Flutter Framework

COURSE 6 (REV 4)

GAVRILUT DRAGOS

Agenda

- Stateful widgets
- Timers and stateful widgets
- Custom painter

Stateful widget

Stateful widget

Regular applications have data members / variable that often change as a result of the interaction with the user. Some of them have an UI reflection (meaning that the change of that specific variable should be reflected in a change in UI).

For example, → if we have a timer that is printed on the screen, whenever the timer changes, the new time has to be reprinted to reflect the change.

In Flutter, this behavior is called a stateful widget ➔ or to simplify a widget that has to be redrawn when its internal state changes (e.g. something else to display).

Stateful widget

A stateful application has the following format:

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(…);
}
```

We still need a main function to start the program (the actual app)

Stateful widget

A stateful application has the following format:

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(…);
}
```

However, the app extends StatefulWidget (that allows the App to create a State object)

Stateful widget

A stateful application has the following format:

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState()
}
class MyAppState extends State<MyApp> {
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(…);
}
```

The state object is created via the **createState** method (that has to be overridden)

Stateful widget

A stateful application has the following format:

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState
  // data members
  @override
  Widget build(BuildContext context) => MaterialApp(…);
}
```

The state object (in our case MyAppState) has a build method that will be called whenever the new Widget / view has to be updated/created.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  // some data member

  Widget GetBody() => …

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("<Title>")),
                        body: GetBody()));
  }
}
```

This is where the data (the state values) are defined. They are usually regular variables / data members from this class.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  // some data member

  Widget GetBody() => …

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("<Title>")),
                   body: GetBody()));
  }
}
```

This is where the widgets and the interaction that leads to a change in the state happens.

Stateful widget

Let’s build a very simple stateful app:

1. Has a button, on the center of the screen that says “Random Values”
2. Whenever that button is being pressed, a new random value will be generated and the text of that button will be change to reflect that value.
3. We will create an internal data member (called “value”) to store the randomly generated value
4. The initial value of that field will be -1 and the generated values will be between 0 and 100

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  int value = -1;
```

First we need to create that state value

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  int value = -1;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("Test")),
                   body: GetBody()));
  }
```

Then we need to add the **build** method

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  int value = -1;

  @override
  Widget build(BuildContext context) => …

  Widget GetBody() {
    return Center(
        child: ElevatedButton(
            child: Text(value == -1 ? "Random" : "Value: ${value}"),
            onPressed: onButtonPressed));
  }

}
```

After this we create the GetBody method that creates an ElevatedButton that has a text form out of either “Random” or “Value:…”

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  int value = -1;

  @override
  Widget build(

  Widget GetBod
    return Cent
        child: ElevatedButton(
          child: Text(value == -1 ? "Random" : "Value: ${value}"),
          onPressed: onButtonPressed));
  }

}
```

We also define a callback called **onButtonPressed** that should be called whenever that button is being pressed.

Stateful widget

The “MyAppState” usually looks like this:

```
class MyAppState extends State<MyApp> {
  int value = -1;

  @override
  Widget build(BuildContext context) => …

  Widget GetBody() => …

  void onButtonPressed() {
    setState(() {
      value = Random().nextInt(100);
    });
  }
}
```

