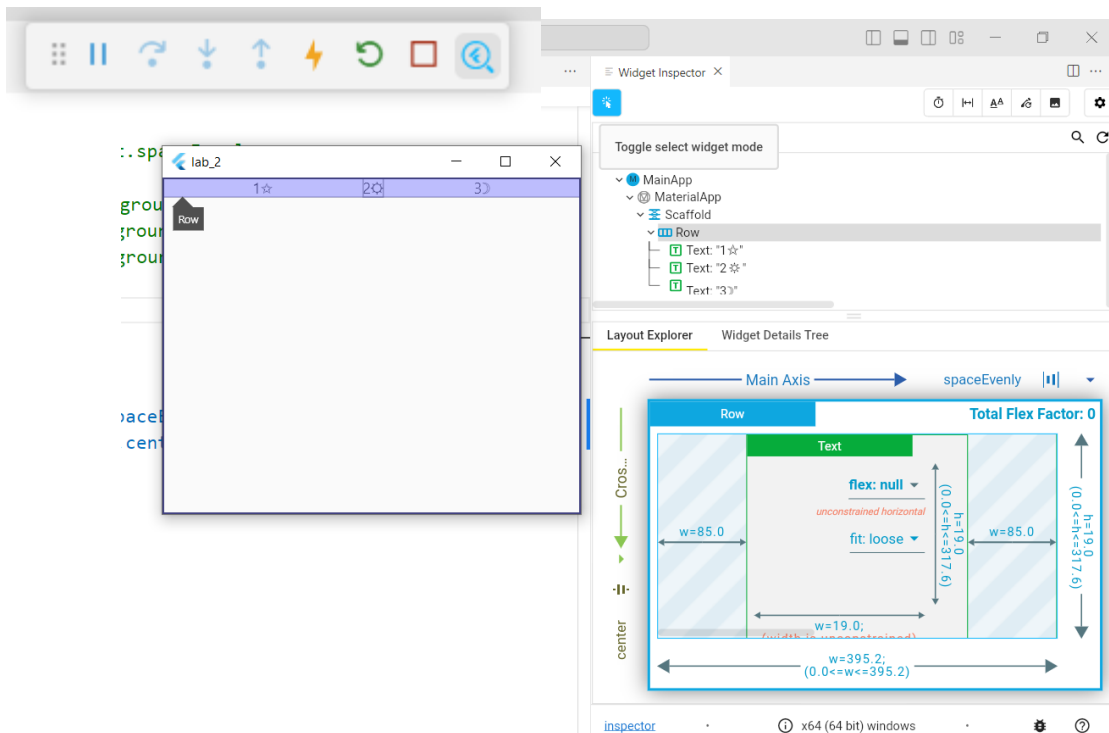


Элемент Row занимает высоту по содержимому.



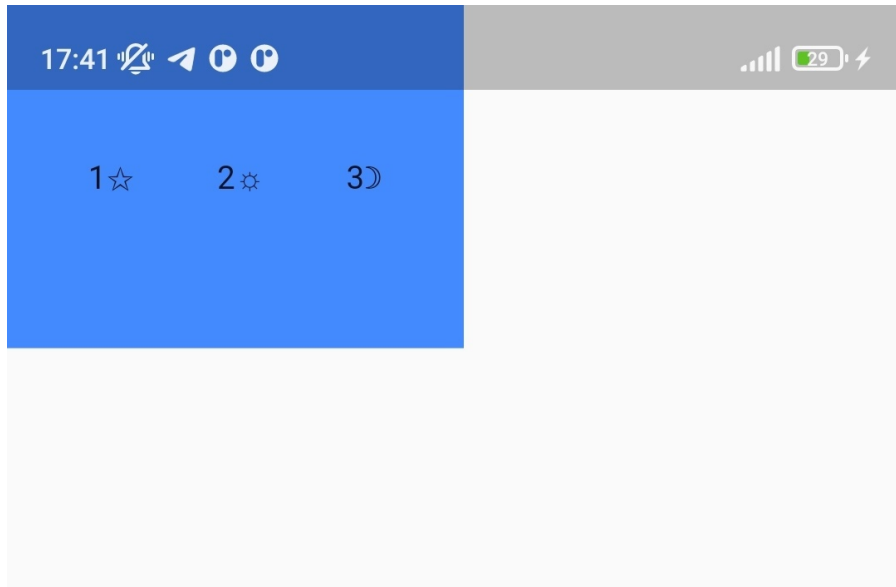
* Для просмотра размеров виджета удобно использовать **инструмент *Widget Inspector***.

В vs code нужно нажать на иконку лупы в панели, появляющейся при запуске проекта.



Пример.

```
Container(  
  color: Colors.blueAccent,  
  height: 150,  
  width: 200,  
  child: Row(  
    mainAxisAlignment:  
MainAxisAlignment.spaceEvenly,  
    crossAxisAlignment:  
CrossAxisAlignment.center,  
    children: [  
      Text ('1 ☆'),  
      Text ('2 ☀'),  
      Text ('3 ☺'),  
    ],  
  ),  
),
```



Column

Виджет *Column* располагает свои дочерние элементы в вертикальном направлении, в виде столбца.

Во многом похож на виджет Row, кроме расположения главной и поперечной осей (main and cross axes): не имеет прокрутки и позволяет создавать гибкие макеты.

Виджеты *Row* и *Column* являются хорошим выбором для создания адаптивных макетов под разные разрешения экрана. Можно использовать Expanded или Flexible виджет в качестве дочернего элемента Row и Column, чтобы дочерние виджеты занимали пропорциональный объем доступного пространства.

Конструктор (имеет те же параметры, что и у Row):

Column({

Key key, // ключ виджета

MainAxisAlignment mainAxisAlignment: MainAxisAlignment.start, // направление по по вертикали

MainAxisSize mainAxisSize: MainAxisSize.max, // пространство, занимаемое виджетом по вертикали

CrossAxisAlignment crossAxisAlignment: CrossAxisAlignment.center, // выравнивание по горизонтали

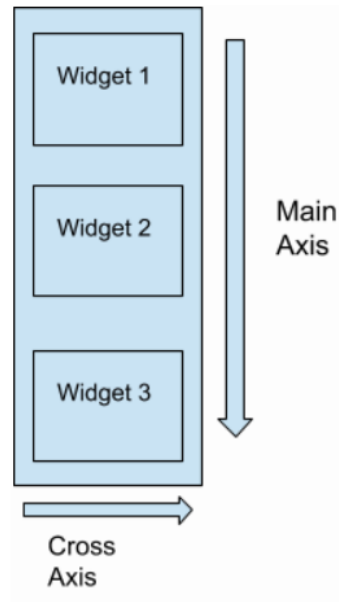
TextDirection textDirection, // порядок расположения вложенных элементов по горизонтали

VerticalDirection verticalDirection: VerticalDirection.down, // порядок расположения вложенных элементов по вертикали

TextBaseline textBaseline, // базовая линия для выравнивания элементов

List<Widget> children: const [] // набор вложенных элементов

})



Параметры ***mainAxisAlignment*** и ***crossAxisAlignment*** будут принимать те же значения, что у виджета Row, с поправкой на то, что у Column главная ось проходит вертикально, а поперечная – горизонтально.

CrossAxis
Alignment



MainAxis
Alignment

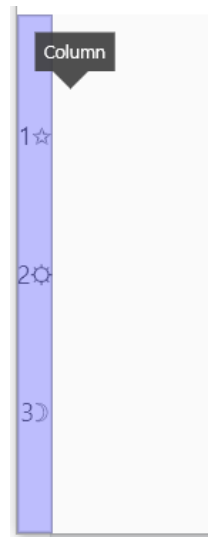


Пример.

```
Column(  
  mainAxisAlignment:  
    MainAxisAlignment.spaceEvenly,  
  crossAxisAlignment:  
    CrossAxisAlignment.center,  
  children: [  
    Text('1★'),  
    Text('2☀'),  
    Text('3☺'),  
  ],  
),
```



Column (по аналогии с Row) занимает всё доступное пространство родителя по главной оси, а по поперечной – по ширине дочерних элементов (у Row по высоте).



List View

Виджет *List View* представляет *прокручиваемый* список элементов.

Класс List View имеет *несколько конструкторов* с большим количеством параметров, поэтому отметим только *некоторые параметры*:

- **children**: объект List<Widget>, который представляет список виджетов, добавляемых в List View
- **scrollDirection**: устанавливает направление элементов. Представляет перечисление Axis, которое определяет две константы:
 - Axis.horizontal: устанавливает горизонтальный список - элементы располагаются слева направо (или справа налево)
 - Axis.vertical: устанавливает вертикальный список - элементы располагаются сверху вниз

Значение по умолчанию - Axis.vertical.

