



## Developing the “Flying balls” application

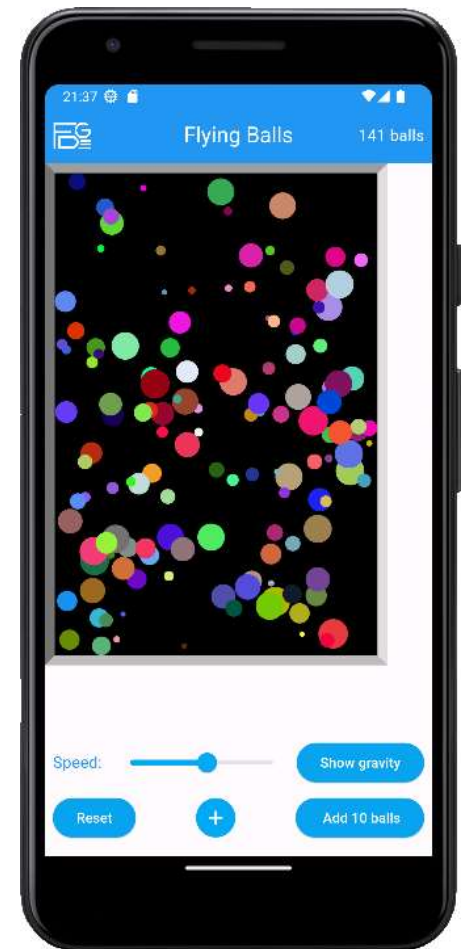
- Control the visibility of Widgets
- React on user gestures
- Use the Stack and Positioned widgets
- Define a periodic Timer
- Create random numbers
- Use object oriented principles when defining the Ball class

# Our Goal

After Start:



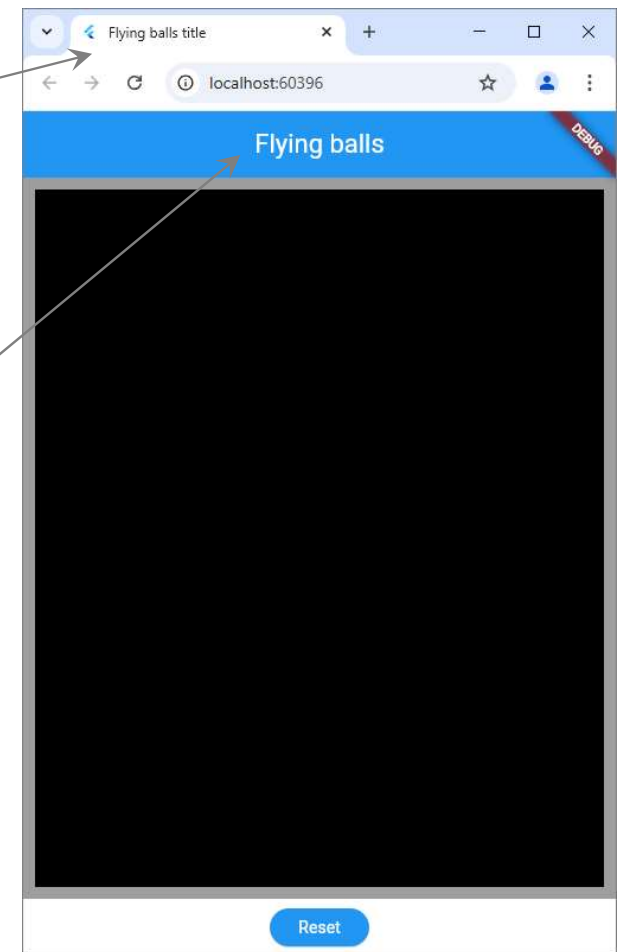
In “Expert Mode”  
after adding balls &  
with a shrunk box:





# Defining colors in the Theme

```
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flying balls title',  
      theme: ThemeData(  
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),  
        scaffoldBackgroundColor: Colors.white,  
        appBarTheme: const AppBarTheme(  
          foregroundColor: Colors.white,  
          backgroundColor: Colors.blue,  
        ), // AppBarTheme  
        elevatedButtonTheme: ElevatedButtonThemeData(  
          style: ElevatedButton.styleFrom(  
            backgroundColor: Colors.blue, // Background color  
            foregroundColor: Colors.white, // Text color  
          ),  
        ), // ElevatedButtonThemeData  
        useMaterial3: true,  
      ), // ThemeData  
      home: const MyHomePage(title: 'Flying balls'),  
    ); // MaterialApp  
  }  
}
```

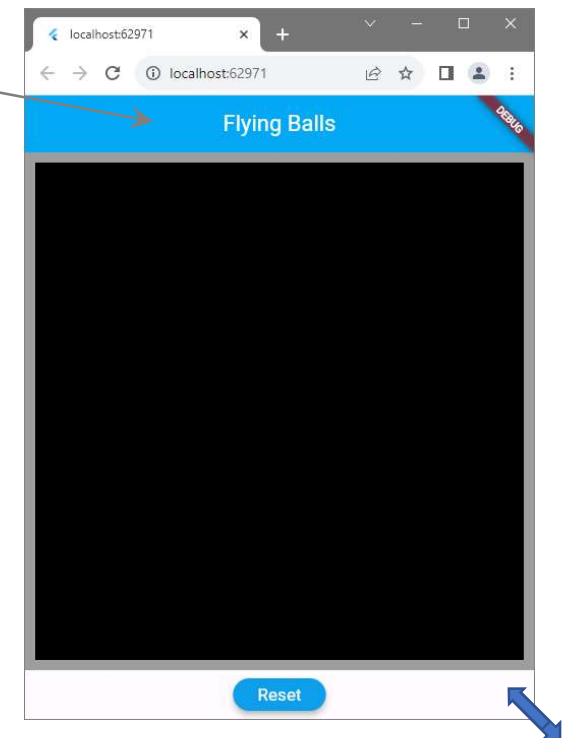


# First layout of the Scaffold



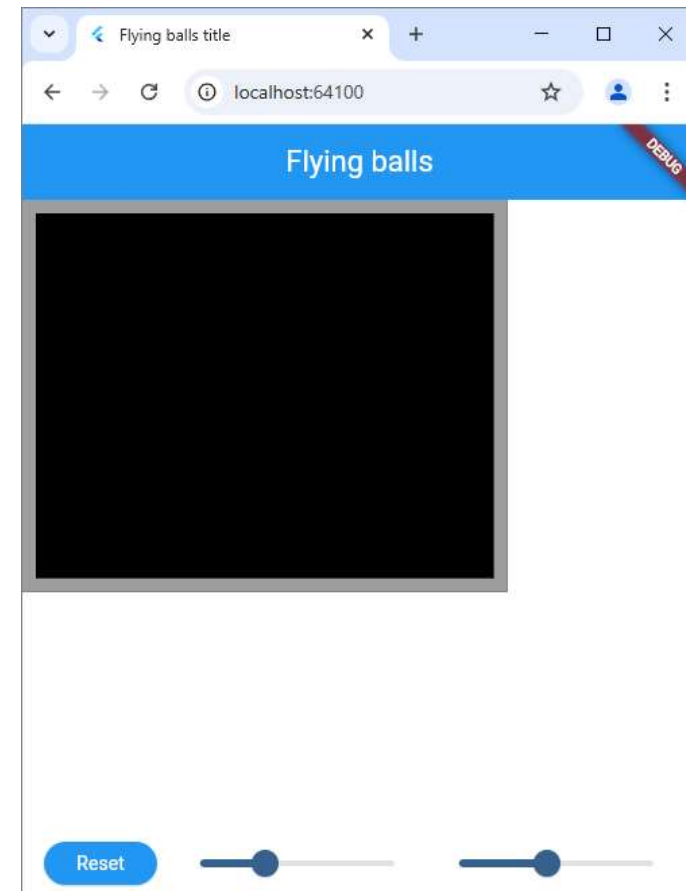
```
class _MyHomePageState extends State<MyHomePage> {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
        centerTitle: true,  
      ), // AppBar  
      body: Column(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: <Widget>[  
          Expanded(  
            child: Container(  
              decoration: BoxDecoration(  
                color: Colors.black,  
                border: Border.all(color: Colors.grey.shade500, width: 10),  
              ), // BoxDecoration  
            ), // Container  
          ), // Expanded  
          Row(mainAxisAlignment: MainAxisAlignment.center, children: [  
            Padding(  
              padding: const EdgeInsets.all(8.0),  
              child:  
                ElevatedButton(onPressed: () {}, child: const Text("Reset")),  
            ), // Padding  
          ]), // Row  
        ], // <Widget>[]  
      ), // Column  
    ); // Scaffold  
  }  
}
```