Then we need to add the **build** method

Stateful widget

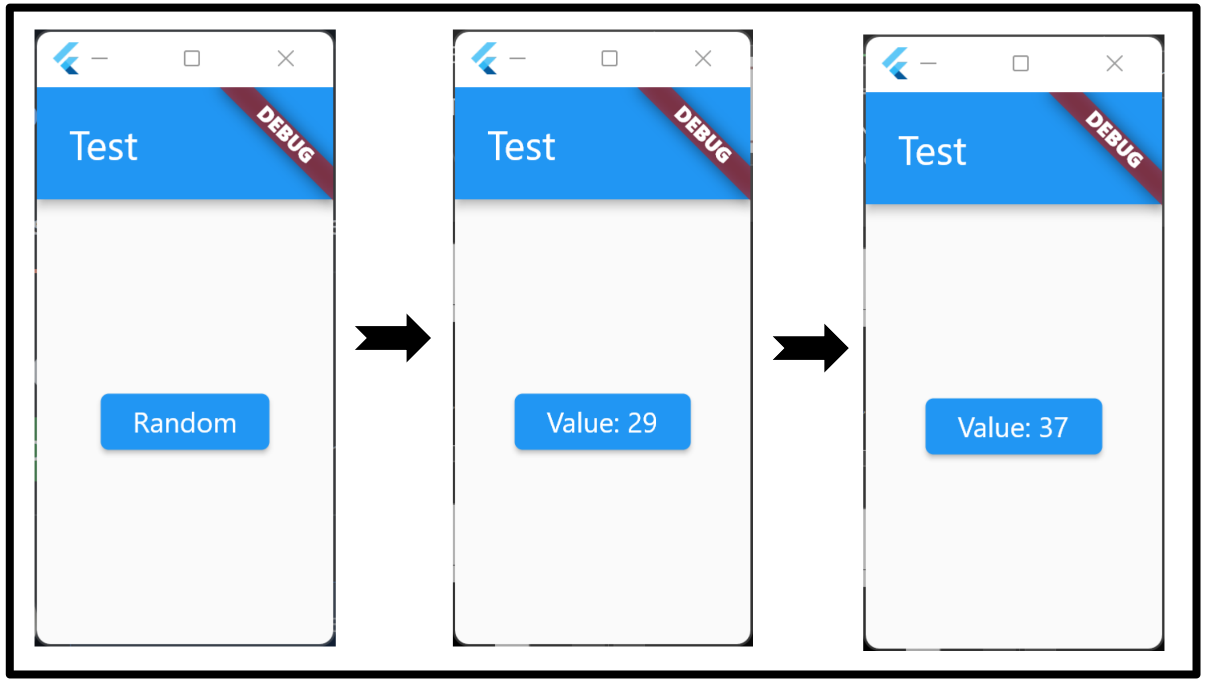
The “MyAppState” usually looks like this:

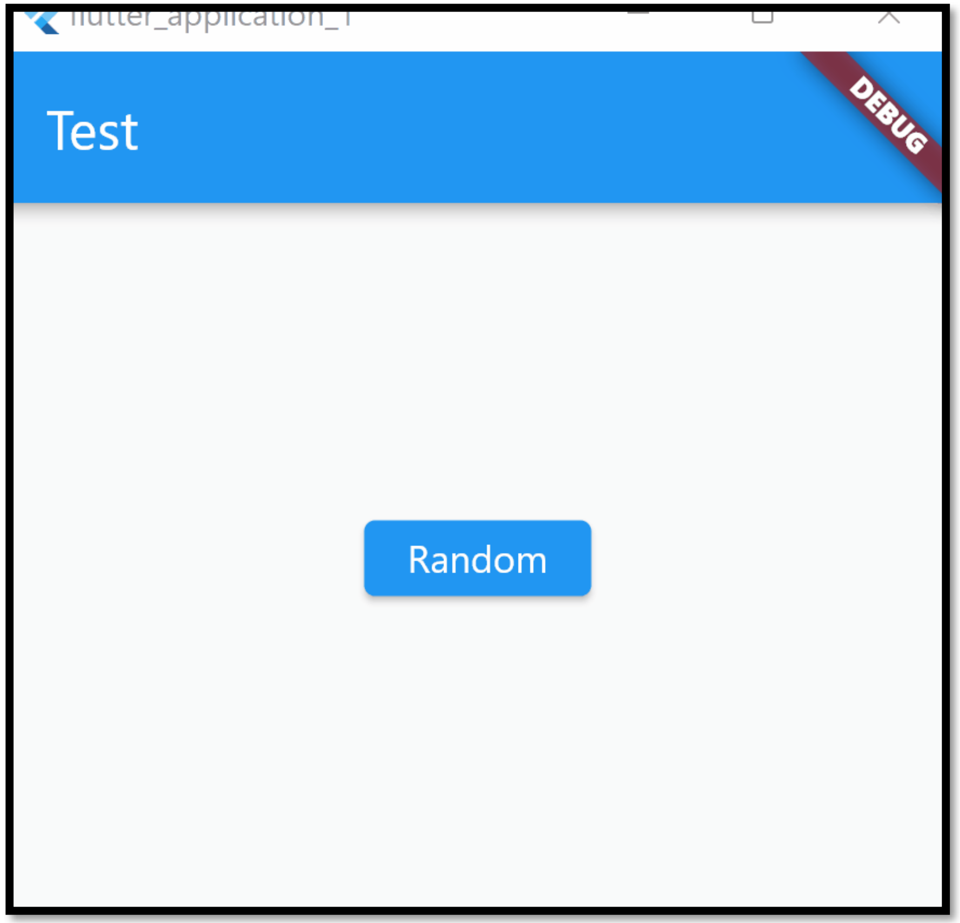
```
class MyAppState extends State<MyApp> {
  int value = -1;

  @override
  Widget build(BuildContext context) => …

  Widget GetBody() => …

  void onButtonPressed() {
    setState(() {
      value = Random().ne
    });
  }
}
```