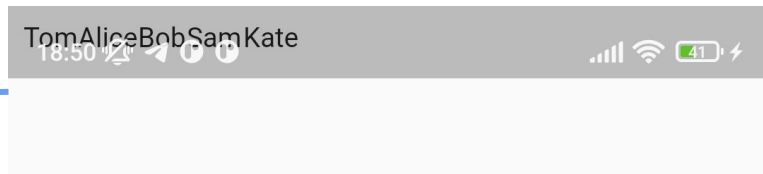
- `padding`: устанавливает отступы элементов от границ `ListView`, представляет объект `EdgeInsetsGeometry`
- `reverse`: если равен `true`, располагает элементы в обратном порядке
- `physics`: задает параметры скроллинга с помощью объекта `ScrollPhysics` (например, можно сделать список не прокручиваемым)

Два простейших примера.

```
ListView(  
  padding: const EdgeInsets.all(8),  
  children: [  
    Text("Tom"),  
    Text("Alice"),  
    Text("Bob"),  
    Text("Sam"),  
    Text("Kate"),  
  ],  
),
```



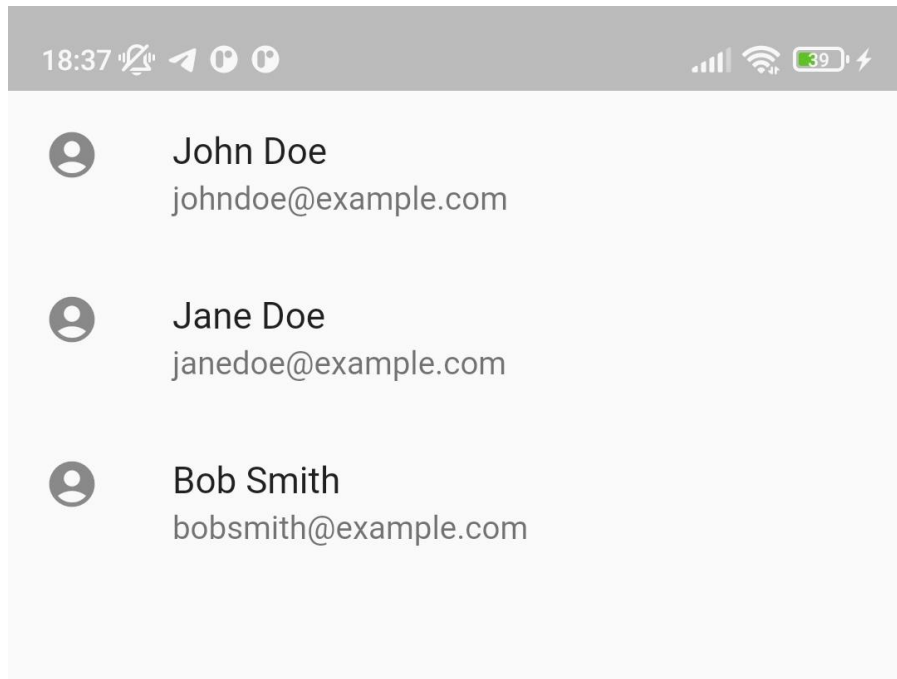
```
ListView(  
  padding: const EdgeInsets.all(8),  
  scrollDirection: Axis.horizontal,  
  children: [  
    Text("Tom"),  
    Text("Alice"),  
    Text("Bob"),  
    Text("Sam"),  
    Text("Kate"),  
  ],  
),
```



Для добавления отступов между элементами списка нужно обернуть их в какой-нибудь виджет, поддерживающий отступы.

Пример.

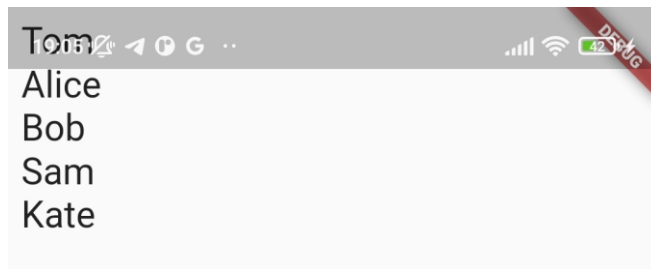
```
ListView(  
  children: [  
    ListTile(  
      leading: Icon(Icons.account_circle),  
      title: Text('John Doe'),  
      subtitle: Text('johndoe@example.com'),  
    ),  
    ListTile(  
      leading: Icon(Icons.account_circle),  
      title: Text('Jane Doe'),  
      subtitle: Text('janedoe@example.com'),  
    ),  
    ListTile(  
      leading: Icon(Icons.account_circle),  
      title: Text('Bob Smith'),  
      subtitle: Text('bobsmith@example.com'),  
    ),  
  ],  
)
```



Для *динамического создания списков* применяется **ListView.builder()** конструктор, который в качестве параметра *itemBuilder* принимает объект **IndexedWidgetBuilder**, который создает элементы списка. Этот конструктор более удобен для создания больших списков.

Пример:

```
const List<String> users = <String>["Tom", "Alice", "Bob", "Sam", "Kate"];
void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: ListView.builder(
        padding: const EdgeInsets.all(8),
        itemCount: users.length,
        itemBuilder: (BuildContext context, int index) {
          return Text(users[index], style: TextStyle(fontSize: 22));
        }
      ),
    ),
  ));
}
```



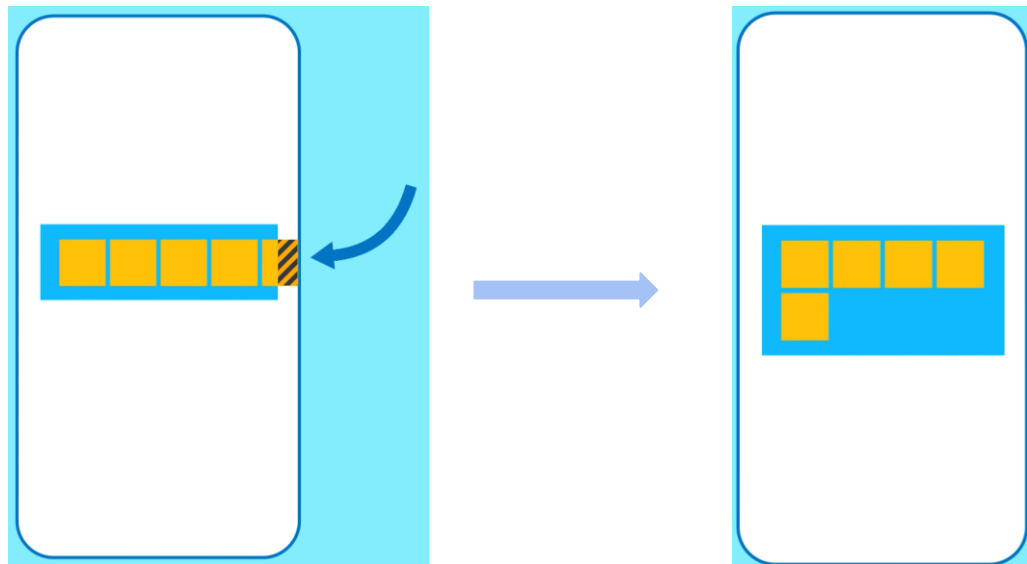
Объект ***IndexedWidgetBuilder***, который передается параметру ***itemBuilder***, по сути представляет функцию **Widget Function(BuildContext context, int index)**, которая получает *контекст виджета*, а также *индекс* и возвращает созданный виджет. Что передается в качестве индекса?

Здесь для ListView также устанавливается количество элементов с помощью параметра ***itemCount*** - оно равно количеству объектов в списке users: ***itemCount: users.length*** (в данном случае 5 элементов). В итоге itemBuilder будет последовательно перебирать числа от 0 до users.length, которые передаются в параметр index, и таким образом для каждого передаваемого индекса создавать виджет Text и передавать в виджет значение users[index].

Фактически по своему действию данную функцию можно сравнивать с циклом, в котором для каждого индекса создается свой виджет Text.

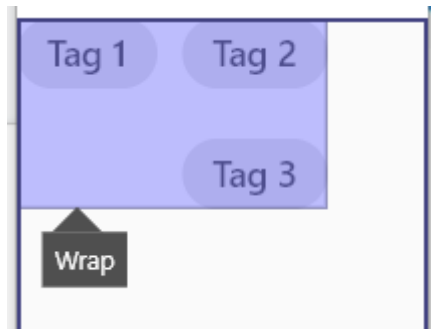
Wrap

Виджет **Wrap** располагает свои дочерние элементы в горизонтальном или вертикальном направлении и переносит элементы на следующую строку (столбец) если в текущей строке (столбце) не хватает места.



