Flutter: n Platforms, 1 Codebase, 0 Problems

Michael P. Rogers, Bill Siever
CCSC:Central Plains 2022, Springfield, MO

Agenda

- Why Cross-Platform?
- Why Flutter?
- In Praise of Dart
- Flutter Basics
- Flutter in the Classroom

Terminology

- Q: What do we mean by cross-platform?

- A: One codebase that can run on multiple platforms (Android, iOS, desktop OSes)

- Q: What do we mean by native?

- A: An app developed for a specific platform, using the SDK (frameworks and tools) provided by the OS maker*

Pros and Cons - Industry

Parameters	Native Apps	Cross-Platform Apps
Cost	High Cost of development	Relatively low cost of development
Code Usability	Works for a single platform	Single code can be used for multiple platforms, for an easy portability
Device Access	Platform SDK ensures access to device's APIs without any hindrance	No assured access to all device APIs
UI Consistency	Consistent with the UI components of the device	Limited consistency with the UI components of the device
Performance	Seamless performance, given the app is developed for the device's OS	High on performance, but lags and hardware compatibility issues are not uncommon

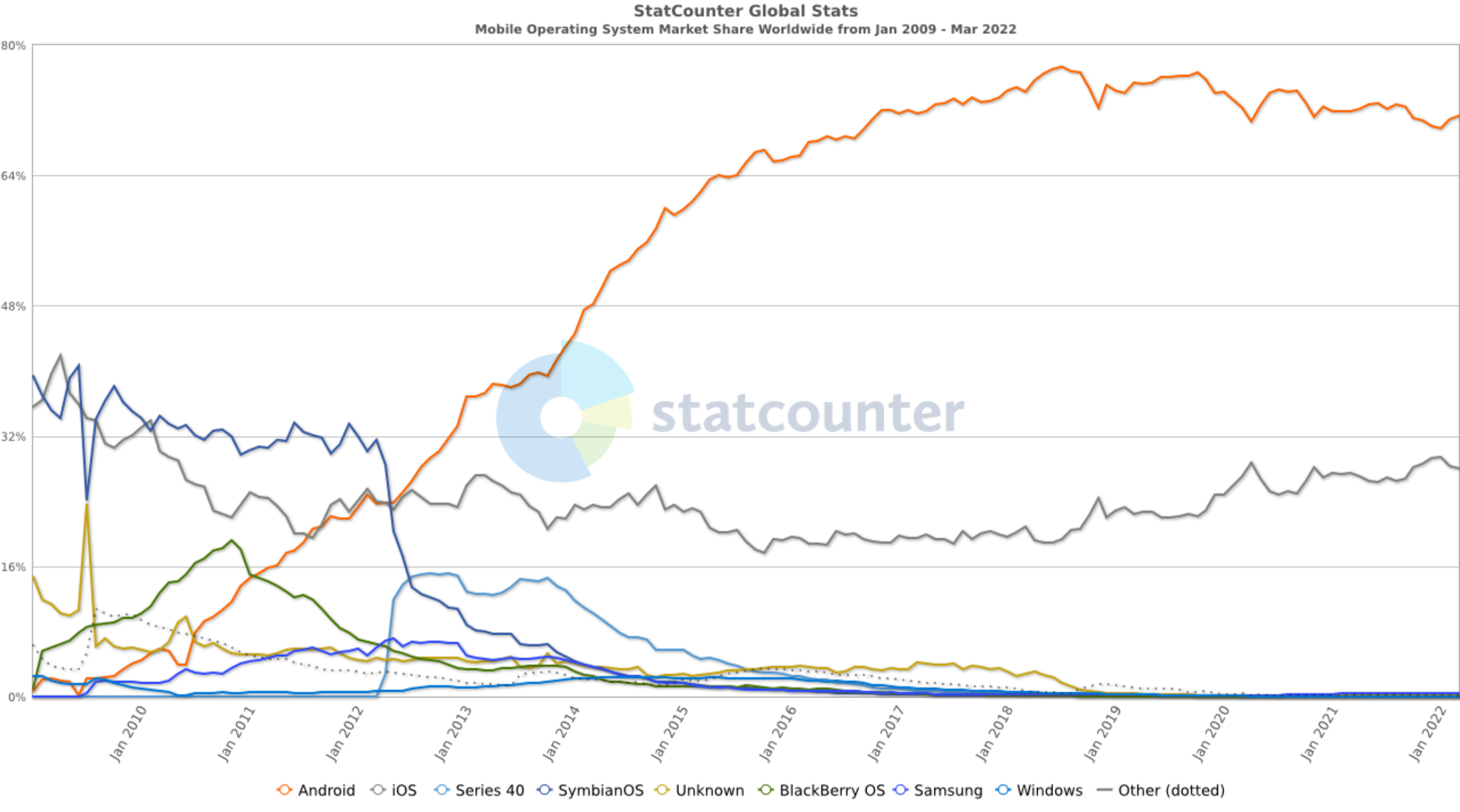
net solutions

Pros and Cons - Academia

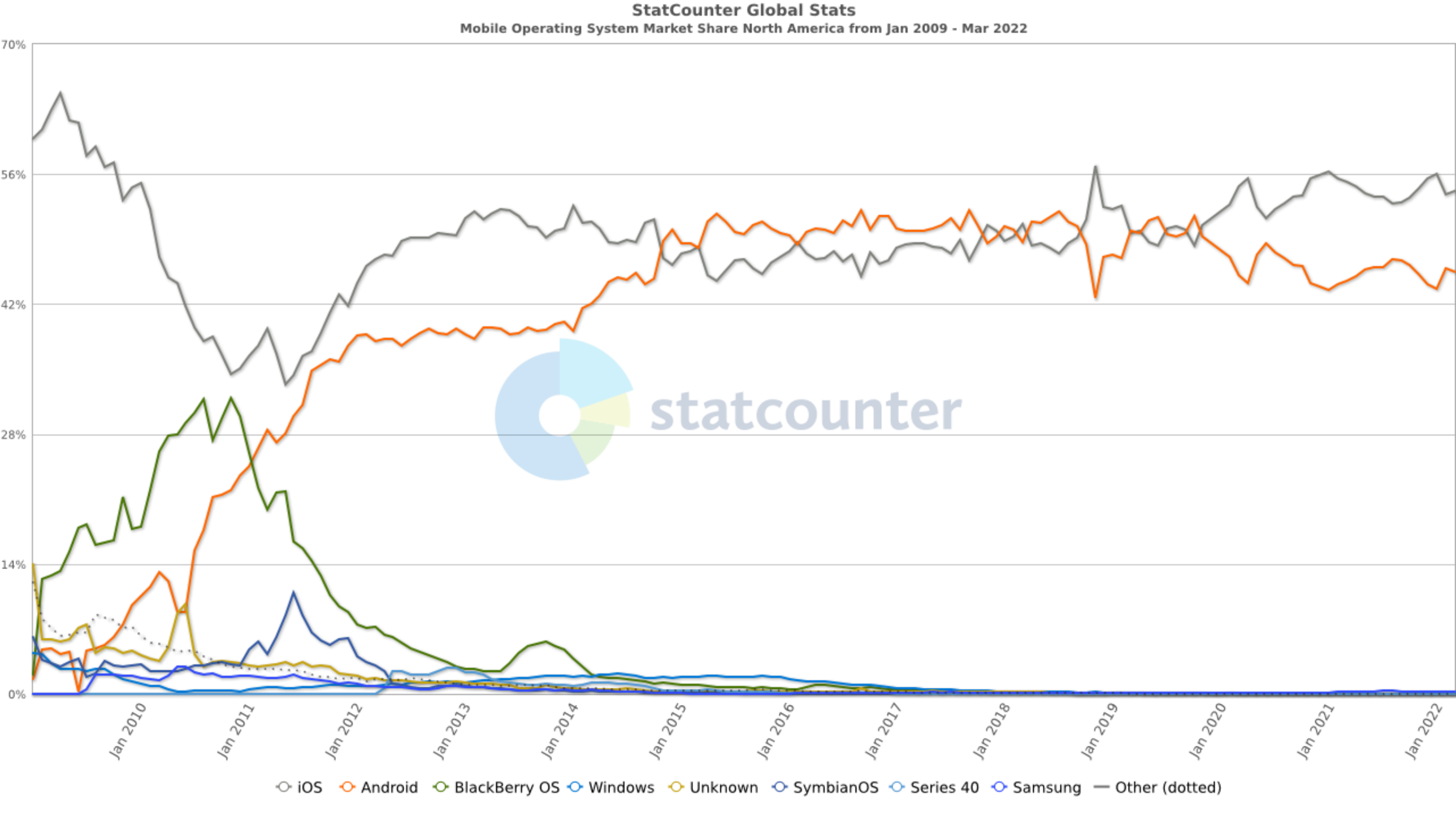
Parameters	Native Apps	Cross-Platform Apps
Cost	High Cost of development	Relatively low cost of development
Code Usability	Works for a single platform	Single code can be used for multiple platforms, for an easy portability
Device Access	Platform SDK ensures access to device's APIs without any hindrance	No assured access to all device APIs
UI Consistency	Consistent with the UI components of the device	Limited consistency with the UI components of the device
Performance	Seamless performance, given the app is developed for the device's OS	High on performance, but lags and hardware compatibility issues are not uncommon

net solutions

Marketshare, Worldwide



Marketshare, North America



Cross-Platform Possibilities

- 5App
- Appcelerator
- Cocos2d
- Codename One
- Corona
- Flutter
- ionic
- Mobile Angular UI
- Monocross
- NativeScript
- .NET MAUI
- PhoneGap
- Qt
- React Native
- RhoMobile
- Sencha Touch
- Unity 3D
- Xamarin
- NativeScript