Expanded widget:  
box fills all available space and  
follows browser window size  
when running in Chrome



# Exercise

Implement 2 Sliders which let the black box shrink or grow while it is fixed on top-left corner:



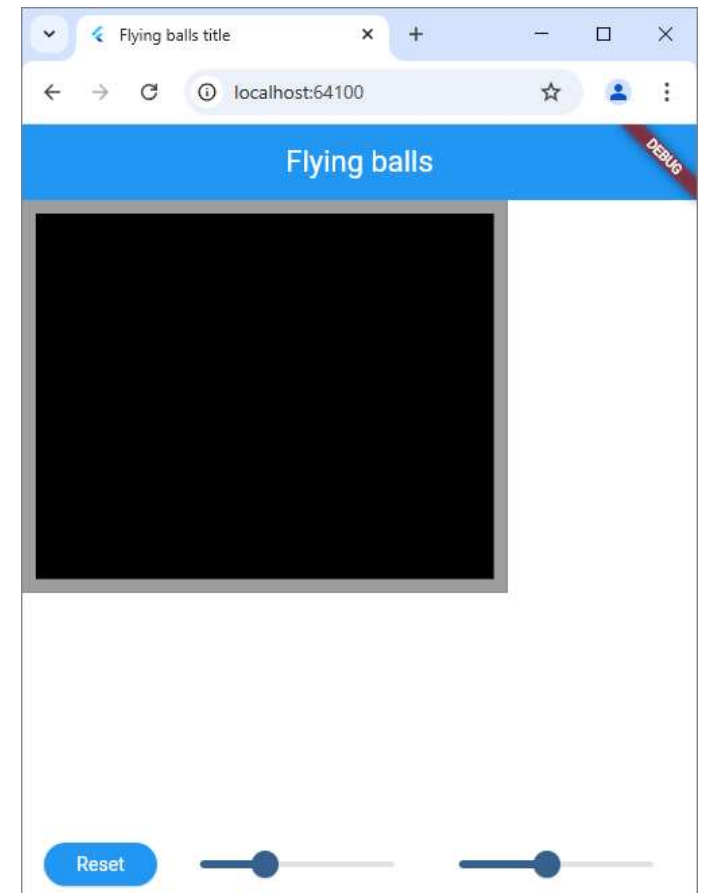
# Possible solution



```
class _MyHomePageState extends State<MyHomePage> {  
  double marginRight = 0;  
  double marginBottom = 0;  
  
  @override  
  Widget build(BuildContext context) {  
    double screenWidth = MediaQuery.of(context).size.width;  
    double screenHeight = MediaQuery.of(context).size.height;  
  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.title),  
        centerTitle: true,  
      ), // AppBar  
      body: Column(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: <Widget>[  
          Expanded(  
            child: Container(  
              margin: EdgeInsets.only(right: marginRight, bottom: marginBottom),  
              decoration: BoxDecoration(  
                color: Colors.black,  
                borderRadius: BorderRadius.circular(10),  
              ),  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

```
Slider(  
  min: 0,  
  max: screenWidth - 80,  
  value: marginRight,  
  onChanged: (value) {  
    setState(() {  
      marginRight = value;  
    });  
  },  
), // Slider
```

```
Slider(  
  min: 0,  
  max: screenHeight - 180,  
  value: marginBottom,  
  onChanged: (value) {  
    setState(() {  
      marginBottom = value;  
    });  
  },  
), // Slider
```



# Modify box size with GestureDetector



```
Expanded(  
  child: GestureDetector(  
    onPanUpdate: (details) {  
      //print(details.delta.dx);  
      setState(() {  
        marginRight -= details.delta.dx;  
        if (marginRight < 0) {  
          marginRight = 0;  
        }  
        if (marginRight > screenWidth - 80) {  
          marginRight = screenWidth - 80;  
        }  
        marginBottom -= details.delta.dy;  
        if (marginBottom < 0) {  
          marginBottom = 0;  
        }  
        if (marginBottom > screenHeight - 180) {  
          marginBottom = screenHeight - 180;  
        }  
      });  
    },  
    child: Container(  
      margin:  
        EdgeInsets.only(right: marginRight, bottom: marginBottom),  
      decoration: BoxDecoration(  
        color: Colors.white,  
        borderRadius: BorderRadius.all(Radius.circular(10)),  
        border: Border.all(color: Colors.black, width: 1),  
      ),  
    ),  
  ),  
),
```

```
(new) GestureDetector GestureDetector({  
  Key? key,  
  Widget? child,  
  void Function(TapDownDetails)? onTapDown,  
  void Function(TapUpDetails)? onTapUp,  
  void Function()? onTap,  
  void Function()? onTapCancel,  
  void Function()? onSecondaryTap,  
  void Function(TapDownDetails)? onSecondaryTapDown,  
  void Function(TapUpDetails)? onSecondaryTapUp,  
})
```

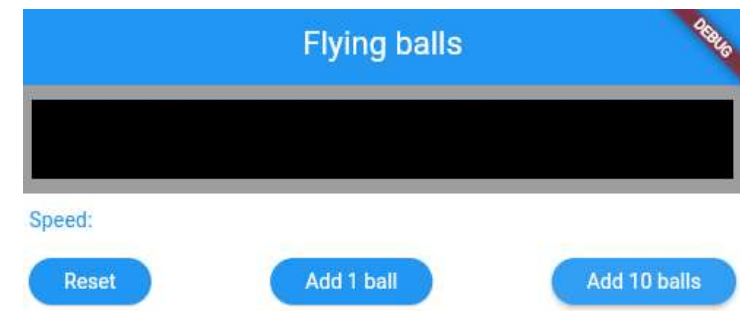
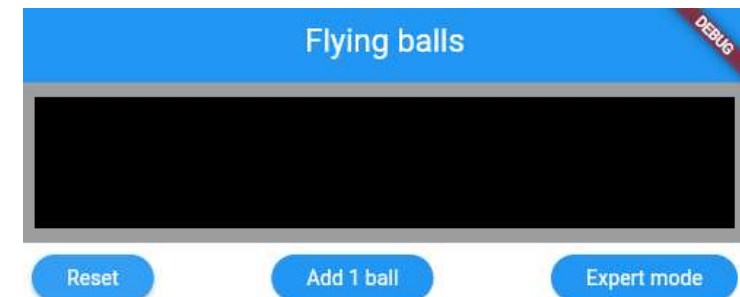
```
void Function(DragDownDetails)? onPanDown,  
void Function(DragStartDetails)? onPanStart,  
void Function(DragUpdateDetails)? onPanUpdate,  
void Function(DragEndDetails)? onPanEnd,  
void Function()? onPanCancel,  
void Function(ScaleStartDetails)? onScaleStart,  
void Function(ScaleUpdateDetails)? onScaleUpdate,  
void Function(ScaleEndDetails)? onScaleEnd,
```



# Implement ExpertMode



