

Introducción (Marcelo Glasberg)

Para poder construir layouts de forma efectiva, es necesario comprender cómo funciona Constraints.

Para ello, es necesario aprender la siguiente regla: Las restricciones son descendentes, el tamaño es ascendentes y el padre determina la posición.

El diseño de Flutter no se puede entender realmente sin conocer esta regla, pero qué significa?

Un widget obtiene sus propias restricciones de su padre. Una restricción es simplemente un conjunto de 4 números dobles: un ancho mínimo y máximo, y una altura mínima y máxima.

Luego, el widget recorre su propia lista de hijos. Uno por uno, el widget le dice a sus hijos cuáles son sus restricciones (que pueden ser diferentes para cada hijo) y luego pregunta a cada hijo qué tamaño desea ser.

Luego, el widget posiciona a sus hijos (horizontalmente en el eje x y verticalmente en el eje y), uno por uno.

Y, finalmente, el widget le informa a su padre sobre su propio tamaño (dentro de las restricciones originales, por supuesto).

...

Código Base

main.dart X

```

1 import 'package:flutter/material.dart';
2
3 Run | Debug | Profile
4 void main() => runApp(MyApp());
5
6 class MyApp extends StatelessWidget { Constructors for public widgets should
7   @override
8   Widget build(BuildContext context) => FlutterLayoutArticle(<Example>[
9     Example1(),
10  ]); // <Example>[] // FlutterLayoutArticle
11
12 abstract class Example extends StatelessWidget { Constructors for public widgets
13   String get code;
14   String get explanation;
15 }
16
17 class FlutterLayoutArticle extends StatefulWidget {
18   final List<Example> examples;
19
20   FlutterLayoutArticle(this.examples); Constructors in '@immutable' classes
21
22   @override
23   _FlutterLayoutArticle createState() => _FlutterLayoutArticle(); Invalid use
24 }
25
26 class _FlutterLayoutArticle extends State<FlutterLayoutArticle> {
27   late int count;
28   late Widget example;
29   late String code;
30   late String explanation;
31
32   @override
33   void initState() {
34     count = 1;
35     code = Example1().code;
36     explanation = Example1().explanation;
37     super.initState();
38   }
39
40   @override
41   void didUpdateWidget(FlutterLayoutArticle oldWidget) {
42     super.didUpdateWidget(oldWidget);
43     var example = widget.examples[count - 1];
44     code = example.code;
45     explanation = example.explanation;
46   }
47
48   @override
49   Widget build(BuildContext context) {
50     return MaterialApp(
51       debugShowCheckedModeBanner: false,
52       title: 'Flutter Layout Article',
53       home: SafeArea(
54         child: Material(
55           color: Colors.black,
56           child: FittedBox(
57             child: Container(
58               width: 400,
59               height: 670,
60               color: const Color(0xFFCCCCCC),
61               child: Column(
62                 crossAxisAlignment: CrossAxisAlignment.center,
63                 children: [
64                   Expanded(
65                     child: ConstrainedBox(
66                       constraints: const BoxConstraints.tightFor(
67                         width: double.infinity, height: double.infinity), //
68                       child: widget.examples[count - 1],
69                     ), // ConstrainedBox
70                   ), // Expanded
71                   Container(

```

main.dart X

```

26 class _FlutterLayoutArticle extends State<FlutterLayoutArticle> {
27   Widget build(BuildContext context) {
28     Expanded(
29       child: ConstrainedBox(
30         constraints: const BoxConstraints.tightFor(
31           width: double.infinity, height: double.infinity), // Box
32         child: widget.examples[count - 1],
33       ), // ConstrainedBox
34     ), // Expanded
35     Container(
36       height: 50,
37       width: double.infinity,
38       color: Colors.black,
39       child: SingleChildScrollView(
40         scrollDirection: Axis.horizontal,
41         child: Row(
42           mainAxisAlignment: MainAxisAlignment.min,
43           children: [
44             for (var i = 0; i < widget.examples.length; i++)
45               Container(
46                 width: 50,
47                 padding:
48                   const EdgeInsets.only(left: 4.0, right: 4.0),
49                 child: button(i + 1),
50               ), // Container
51             ), // Row
52           ), // SingleChildScrollView
53         ), // Container
54         Container(
55           child: Scrollbar( The 'child' argument should be last in widget
56             child: SingleChildScrollView(
57               key: ValueKey(count),
58               child: Padding(
59                 padding: const EdgeInsets.all(10.0),
60                 child: Column(
61                   children: <Widget>[
62                     Center(
63                       child: Text(code),
64                     ), // Center
65                     SizedBox(height: 16), Use 'const' with the constructor
66                     Text(
67                       explanation,
68                       style: TextStyle(
69                         color: Colors.blue[900],
70                         fontStyle: FontStyle.italic), // TextStyle
71                     ), // Text
72                   ], // <Widget>[]
73                 ), // Column
74               ), // Padding
75             ), // SingleChildScrollView
76           ), // Scrollbar
77           height: 275,
78           color: Colors.grey[200],
79         ), // Container
80       ), // Column
81     ), // Container
82     ), // FittedBox
83     ), // Material
84   ), // SafeArea
85 ); // MaterialApp
86
87 Widget button(int exampleNumber) => Button(
88   key: ValueKey(exampleNumber),
89   isSelected: count == exampleNumber,
90   exampleNumber: exampleNumber,
91   onPressed: () {
92     showExample(
93       exampleNumber,
94       widget.examples[exampleNumber - 1].code
95     );
96   }
97 }

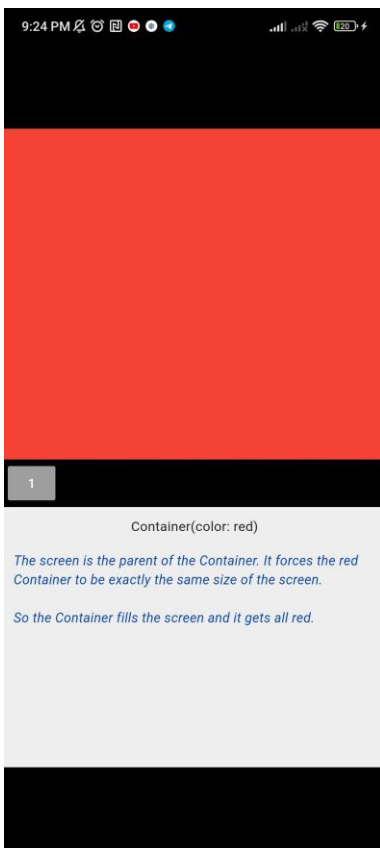
```

```

main.dart X
120 class _FlutterLayoutArticle extends State<FlutterLayoutArticle> {
121   Widget build(BuildContext context) {
122     // SafeArea
123     // MaterialApp
124   }
125
126   Widget button(int exampleNumber) => Button(
127     key: ValueKey<int>(exampleNumber),
128     isSelected: count == exampleNumber,
129     exampleNumber: exampleNumber,
130     onPressed: () {
131       showExample(
132         exampleNumber,
133         widget.examples[exampleNumber - 1].code,
134         widget.examples[exampleNumber - 1].explanation,
135       );
136     },
137   ); // Button
138
139   void showExample(int exampleNumber, String code, String explanation) =>
140     setState(() {
141       this.count = exampleNumber; // Unnecessary 'this.' qualifier. Try removing
142       this.code = code;
143       this.explanation = explanation;
144     });
145   }
146
147 class Button extends StatelessWidget {
148   final Key key; // The member 'key' overrides an inherited member but isn't annotated
149   final bool isSelected;
150   final int exampleNumber;
151   final VoidCallback onPressed;
152
153   Button({ // Constructors in '@immutable' classes should be declared as 'const'.
154     required this.key,
155     required this.isSelected,
156     required this.exampleNumber,
157     required this.onPressed,
158   }) : super(key: key);
159
160   @override
161   Widget build(BuildContext context) {
162     return MaterialButton(
163       color: isSelected ? Colors.grey : Colors.grey[800],
164       child: Text(
165         exampleNumber.toString(),
166         style: TextStyle(color: Colors.white), // Use 'const' with the constructor
167       ), // Text
168       onPressed: () {
169         Scrollable.ensureVisible(
170           context,
171           duration: Duration(milliseconds: 350), // Use 'const' with the constructor
172           curve: Curves.easeOut,
173           alignment: 0.5,
174         );
175         onPressed();
176       },
177     ); // MaterialButton
178   }
179 }
180
181 class Example1 extends Example { // Constructors for public widgets should have a
182   final String code = "Container(color: red);"; // The member 'code' overrides an
183   final String explanation = "The screen is the parent of the Container. " // The
184     "It forces the red Container to be exactly the same size of the screen."
185     "\n\n"
186     "So the Container fills the screen and it gets all red.";
187
188   @override
189   Widget build(BuildContext context) => Container(color: Colors.red);
190 }