It’s very important to change the value of the state within a **setState**(…) call. This makes sure that .build method is called after the value is changed, and as a result the text from the button will be changed as well.

Stateful widget

Upon execution, the app should behave like in the following images:





Using timers with stateful widgeds

Stateful widget & Timers

A stateful widget can be used with a Timer object to create an animation logic (a loop that will be called at a specific period of time).

A very simple example will increase a value and print it every “x” seconds

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
…
}
```

Let’s consider this a template for a very simple program.

Stateful widget & Timers

A stateful widget can be used with a Timer object to create an animation logic (a loop that will be called at a specific period of time).

A very simple example will increase a value and print it every “x” seconds

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

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

class                        t {
  @ove
  State<MyApp> cre  eState() => MyAppState();
}
class MyAppState extends State<MyApp> {
…
}
```

This means that in the next slides we will focus on the **MyAppState** class

Let’s consider this a template for a very simple program.

Stateful widget & Timers

```dart
class MyAppState extends State<MyApp> {
  int value = 0;









}
```

First, we need to create a value (the timer value) that will be increase every seconds.

Stateful widget & Timers

The app consists in a **Center** layout that contains a **Text** object with the value ("Value: ${value}"), practically showing the value of **value** data member

```
  int value = 0;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("Test")),
            body: Center(child: Text("Value: ${value}"))));

}
```

Next **build** method has to be created.