```
Padding(  
  padding: const EdgeInsets.all(8.0),  
  child: Row(  
    mainAxisAlignment: MainAxisAlignment.spaceBetween,  
    children: [  
      ElevatedButton(  
        onPressed: () {  
          setState(() {  
            expertMode = false;  
          });  
        },  
        child: const Text("Reset"), // ElevatedButton  
      ),  
      ElevatedButton(  
        onPressed: () {}, child: const Text("Add 1 ball")), // ElevatedButton  
      if (!expertMode)  
        ElevatedButton(  
          onPressed: () {  
            setState(() {  
              expertMode = !expertMode;  
            });  
          },  
          child: const Text("Expert mode"), // ElevatedButton  
        ),  
      if (expertMode)  
        ElevatedButton(  
          onPressed: () {}, child: const Text("Add 10 balls")), // ElevatedButton  
    ], // Row  
  ), // Padding  
)
```







# Text color in Theme (ChatGPT)

flutter text color in theme

In Flutter, to define **text color in a theme**, you typically customize the `ThemeData`'s `textTheme`. Here's how to set the default text colors app-wide using your theme:

```
MaterialApp(  
  theme: ThemeData(  
    textTheme: TextTheme(  
      bodyLarge: TextStyle(color: Colors.black),  
      bodyMedium: TextStyle(color: Colors.grey[800]),  
    ),  
  ),  
  home: MyHomePage(),  
)
```

When you want to use the theme's text style:

```
dart  
  
Text(  
  'Hello Flutter',  
  style: Theme.of(context).textTheme.bodyLarge,  
)
```

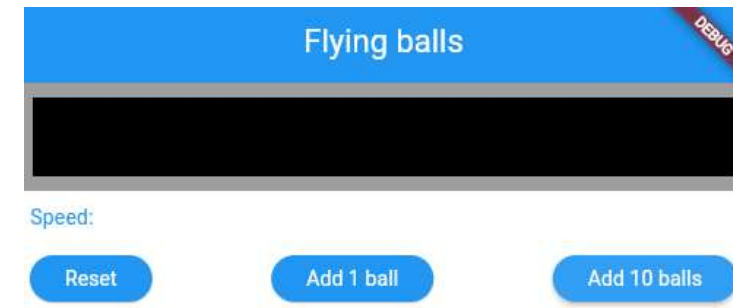
Using this for the text “Speed:” in Expert row



```
theme: ThemeData(  
  colorScheme: ColorScheme.fromSeed(seedColor: Colors.blue),  
  scaffoldBackgroundColor: Colors.white,  
> appBarTheme: const AppBarTheme( // AppBarTheme ...  
> elevatedButtonTheme: ElevatedButtonThemeData( // ElevatedButtonThemeData ...  
  textTheme: const TextTheme(  
    bodyMedium: TextStyle(color: Colors.blue),  
  ), // TextTheme  
  useMaterial3: true,  
) // ThemeData
```

```
TextTheme TextTheme({  
  TextStyle? displayLarge,  
  TextStyle? displayMedium,  
  TextStyle? displaySmall,  
  TextStyle? headlineLarge,  
  TextStyle? headlineMedium,  
  TextStyle? headlineSmall,  
  TextStyle? titleLarge,  
  TextStyle? titleMedium,  
  TextStyle? titleSmall,  
  TextStyle? bodyLarge,  
  TextStyle? bodyMedium,  
  TextStyle? bodySmall,
```

```
if (expertMode)  
  Padding(  
    padding: const EdgeInsets.all(8.0),  
    child: Row(  
      mainAxisAlignment: MainAxisAlignment.spaceBetween,  
      children: [  
        Text(  
          "Speed:",  
          style: Theme.of(context).textTheme.bodyMedium,  
        ) // Text  
      ]), // Row  
    ), // Padding
```

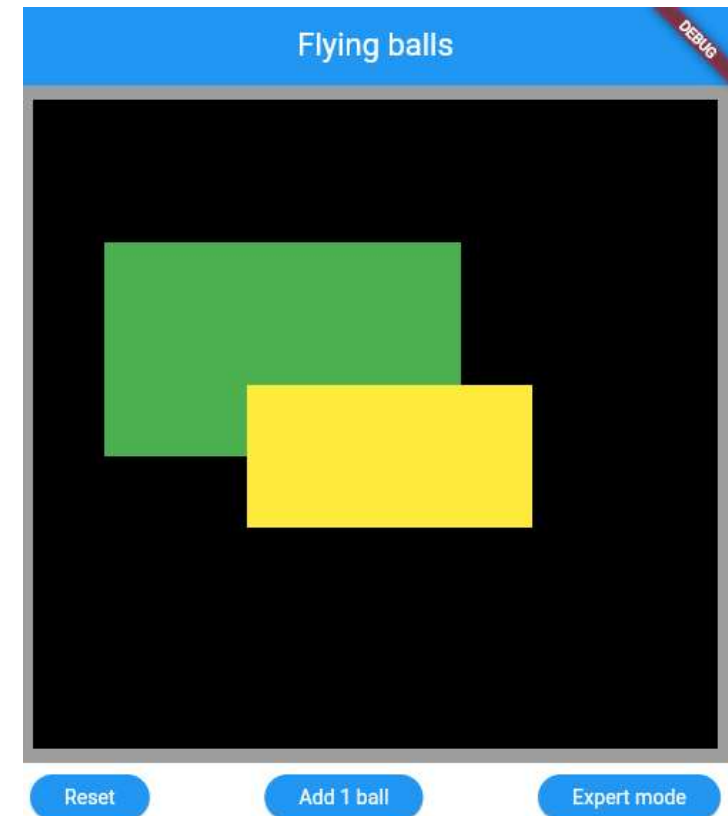


# Stack with two positioned containers in “the box”



```
child: Container(  
  margin:  
    EdgeInsets.only(right: marginRight, bottom: marginBottom),  
  decoration: BoxDecoration(  
    color: Colors.black,  
    border: Border.all(color: Colors.grey.shade500, width: 10),  
  ), // BoxDecoration  
  child: Stack(children: [  
    Positioned(  
      top: 100,  
      left: 50,  
      child: Container(  
        width: 250, height: 150, color: Colors.green), // Container // Positioned  
      ),  
    Positioned(  
      top: 200,  
      left: 150,  
      child: Container(  
        width: 200, height: 100, color: Colors.yellow), // Container // Positioned  
      ),  
  ]), // Stack  
), // Container
```

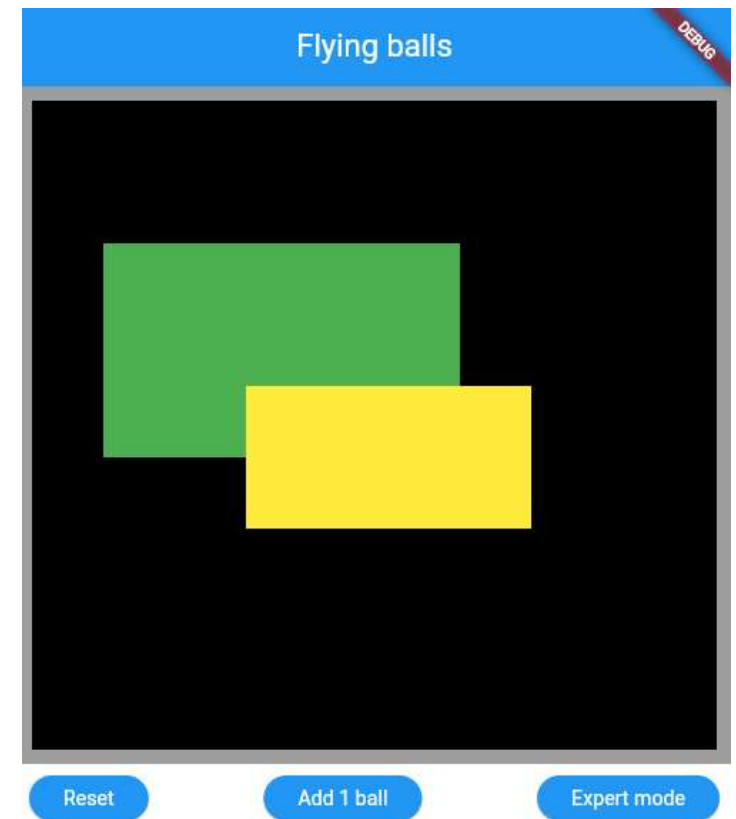
Remark: Last child is drawn above previous ones.



# Exercise



- 1) Implement a Gesture Detection so that the yellow container follows a user's panning.
- 2) Enforce that the yellow container stays completely inside the "black box" (to imitate the behavior of the flying balls). You can do this in a first step only horizontally, because vertically it is a bit more complex.



# Possible solution