```

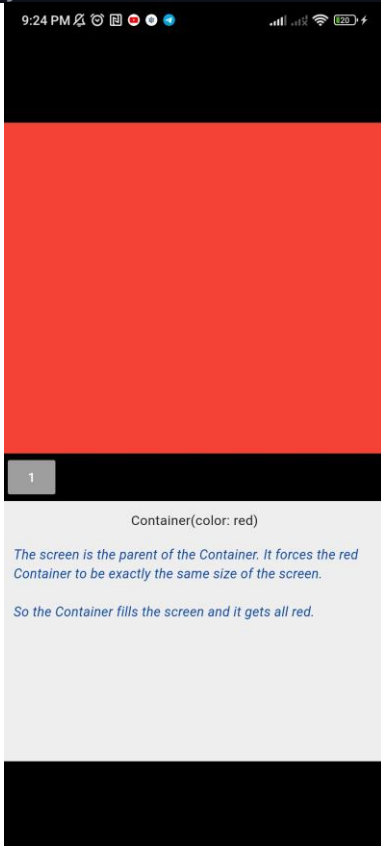
App Base:



Ejemplos:

```
class Example1 extends Example {
  final String code = "Container(color: red)";
  final String explanation = "The screen is the parent of the Container. "
    "It forces the red Container to be exactly the same size of the screen."
    "\n\n"
    "So the Container fills the screen and it gets all red.";

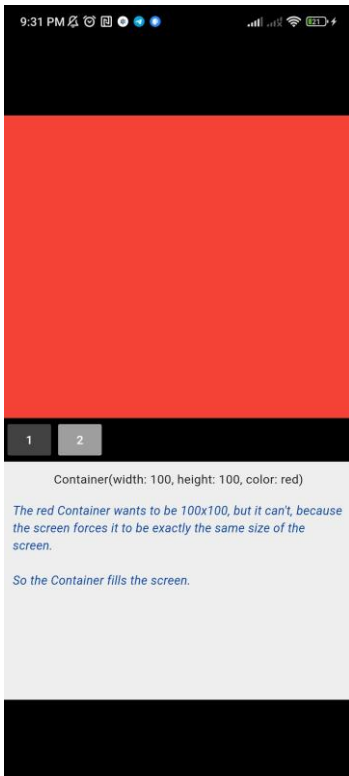
  @override
  Widget build(BuildContext context) => Container(color: Colors.red);
}
```



```
class Example2 extends Example {
  final String code = "Container(width: 100, height: 100, color: red)";

  final String explanation =
    "The red Container wants to be 100x100, but it can't, "
    "because the screen forces it to be exactly the same size of the screen."
    "\n\n"
    "So the Container fills the screen.";

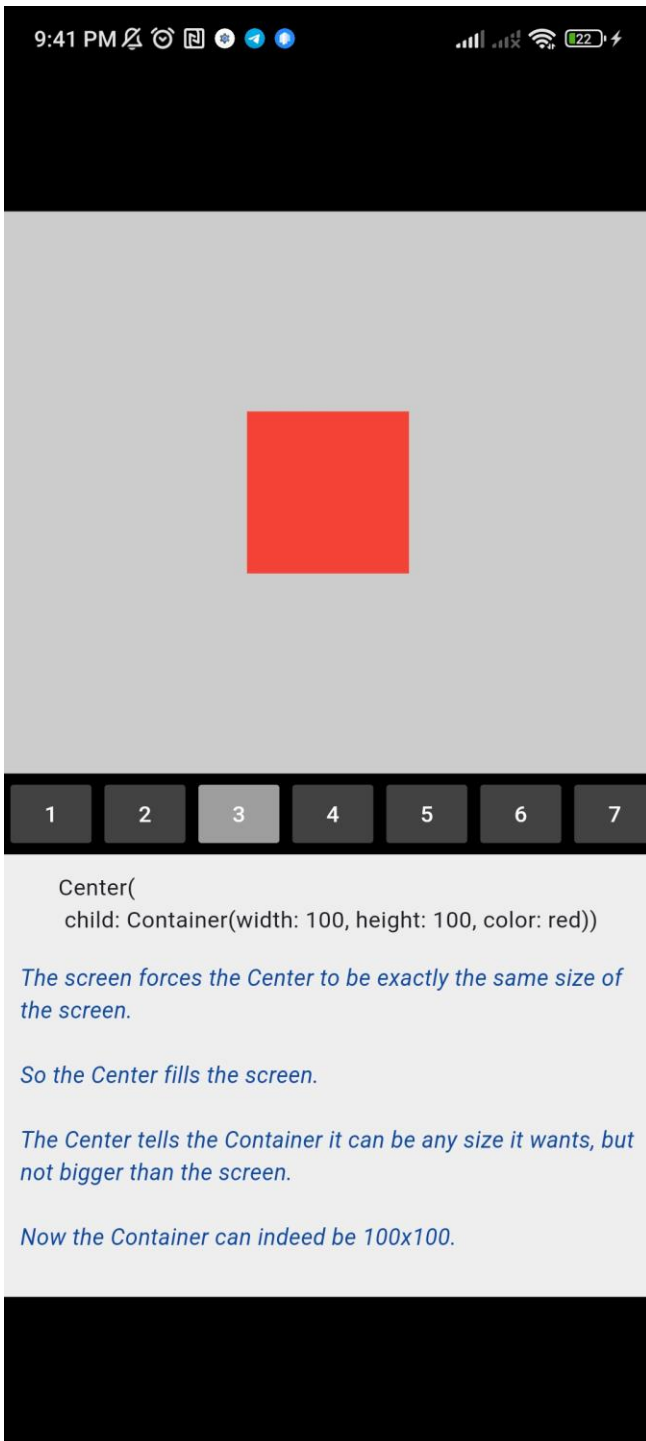
  @override
  Widget build(BuildContext context) {
    return Container(width: 100, height: 100, color: red);
  }
}
```



```
class Example3 extends Example {
  final String code = "Center(\n"
    "  child: Container(width: 100, height: 100, color: red));"

  final String explanation =
    "The screen forces the Center to be exactly the same size of the screen."
    "\n\n"
    "So the Center fills the screen."
    "\n\n"
    "The Center tells the Container it can be any size it wants, but not bigger than the"
    "screen."
    "\n\n"
    "Now the Container can indeed be 100x100.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(width: 100, height: 100, color: red),
    );
  }
}
```