https://stackshare.io/cross-platform-mobile-development

Dart

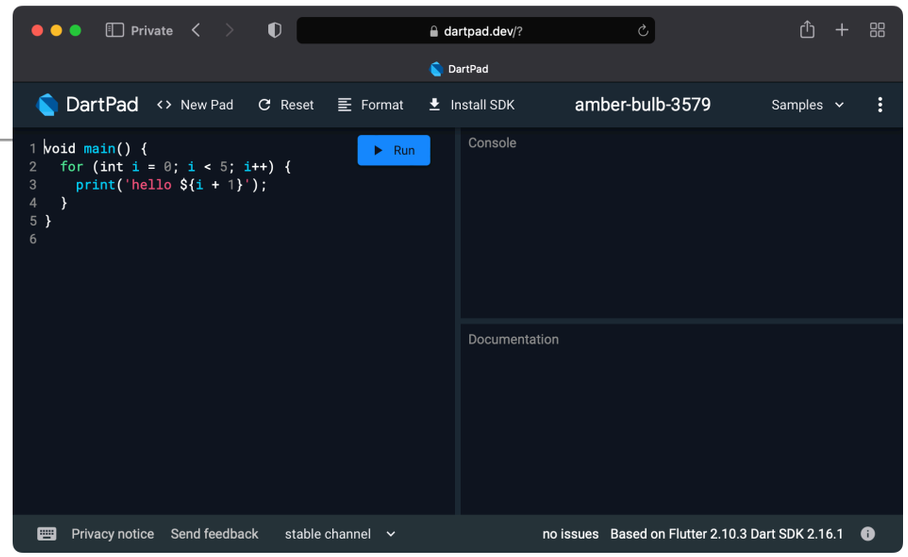
In Praise of Dart

Other programming languages

The complete top 50 of programming languages is listed below. This overview is published unofficially, because it could be the case that we missed a language. If you have the impression there is a programming language lacking, please notify us at tpci@tiobe.com. Please also check the overview of all programming languages that we monitor.

Position	Programming Language	Ratings
21	Prolog	0.55%
22	COBOL	0.54%
23	Scratch	0.53%
24	SAS	0.52%
25	Groovy	0.52%
26	Rust	0.51%
27	(Visual) FoxPro	0.50%
28	Ada	0.42%
29	PL/SQL	0.39%
30	Fortran	0.39%
31	Kotlin	0.38%
32	Julia	0.32%
33	Lisp	0.31%
34	VBScript	0.31%
35	Dart	0.26%
36	Scala	0.25%
37	D	0.25%

Quick Start: Running an App



1. Visit dartpad.dev

- There is no step 2 😍

- Note the numerous samples that show off the power of Dart and/or Flutter

Hello, Dart!

```
/*
 * Author: Tyler Perry
 * Date: 1/4/2022
 * Description: A Simple Dart Program
 */

//Dart programs start with main() ... oh, and single line comments work ;-)
void main(List<String> args) {
  var name = 'Christo'; // declare and initialize a variable using type inference
  String profession = 'artist'; // both name and profession are of type String
  var birthYear = 1935; // type inference is preferred where possible

  printInfo(name, 2021 - birthYear);
}

// A function with two parameters -- can't use var here
void printInfo(String name, int age) {
  print('My name is $name'); // string interpolation
  print('I am $age years old, or ${age * 365.25} days old');
}
```

```
My name is Christo
I am 86 years old, or 31411.5 days old
```

Exercise: Hello, World!

- Write a complete Dart program to print "Hello, World!" (or if you are a purist, "hello, world"

Built-In Types

- numbers (int, double, subclasses of num)
- Strings
- bool
- List< >, Set< >, Map< , >.
  - Specify type between < >
- runes (for expressing Unicode characters in a string)
- symbols
- Everything in Dart is an object

What's missing?

Null Safety

- A type is nullable if it can be assigned the value null
- Nullable types are marked by a ? after the type name

```
int x;
print(x); // error: non-nullable local variable 'x'
          // must be assigned  before it can be used.

int? y;      // nullable type
print(y);    // prints null

y = 15;
print(y);    // no surprise, 15

y = null;
print(y);    // prints null again
```

Null Assertion Operator

- Use the null assertion operator (!) if you know that a variable is non-null -- it allows assignment to a non-null value

```dart
String? name = 'Dua Lipa';

String famousSinger = name!; // name isn't null - safe to assign
                             // to non-null String
print(famousSinger);
```

Numbers & Precision

- There are 3 flavors of integer

Internal representation	smi (**sm**all **i**nteger)*	mint (**m**edium **int**eger)	bigint
Minimum value	-2^{30} (on a 32-bit machine) -2^{62} (on a 64-bit machine)	-2^{63}	Limited by RAM
Maximum value	$2^{30} - 1$ (on a 32-bit machine) $2^{62} - 1$ (on a 64-bit machine)	$2^{63} - 1$	Limited by RAM

- Floats are stored in IEEE-754 double-precision format

*rhymes with pi (or, if you prefer, pie 😀)