```
class _MyHomePageState extends State<MyHomePage> {
  double marginRight = 0;
  double marginBottom = 0;
  bool expertMode = false;
  double yellowTop = 200;
  double yellowLeft = 150;
  double boxBorderWidth = 10;
  double yellowWidth = 200;
  double yellowHeight = 100;
  double bottomRowHeight = 45;

  @override
  Widget build(BuildContext context) {
    double screenWidth = MediaQuery.of(context).size.width;
    double screenHeight = MediaQuery.of(context).size.height;
    double paddingTop = MediaQuery.of(context).padding.top;
    double stackWidth = screenWidth - 2 * boxBorderWidth;
    double stackHeight = screenHeight -
      2 * boxBorderWidth -
      paddingTop -
      kToolbarHeight -
      bottomRowHeight;

    if (expertMode) {
      stackHeight -= bottomRowHeight;
    }
  }
}
```

```
Positioned(
  top: yellowTop,
  left: yellowLeft,
  child: GestureDetector(
    onPanUpdate: (details) {
      setState(() {
        yellowLeft += details.delta.dx;
        if (yellowLeft < 0) {
          yellowLeft = 0;
        }
        if (yellowLeft > stackWidth - yellowWidth) {
          yellowLeft = stackWidth - yellowWidth;
        }
        yellowTop += details.delta.dy;
        if (yellowTop < 0) {
          yellowTop = 0;
        }
        if (yellowTop > stackHeight - yellowHeight) {
          yellowTop = stackHeight - yellowHeight;
        }
      });
    },
    child: Container(
      width: yellowWidth,
      height: yellowHeight,
      color: Colors.yellow, // Container
    ), // GestureDetector // Positioned
  ),
```



## Possible solution (continued)



We set the size of the two bottom-rows to be able to calculate the height of the stack:

```
if (expertMode)
  Padding(
    padding: const EdgeInsets.all(8.0),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Text("Speed:",
          style: Theme.of(context).textTheme.bodyMedium) //
      ], // Row
    ), // Padding
  Padding(
    padding: const EdgeInsets.all(8.0),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        ElevatedButton(
          onPressed: () {
            setState(() {
              expertMode = false;
            });
          },
          child: const Text("Reset")), // ElevatedButton
        ElevatedButton(
          onPressed: () {}, child: const Text("Add 1 ball")),
```



```
if (expertMode)
  Container(
    height: bottomRowHeight,
    padding: const EdgeInsets.all(8.0),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        Text("Speed:",
          style: Theme.of(context).textTheme.bodyMedium) //
      ], // Row
    ), // Container
  Container(
    height: bottomRowHeight,
    padding: const EdgeInsets.all(8.0),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        ElevatedButton(
          onPressed: () {
            setState(() {
              expertMode = false;
            });
          },
          child: const Text("Reset")), // ElevatedButton
        ElevatedButton(
          onPressed: () {}, child: const Text("Add 1 ball")),
        if (expertMode)
```

Move yellow container up when Chrome is resized or when it is near the bottom and Expert Mode is switched on



Moved here  
from  
onPanUpdate

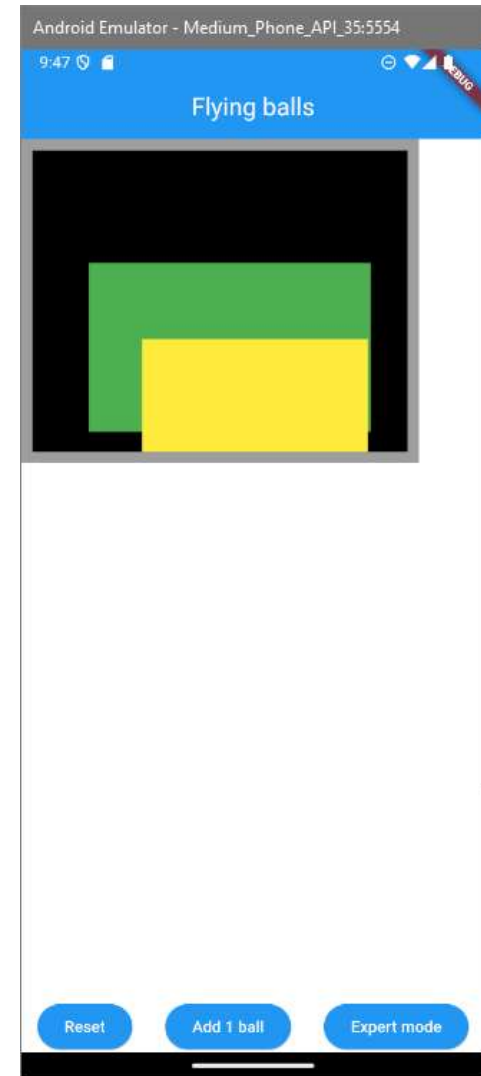
```
Widget build(BuildContext context) {  
  double screenWidth = MediaQuery.of(context).size.width;  
  double screenHeight = MediaQuery.of(context).size.height;  
  double paddingTop = MediaQuery.of(context).padding.top;  
  double stackWidth = screenWidth - 2 * boxBorderWidth;  
  double stackHeight = screenHeight -  
    2 * boxBorderWidth -  
    paddingTop -  
    kToolbarHeight -  
    bottomRowHeight;  
  
  if (expertMode) {  
    stackHeight -= bottomRowHeight;  
  }  
  
  if (yellowLeft < 0) {  
    yellowLeft = 0;  
  }  
  if (yellowLeft > stackWidth - yellowWidth) {  
    yellowLeft = stackWidth - yellowWidth;  
  }  
  if (yellowTop < 0) {  
    yellowTop = 0;  
  }  
  if (yellowTop > stackHeight - yellowHeight) {  
    yellowTop = stackHeight - yellowHeight;  
  }  
  
  return Scaffold(  
    appBar: AppBar(  
      title: Text('Flutter Demo App'),  
      actions: [  
        IconButton(  
          icon: Icon(Icons.settings),  
          onPressed: () {  
            // Open settings dialog  
          }  
        ),  
      ],  
    ),  
    body: Stack(  
      children: [  
        Positioned(  
          top: yellowTop,  
          left: yellowLeft,  
          child: GestureDetector(  
            onPanUpdate: (details) {  
              setState(() {  
                yellowLeft += details.delta.dx;  
                yellowTop += details.delta.dy;  
              });  
            },  
            child: Container(  
              width: yellowWidth,  
              height: yellowHeight,  
              color: Colors.yellow, // Container  
            ),  
          ),  
        ),  
        Positioned(  
          top: 0,  
          left: 0,  
          child: Stack(  
            children: [  
              Positioned(  
                top: 0,  
                left: 0,  
                child: Container(  
                  width: stackWidth,  
                  height: stackHeight,  
                  color: Colors.white, // Stack  
                ),  
              ),  
              Positioned(  
                top: 0,  
                left: 0,  
                child: Container(  
                  width: stackWidth,  
                  height: stackHeight,  
                  color: Colors.white, // Stack  
                ),  
              ),  
            ],  
          ),  
        ),  
      ],  
    ),  
  );  
}
```