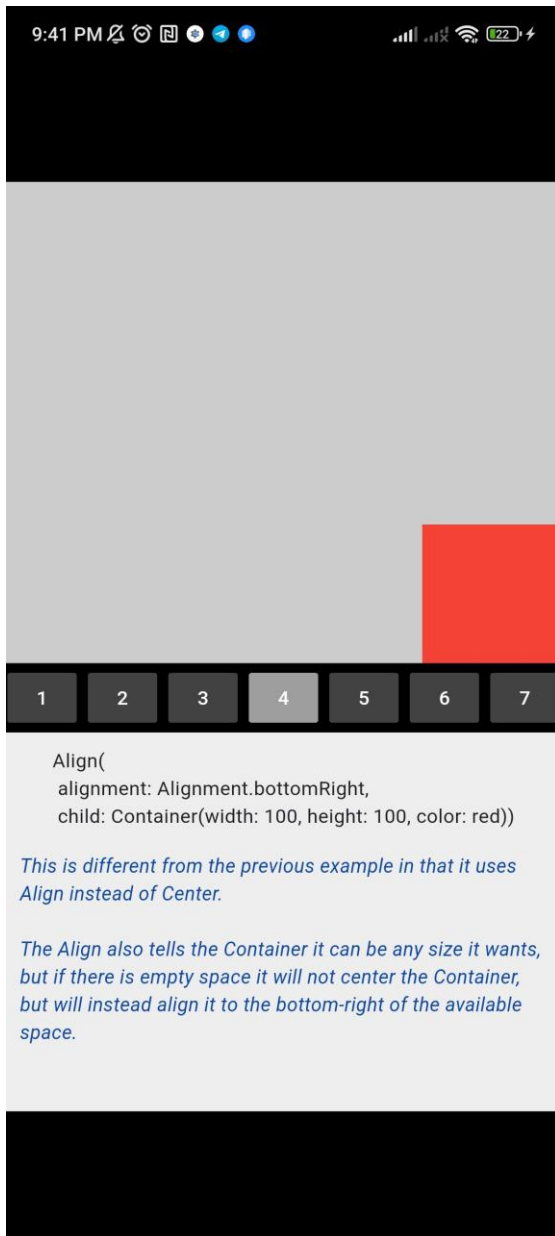


```
class Example4 extends Example {  
  final String code = "Align(\n"  
    " alignment: Alignment.bottomRight,\n"  
    " child: Container(width: 100, height: 100, color: red))";  
  
  final String explanation =  
    "This is different from the previous example in that it uses Align instead of Center."  
    "\n\n"
```

"The Align also tells the Container it can be any size it wants, but if there is empty space it will not center the Container, "

"but will instead align it to the bottom-right of the available space.";

```
@override
Widget build(BuildContext context) {
  return Align(
    alignment: Alignment.bottomRight,
    child: Container(width: 100, height: 100, color: red),
  );
}
```



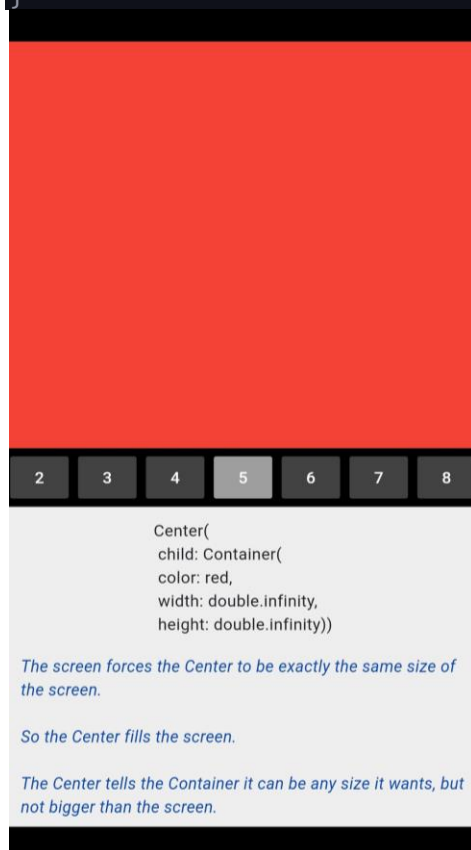
```

class Example5 extends Example {
  final String code = "Center(\n"
    "  child: Container(\n"
    "    color: red,\n"
    "    width: double.infinity,\n"
    "    height: double.infinity));"

  final String explanation =
    "The screen forces the Center to be exactly the same size of the screen."
    "\n\n"
    "So the Center fills the screen."
    "\n\n"
    "The Center tells the Container it can be any size it wants, but not bigger than the"
    "screen."
    "\n\n"
    "The Container wants to be of infinite size, but since it can't be bigger than the"
    "screen it will just fill the screen.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        width: double.infinity, height: double.infinity, color: red),
    );
  }
}

```



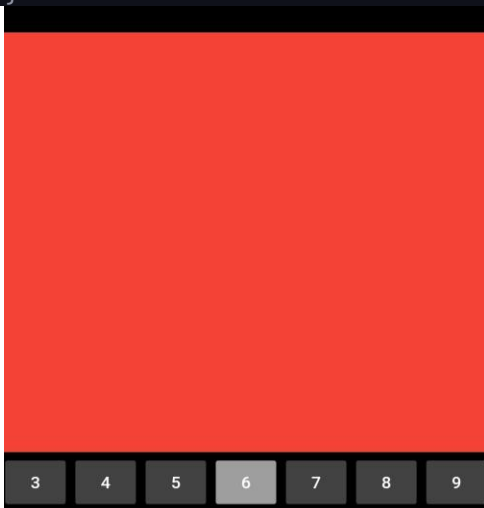

```

class Example6 extends Example {
  final String code = "Center(child: Container(color: red));

  final String explanation =
    "The screen forces the Center to be exactly the same size of the screen."
    "\n\n"
    "So the Center fills the screen."
    "\n\n"
    "The Center tells the Container it is free to be any size it wants, but not bigger than
the screen."
    "\n\n"
    "Since the Container has no child and no fixed size, it decides it wants to be as big
as possible, so it fits the whole screen."
    "\n\n"
    "But why does the Container decide that? "
    "Simply because that's a design decision by those who created the Container widget. "
    "It could have been created differently, and you actually have to read the Container's
documentation to understand what it will do depending on the circumstances. ";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(color: red),
    );
  }
}

```



Center(child: Container(color: red))

The screen forces the Center to be exactly the same size of the screen.

So the Center fills the screen.

The Center tells the Container it is free to be any size it wants, but not bigger than the screen.

Since the Container has no child and no fixed size, it decides it wants to be as big as possible, so it fits the whole screen.