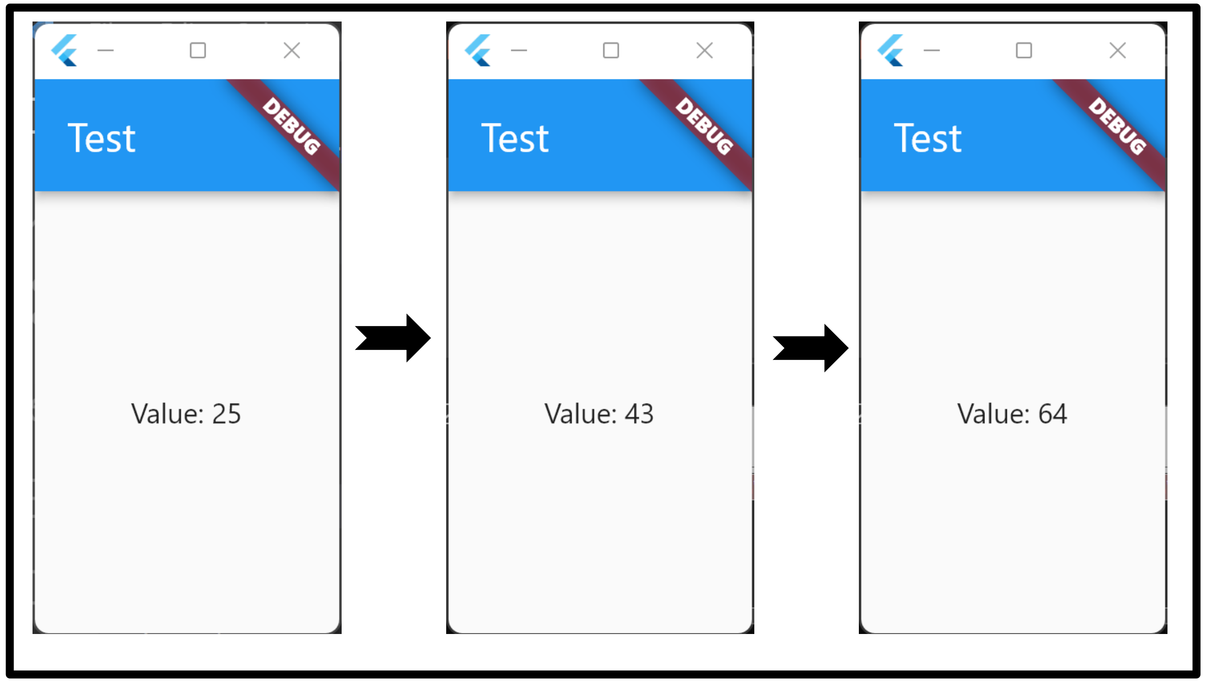
Stateful widget & Timers

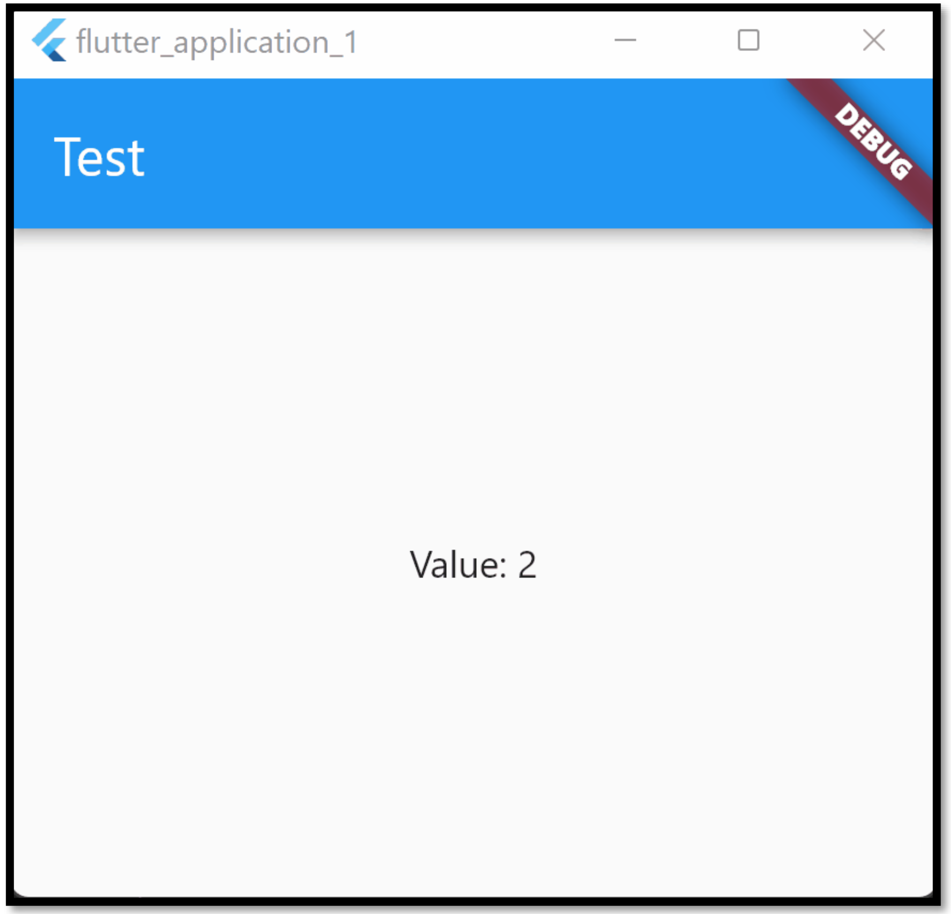
```
class MyAppState extends State<MyApp> {
  int value = 0;

  @override
  Widget build(BuildContext context) {…}

  MyAppState() {
    Timer.periodic(
        Duration(seconds: 1),
        (t) => setState(() {
              value++;
            }));

}
```