Пример.

```
Wrap(  
  direction: Axis.horizontal,  
  alignment: WrapAlignment.end,  
  spacing: 10.0, // пространство до следующего виджета  
  runSpacing: 20.0, // пространство между строками/столбцами  
  children: [  
    Chip(  
      label: Text('Tag 1'),  
    ),  
    Chip(  
      label: Text('Tag 2'),  
    ),  
    Chip(  
      label: Text('Tag 3'),  
    ),  
  ],  
)
```



Flow

Виджет **Flow** располагает позволяет позиционировать дочерние элементы с использованием пользовательской логики.

Flow может быть особенно полезен, когда вам **нужно создать собственный макет** с нестандартно расположенными дочерними виджетами. Например, если вы создаете игровое поле, на котором плитки должны располагаться непрямоугольным узором, вы можете использовать Flow , чтобы расположить плитки вручную.

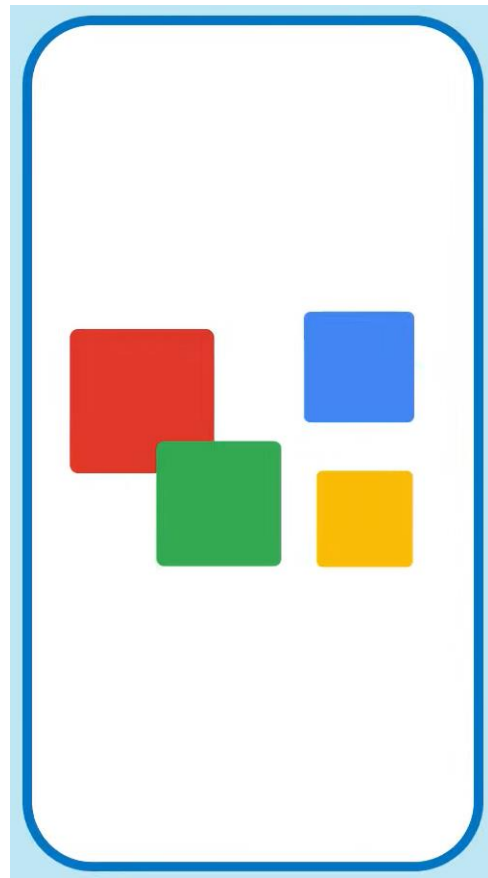
Flow — это низкоуровневый примитив макета, который дает вам детальный контроль над позиционированием дочерних виджетов. Обычно он не используется напрямую в большинстве сценариев разработки приложений, поскольку виджеты макета более высокого уровня, такие как Row, Column, Stack и Wrap, предоставляют более удобные абстракции для большинства потребностей макета. Однако могут быть случаи, когда макет Flow является лучшим инструментом для работы.



Например, Flow можно использовать для создания выпадающего меню.

Stack

Stack используется для размещения одного виджета поверх другого в виде стека. Stack можно использовать для создания сложных макетов, в которых *виджеты перекрывают друг друга* или где виджеты должны *располагаться точно по отношению друг к другу*.



Дочерние элементы Stack могут быть позиционированы или не позиционированы.

- Позиционированный виджет: обернут в виджет Positioned и работает с комбинацией параметров — по вертикали (сверху, снизу, по высоте) и по горизонтали (слева, справа и по ширине) для позиционирования виджетов в стеке.

К позиционированному виджету относится так же виджет обернутый в Align (для Align можно использовать свойство alignment).

- Виджет без позиционирования: если элемент Stack не обернут виджетом Align или Positioned, то он считается виджетом без позиционирования.













Непозиционированные виджеты оказываются на экране в зависимости от свойства alignment стека. По умолчанию верхний левый угол экрана.

Конструктор:

```
Stack({  
  Key key,  
  AlignmentGeometry alignment: AlignmentDirectional.topStart, // расположение вложенных виджетов  
  TextDirection textDirection, // порядок расположения вложенных элементов по горизонтали  
  StackFit fit: StackFit.loose, // размеры для вложенных виджетов  
  Overflow overflow: Overflow.clip, // устанавливает, надо ли усекать вложенное содержимое  
  Clip clipBehavior: Clip.hardEdge, // устанавливает, как вложенные элементы будут усекаться  
  List<Widget> children: const <Widget>[]  
})
```

По умолчанию вложенные виджеты располагаются в верхнем левом углу элемента Stack (по умолчанию для свойства alignment установлено значение `AlignmentDirectional.topStart`), но естественно с помощью свойства alignment, которое представляет тип `AlignmentGeometry` мы можем переопределить положение вложенных виджетов с помощью одного из следующих значений:

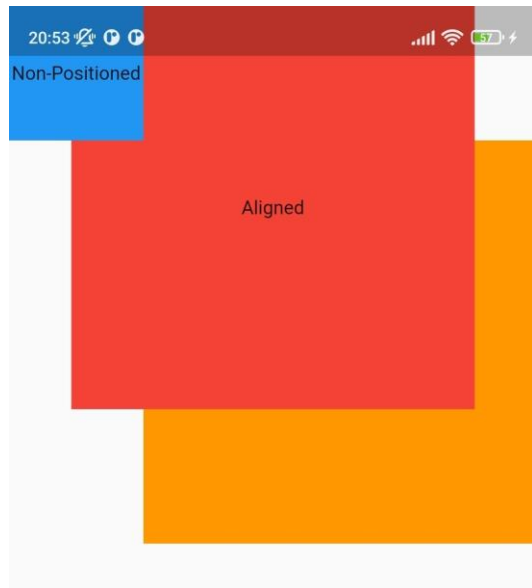
- `AlignmentDirectional.topStart`
- `AlignmentDirectional.topEnd`
- `AlignmentDirectional.topCenter`
- `AlignmentDirectional.bottomStart`
- `AlignmentDirectional.bottomEnd`
- `AlignmentDirectional.bottomCenter`
- `AlignmentDirectional.center`
- `Alignment.centerStart`
- `Alignment.centerEnd`

Enum	LTR	RTL
<code>AlignmentDirectional.topStart</code>		
<code>AlignmentDirectional.topEnd</code>		
<code>AlignmentDirectional.centerStart</code>		
<code>AlignmentDirectional.centerEnd</code>		
<code>AlignmentDirectional.bottomStart</code>		
<code>AlignmentDirectional.bottomEnd</code>		

Пример.

```
Stack(  
  alignment: Alignment.topLeft,  
  children: <Widget>[  
    Positioned(  
      top: 100,  
      left: 100,  
      child: Container(  
        height: 300,  
        width: 300,  
        child: Center(child: Text('Positioned')),  
        color: Colors.orange,  
      ),  
    ),  
    Align(  
      alignment: Alignment.topCenter,  
      child: Container(  
        height: 300,  
        width: 300,  
        child: Center(child: Text('Aligned')),  
        color: Colors.red,  
      ),  
    ),  
  ],  
)
```

```
Container(  
  height: 100,  
  width: 100,  
  child: Center(child: Text('Non-Positioned')),  
  color: Colors.blue,  
)
```

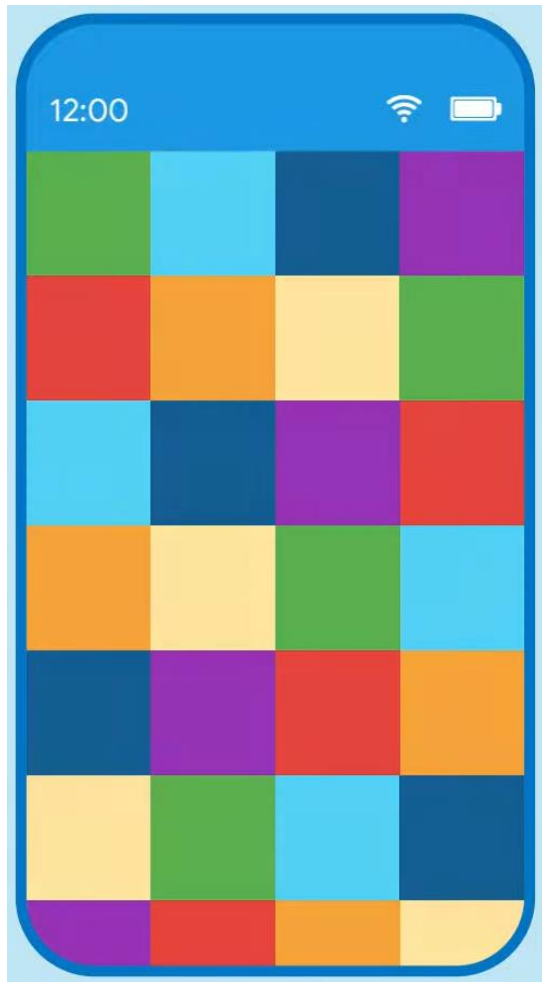


GridView

GridView — это прокручиваемая сетка виджетов.