Selected by the Container child: The Container

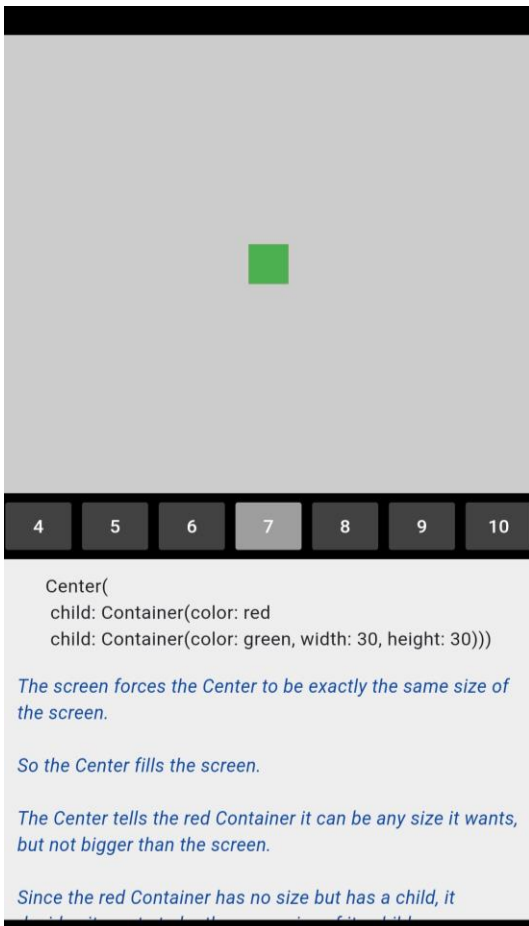
```

class Example7 extends Example {
  final String code = "Center(\n"
    "  child: Container(color: red\n"
    "  child: Container(color: green, width: 30, height: 30)))";

  final String explanation =
    "The screen forces the Center to be exactly the same size of the screen."
    "\n\n"
    "So the Center fills the screen."
    "\n\n"
    "The Center tells the red Container it can be any size it wants, but not bigger than"
    "the screen."
    "\n\n"
    "Since the red Container has no size but has a child, it decides it wants to be the"
    "same size of its child."
    "\n\n"
    "The red Container tells its child that it can be any size it wants, but not bigger"
    "than the screen."
    "\n\n"
    "The child happens to be a green Container, that wants to be 30x30."
    "\n\n"
    "As said, the red Container will size itself to its children size, so it will also be"
    "30x30. "
    "No red color will be visible, since the green Container will occupy all of the red"
    "Container.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        color: red,
        child: Container(color: green, width: 30, height: 30),
      ),
    );
  }
}

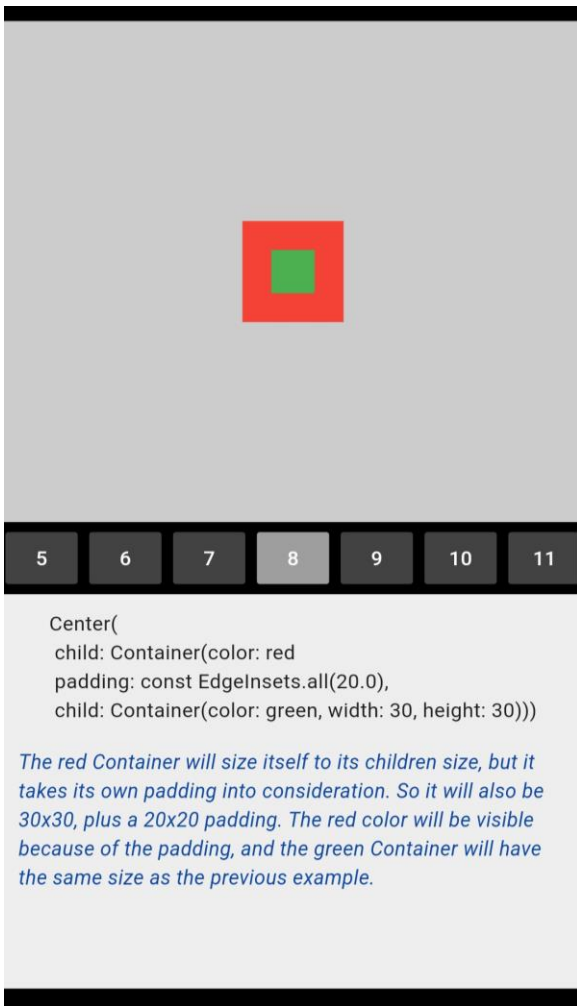
```



```
class Example8 extends Example {
  final String code = "Center(\n"
    "  child: Container(color: red\n"
    "    padding: const EdgeInsets.all(20.0),\n"
    "    child: Container(color: green, width: 30, height: 30)))";

  final String explanation =
    "The red Container will size itself to its children size, but it takes its own padding\n"
    "into consideration. "\n"
    "So it will also be 30x30, plus a 20x20 padding. "\n"
    "The red color will be visible because of the padding, and the green Container will\n"
    "have the same size as the previous example.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        padding: const EdgeInsets.all(20.0),
        color: red,
        child: Container(color: green, width: 30, height: 30),
      ),
    );
  }
}
```

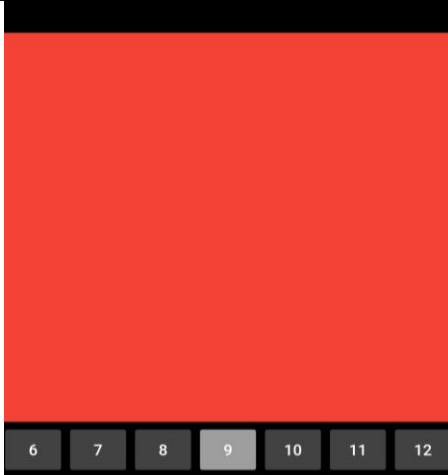


```
class Example9 extends Example {  
  final String code = "ConstrainedBox(\n"  
    " constraints: BoxConstraints(\n"  
    " minWidth: 70, minHeight: 70,\n"  
    " maxWidth: 150, maxHeight: 150),\n"  
    " child: Container(color: red, width: 10, height: 10)))";  
  
  final String explanation =  
    "You would guess the Container would have to be between 70 and 150 pixels, but you  
would be wrong. "  
    "The ConstrainedBox only imposes ADDITIONAL constraints than the ones it received from  
its parent."  
    "\n\n"  
    "Here, the screen forces the ConstrainedBox to be exactly the same size of the screen,  
"  
    "so it will tell its child Container to also assume the size of the screen, "  
    "thus ignoring its 'constraints' parameter.";  
  
  @override  
  Widget build(BuildContext context) {
```

```

return ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 70, minHeight: 70, maxWidth: 150, maxHeight: 150),
  child: Container(color: red, width: 10, height: 10),
);
}
}

```



```

ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 70, minHeight: 70,
    maxWidth: 150, maxHeight: 150),
  child: Container(color: red, width: 10, height: 10)))

```

You would guess the Container would have to be between 70 and 150 pixels, but you would be wrong. The ConstrainedBox only imposes ADDITIONAL constraints than the ones it received from its parent.