Lists (aka Arrays in other languages)

```
main() {
  var population = <int>[]; // an empty list of ints
  var population2 = List<int>(); // same, with constructor
  List<int> population3 = []; // same, not ideal

  var years = [1812, 1914, 2000];
  print('years, with ${years.length} elements, is $years');
  years.add(2021); //
  print('Now years is $years');
  years.addAll([2020, 2025, 2030]);
  print('After addAll(), years is $years');
  years.removeWhere((year) => year % 5 == 0);
  print('Getting rid of years divisible by 5, we have ... $years');

  var moreYears = [...years, 1400, 1401, 1402, ];
  print('using the spread operator, we get $moreYears');

  final centuries = List.generate(5, (i) => 1900 + 100 * i, growable: true);
  print(centuries); // [1900, 2000, 2100, 2200, 2300]

}
```

```
years, with 3 elements, is [1812, 1914, 2000]
Now years is [1812, 1914, 2000, 2021]
After addAll(), years is [1812, 1914, 2000, 2021, 2020, 2025, 2030]
Getting rid of years divisible by 5, we have ... [1812, 1914, 2021]
using the spread operator, we get [1812, 1914, 2021, 1400, 1401, 1402]
[1900, 2000, 2100, 2200, 2300]
```

Exercise

- In main(), declare a list of Strings:
  - ["to", "boldy", "go"]
- and iterate through it (the for-loops you know and love are at your disposal)

Exercise Solution

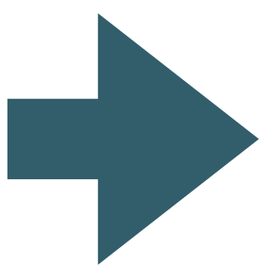
```
void main() {
  var words = ["to", "boldly", "go"];
  for(int i=0; i < words.length; i++){
    print(words[i]);
  }
}
```

```
void main() {
  var words = ["to", "boldly", "go"];
  for(var word in words){
    print(word);
  }
}
```

Trailing Commas

- Dart supports trailing commas
- Helps format / legibility

```
Row(children: [
  const Radio(value: true, groupValue: 'radiogaga', onChanged: null),
  Text(answers[0]),
  const Radio(value: true, groupValue: 'radiogaga', onChanged: null),
  Text(answers[1]),
]),
```



```
Row(children: [
  const Radio(
    value: true,
    groupValue: 'radiogaga',
    onChanged: null,
  ),
  Text(answers[0]),
  const Radio(
    value: true,
    groupValue: 'radiogaga',
    onChanged: null,
  ),
  Text(answers[1]),
      ]),
```

Maps

```
main(){

  var chess = <String, int>{}; // Empty Map<String,int>

  chess['Pawn'] = 1;
  chess['Knight'] = 3;
  chess['Rook'] = 5;

 var population = {
   'China' : 1427647786,
   'India' : 1352642280,
   'United States' : 327096265,
 }; // type is Map<String,int>


 population['Pakistan'] = 212228286;

 print(population.keys); // (China, India, United States, Pakistan)
 print(population.values); // (1427647786, 1352642280, 327096265, 212228286)
 print('India has ${population['India']} people');
}
```

{"Flutter", "also", "has", "Sets"}

Functions and Optional Positional Parameters: [ ]

```
void sayHello(String to, [bool? inEnglish]){

  if(inEnglish == null || inEnglish){
    print('Hello, $to');
  } else {
    print('Bonjour, $to');
  }
}

// somewhere in main() ...
 sayHello('Michael', true);
 sayHello('Eleanor'); // inEnglish is an optional positional parameter,
 sayHello('Chidi');   // it can be omitted (and will have the value null)
```

Exercise

- Create a function to iterate through a list of Strings and return the fraction of words that include the letter 'o'

```
void main() {
  var phrase = ["to", "boldly", "go", "where"];
  print(fractionOfOs(phrase)); // output: 0.75
}
```

- Hint: the **contains()** method works more or less like in other languages ... in fact *everything* in Dart works more or les like in other languages, but with just less fuss.

Exercise Solution

```
double fractionOfOs(List<String> words){
  double numOs = 0;
  for(var word in words){
    if(word.contains('o') || word.contains('O')){
      numOs++;
    }
  }

  return numOs / words.length;
}
```

Named Parameters

- named parameters are enclosed in { }
- they are optional unless marked as required with **required**
- arguments for named parameters appear in the form *parameterName : argumentValue*
- named parameters are ubiquitous in the Flutter APIs

Named Parameters

```
void sayHello({String to, required bool inEnglish}){

  if(inEnglish == null || inEnglish){
    print('Hello, $to');
  } else {
    print('Bonjour, $to');
  }
}

void main(){
 sayHello(to: 'Michael', inEnglish: true);
 sayHello(to: 'Eleanor', inEnglish : false);
 sayHello(inEnglish: true, to: 'Tahani'); // let's not switch order ...
 sayHello(to: 'Chidi'); // inEnglish is optional
 sayHello(inEnglish: true); // to is also optional - output: Hello, null

}

// void sayHello({@required String to, bool inEnglish}){ ... }
```

Exercise

- Modify fractionOfOs() to include a named parameter, **words**, and then call it from main

Exercise Solution

```
void main() {
  var phrase = ["to", "boldly", "go", "where"];
  print(fractionOfOs(words: phrase));
}

double fractionOfOs({required List<String> words}){
  double numOs = 0;
  for(var word in words){
    if(word.contains('o') || word.contains('O')){
      numOs++;
    }
  }

  return numOs / words.length;
}
```