Finally, the constructor creates a Timer object that will be triggered every one second, and uses **setState** to increment the value.

Stateful widget & Timers

Upon execution, the app should behave like in the following images:





Custom paint

Custom paint

Every UX has to have a component that can be use for custom drawing. In Flutter this is materialized through the CustomPaint widget that is usually organized in the following way:

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text(“<Title>")),
            body: CustomPaint(painter: <painter>,
                              child: Container())));
  }
}
```

Custom paint

Every UX has to have a component that can be use for custom drawing. In Flutter this is materialized through the CustomPaint widget that is usually organized in the following way:

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("<Title>")),
            body: CustomPaint(painter: <painter>,
                              child: Container())));
  }
}
```

A *painter* parameter must be provided (this will include the actual custom drawing).

Custom paint

Every UX has to have a component that can be use for custom drawing. In Flutter this is materialized through the CustomPaint widget that is usually organized in the following way:

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text(“<Title>")),
            body: CustomPaint(painter: <painter>,
                                 child: Container()));
  }
}
```

A *child* parameter will reflect the widget over which the drawing will happen.

Custom paint

The painter object has to be derived from CustomPainter (not to be confused with CustomPaint widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) { … }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) { … }
}
```

Custom paint

The painter object has to be derived from CustomPainter (not to be confused with CustomPaint widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) { … }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) { … }
}
```

This is the actual method where the custom drawing will happen

Custom paint

The painter object has to be derived from CustomPainter (not to be confused with CustomPaint widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) { … }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) { … }
}
```

This is called whenever a new instance of the class is created. The idea is to check if a repaint should be performed or not.

Custom paint

The painter object has to be derived from CustomPainter (not to be confused with CustomPaint widget) where two methods have to be overridden:

```
class <my_painter> extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) { … }
}
```

the .paint method has two parameters:

1. A canvas object that will be used for the actual drawing
2. The size of the space where the drawing will occur

Custom paint

A canvas is a generic interface for drawing. It is similar to the capabilities of a SVG format and it includes methods for:

- Primitives (drawing a line, a rectangle, a circle, etc)
- Paths
- Clipping support
- Scale / skew / various transformations
- Image manipulations

A complete list of all these methods can be found here: https://api.flutter.dev/flutter/dart-ui/Canvas-class.html

Custom paint

Let’s put all of these together and build an app that uses custom paint to draw a smiling face ☺
First → we will use the previous template for the basic app, and we will focus on the **MyAppState** class.

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

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

class MyApp extends StatefulWidget {
  @override
  State<MyApp> createState() => MyAppState();
}
class MyAppState extends State<MyApp> {
…
}
```

Custom paint

The MyAppState class creates the widget hierarchy. Its body contains of a CustomPaint widget that uses MyPainter class to draw a smiley face over a Container object. This means that the drawing will happen over the entire app space.