```
Positioned(  
  top: yellowTop,  
  left: yellowLeft,  
  child: GestureDetector(  
    onPanUpdate: (details) {  
      setState(() {  
        yellowLeft += details.delta.dx;  
        yellowTop += details.delta.dy;  
      });  
    },  
    child: Container(  
      width: yellowWidth,  
      height: yellowHeight,  
      color: Colors.yellow, // Container  
    ),  
  ),  
) // GestureDetector // Positioned
```



Move yellow container up when user changes the size of the “black box”

```
Widget build(BuildContext context) {  
  double screenWidth = MediaQuery.of(context).size.width;  
  double screenHeight = MediaQuery.of(context).size.height;  
  double paddingTop = MediaQuery.of(context).padding.top;  
  double stackWidth = screenWidth - 2 * boxBorderWidth - marginRight;  
  double stackHeight = screenHeight -  
    2 * boxBorderWidth -  
    paddingTop -  
    kToolbarHeight -  
    bottomRowHeight -  
    marginBottom;  
}
```





# Periodic timer in Flutter

periodic timer in Flutter



Ah, a periodic timer in Flutter! You're looking to execute a piece of code repeatedly at a set interval, right? Flutter provides a straightforward way to achieve this using the `Timer` class from the `dart:async` library. ▾

```
class _PeriodicCounterState extends State<PeriodicCounter> {  
  int _counter = 0;  
  Timer? _timer;  
  
  @override  
  void initState() {  
    super.initState();  
    _timer = Timer.periodic(const Duration(seconds: 1), (Timer timer) {  
      setState(() {  
        _counter++;  
      });  
    });  
  }  
  
  @override  
  void dispose() {  
    _timer?.cancel(); // Cancel the timer when the widget is disposed to prevent memory leaks  
    super.dispose();  
  }  
}
```



# When is a widget disposed ?

when is a flutter widget disposed



In Flutter, the `dispose()` method is called when a `State` object (associated with a `StatefulWidget`) is about to be permanently removed from the widget tree. This is a crucial part of the widget lifecycle for resource management.

Here's a breakdown of when a Flutter widget is disposed:

- **When the `State` object is permanently removed from the widget tree:** This is the primary scenario where `dispose()` is called. It happens when:
  - You navigate away from a screen that contains the widget (e.g., popping a route from the navigation stack).
  - The widget is conditionally removed from the tree (e.g., an `if` statement or `Visibility` widget causes it to no longer be built).
  - A parent widget rebuilds and no longer includes the `StatefulWidget` in its child list.
- **The `dispose()` method is the last stage of a `State` object's lifecycle.** Once `dispose()` is called, the `State` object is considered unmounted, and its `mounted` property becomes `false`. It's an error to call `setState()` on an unmounted `State` object. [🔗](#)

# Periodic timer moving the yellow rectangle



```
double speedX = 1;
double speedY = 2;
Timer? timer;

@override
void initState() {
  super.initState();
  timer = Timer.periodic(const Duration(milliseconds: 20), timerCallback);
}

void timerCallback(Timer t) {
  setState(() {
    yellowLeft += speedX;
    yellowTop += speedY;
  });
}

@override
void dispose() {
  // Cancel the timer when the widget is disposed to prevent memory leaks
  timer?.cancel();
  super.dispose();
}
```

```
if (yellowLeft < 0) {
  yellowLeft = 0;
  speedX = -speedX;
}
if (yellowLeft > stackWidth - yellowWidth) {
  yellowLeft = stackWidth - yellowWidth;
  speedX = -speedX;
}
if (yellowTop < 0) {
  yellowTop = 0;
  speedY = -speedY;
}
if (yellowTop > stackHeight - yellowHeight) {
  yellowTop = stackHeight - yellowHeight;
  speedY = -speedY;
}
```

# Not every setState results in a build



```
int timerCounter = 0;
DateTime timerStartDate = DateTime.now();
bool firstTimerCall = true;
int buildCounter = 0;
DateTime buildStartDate = DateTime.now();
bool firstBuildCall = true;
```

```
void timerCallback(Timer t) {
  if (firstTimerCall) {
    timerStartDate = DateTime.now();
    firstTimerCall = false;
  }
  setState(() {
    yellowLeft += speedX;
    yellowTop += speedY;
    timerCounter++;
  });
}
```

```
Widget build(BuildContext context) {
  double screenWidth = MediaQuery.of(context).size.width;
  double screenHeight = MediaQuery.of(context).size.height;
  double paddingTop = MediaQuery.of(context).padding.top;
  double stackWidth = screenWidth - 2 * boxBorderWidth - marginRight;
  double stackHeight = screenHeight -
    2 * boxBorderWidth -
    paddingTop -
    kToolbarHeight -
    bottomRowHeight -
    marginBottom;

  if (firstBuildCall) {
    buildStartDate = DateTime.now();
    firstBuildCall = false;
  }
  buildCounter++;
}
```

```
var timersPerSecond = timerCounter *
  1000 /
  DateTime.now().difference(timerStartDate).inMilliseconds;
```

```
var buildsPerSecond = buildCounter *
  1000 /
  DateTime.now().difference(buildStartDate).inMilliseconds;
```

# Not every setState results in a build (measurements)



```
timer = Timer.periodic(const Duration(milliseconds: 1), timerCallback);

child: Stack(children: [
  Positioned(
    top: 0,
    left: 0,
    child: Text(
      "timersPerSecond: ${timersPerSecond.toStringAsFixed(1)}  "
      "buildsPerSecond: ${buildsPerSecond.toStringAsFixed(1)}  \"
      "timerCounter: $timerCounter  buildCounter: $buildCounter\")),
```

Chrome Debug:

```
timersPerSecond: 130.3  buildsPerSecond: 55.5
timerCounter: 1385  buildCounter: 615
```

Emulator Debug:

```
timersPerSecond: 135.6  buildsPerSecond: 33.5
timerCounter: 2912  buildCounter: 745
```

Samsung A33 Debug:

```
timersPerSecond: 443.3  buildsPerSecond: 55.7
timerCounter: 10846  buildCounter: 1401
```

Emulator Release:

```
timersPerSecond: 221.1  buildsPerSecond: 55.4
timerCounter: 3455  buildCounter: 868
```

Samsung A33 Release:

```
timersPerSecond: 818.7  buildsPerSecond: 59.7
timerCounter: 17067  buildCounter: 1244
```

With

```
Timer.periodic(const Duration(milliseconds: 20), timerCallback);
```

Samsung A33 Release:

```
timersPerSecond: 50.0  buildsPerSecond: 49.9
timerCounter: 1190  buildCounter: 1187
```



# Introduce class Ball

```
class Ball {  
  double diameter = 20;  
  double left = 0;  
  double top = 100;  
  double speedX = 1;  
  double speedY = 2;  
  
  void move() {  
    left += speedX;  
    top += speedY;  
  }  
  
  void bounce(double boxWidth, double boxHeight) {  
    if (left < 0) {  
      left = 0;  
      speedX *= -1;  
    }  
    if (left > boxWidth - diameter) {  
      left = boxWidth - diameter;  
      speedX *= -1;  
    }  
  
    if (top < 0) {  
      top = 0;  
      speedY *= -1;  
    }  
    if (top > boxHeight - diameter) {  
      top = boxHeight - diameter;  
      speedY *= -1;  
    }  
  }  
}
```

In class `_MyHomePageState` :