Objects Overview

- Dart is an OO language, with classes and mixins
- Every object is an instance of a class and descends from Object
- Methods can be added to existing classes using extensions. That avoids having to subclass, and can be used when you don't have access to the source

Setting Instance Variables 😍

```dart
void main() {
  var stu = Student(firstName: 'Janet', lastName: 'Reno', gpa: 4.0);
  print(stu);
}

class Student {
  String? lastName;  // ? ==> variable could store a String or null

  String? firstName;
  double? gpa;

  // If the constructor's only job is to initialize instance variables
  // eliminate the constructor body:
  // Student({this.firstName, this.lastName, this.gpa});

  Student({this.firstName, this.lastName, this.gpa}) {
    // instance variables get initialized automatically
    // other stuff can happen here
  }

  @override
  String toString() => '$lastName, $firstName ($gpa)';
}
```

Named Constructors

- Dart doesn't allow multiple constructors with different parameter lists, like Java

- Instead, define one or more **named constructors**, with this syntax:

- ClassName.constructorName(param list)

Named Constructors

```
class Movie {
  int? length = 0;
  String? title;

  Movie.undecided() {
    title = '';
    length = 0;
  }

  Movie.short({this.title, this.length});

  Movie(String title, int length);

  @override
  String toString() {
    return 'title: $title, length: $length';
  }
}
```

```
int main() {
  var movie = Movie.undecided();
  var movie2 = Movie.short(title: 'WW 1984');
  var movie3 = Movie.short(length: 150);
  var movie4 = Movie('Mulan', 150);
  print([movie, movie2, movie3, movie4]);
  return 0;
}
```

```
[title: , length: 0,
title: WW 1984, length: null,
title: null, length: 150,
title: null, length: 0]
```

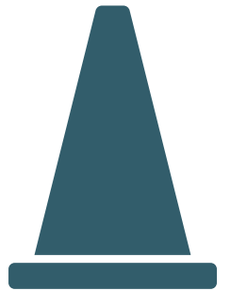
Named Constructors

```dart
int main() {
  var movie = Movie.undecided();
  var movie2 = Movie.short(title: 'WW 1984'); // length of 0
  // var movie3 = Movie.short(length: 150); won't compile
  var movie4 = Movie('Mulan', 150);
  print([movie, movie2, movie3, movie4]);
  return 0;
}
```

```dart
class Movie {
  int length = 0;
  String title;

// instance variables must be assigned before entering body of constructor
  Movie.undecided()
      : title = '',
        length = 0;

  Movie.short(
      {required this.title,
      this.length = 0}); // one required, other defaults to 0

  Movie(this.title, this.length); // both required

  @override
  String toString() {
    return 'title: $title, length: $length';
  }
}
```

```
[title: , length: 0,
title: WW 1984, length: 0,
title: Mulan, length: 150]
```

Extension Methods



- Extension methods let you add methods (and getters, setters and operators) to a pre-existing class which you don't have the source code for

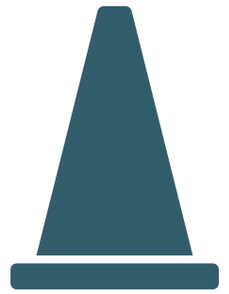
- To use an extension method just import its library, then it's available to you

Extension Syntax

```
extension <extension name> on <type> {
    <method definition(s)>
}
```

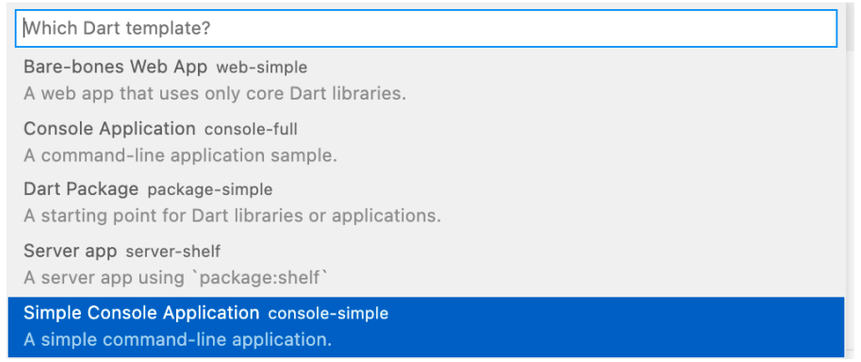
- Inside the method, use **this** to reference the invoking object

An Example



```dart
void main(List<String> arguments) {
  print('hello flutter'.vowelCount()); // output: 4
}


extension VowelCounter on String {
  int vowelCount() {
    var numVowels = 0;
    for (int i = 0; i < this.length; i++) {
      if ('aeiou'.contains(this[i])) {
        numVowels++;
      }
    }
    return numVowels;
  }
}
```

Exercise



- In the same file as main()
- Define a class, **Person**, with instance variables **name** (String), **age** (int)
- Create a constructor with named parameters for both instance variables. Make name required, provide defaults of -1 for age
- Instantiate one Person using the constructor