```
class MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("Test")),
            body: CustomPaint(painter: MyPainter(),
                              child: Container())));
  }
}
```

Custom paint

The MyPainter class overrides:

1. *paint* method
2. shouldRepaint method

```
class MyPainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
   …
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) { return false;}
}
```

This mean that the entire drawing logic should happen in the .paint method:

Custom paint

Let’s start by creating a method (DrawSmileyFace) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

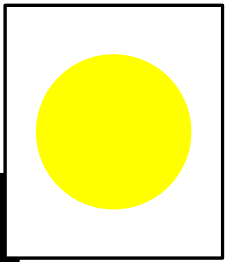
```
class MyPainter extends CustomPainter {
  void DrawSmileyFace(Offset pos, Canvas canvas) { … }

  @override
  void paint(Canvas canvas, Size size) {
    Offset center = Offset(size.width / 2, size.height / 2);
    DrawSmileyFace(center, canvas);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) { return false;}
}
```

Custom paint

Let’s start by creating a method (DrawSmileyFace) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

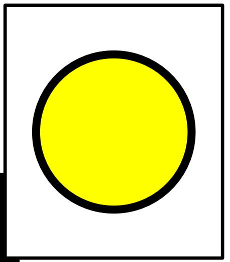
```
class MyPainter extends CustomPainter {
  void DrawSmileyFace(Offset pos, Canvas canvas) {
    var paint = Paint()
                  ..color = Colors.yellow
                  ..style = PaintingStyle.fill;
    canvas.drawCircle(pos, 100, paint);




   }
}
```

First, we will draw a yellow circle with the ray of 100. The paint object describe how painting has to be performed.



Custom paint

Let's start by creating a method (DrawSmileyFace) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.



```
class MyPainter extends CustomPainter {
  void DrawSmileyFace(Offset pos, Canvas canvas) {
    var paint = Paint()…
    canvas.drawCircle(pos, 100, paint);
    paint
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5
      ..style = PaintingStyle.stroke;
    canvas.drawCircle(pos, 100, paint);
   }
}
```

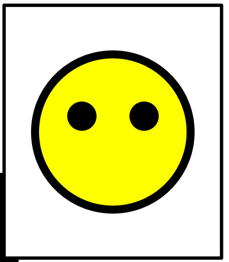
Secondly, we will draw a black line circle of width 5 around the existing yellow circle.

Custom paint

Let’s start by creating a method (DrawSmileyFace) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

```
class MyPainter extends CustomPainter {
  void DrawSmileyFace(Offset pos, Canvas canva
    …
    paint
      ..color = Colors.black
      ..style = PaintingStyle.fill;
    canvas.drawCircle(Offset(pos.dx - 33, pos.dy - 10), 20, paint);
    canvas.drawCircle(Offset(pos.dx + 33, pos.dy - 10), 20, paint);