Here, the screen forces the ConstrainedBox to be exactly the same size of the screen, so it will tell its child Container to be exactly the same size of the screen.

```

class Example10 extends Example {
  final String code = "Center(\n"
    "  child: ConstrainedBox(\n"
    "    constraints: BoxConstraints(\n"
    "      minWidth: 70, minHeight: 70,\n"
    "      maxWidth: 150, maxHeight: 150),\n"
    "    child: Container(color: red, width: 10, height: 10))))";

  final String explanation =
    "Now, Center will allow ConstrainedBox to be any size up to the screen size."
    "\n\n"
    "The ConstrainedBox will impose its child the ADDITIONAL constraints from its"
    "'constraints' parameter."
    "\n\n"
    "So the Container must be between 70 and 150 pixels. It wants to have 10 pixels, so it"
    "will end up having 70 (the MINIMUM).";

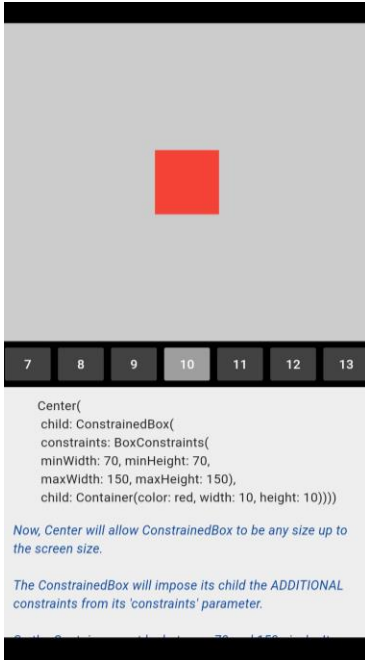
  @override
  Widget build(BuildContext context) {
    return Center(

```

```

child: ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 70, minHeight: 70, maxWidth: 150, maxHeight: 150),
  child: Container(color: red, width: 10, height: 10),
),
);
}
}

```



```

class Example11 extends Example {
  final String code = "Center(\n"
    " child: ConstrainedBox(\n"
    " constraints: BoxConstraints(\n"
    " minWidth: 70, minHeight: 70,\n"
    " maxWidth: 150, maxHeight: 150),\n"
    " child: Container(color: red, width: 1000, height: 1000))));";

  final String explanation =
    "Center will allow ConstrainedBox to be any size up to the screen size."
    "\n\n"
    "The ConstrainedBox will impose its child the ADDITIONAL constraints from its 'constraints' parameter."
    "\n\n"
    "So the Container must be between 70 and 150 pixels. It wants to have 1000 pixels, so it will end up having 150 (the MAXIMUM).";

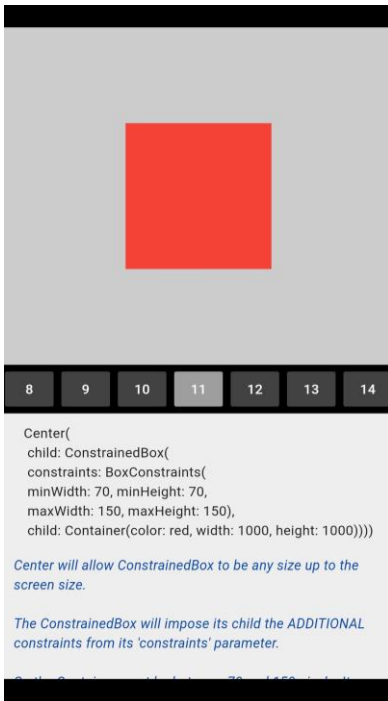
  @override
  Widget build(BuildContext context) {
    return Center(

```

```

child: ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 70, minHeight: 70, maxWidth: 150, maxHeight: 150),
  child: Container(color: red, width: 1000, height: 1000),
),
);
}
}

```



```

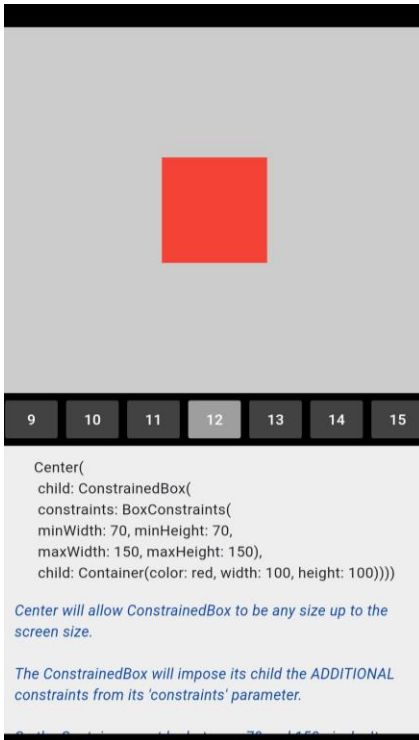
class Example12 extends Example {
  final String code = "Center(\n"
    " child: ConstrainedBox(\n"
    " constraints: BoxConstraints(\n"
    "   minWidth: 70, minHeight: 70,\n"
    "   maxWidth: 150, maxHeight: 150),\n"
    " child: Container(color: red, width: 100, height: 100))));";

  final String explanation =
    "Center will allow ConstrainedBox to be any size up to the screen size."
    "\n\n"
    "The ConstrainedBox will impose its child the ADDITIONAL constraints from its"
    "'constraints' parameter."
    "\n\n"
    "So the Container must be between 70 and 150 pixels. It wants to have 100 pixels, and"
    "that's the size it will have, since that's between 70 and 150.";

  @override

```

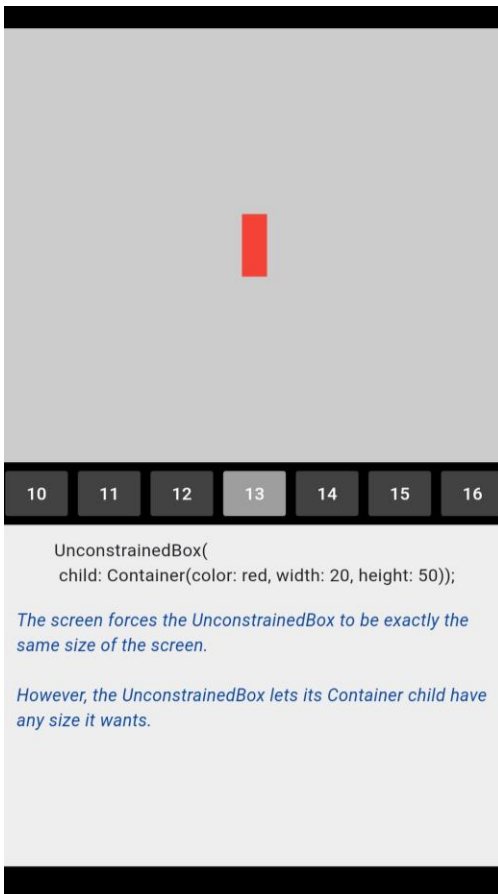
```
Widget build(BuildContext context) {
  return Center(
    child: ConstrainedBox(
      constraints: BoxConstraints(
        minWidth: 70, minHeight: 70, maxWidth: 150, maxHeight: 150),
      child: Container(color: red, width: 100, height: 100),
    ),
  );
}
```