Create a named constructor, **Person.unnamed()** to create an unnamed person, of age -1, and no friends. Instantiate a Person using that named constructor

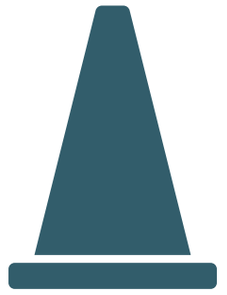
Exercise Solution

```
int main() {
  var percy = Person(name: 'Dua Lipa', age: 42);
  var anonymous = Person.unnamed();
  return 0;
}
```

```
class Person {
  String name;
  int age;

  Person({required this.name, this.age = -1});

  Person.unnamed()
      : name = '',
        age = -1,
}
```

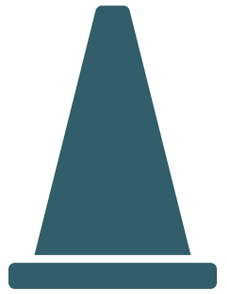
Flutter

A Brief History of Flutter

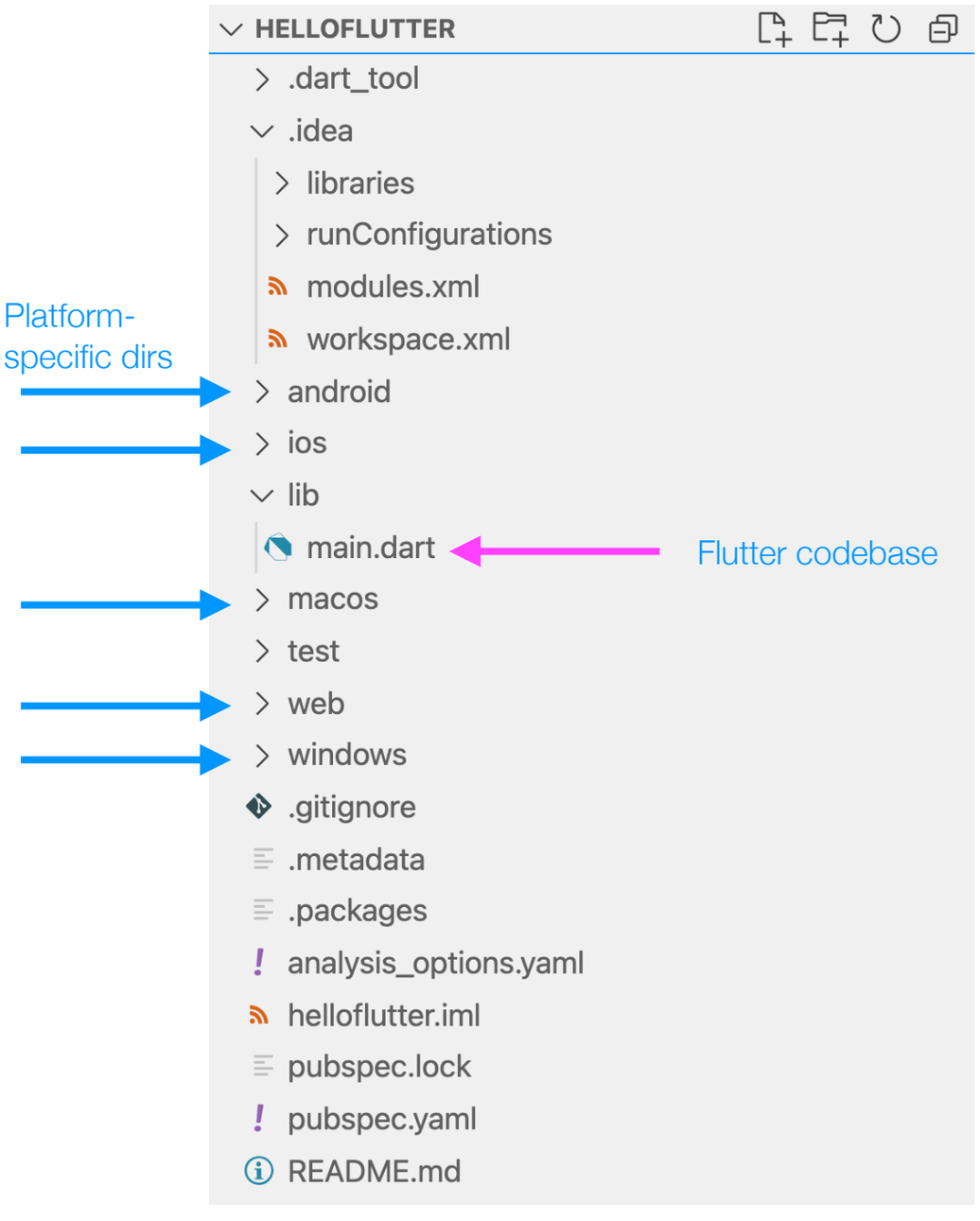


- Started off as a quest for speed, a rendering engine to make Chrome run faster
- Produced something 20x faster
- Languages: C++ >> JavaScript
- Had problems with JavaScript, went looking for a language and found Dart
- Eventually excised most of the C++
- The name, "Flutter" was in a cache of names that Google keeps on hand - companies that they bought, or names that they'd previously rejected
- Version 0.0.6 was released in 2017
- Started showing off multi-platform capabilities at Google I/O 2017
- Announced support for desktop and web at Google I/O 2019