GridView — это мощный инструмент для создания сложных макетов сетки в приложении, который поддерживает широкий спектр параметров настройки, помогающих создать идеальную сетку для нужд вашего приложения.

Направление главной оси сетки — это направление, в котором она прокручивается.



Наиболее часто используемый конструктор — ***GridView.count***.

Некоторые из важных свойств:

- **scrollDirection**: указывает направление, в котором будет прокручиваться GridView.

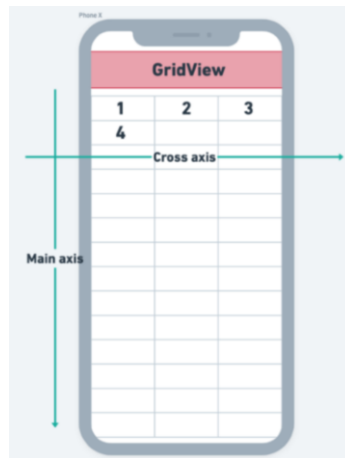
По умолчанию он прокручивается в вертикальном направлении

- **crossAxisCount**: указывает количество столбцов

- **crossAxisSpacing**: определяет расстояние между столбцами/строками по поперечной оси

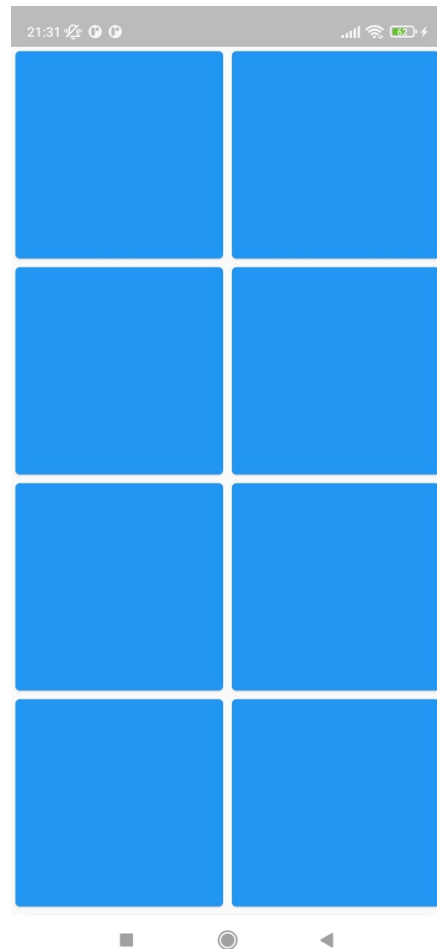
- **mainAxisSpacing**: определяет расстояние между столбцами/строками по главной оси

по умолчанию



Пример.

```
GridView.count(  
  crossAxisCount: 2, // => каждая строка будет содержать 2 виджета  
  // генерируем 10 виджетов Container  
  children: List.generate(10, (index) {  
    return Container(  
      child: Card(  
        color: Colors.blue,  
      ),  
    );  
  }  
)
```



Общие проблемы компоновки и их решения

Наиболее распространенными проблемами компоновки во Flutter являются такие проблемы как:

- ❑ адаптивный дизайн и изменение ориентации устройства
- ❑ выравнивание виджетов
- ❑ обработка переполнения

В этом разделе также будет найдено решение этих проблем с примерами кода и демонстрацией

Адаптивный дизайн и изменение ориентации устройства

Постановка задачи или проблемы адаптивной верстки

«Делай то, что тебе нравится».

Звучит как духовно обогащенный мотиватор, но это — реальный взгляд разработчиков фреймворка на эту проблему. У Flutter нет одного решения, «прибитого гвоздями», здесь у разработчика полная свобода и возможность выбирать способ решения этих проблем.

На данный момент Flutter поддерживает мобильные платформы (Android, iOS), Web, редко используется для desktop. Это значит, что приложение должно поддерживать широкий диапазон разрешений экранов устройств и ориентации. Также мобильное устройство (если оно не квадратное) может быть повернуто пользователем в портретную или ландшафтную ориентацию. Пользователи мобильных устройств любят и умеют это делать во время работы приложения, чтоб рассмотреть подробнее содержимое экрана. Так что, чтобы не разочаровывать пользователя, мы должны позаботиться о проблеме поворота экрана во время работы приложения.

И при всем этом приложению желательно еще и работать, отображая информацию о своей жизнедеятельности, несмотря на характеристики и параметры устройства, на которое его занесло, и на то, какие действия с ним может совершать пользователь.

Как это уже работает в Android

Проблема и способы ее решения известны. На Android в свое время это называлось отзывчивый (responsive) дизайн.

Как это выглядит на практике для разработчика? В самом начале существует один макет, к которому по мере необходимости добавляем макеты для разных по разрешению экрана устройств, а затем обе ориентации для каждого типа устройств. Далее эти макеты выбираются в зависимости от того, на каком устройстве (с каким разрешением) и в какой ориентации они работают.

Для экранов с большой площадью обычно используют макеты с большим количеством информации, которую можно отобразить.

Реализация

Перед тем как писать код, опишем сценарии:

1. В пределах одного макета на разных устройствах с разной плотностью пикселей должно выполняться масштабирование контента.
2. Я хочу иметь возможность добавлять макеты в любом порядке (сделали макет для landscape — отлично, используем его везде, сделали для portrait — отлично, автоматически подставляем его для портретной ориентации).
3. Я хочу иметь возможность добавлять макеты для landscape, для portrait, для более широких экранов, для более узких.
4. Добавление новых макетов не должно влиять на существующие макеты.
5. Цепочка вызовов для созданного макета не должна без моего ведома изменяться со временем.
6. Я хочу, чтобы логика выбора макета не изменялась, ее не нужно было изменять при добавлении/удалении макетов.

Инициализация

Начнем с инициализации пакета. Вызов нашей MaterialApp оборачиваем в вызов инициализации виджета подстройки размеров виджетов под dpi экранов:

```
@override
```

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

```
  return LayoutBuilder(
```

```
    builder: (context, constraints) {
```

```
      return OrientationBuilder(
```

```
        builder: (context, orientation) {
```

```
         SizerUtil().init(constraints, orientation);
```

```
          return MaterialApp();
```

Шаг 1. Настройка виджетов под изменяемое разрешение экрана

Мы хотим отображать виджеты на разных экранах наиболее близко к задуманному при первоначальном дизайне.



Same UI in any Device
Responsiveness made Simple

Для реализации можно использовать следующий метод, основанный на `SizerUtil.orientation` свойстве для более вариабельной настройки параметров виджета. Пример:

```
appBar: AppBar(  
  title: SizerUtil.orientation == Orientation.portrait  
    ? const Text('portrait')  
    : const Text('landscape'),  
),
```

На AppBar выдается заголовок с названием текущей ориентации.

Проверим соответствие сценариям.

Этот способ выполняет наше пожелание из пункта 1 списка *«В пределах одного макета на разных устройствах с разной плотностью пикселей должно выполняться масштабирование контента»*.

Шаг 2. Используем специализированный виджет для работы с разными разрешениями экрана и ориентацией

Поддержка изменения ориентации и переключения виджетов под разные размеры экрана реализована с использованием виджета `ResponsiveWidget`. В его поля мы подставляем виджет для каждой пары разрешения/ориентации. В начале у него есть одно обязательное поле `landscapeLargeScreen`. Дополнительные виджеты под другие значения разрешения/ориентации опциональны, т.е их можно добавить по мере появления: `landscapeMediumScreen`, `landscapeSmallScreen`, `portraitMediumScreen`, `portraitSmallScreen`, `portraitLargeScreen`.

В приведенном ниже примере с помощью виджета `WelcomePage` создается страница в ориентации `landscape` и строится макет с использованием прокрутки контента в вертикальной плоскости, а для ориентации `portrait` используется такой же виджет с таким же набором страниц, только контент прокручивается в горизонтальной плоскости:

```
body: ResponsiveWidget(  
  landscapeLargeScreen: WelcomePage(  
    pageIndex: 0,  
    scrollDirection: Axis.vertical,  
    children: listOfPages, ),  
  portraitLargeScreen: WelcomePage(  
    pageIndex: 0,  
    scrollDirection: Axis.horizontal,  
    children: listOfPages, ), ),
```

portrait



Lorem ipsum

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s



Заключения

Мы получили экран, который имеет разные макеты для портретной и альбомной ориентации. Когда устройство меняет ориентацию, мы перестраиваем наш макет.