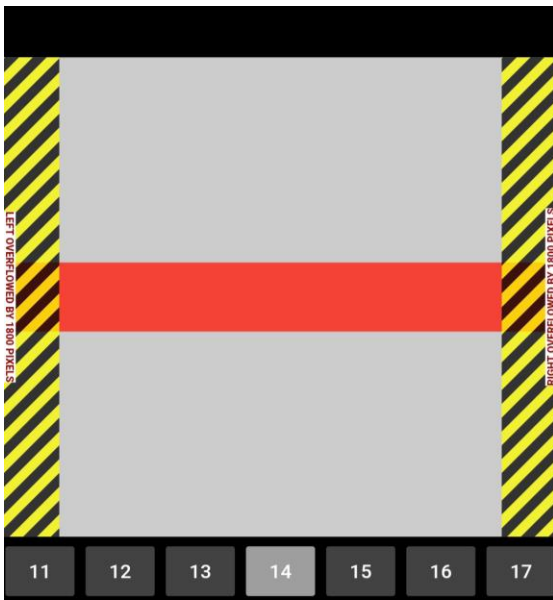
```
class Example13 extends Example {
  final String code = "UnconstrainedBox(\n"
    "  child: Container(color: red, width: 20, height: 50));";

  final String explanation =
    "The screen forces the UnconstrainedBox to be exactly the same size of the screen."
    "\n\n"
    "However, the UnconstrainedBox lets its Container child have any size it wants.";

  @override
  Widget build(BuildContext context) {
    return UnconstrainedBox(
      child: Container(color: red, width: 20, height: 50),
    );
  }
}
```

```
class Example14 extends Example {  
  final String code = "UnconstrainedBox(\n"  
    "  child: Container(color: red, width: 4000, height: 50));";  
  
  final String explanation =  
    "The screen forces the UnconstrainedBox to be exactly the same size of the screen, "  
    "and UnconstrainedBox lets its Container child have any size it wants."  
    "\n\n"  
    "Unfortunately, in this case the Container has 4000 pixels of width and is too big to  
fix UnconstrainedBox, "  
    "so the UnconstrainedBox will display the much dreaded \"overflow warning\".";  
  
  @override  
  Widget build(BuildContext context) {  
    return UnconstrainedBox(  
      child: Container(color: red, width: 4000, height: 50),  
    );  
  }  
}
```



```
UnconstrainedBox(
  child: Container(color: red, width: 4000, height: 50));
```

The screen forces the UnconstrainedBox to be exactly the same size of the screen, and UnconstrainedBox lets its Container child have any size it wants.

Unfortunately, in this case the Container has 4000 pixels of width and is too big to fit UnconstrainedBox, so the UnconstrainedBox will display the much dreaded "overflow warning".

```
class Example15 extends Example {
  final String code = "OverflowBox(\n"
    "  child: Container(color: red, width: 4000, height: 50));";

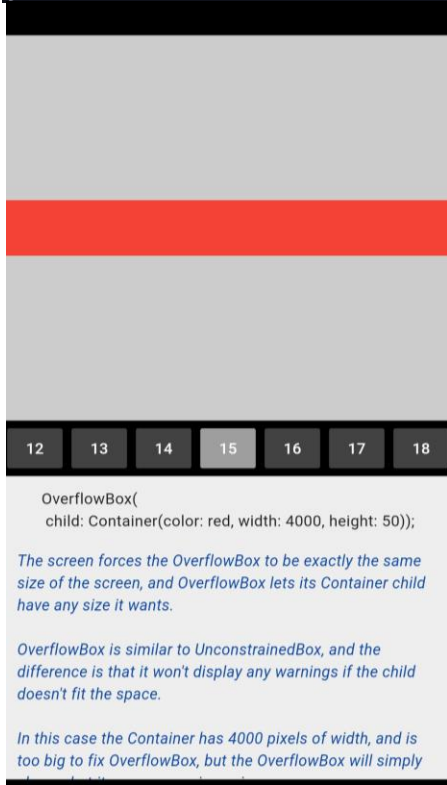
  final String explanation =
    "The screen forces the OverflowBox to be exactly the same size of the screen, "
    "and OverflowBox lets its Container child have any size it wants."
    "\n\n"
    "OverflowBox is similar to UnconstrainedBox, and the difference is that it won't "
    "display any warnings if the child doesn't fit the space."
    "\n\n"
    "In this case the Container has 4000 pixels of width, and is too big to fit "
    "OverflowBox, "
    "but the OverflowBox will simply show what it can, no warnings given.";

  @override
  Widget build(BuildContext context) {
    return OverflowBox(
      minWidth: 0.0,
      minHeight: 0.0,
```

```

    maxWidth: double.infinity,
    maxHeight: double.infinity,
    child: Container(color: red, width: 4000, height: 50),
  );
}
}

```



```

class Example16 extends Example {
  final String code = "UnconstrainedBox(\n"
    "  child: Container(color: Colors.red, width: double.infinity, height: 100));";

  final String explanation =
    "This won't render anything, and you will get an error in the console."
    "\n\n"
    "The UnconstrainedBox lets its child have any size it wants, "
    "however its child is a Container with infinite size."
    "\n\n"
    "Flutter can't render infinite sizes, so it will throw an error with the following"
    "message: "
    "'BoxConstraints forces an infinite width.'";

  @override
  Widget build(BuildContext context) {
    return UnconstrainedBox(
      child: Container(color: Colors.red, width: double.infinity, height: 100),
    );
  }
}

```

```
}
```

13 14 15 16 17 18 19

```
UnconstrainedBox(  
  child: Container(color: Colors.red, width: double.infinity,  
  height: 100));
```

This won't render anything, and you will get an error in the console.

The UnconstrainedBox lets its child have any size it wants, however its child is a Container with infinite size.

Flutter can't render infinite sizes, so it will throw an error with the following message: 'BoxConstraints forces an infinite width.'

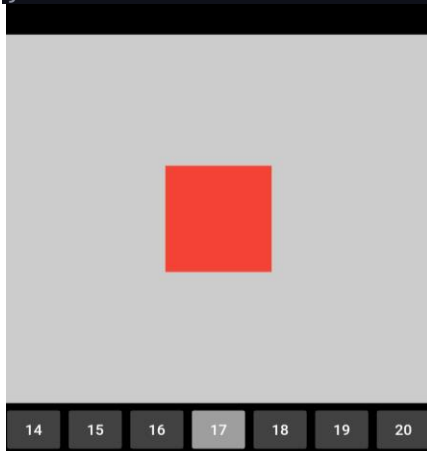
```
class Example17 extends Example {  
  final String code = "UnconstrainedBox(\n"  
    " child: LimitedBox(maxWidth: 100,\n"  
    " child: Container(color: Colors.red,\n"  
    " width: double.infinity, height: 100));";  
  
  final String explanation = "Here you won't get an error anymore, "  
    "because when the LimitedBox is given an infinite size by the UnconstrainedBox, "  
    "it will pass down to its child the maximum width of 100."  
    "\n\n"  
    "Note, if you change the UnconstrainedBox to a Center widget, "  
    "the LimitedBox will not apply its limit anymore (since its limit is only applied when  
it gets infinite constraints), "  
    "and the Container width will be allowed to grow past 100."  
    "\n\n"  
    "This makes it clear the difference between a LimitedBox and a ConstrainedBox.";  
  
  @override  
  Widget build(BuildContext context) {  
    return UnconstrainedBox(  

```

```

child: LimitedBox(
  maxWidth: 100,
  child:
    Container(color: Colors.red, width: double.infinity, height: 100),
),
);
}
}

```



```

UnconstrainedBox(
  child: LimitedBox(maxWidth: 100,
    child: Container(color: Colors.red,
      width: double.infinity, height: 100));

```

Here you won't get an error anymore, because when the LimitedBox is given an infinite size by the UnconstrainedBox, it will pass down to its child the maximum width of 100.

Note, if you change the UnconstrainedBox to a Center widget, the LimitedBox will not apply its limit anymore (since its limit is only applied when it gets infinite constraints), and the Container width will be allowed to grow past 100.

```

class Example18 extends Example {
  final String code = "FittedBox(\n"
    "  child: Text('Some Example Text.'));";

  final String explanation =
    "The screen forces the FittedBox to be exactly the same size of the screen."
    "\n\n"
    "The Text will have some natural width (also called its intrinsic width) that depends on the amount of text, its font size, etc."
    "\n\n"
    "The FittedBox will let the Text have any size it wants, "
    "but after the Text tells its size to the FittedBox, "
    "the FittedBox will scale it until it fills all of the available width.";

  @override
  Widget build(BuildContext context) {
    return FittedBox(
      child: Text("Some Example Text."),
    );
  }
}

```

```
}
```

Some Example Text.