   }
}
```

Similarly, we draw the eyes

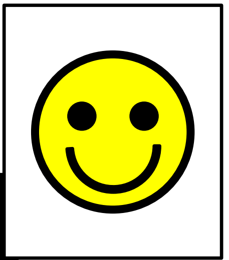


Custom paint

Let's start by creating a method (DrawSmileyFace) that will receive a coordinate at the screen and will draw a smiley face there. We will draw a smiley face at the center of the screen.

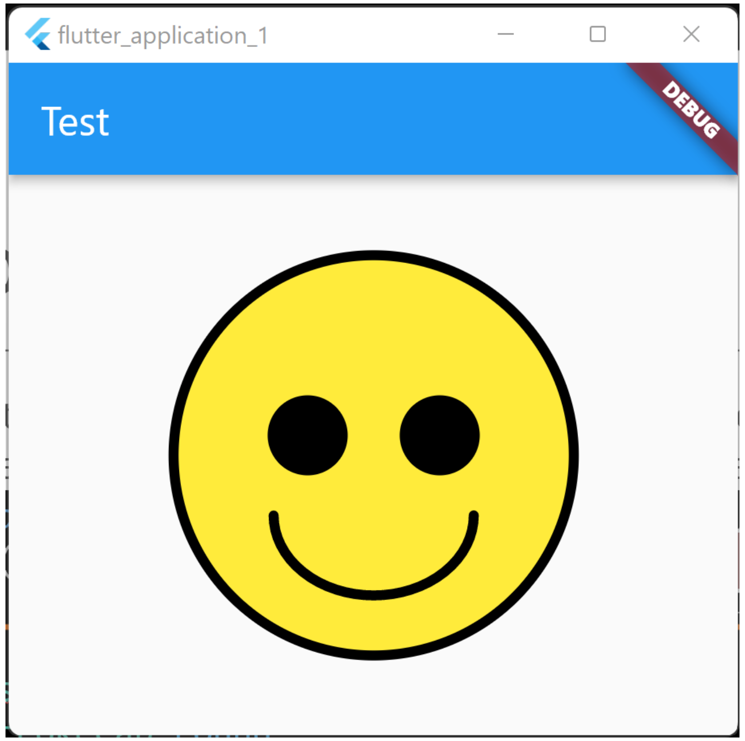
```
class MyPainter extends CustomPainter {
  void DrawSmileyFace(Offset pos, Canvas canva
    …
    paint
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5
      ..style = PaintingStyle.stroke;
    canvas.drawArc(Rect.fromCenter(center: Offset(pos.dx, pos.dy + 30),
                  width: 100, height: 80), 3.14, -3.14,false,paint);
    }
}
```

Finally, we draw the mouth



Custom paint

Upon execution the image should look like the following one. Keep in mind the changed the parameters / distances can be used to change the way this image looks like.



Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {
  double x, y;
  MyPainter(this.x, this.y);
  void DrawSmileyFace(Offset pos, Canvas canvas) {…}

  @override
  void paint(Canvas canvas, Size size) {
    DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => true;
}
```

Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {
  double x, y;
  MyPainter(this.x, this.y);
  void DrawSmileyFace(Offset pos, Canvas canvas) {…}

  @override
  void paint(Canvas canvas, Size size) {
    DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => true;
}
```

MyPainter will now receive the (x,y) coordinates of the Smiley object as percentages (0 .. 1) [0 to 100%]

Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {
  dou
  MyP
  voi
```

The .paint method just calls **DrawSmileyFace** with coordinates based on “X” percentage and “y” percentage.

```
  @override
  void paint(Canvas canvas, Size size) {
    DrawSmileyFace(Offset(size.width * x, size.height * y), canvas);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => true;
}
```

Custom paint

What if we want to create a simple animation → for example to move a smiley face within the screen. First, we will change the **MyPainter** class in the following way:

```
class MyPainter extends CustomPainter {
  double x, y;
  MyPainter(this.x, this.y);
  void DrawSmileyFace(Offset pos, Canvas canvas) {…}

  @override
                                                              , canvas);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => true;
}
```

Finally, the .shouldRepaint method will return true to tell the app that it should redraw whenever a new MyPainter object is created.

Custom paint

Let’s see how the state class should look like:

```
class MyAppState extends State<MyApp> {
  double x = 0.3, y = 0.4;
  double addX = 0.05, addY = 0.05;









}
```

We define a start-up position (30% x, 40% y) and addX /addY with values to be added on X/Y axes.