Как мы обнаруживаем изменения ориентации? Мы смотрим на отношение ширины к длине страницы. И в зависимости от этого переключаем макеты.

Это полностью соответствует пунктам 2-6 из исходных пожеланий.

Выравнивание виджетов

Здесь мы рассмотрим как создать виджеты, а также различные способы по их расположению на экране.

1. Выбираем виджет макета

Выберите из множества рассмотренных ранее виджетов макета в зависимости от того, как вы хотите выровнять или ограничить видимый виджет, поскольку эти характеристики обычно передаются в содержащийся виджет. В этом примере используется Center, который центрирует свое содержимое по горизонтали и вертикали.

2. Создаем видимый виджет

Например, создание текстового виджета Text:

```
Text('Hello World'),
```

Создание виджета изображения Image:

```
Image.asset(  
  'images/lake.jpg',  
  fit: BoxFit.cover,)
```

Создание виджета иконки Icon:

```
Icon(  
  Icons.star,  
  color: Colors.red[500],  
)
```

3. Добавляем видимый виджет в виджет макета

Все виджеты макета имеют следующие положения:

- Используется свойство **child**, если виджет имеет только один дочерний элемент – к примеру, **Center** или **Container**
- Используется свойство **children**, если виджет имеет список дочерних элементов – к примеру, **Row**, **Column**, **ListView**, или **Stack**

Ниже показан пример добавление текстового виджета **Text** в виджет **Center**:

```
Center(  
  child: Text('Hello World'),  
),
```

4. Добавление виджета макета на страницу

Приложение Flutter само по себе является виджетом, и большинство виджетов имеют метод **build()**. Создание и возврат виджета в методе приложения **build()** отображает виджет.

Hello World

Приложение с Material Design

Для приложения **Material** вы можете использовать виджет Scaffold он предоставляет баннер по умолчанию с цветом фона и имеет API для добавления перекрывающей меню(**drawers**), полосы уведомлений(**snack bars**) и нижних вкладок(**sheets**). Затем вы можете добавить виджет **Center** непосредственно в свойство **body** домашней страницы.

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter layout demo',  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text('Flutter layout demo'),  
        ),  
        body: Center(  
          child: Text('Hello World'),  
        ),  
      ),  
    );  
  }  
}
```

Приложение без Material Design

Для приложений, не относящихся к Material Design, вы можете добавить виджет **Center** в метод **build()** приложения:

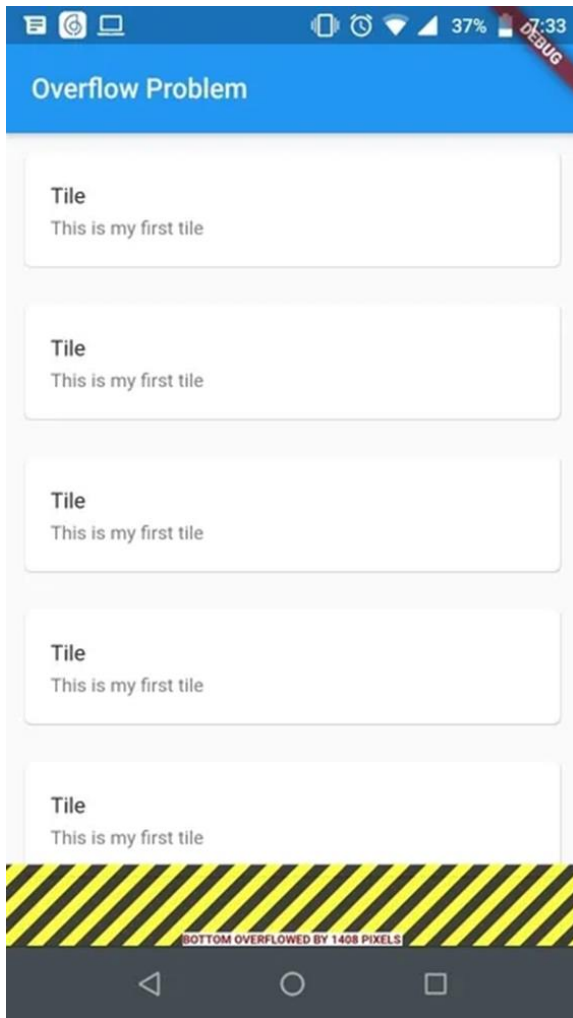
```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Container(  
      decoration: BoxDecoration(color: Colors.white),  
      child: Center(  
        child: Text(  
          'Hello World',  
          textDirection: TextDirection.ltr,  
          style: TextStyle(  
            fontSize: 32,  
            color: Colors.black87,  
          ),  
        ),  
      ),  
    );  
  }  
}
```

Hello World

Переполнения (overflow)

Здесь мы рассмотрим как выглядит переполнение, как оно появляется и что нужно делать, чтобы избавиться от него.

Переполнения бывают двух видов в зависимости от расположения: bottom overflow, horizontal overflow.



Bottom overflow

На изображении внутри виджета столбца просто повторена плитка столько раз, чтобы она требовала большего размера экрана и создавала для нас проблему переполнения. Как разработчик, мы видим черно-желтый полосатый индикатор переполнения с информацией о том, на сколько пикселей переполняется дно; в данном случае 1408. Как пользователь, мы не сможем увидеть эту полосатую рамку, что приведет к плохому пользовательскому интерфейсу; невозможно прокручивать, пропускать скрытые виджеты.

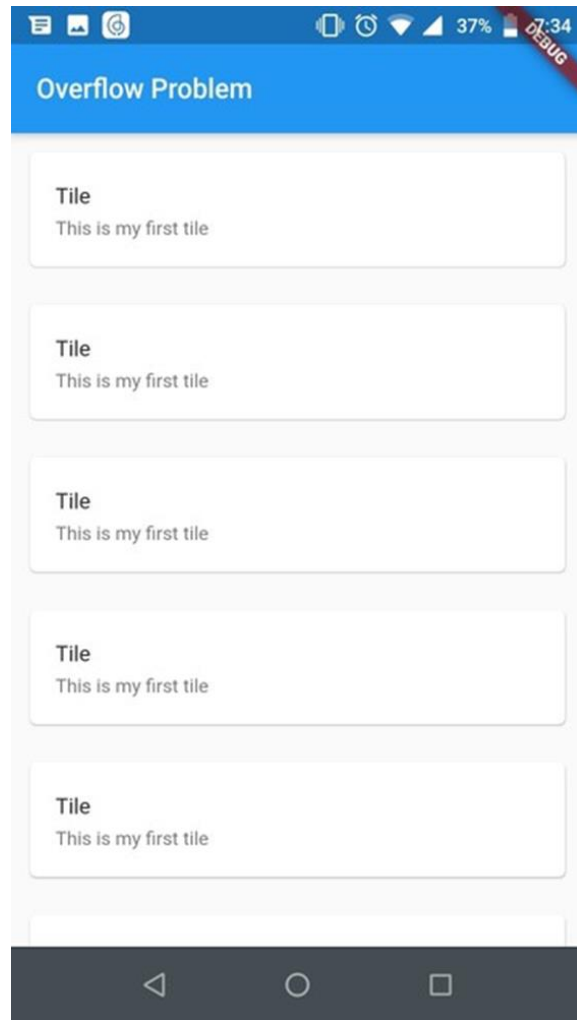
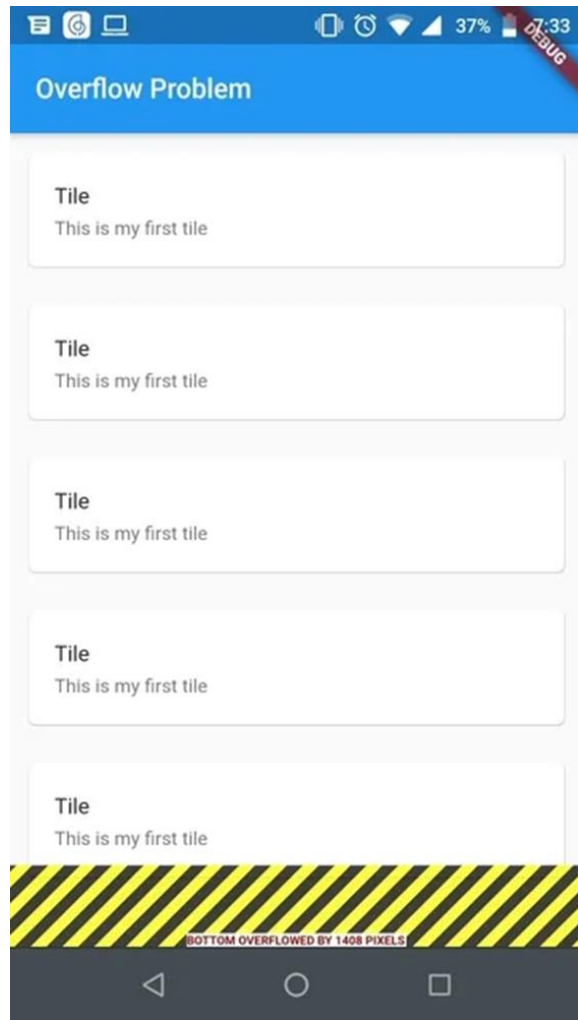
Как же исправить эту проблему?

Здесь представлен код, когда было переполнение

[illegible]

Здесь мы исправили уже эту проблему (обернули виджет столбца другим виджетом с именем `SingleChildScrollView`, который позволяет пользователю прокручивать содержимое своего дочернего элемента, если они занимают больше размера экрана)

[illegible]



Overflow Problem

This is my first text This is my second text This is my third text



RIGHT OVERFLOWED BY

Horizontal overflow

Мы смогли сделать экран прокручиваемым, когда дело дошло до вертикального переполнения. Можем ли мы сделать то же самое для горизонтального переполнения? Да, но это выглядело бы некрасиво. Здесь создано горизонтальное переполнение путем помещения слишком большого содержимого внутри виджета строки.

Решение

Для избавления от данного вида переполнения воспользуемся внедрением Flexible виджета.

Flexible виджет можно разместить внутри столбца, строки и подобных виджетов, чтобы окружить любой из их дочерних элементов. Flexible помогает дочернему избежать переполнения, отображая его только в доступном пространстве.