15 16 17 18 19 20 21

```
FittedBox(  
  child: Text('Some Example Text.');
```

The screen forces the FittedBox to be exactly the same size of the screen.

The Text will have some natural width (also called its intrinsic width) that depends on the amount of text, its font size, etc.

The FittedBox will let the Text have any size it wants, but after the Text tells its size to the FittedBox, the FittedBox will scale it until it fills all of the available width.

```
class Example19 extends Example {  
  final String code = "Center(\n"  
    " child: FittedBox(\n"  
    " child: Text('Some Example Text.')));";  
  
  final String explanation =  
    "But what happens if we put the FittedBox inside of a Center? "  
    "The Center will let the FittedBox have any size it wants, up to the screen size."  
    "\n\n"  
    "The FittedBox will then size itself to the Text, and let the Text have any size it  
wants."  
    "\n\n"  
    "Since both FittedBox and the Text have the same size, no scaling will happen.";  
  
  @override  
  Widget build(BuildContext context) {  
    return Center(  
      child: FittedBox(  
        child: Text("Some Example Text."),  
      ),  
    );  
  }  
}
```

```
);
}
}
```

Some Example Text.

16 17 18 19 20 21 22

```
Center(
  child: FittedBox(
    child: Text('Some Example Text.'));
```

But what happens if we put the FittedBox inside of a Center? The Center will let the FittedBox have any size it wants, up to the screen size.

The FittedBox will then size itself to the Text, and let the Text have any size it wants.

Since both FittedBox and the Text have the same size, no scaling will happen.

```
class Example20 extends Example {
  final String code = "Center(\n"
    "  child: FittedBox(\n"
    "    child: Text('...')));\n";

  final String explanation =
    "However, what happens if FittedBox is inside of Center, but the Text is too large to\n"
    "fit the screen?\n"
    "\n\n"
    "FittedBox will try to size itself to the Text, but it cannot be bigger than the\n"
    "screen. "\n"
    "It will then assume the screen size, and resize the Text so that it fits the screen\n"
    "too.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: FittedBox(
        child: Text(
          "This is some very very very large text that is too big to fit a regular screen\n"
          "in a single line."),
```

```

    ),
  );
}
}

```

This is some very very very large text that is too big to fit a regular screen in a single line.

17 18 19 20 21 22 23

```

Center(
  child: FittedBox(
    child: Text('...'));

```

However, what happens if FittedBox is inside of Center, but the Text is too large to fit the screen?

FittedBox will try to size itself to the Text, but it cannot be bigger than the screen. It will then assume the screen size, and resize the Text so that it fits the screen too.

```

class Example21 extends Example {
  final String code = "Center(\n"
    "  child: Text('...'))";

  final String explanation = "If, however, we remove the FittedBox, "
    "the Text will get its maximum width from the screen, "
    "and will break the line so that it fits the screen.";

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        "This is some very very very large text that is too big to fit a regular screen in a single line."),
    );
  }
}

```


This is some very very very large text that is too big to fit a regular screen in a single line.

18 19 20 21 22 23 24

```
Center(  
  child: Text('...'));
```

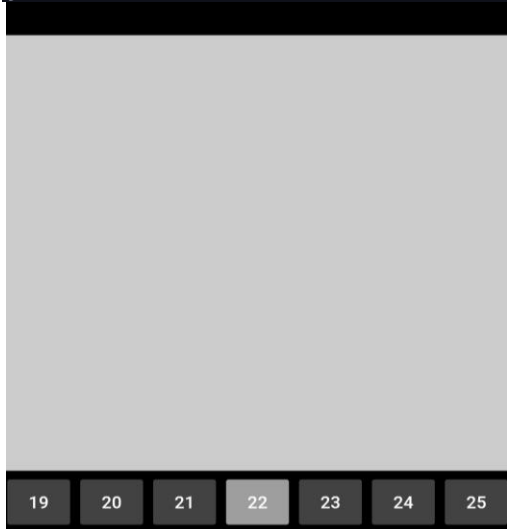
If, however, we remove the FittedBox, the Text will get its maximum width from the screen, and will break the line so that it fits the screen.

```
class Example22 extends Example {  
  final String code = "FittedBox(\n"  
    " child: Container(\n"  
    " height: 20.0, width: double.infinity));";  
  
  final String explanation =  
    "Note FittedBox can only scale a widget that is BOUNDED (has non infinite width and  
height)."  
    "\n\n"  
    "Otherwise, it won't render anything, and you will get an error in the console.";  
  
  @override  
  Widget build(BuildContext context) {  
    return FittedBox(  
      child: Container(  
        height: 20.0,  
        width: double.infinity,  
        color: Colors.red,
```

```

    ),
  );
}
}

```



```

FittedBox(
  child: Container(
    height: 20.0, width: double.infinity));

```

*Note FittedBox can only scale a widget that is **BOUNDED** (has non infinite width and height).*

Otherwise, it won't render anything, and you will get an error in the console.

```

class Example23 extends Example {
  final String code = "Row(children:[\n"
    "  Container(color: red, child: Text('Hello!'))\n"
    "  Container(color: green, child: Text('Goodbye!'))]\n");

  final String explanation =
    "The screen forces the Row to be exactly the same size of the screen."
    "\n\n"
    "Just like an UnconstrainedBox, the Row won't impose any constraints to its children, "
    "and will instead let them have any size they want."
    "\n\n"
    "The Row will then put them side by side, and any extra space will remain empty.";

  @override
  Widget build(BuildContext context) {
    return Row(
      children: [

```

```

        Container(color: red, child: Text("Hello!", style: big)),
        Container(color: green, child: Text("Goodbye!", style: big)),
    ],
);
}
}

```



Hello!Goodbye!

20 21 22 23 24 25 26

```

Row(children:[
  Container(color: red, child: Text('Hello!'))
  Container(color: green, child: Text('Goodbye!'))])

```

The screen forces the Row to be exactly the same size of the screen.

Just like an UnconstrainedBox, the Row won't impose any constraints to its children, and will instead let them have any size they want.