Demo/Exercise: Running an App via VSC



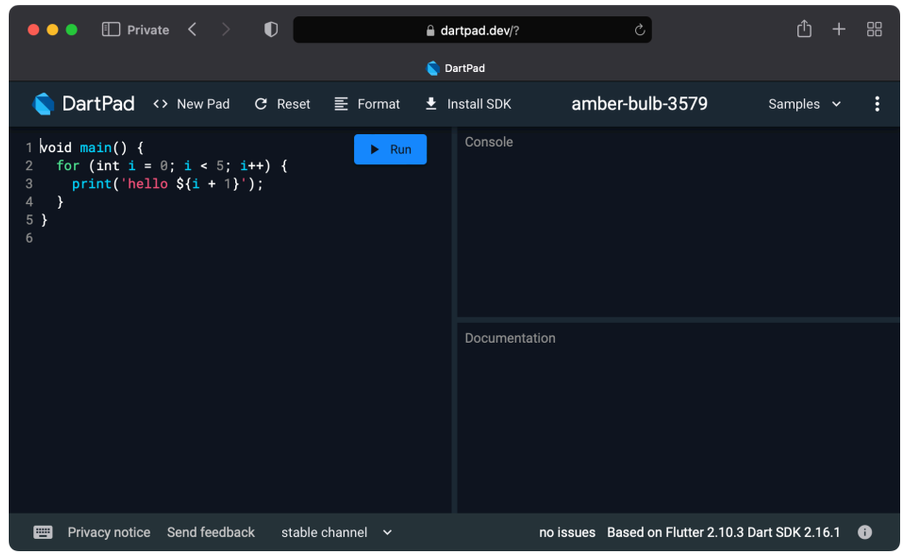
1. Install Flutter (use Visual Studio Code for the IDE)
2. Create a new project
    1. Command Palette (⌘+⇧+P or Ctrl+⇧+P)
    2. Flutter: New Project
    3. Application Template
    4. Select a folder - any one will do
    5. Name the project **helloflutter**
3. Launch a simulator
4. Start Debugging (F5)

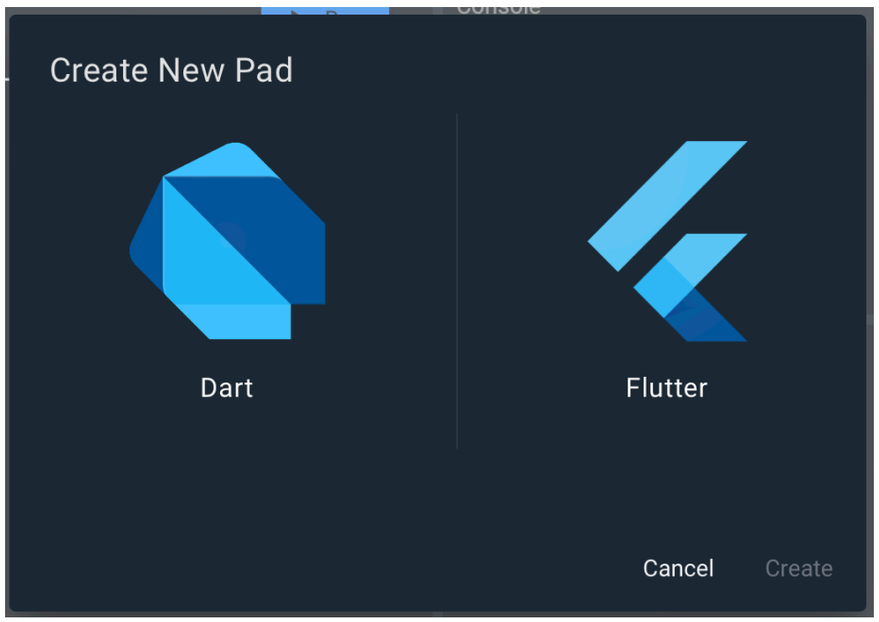


Exercise: Running an App in DartPad

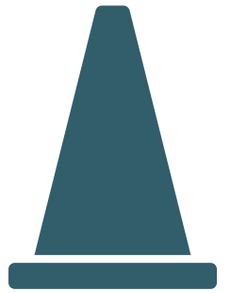
On dartpad.dev ...