```
Padding(  
  padding: const EdgeInsets.all(8.0),  
  child: Row(  
    mainAxisAlignment:  
      MainAxisAlignment.spaceEvenly,  
    children: [  
      Flexible(child: Card(child: Text('This  
is my first text'))),  
      Card(child: Text('This is my second  
text')),  
      Card(child: Text('This is my third  
text'))  
    ],),  
  ),  
),
```

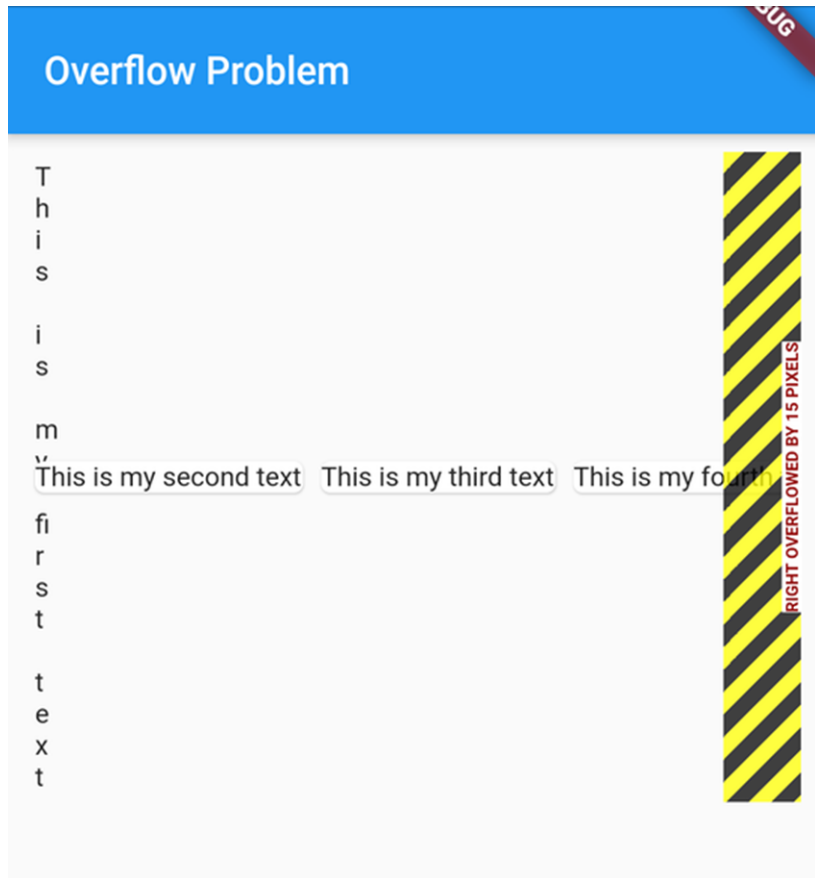
Overflow Problem

This is my first
text

This is my second text

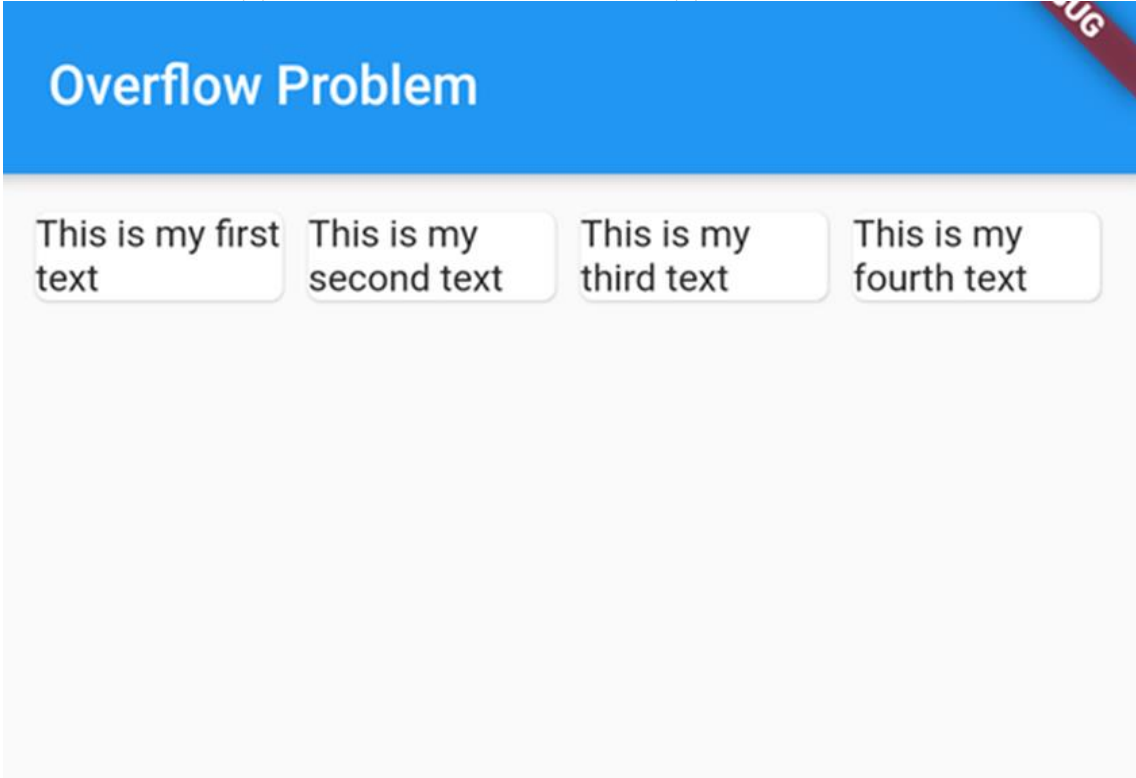
This is my third text

Но что будет, если мы захотим добавить еще один похожий дочерний элемент внутри нашего виджета **Row**? Мы видим, что гибкость не сохранилась.



```
Padding(  
  padding: const EdgeInsets.all(8.0),  
  child: Row(  
    mainAxisAlignment:  
    MainAxisAlignment.spaceEvenly,  
    children: [  
      Flexible(child: Card(child: Text('This  
is my first text'))),  
      Card(child: Text('This is my second  
text')),  
      Card(child: Text('This is my third  
text')),  
      Card(child: Text('This is my fourth  
text')),  
    ],  
  ),  
)
```

Простое решение этой проблемы заключается в том, что мы помещаем каждый из наших дочерних элементов в один и тот же **гибкий** виджет.