The Row will then put them side by side, and any extra space will remain empty.

```

class Example24 extends Example {
  final String code = "Row(children:[\n"
    "  Container(color: red, child: Text('...'))\n"
    "  Container(color: green, child: Text('Goodbye!'))]\n";

  final String explanation =
    "Since the Row won't impose any constraints to its children, "\n"
    "it's quite possible that the children will be too big to fit the available Row width."
    "\n\n"
    "In this case, just like an UnconstrainedBox, the Row will display the \"overflow\n"
    "warning\".";

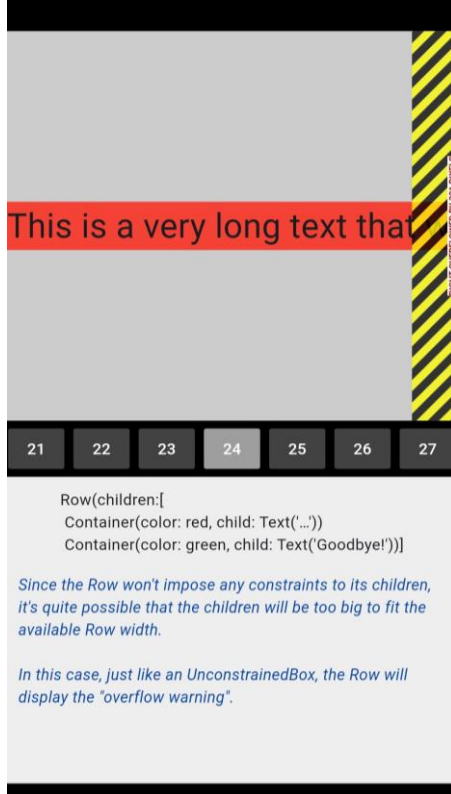
  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        Container(
          color: red,

```

```

        child: Text("This is a very long text that won't fit the line.",
            style: big)),
        Container(color: green, child: Text("Goodbye!", style: big)),
    ],
);
}
}

```



```

class Example25 extends Example {
  final String code = "Row(children:[\n"
    "  Expanded(\n"
    "    child: Container(color: red, child: Text('...')))\n"
    "    Container(color: green, child: Text('Goodbye!'))]);"

  final String explanation =
    "When a Row child is wrapped in an Expanded widget, the Row will not let this child"
    "define its own width anymore."
    "\n\n"
    "Instead, it will define the Expanded width according to the other children, and only"
    "then the Expanded widget will force the original child to have the Expanded's width."
    "\n\n"
    "In other words, once you use Expanded, the original child's width becomes irrelevant,"
    "and will be ignored.";

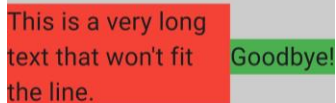
  @override
  Widget build(BuildContext context) {

```

```

return Row(
  children: [
    Expanded(
      child: Center(
        child: Container(
          color: red,
          child: Text("This is a very long text that won't fit the line.",
            style: big)),
      ),
    ),
    Container(color: green, child: Text("Goodbye!", style: big)),
  ],
);
}
}

```



This is a very long text that won't fit the line. Goodbye!

22 23 24 25 26 27 28

```

Row(children:[
  Expanded(
    child: Container(color: red, child: Text('...'))
  ),
  Container(color: green, child: Text('Goodbye!'))
])

```

When a Row child is wrapped in an Expanded widget, the Row will not let this child define its own width anymore.

Instead, it will define the Expanded width according to the other children, and only then the Expanded widget will force the original child to have the Expanded's width.

In other words, once you use Expanded, the original child's

```

class Example26 extends Example {
  final String code = "Row(children:[\n"
    "  Expanded(\n"
    "    child: Container(color: red, child: Text('...')))\n"
    "  Expanded(\n"
    "    child: Container(color: green, child: Text('Goodbye!')))]";

  final String explanation =
    "If all Row children are wrapped in Expanded widgets, each Expanded will have a size\n"
    "proportional to its flex parameter, "\n"
    "and only then each Expanded widget will force their child to have the Expanded's\n"
    "width."
    "\n\n"
    "In other words, the Expanded ignores their children preferred width.";

  @override
  Widget build(BuildContext context) {
    return Row(

```

```

children: [
  Expanded(
    child: Container(
      color: red,
      child: Text("This is a very long text that won't fit the line.",
        style: big))),
  Expanded(
    child:
      Container(color: green, child: Text("Goodbye!", style: big))),
],
);
}
}

```

This is a very
long text that
won't fit the
line.

Goodbye!

23 24 25 26 27 28 29

```

Row(children:[
  Expanded(
    child: Container(color: red, child: Text('...'))
  Expanded(
    child: Container(color: green, child: Text('Goodbye!'))])

```

If all Row children are wrapped in Expanded widgets, each Expanded will have a size proportional to its flex parameter, and only then each Expanded widget will force their child to have the Expanded's width.

In other words, the Expanded ignores their children preferred width.

```

class Example27 extends Example {
  final String code = "Row(children:[\n"
    " Flexible(\n"
    " child: Container(color: red, child: Text('...')))\n"
    " Flexible(\n"
    " child: Container(color: green, child: Text('Goodbye!')))]";

  final String explanation =
    "The only difference if you use Flexible instead of Expanded, "
    "is that Flexible will let its child be SMALLER than the Flexible width, "
    "while Expanded forces its child to have the same width of the Expanded."
    "\n\n"
    "But both Expanded and Flexible will ignore their children width when sizing themselves."
    "\n\n"
    "Note, this means it's IMPOSSIBLE to expand Row children proportionally to their sizes."
    "

```

"The Row will either use the exact child's with, or ignore it completely when you use Expanded or Flexible.";

```
@override
Widget build(BuildContext context) {
  return Row(
    children: [
      Flexible(
        child: Container(
          color: red,
          child: Text("This is a very long text that won't fit the line.",
            style: big))),
      Flexible(
        child:
          Container(color: green, child: Text("Goodbye!", style: big))),
    ],
  );
}
```

This is a very
long text that
won't fit the
line.

Goodbye!

23

24

25

26

27

28

29

```
Row(children:[  
  Flexible(  
    child: Container(color: red, child: Text('...'))  
  ),  
  Flexible(  
    child: Container(color: green, child: Text('Goodbye!'))  
  )  
])
```

The only difference if you use Flexible instead of Expanded, is that Flexible will let its child be SMALLER than the Flexible width, while Expanded forces its child to have the same width of the Expanded.

But both Expanded and Flexible will ignore their children width when sizing themselves.


```

class Example28 extends Example {
  final String code = "Scaffold(\n"
    "  body: Container(color: blue,\n"
    "    child: Column(\n"
    "      children: [\n"
    "        Text('Hello!'),\n"
    "        Text('Goodbye!')])))";

  final String explanation =
    "The screen forces the Scaffold to be exactly the same size of the screen."
    "\n\n"
    "So the Scaffold fills the screen."
    "\n\n"
    "The Scaffold tells the Container it can be any size it wants, but not bigger than the"
    "screen."
    "\n\n"
    "Note: When a widget tells its child it can be smaller than a certain size, "
    "we say the widget supplies \"loose\" constraints to its child. More on that later.";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        color: blue,
        child: Column(
          children: [
            Text('Hello!'),
            Text('Goodbye!'),
          ],
        ),
      ),
    );
  }
}

```

Hello!
Goodbye!

23

24

25

26

27

28

29

```
Scaffold(  
  body: Container(color: blue,  
  child: Column(  
    children: [  
      Text('Hello!'),  
      Text('Goodbye!')] ]))
```

The screen forces the Scaffold to be exactly the same size of the screen.

So the Scaffold fills the screen.

The Scaffold tells the Container it can be any size it wants,

```

class Example29 extends Example {
  final String code = "Scaffold(\n"
    "  body: Container(color: blue,\n"
    "    child: SizedBox.expand(\n"
    "      child: Column(\n"
    "        children: [\n"
    "          Text('Hello!'),\n"
    "          Text('Goodbye!')] ]))));";

  final String explanation =
    "If we want the Scaffold's child to be exactly the same size as the Scaffold itself, "\n"
    "we can wrap its child into a SizedBox.expand."
    "\n\n"
    "Note: When a widget tells its child it must be of a certain size, "\n"
    "we say the widget supplies \"tight\" constraints to its child. More on that later.";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: SizedBox.expand(
        child: Container(
          color: blue,
          child: Column(
            children: [
              Text('Hello!'),
              Text('Goodbye!'),
            ],
          ),
        ),
      ),
    );
  }
}

```