```
var ball = Ball();
```

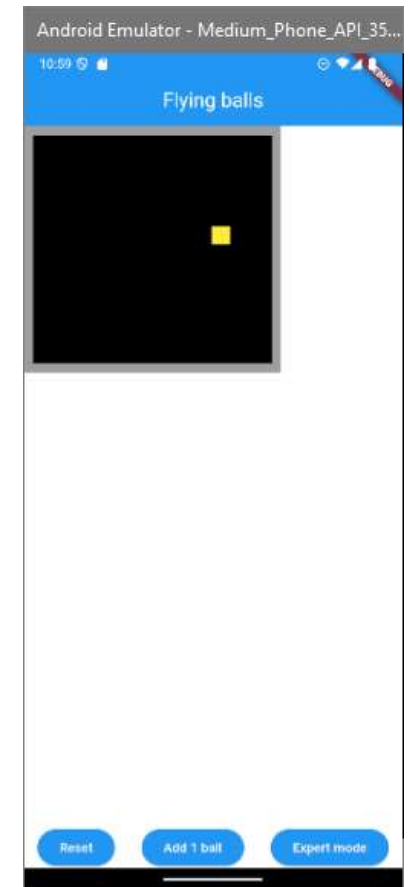
In `timerCallBack`:

```
ball.move();
```

In `build` method:

```
ball.bounce(stackWidth, stackHeight);
```

```
Positioned(  
  top: ball.top,  
  left: ball.left,  
  child: Container(  
    width: ball.diameter,  
    height: ball.diameter,  
    color: Colors.yellow), // Container  
), // Positioned
```





# Manage many balls

In class `_MyHomePageState` :

```
List<Ball> balls = [Ball()];
```

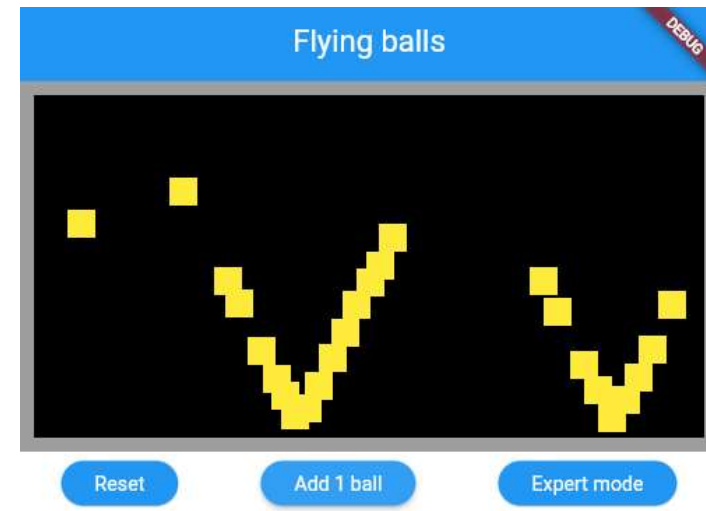
In `timerCallback`:

```
for (var ball in balls) {  
  ball.move();  
}
```

In `build` method:

```
for (var ball in balls) {  
  ball.bounce(stackWidth, stackHeight);  
}
```

```
child: Stack(children: [  
  for (var ball in balls)  
    Positioned(  
      top: ball.top,  
      left: ball.left,  
      child: Container(  
        width: ball.diameter,  
        height: ball.diameter,  
        color: Colors.yellow), // Container  
    ), // Positioned  
], // Stack
```



```
ElevatedButton(  
  onPressed: () {  
    setState(() {  
      expertMode = false;  
      balls.clear();  
      balls.add(Ball());  
    });  
  },  
  child: const Text("Reset")), // ElevatedButton  
ElevatedButton(  
  onPressed: () {  
    setState(() {  
      balls.add(Ball());  
    });  
  },  
  child: const Text("Add 1 ball")), // ElevatedButton
```

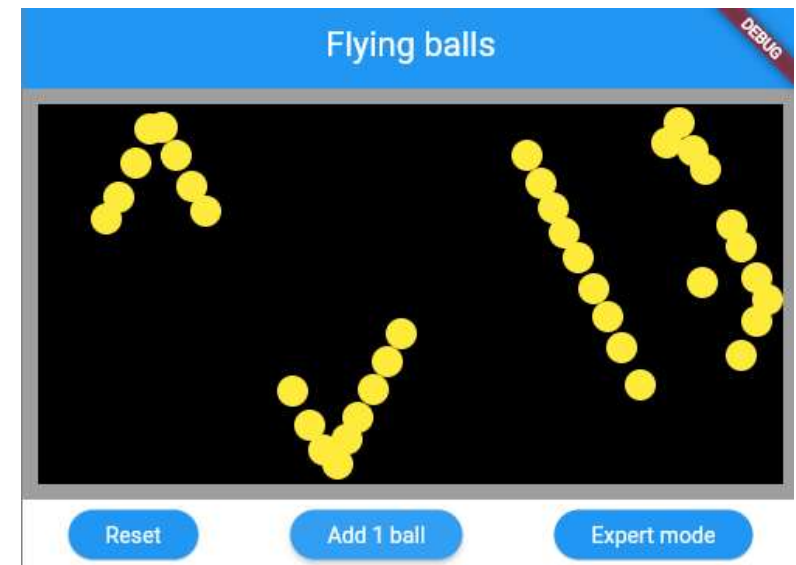
# Show circles instead of squares



```
Positioned(  
  top: ball.top,  
  left: ball.left,  
  child: Container(  
    width: ball.diameter,  
    height: ball.diameter,  
    color: Colors.yellow), // Container  
), // Positioned
```



```
Positioned(  
  top: ball.top,  
  left: ball.left,  
  child: Container(  
    width: ball.diameter,  
    height: ball.diameter,  
    decoration: const BoxDecoration(  
      color: Colors.yellow,  
      shape: BoxShape.circle,  
    ), // BoxDecoration  
  ), // Container  
), // Positioned
```



# Randomize balls

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

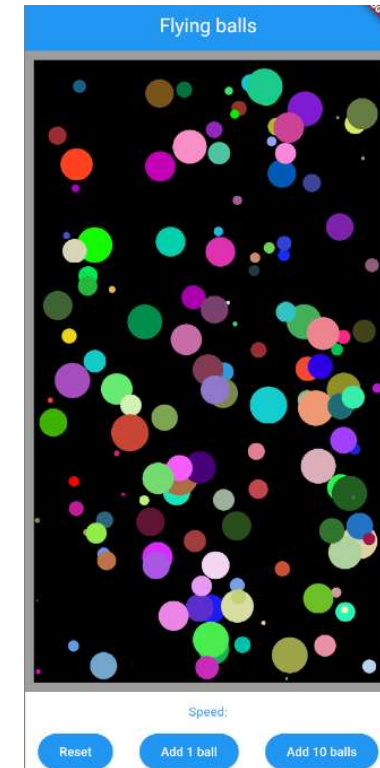
class Ball {
  double diameter = 20;
  double left = 0;
  double top = 100;
  double speedX = 1;
  double speedY = 2;
  Color color = Colors.yellow;

  Ball.random() {
    var random = Random();
    top = 50 + 200 * random.nextDouble();
    diameter = 2 + 40 * random.nextDouble();
    speedX = 0.5 + 4 * random.nextDouble();
    speedY = 0.5 + 4 * random.nextDouble();
    color = Color.fromARGB(
      255, random.nextInt(256), random.nextInt(256), random.nextInt(256));
  }
}
```

```
ElevatedButton(
  onPressed: () {
    setState(() {
      expertMode = false;
      balls.clear();
      balls.add(Ball.random());
    });
  },
  child: const Text("Reset"),
```

```
ElevatedButton(
  onPressed: () {
    setState(() {
      balls.add(Ball.random());
    });
  },
  child: const Text("Add 1 ball"),
```

```
if (expertMode)
  ElevatedButton(
    onPressed: () {
      setState(() {
        for (var i = 0; i < 10; i++) {
          balls.add(Ball.random());
        }
      });
    },
    child: const Text("Add 10 balls"),
```