```
Padding(
  padding:
    const EdgeInsets.all(8.0),
  child: Row(
    mainAxisAlignment:
      MainAxisAlignment.spaceEvenly,
    children: [
      Flexible(child: Care(child: Text('This
is my first text'))),
      Flexible(child: Care(child: Text('This
is my second text'))),
      Flexible(child: Care(child: Text('This
is my third text'))),
      Flexible(child: Care(child: Text('This
is my fourth text'))),
    ],
```


Лучшие практики для работы с layout виджетами. Как оптимизировать макеты для повышения производительности и скорости отклика в приложениях Flutter

Есть много вопросов, решений и заблуждений, как мы можем улучшить производительность нашего Flutter приложения. Необходимо сразу уточнить, что Flutter является производительным по умолчанию, но мы должны избегать некоторых ошибок при написании кода, чтобы приложение всегда работало хорошо и быстро.

Советы

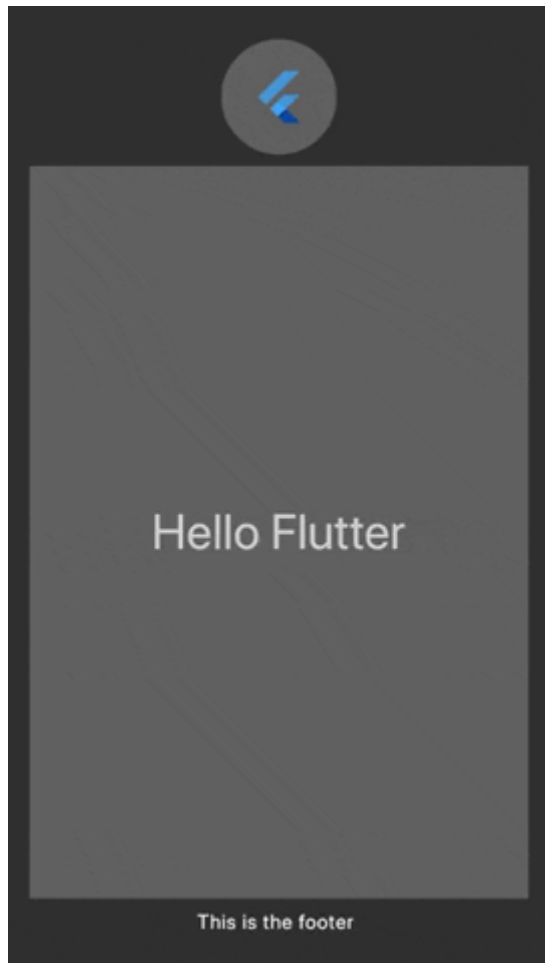
- ☐ Не выносите виджеты в методы класса
- ☐ Избегайте повторных перестроений всех виджетов
- ☐ Используйте `const`
- ☐ Используйте `itemExtent` в `ListView` для больших списков

Не выносите виджеты в методы класса

Когда у нас есть сложное представление, чтобы реализовать в один виджет, то обычно мы разделяем его на виджеты поменьше, которые помещаем в методы класса.

В следующем примере представлен виджет, содержащий заголовок, основной контент и "подвал" (*footer*).

```
class MyHomePage extends StatelessWidget {  
  Widget _buildHeaderWidget() {  
    final size = 40.0;  
    return Padding(  
      padding: const EdgeInsets.all(8.0),  
      child: CircleAvatar(  
        backgroundColor: Colors.grey[700],  
        child: FlutterLogo(  
          size: size,  
        ),  
        radius: size,  
      ),  
    );  
  }  
  
  Widget _buildMainWidget(BuildContext context) {  
    return Expanded(  
      child: Container(  
        color: Colors.grey[700],  
        child: Center(  
          child: Text(  
            'Hello Flutter',  
            style: Theme.of(context).textTheme.display1,  
          ),  
        ),  
      ),  
    );  
  }  
  
  Widget _buildFooterWidget() {  
    return Padding(  
      padding: const EdgeInsets.all(8.0),  
      child: Text('This is the footer '),);  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Padding(  
        padding: const EdgeInsets.all(15.0),  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.min,  
          children: [  
            _buildHeaderWidget(),  
            _buildMainWidget(context),  
            _buildFooterWidget(),  
          ],  
        ),  
      ),  
    );  
  }  
}
```



То, что мы увидели выше, – это антипаттерн. Почему так? Всё потому, что, когда мы вносим изменения и обновляем `MyHomePage` виджет, то виджеты, которые у нас вынесены в методах, также обновляются, даже если в этом нет никакой необходимости. Чтобы не было лишних вычислений, мы можем переделать эти методы в `StatelessWidget` следующим образом.

```
class MyHomePage extends  
StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Padding(  
        padding: const EdgeInsets.all(15.0),  
        child: Column(  
          mainAxisAlignment: MainAxisAlignment.min,  
          children: [  
            HeaderWidget(),  
            MainWidget(),  
            FooterWidget(),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

```
class MainWidget extends  
StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Expanded(  
      child: Container(  
        color: Colors.grey[700],  
        child: Center(  
          child: Text(  
            'Hello Flutter',  
            style:  
              Theme.of(context).textTheme.display1,  
          ),  
        ),  
      ),  
    );  
  }  
}
```

Избегайте повторных перестроений всех виджетов

Это обычная ошибка, которую многие совершают, когда начинают использовать Flutter и впервые сталкиваются с необходимостью обновить StatefulWidget с помощью setState.

Следующий пример — это представление, содержащее квадрат в центре и FloatingActionButton кнопку, при каждом нажатии на которую вызывается изменение цвета. На странице также есть виджет с фоновым изображением. Кроме того, мы добавим вызов print оператора внутри build метода каждого виджета, чтобы посмотреть, как он работает.

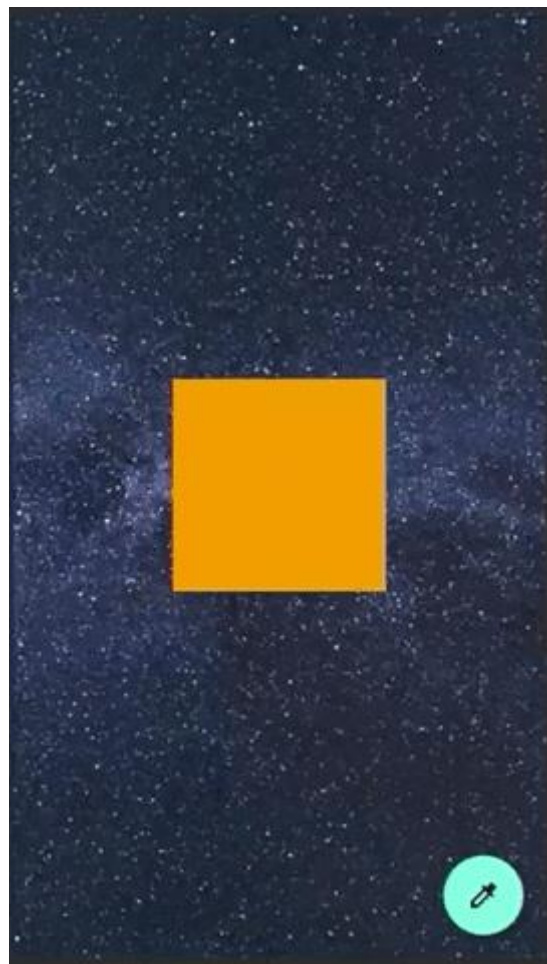
```
class _MyHomePageState extends State<MyHomePage> { @override
  Color _currentColor = Colors.grey;

  Random _random = new Random();

  void _onPressed() {
    int randomNumber = _random.nextInt(30);
    setState(() {
      _currentColor = Colors.primaryes[randomNumber %
Colors.primaryes.length];
    });
  }

  Widget build(BuildContext context) {
    print('building `MyHomePage`');
    return Scaffold(
      floatingActionButton: FloatingActionButton(
        onPressed: _onPressed,
        child: Icon(Icons.colorize), ),
      body: Stack(
        children: [
          Positioned.fill(
            child: BackgroundWidget(),
          ),
          Center(
            child: Container(
              height: 150,
              width: 150,
              color: _currentColor,)),]),
    );
  }
}

class BackgroundWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    print('building `BackgroundWidget`');
    return Image.network(
      'https://cdn.pixabay.com/photo/2017/08/30/01/05/
milky-way-2695569_960_720.jpg',
      fit: BoxFit.cover,
    );
  }
}
```



После клика по кнопке в консоли мы увидим два вывода

```
flutter: building `MyHomePage`
```