Custom paint

Let's see how the

The .build method is pretty standard (it create a custom paint that uses a MyPainter with a customize X and Y

```
class MyAppSt
  double x = 0.3, y = 0.4;
  double addX = 0.05, addY = 0.05;
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(title: Text("Test")),
            body: CustomPaint(painter: MyPainter(x, y), child: Container())));
  }




}
```

Custom paint

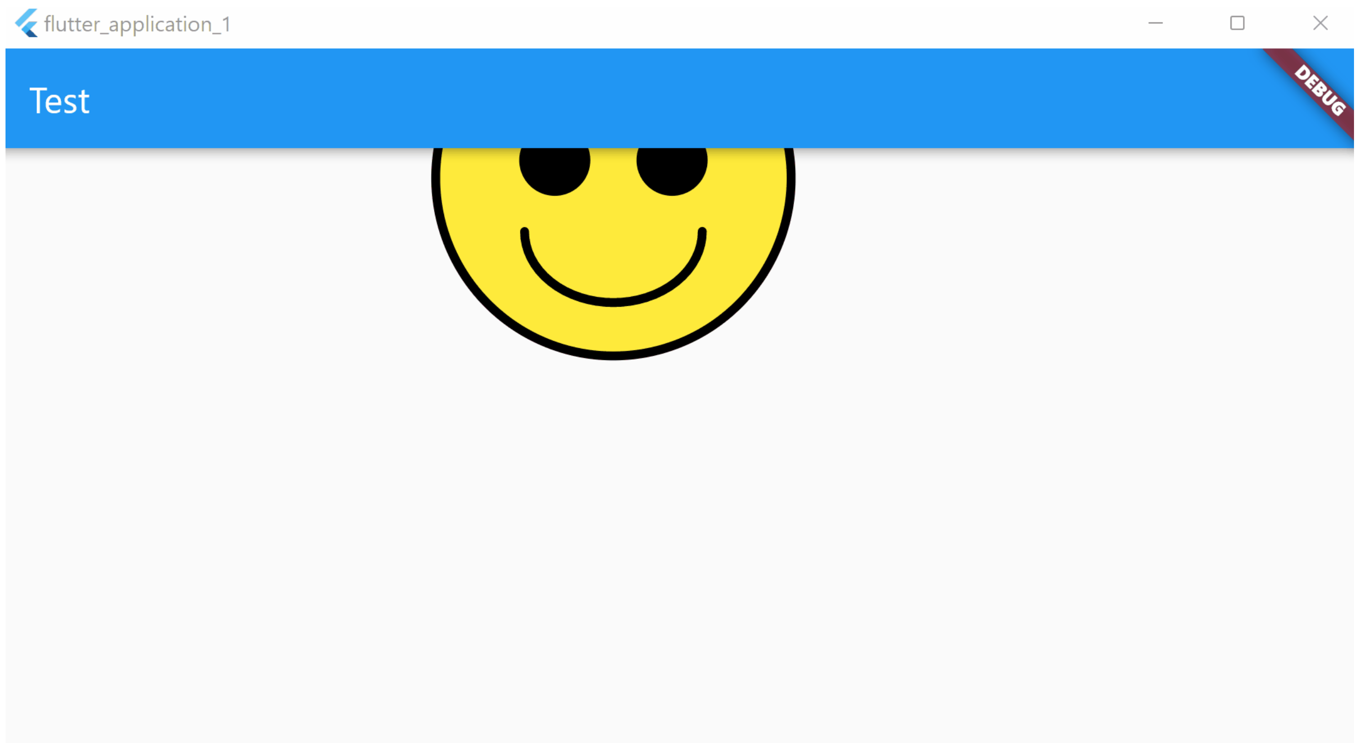
Let’s see how the state class should look like:

```
class M
  doubl
  doubl
  @over
  Widget build(BuildContext contex  { … }
  MyAppState() {
    Timer.periodic( Duration(milliseconds: 100), (t) => setState(() {
              x += addX;
              y += addY;
              if ((x >= 1) || (x <= 0)) addX = -addX;
              if ((y >= 1) || (y <= 0)) addY = -addY;
            }));
  }
}
```

The constructor uses a timer object set up to be triggered every 100 milliseconds that changes the X and Y with a very simple algorithm.

Custom paint

Upon execution the image should look like the following one with a smiley face moving.



Q & A