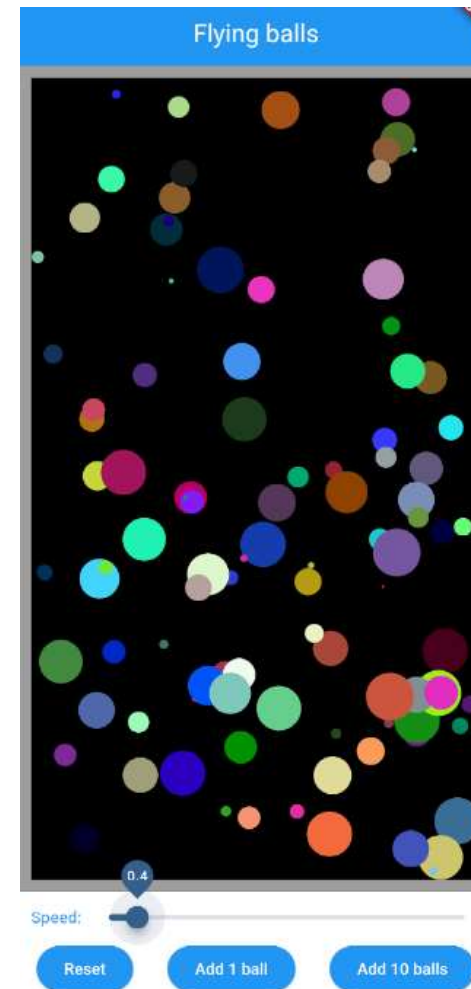


# Modify the speed of all balls

```
class Ball {  
  static double speedFactor = 1;  
  double diameter = 20;  
  double left = 0;
```

```
  void move() {  
    left += speedX * speedFactor;  
    top += speedY * speedFactor;  
  }  
}
```

```
Padding(  
  padding: const EdgeInsets.only(left: 10),  
  child: Text(  
    "Speed:",  
    style: Theme.of(context).textTheme.bodyMedium,  
  ), // Text  
) // Padding  
Expanded(  
  child: Slider(  
    divisions: 100,  
    label: Ball.speedFactor.toStringAsFixed(1),  
    min: 0,  
    max: 5,  
    value: Ball.speedFactor,  
    onChanged: (value) {  
      setState(() {  
        Ball.speedFactor = value;  
      });  
    },  
  ), // Slider  
) // Expanded
```





## Some “physics” around speed and gravity

Movement with constant speed:

$$s(t_1) = s(t_0) + v * (t_1 - t_0)$$

Velocity with constant acceleration:

$$v(t_1) = v(t_0) + a * (t_1 - t_0)$$



# Simulate gravity



```
class Ball {  
  static double speedFactor = 1;  
  static double yAcceleration = 0.0;  
  double diameter = 20;  
  double left = 0;  
}
```

```
void move() {  
  left += speedX * speedFactor;  
  speedY += yAcceleration;  
  top += speedY * speedFactor;  
}
```

```
class _MyHomePageState extends State<MyHomePage> {  
  double marginRight = 0;  
  double marginBottom = 0;  
  bool expertMode = false;  
  bool showSpeed = true;  
  double boxBorderWidth = 10;  
}
```

```
if (!showSpeed)  
  Expanded(  
    child: Slider(  
      min: 0,  
      max: 0.5,  
      value: Ball.yAcceleration,  
      onChanged: (value) {  
        setState(() {  
          Ball.yAcceleration = value;  
        });  
      },  
    ), // Slider  
  ), // Expanded  
  Padding(  
    padding: const EdgeInsets.only(right: 18.0),  
    child: ElevatedButton(  
      onPressed: () {  
        setState(() {  
          showSpeed = !showSpeed;  
        });  
      },  
      child:  
        Text(showSpeed ? "Show gravity" : "Show speed"),  
    ), // Padding  
  ),
```





## Beautify the border of the “black box”

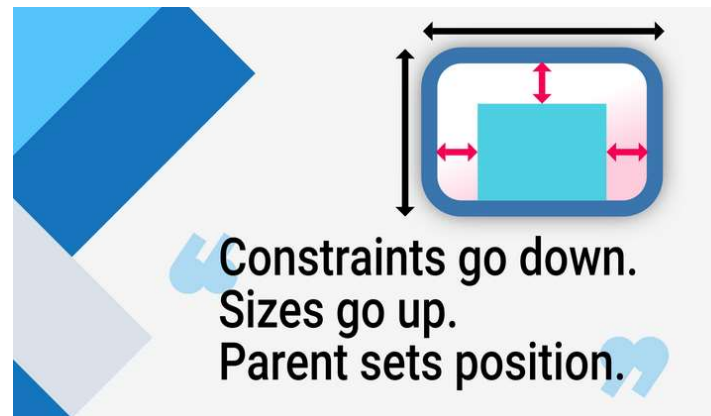
```
child: Container(  
  margin:  
    EdgeInsets.only(right: marginRight, bottom: marginBottom),  
  decoration: const BoxDecoration(  
    color: Colors.black,  
    border: Border(  
      top: BorderSide(color: Colors.grey, width: 10),  
      left: BorderSide(  
        color: Color.fromARGB(255, 119, 117, 117),  
        width: 10), // BorderSide  
      right: BorderSide(  
        color: Color.fromARGB(255, 223, 217, 217),  
        width: 10), // BorderSide  
      bottom: BorderSide(  
        color: Color.fromARGB(255, 207, 203, 203),  
        width: 10))), // BorderSide // Border // BoxDecoration  
  child: Stack(children: [  
    for (var ball in balls)  
      Positioned(  
        top: ball.top,
```







# Appendix 1: Understanding Flutter's layout algorithm



Flutter layout can't really be understood without knowing this rule, so Flutter developers should learn it early on.

In more detail:

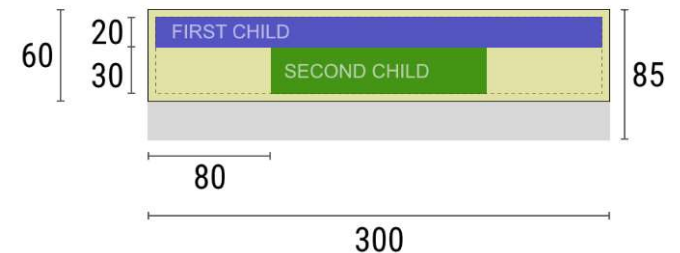
- A widget gets its own **constraints** from its **parent**. A *constraint* is just a set of 4 doubles: a minimum and maximum width, and a minimum and maximum height.
- Then the widget goes through its own list of **children**. One by one, the widget tells its children what their **constraints** are (which can be different for each child), and then asks each child what size it wants to be.
- Then, the widget positions its **children** (horizontally in the *x* axis, and vertically in the *y* axis), one by one.
- And, finally, the widget tells its parent about its own **size** (within the original constraints, of course).

This was copied from <https://docs.flutter.dev/ui/layout/constraints>



# Example for the Flutter layout algorithm

For example, if a composed widget contains a column with some padding, and wants to lay out its two children as follows:



The negotiation goes something like this:

**Widget:** "Hey parent, what are my constraints?"

**Parent:** "You must be from 0 to 300 pixels wide, and 0 to 85 tall."

**Widget:** "Hmmm, since I want to have 5 pixels of padding, then my children can have at most 290 pixels of width and 75 pixels of height."

**Widget:** "Hey first child, You must be from 0 to 290 pixels wide, and 0 to 75 tall."

**First child:** "OK, then I wish to be 290 pixels wide, and 20 pixels tall."

**Widget:** "Hmmm, since I want to put my second child below the first one, this leaves only 55 pixels of height for my second child."