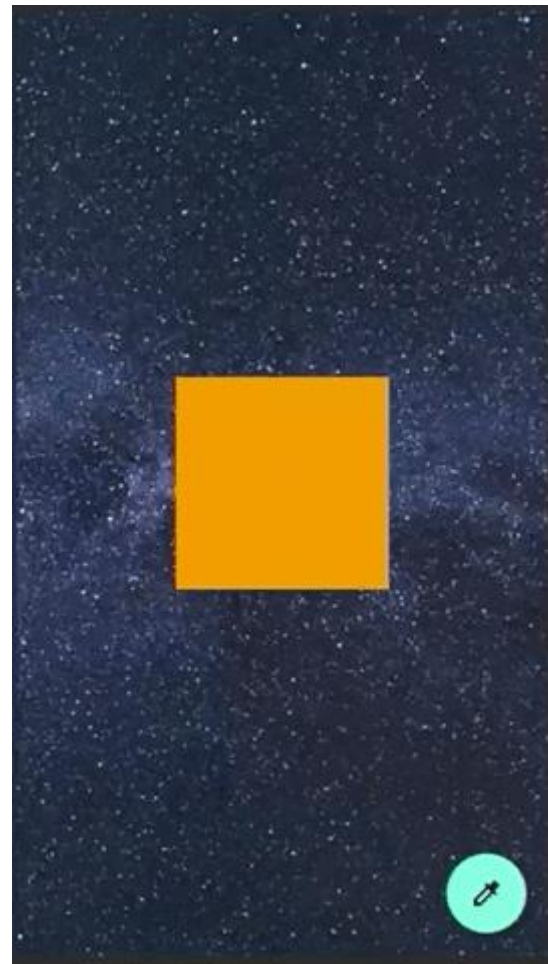
```
flutter: building `BackgroundWidget`
```

Каждый раз, когда мы нажимаем кнопку, мы обновляем весь экран: Scaffold, BackgroundWidget и, наконец, то, что и хотели обновить, – квадрат-Container.

Как мы уже выяснили выше, перестраивать виджеты без необходимости – нехорошая практика. Обновляем только то, что нам нужно. Многие знают, что это можно сделать с помощью различных пакетов управления состоянием: flutter_bloc, mobx, provider. Но мало кто знает, что также это можно сделать с помощью классов, которые Flutter уже предлагает из коробки, без каких-либо сторонних библиотек.

Используем класс ValueNotifier

```
class _MyHomePageState extends  
State<MyHomePage> {  
  final _colorNotifier =  
ValueNotifier<Color>(Colors.grey);  
  Random _random = new Random();  
  
  void _onPressed() {  
    int randomNumber =  
_random.nextInt(30);  
    _colorNotifier.value =  
Colors primaries[randomNumber  
% Colors primaries.length];  
  }  
}
```

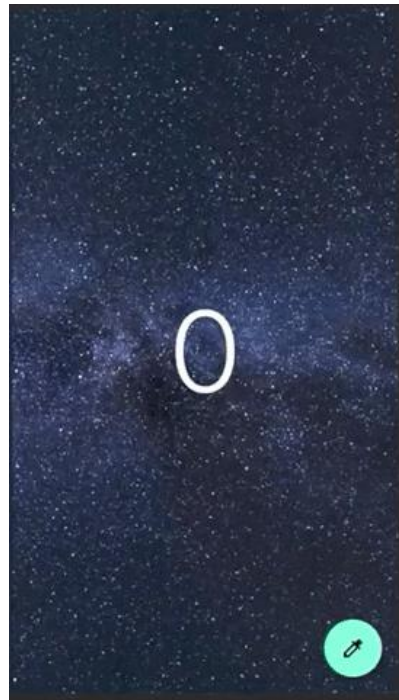


Используйте const

Рекомендуется использовать ключевое слово `const` для значений, которые возможно инициализировать во время компиляции, а также при вызове конструктора виджета (если он поддерживает `const`, конечно), что позволяет работать с одним и тем же экземпляром, тем самым избегая повторных вычислений.

Давайте ещё раз используем наш пример с `setState`, но в этот раз мы добавим счетчик, который будет увеличивать значение на 1 каждый раз при нажатии на кнопку.

```
class _MyHomePageState extends  
State<MyHomePage> {  
  int _counter = 0;  
  
  void _onPressed() {  
    setState() {  
      _counter++;  
    });  
  }  
}
```



У нас снова 2 вывода в консоли, один из которых относится к основному виджету, а другой – к BackgroundWidget. Каждый раз, когда мы нажимаем кнопку, мы видим, что дочерний виджет также перестраивается, хотя его содержимое никак не меняется.

```
flutter: building `MyHomePage`
```

```
flutter: building `BackgroundWidget`
```

А сейчас добавим const при работе с виджетом BackgroundWidget:

```
class BackgroundWidget extends
```

```
StatelessWidget {
```

```
  const BackgroundWidget();
```

```
  @override
```

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

```
    print('building `BackgroundWidget`');
```

```
    return Image.network(
```

```
      'https://cdn.pixabay.com/photo/2017/08/30/0
```

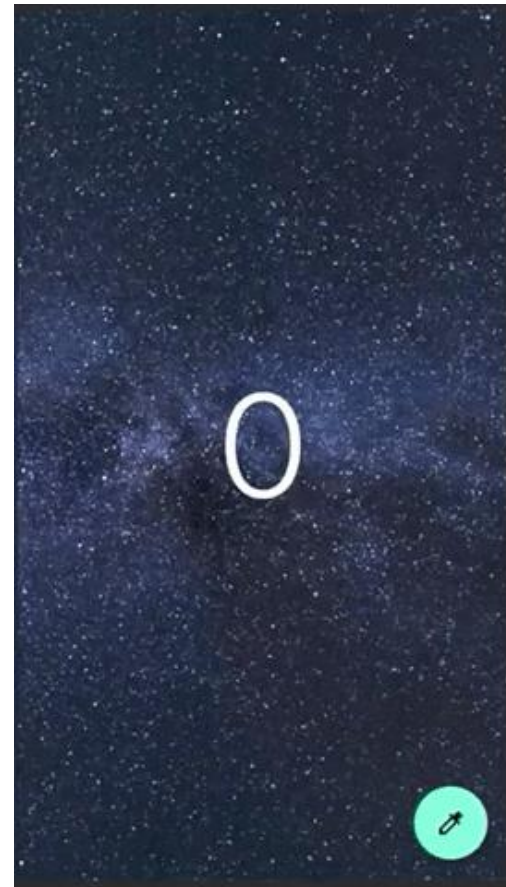
```
1/05/milky-way-2695569_960_720.jpg',
```

```
      fit: BoxFit.cover,
```

```
    );
```

```
  }
```

```
}
```



Используйте `itemExtent` в `ListView` для больших списков

Иногда, когда у нас есть очень длинный список, и мы хотим быстро переместиться по нему, например, в самый конец, очень важно использовать `itemExtent`.

Давайте рассмотрим простой пример. У нас есть список из 10 тысяч элементов. При нажатии на кнопку мы перейдем к последнему элементу. В этом примере мы не будем использовать `itemExtent` и позволим элементам списка самим определить свой размер.

```
class MyHomePage extends StatelessWidget {  
  final widgets = List.generate(  
    10000,  
    (index) => Container(  
      height: 200.0,  
      color: Colors.primaryes[index %  
Colors.primaryes.length],  
      child: ListTile(  
        title: Text('Index: $index'),  
      ),  
    ),  
  );  
  
  final _scrollController = ScrollController();  
  
  void _onPressed() async {  
    _scrollController.jumpTo(  
      _scrollController.position.maxScrollExtent,  
    );  
  }  
}
```



Как можно увидеть на предыдущей анимации, переход происходит очень долго (~10 секунд). Так получается из-за того, что дочерние элементы сами определяют свой размер. Это даже блокирует UI!

Чтобы избежать этого, мы должны использовать свойство `itemExtent`, благодаря которому при прокрутке не совершается лишней работы по расчету позиции скролла, так как размеры элементов заранее известны.

```
@override
```

```
Widget build(BuildContext context) {  
  return Scaffold(  
    floatingActionButton:  
    FloatingActionButton(  
      onPressed: _onPressed,  
      splashColor: Colors.red,  
      child:  
      Icon(Icons.slow_motion_video),  
    ),  
    body: ListView(  
      controller: _scrollController,  
      children: widgets,  
      itemExtent: 200,  
    ),  
  );}}
```



Заключение

Flutter достаточно мощен, чтобы запускать наши приложения без проблем. Однако, всегда полезно следовать хорошим практикам и максимально оптимизировать наше приложение.