**Widget:** "Hey second child, You must be from 0 to 290 wide, and 0 to 55 tall."

**Second child:** "OK, I wish to be 140 pixels wide, and 30 pixels tall."

**Widget:** "Very well. My first child has position x: 5 and y: 5, and my second child has x: 80 and y: 25."

**Widget:** "Hey parent, I've decided that my size is going to be 300 pixels wide, and 60 pixels tall."

Also copied from <https://docs.flutter.dev/ui/layout/constraints>



## Appendix 2: Gradient colors

```
Container(  
  alignment: Alignment.topCenter,  
  width: 300,  
  height: 200,  
  decoration: const BoxDecoration(  
    //color: Colors.green,  
    gradient: LinearGradient(colors: [Colors.yellow, Colors.blue]),  
  ), // BoxDecoration  
  child: null), // Container
```

```
gradient: LinearGradient(  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7]),
```

```
gradient: LinearGradient(  
  colors: [Colors.yellow, Colors.blue], stops: [0.5, 0.5]),
```



# More Gradients

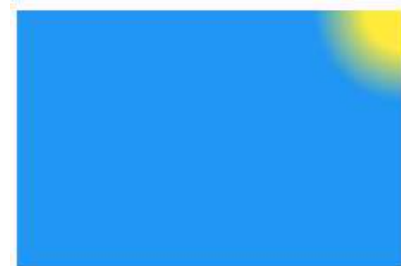
```
gradient: LinearGradient(  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7],  
  begin: Alignment.topLeft, end: Alignment.bottomRight), // LinearGradient
```



```
gradient: RadialGradient(  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7]),
```



```
gradient: RadialGradient(  
  center: Alignment.topRight,  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7]),
```



# Sweep Gradients

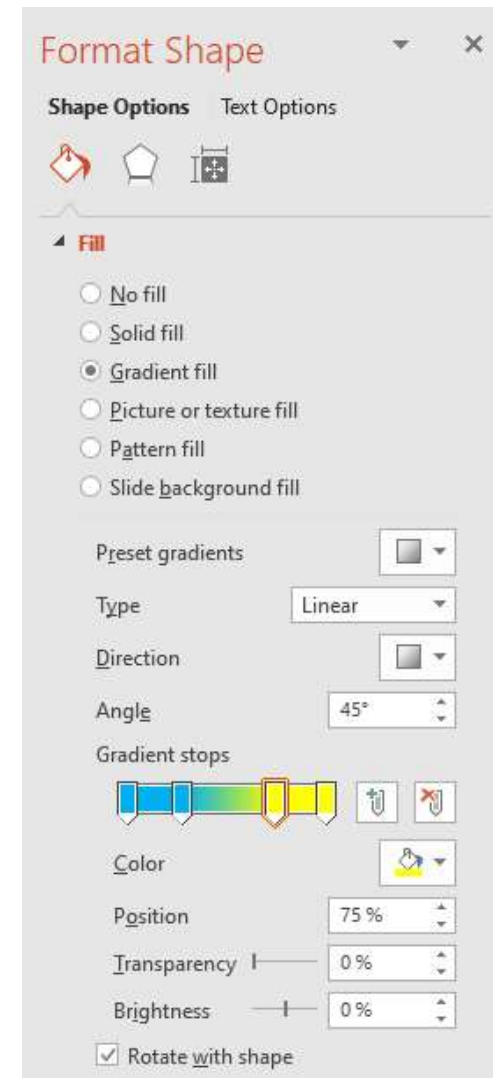
```
gradient: SweepGradient(  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7]),
```

```
gradient: SweepGradient(  
  startAngle: pi * 0.5,  
  endAngle: pi,  
  colors: [Colors.yellow, Colors.blue], stops: [0.3, 0.7]),
```

```
gradient: SweepGradient(  
  colors: [Colors.red, Colors.green,  
           Colors.yellow, Colors.blue, Colors.red],  
  stops: [0.0, 0.25, 0.5, 0.75, 1]) // SweepGradient
```



# Gradient colors in Powerpoint





# Using gradient colors for the balls



```
class Ball {  
  static double speedFactor = 1;  
  static double yAcceleration = 0.0;  
  double diameter = 20;  
  double left = 0;  
  double top = 100;  
  double speedX = 1;  
  double speedY = 2;  
  Color _color1 = Colors.yellow;  
  Color _color2 = Colors.red;  
  bool _useLinearGradient = true;  
}
```

```
Gradient getGradientColor() {  
  Gradient result = RadialGradient(colors: [_color1, _color2]);  
  
  if (_useLinearGradient) {  
    result = LinearGradient(colors: [_color1, _color2]);  
  }  
  
  return result;  
}
```

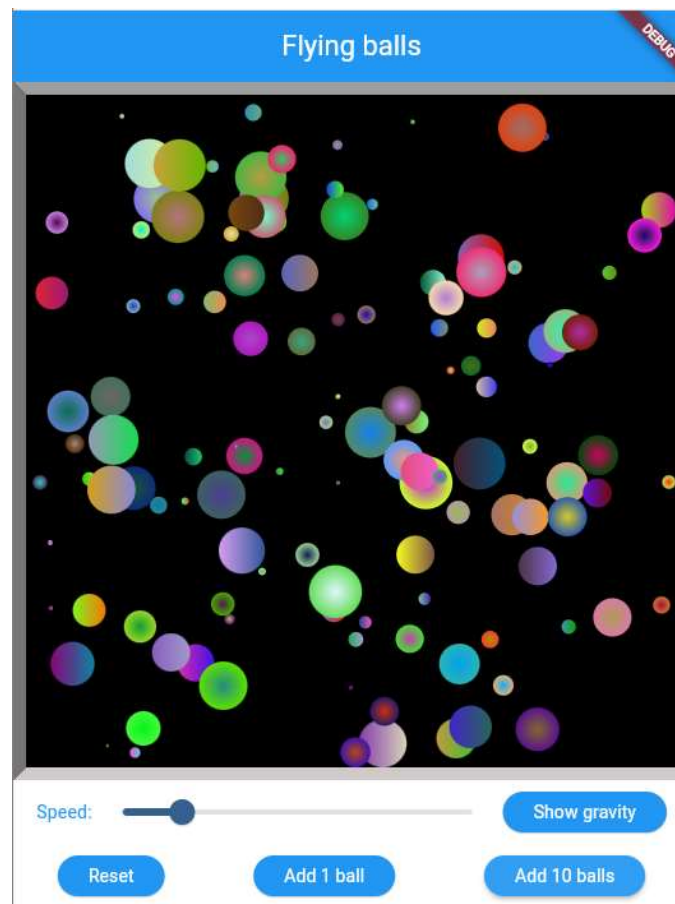
In main.dart:

```
Ball.random() {  
  var random = Random();  
  top = 50 + 200 * random.nextDouble();  
  diameter = 2 + 40 * random.nextDouble();  
  speedX = 0.5 + 4 * random.nextDouble();  
  speedY = 0.5 + 4 * random.nextDouble();  
  _color1 = Color.fromARGB(  
    255, random.nextInt(256), random.nextInt(256), random.nextInt(256));  
  _color2 = Color.fromARGB(  
    255, random.nextInt(256), random.nextInt(256), random.nextInt(256));  
  _useLinearGradient = random.nextBool();  
}
```

```
for (var ball in balls)  
  Positioned(  
    top: ball.top,  
    left: ball.left,  
    child: Container(  
      width: ball.diameter,  
      height: ball.diameter,  
      decoration: BoxDecoration(  
        //color: ball.color,  
        gradient: ball.getGradientColor(),  
        shape: BoxShape.circle,  
      ), // BoxDecoration  
    ), // Container  
  ), // Positioned
```



Decide yourself what you like more ...



or implement an  
even better idea for  
coloring the balls 😊