1. Click on New Pad
2. Click on **Flutter**
3. There is no step 3.





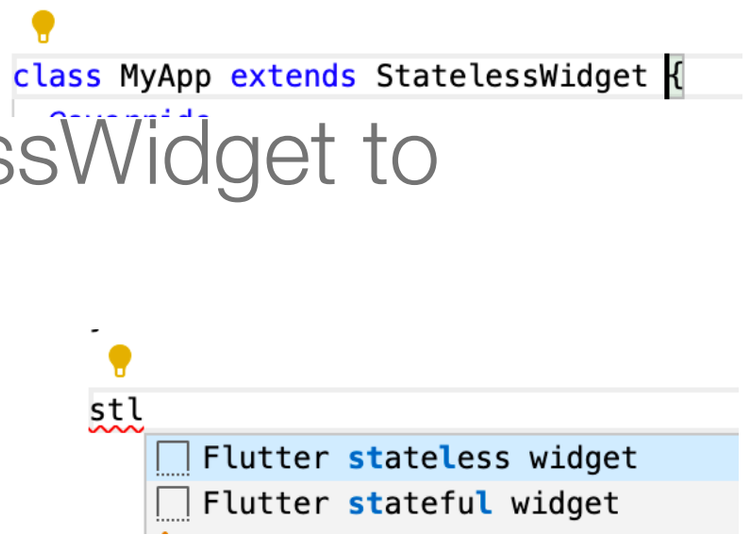
Aside: In Praise of Visual Studio Code Shortcuts



- Hover over a class/method/constructor/parameter to see a description. With methods and constructors, you can see available parameters, very handy!

- You can refactor (^⇧R) a StatelessWidget into a StatefulWidget (another shortcut: click on the class definition, then on the lightbulb)

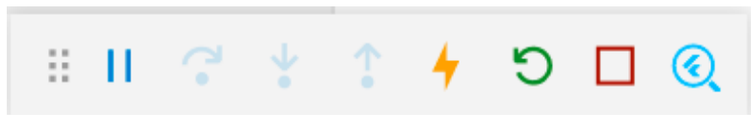
- Type stl then choose Stateful or StatelessWidget to create one on the fly



See also this: https://medium.com/flutter-community/flutter-visual-studio-code-shortcuts-for-fast-and-efficient-development-7235bc6c3b7d

Hot Reload v. Hot Restart (VSC-only)

- Hot reload is a feature of Flutter that allows you to change code and instantaneously see the result -- the app doesn't need to be recompiled and relaunched 😍



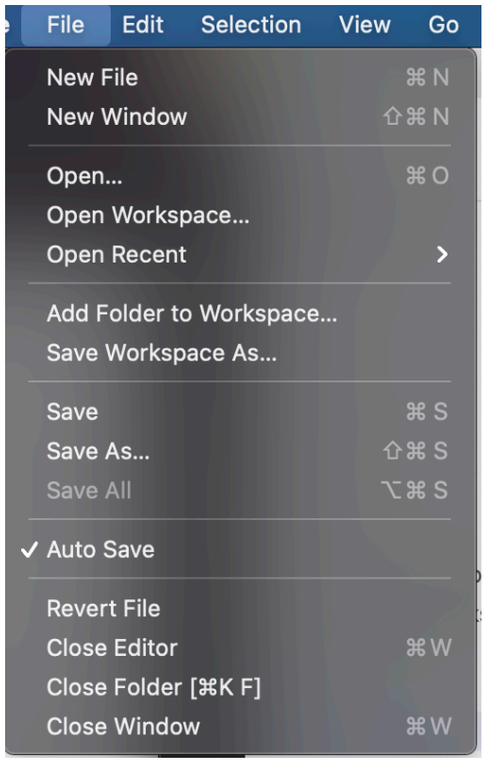
- It can be triggered by saving the code, clicking on the hot reload button, or ^F5

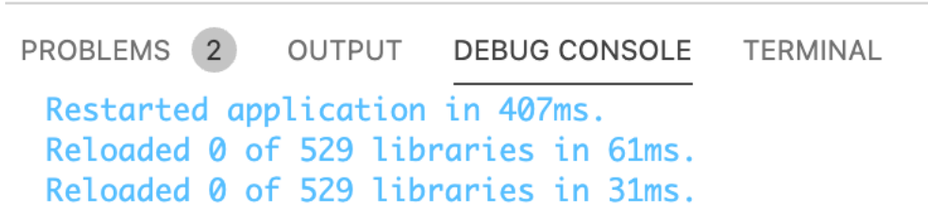
- It rebuilds the widget tree, but it preserves state -- it does not execute main() or initState() [

- Only Flutter apps in debug mode can be hot reloaded

- Hot restart loads code changes into the VM and restarts the app. App state will be lost.

- See the docs for details





You Will Never Need This Slide

- Flutter communicates with Xcode and Android Studio tools, but some times things get lost in transmission

- So, when things go awry:

  - run the project directly from the native IDE

  - clean the project, then choose flutter pub get to retrieve all the flutter packages

  - **macOS:** avoid iCloud Drive: cf https://code-examples.net/en/q/25d0e03

  - text us - or just avoid the middle-person and head right to Google 😉

Widgets

See the Widget catalog
for a comprehensive list

- A Flutter app's UI is built out of **widgets** - think of them as the building blocks for the UI. There are widgets to ...
  - display **information:** Text, Image, ListView, AlertDialog, SnackBar
  - get user **input**: TextField, RaisedButton, Checkbox, DatePicker
  - **layout** other widgets!: Row, Column, Stack, Container
  - work with **forms**: Form, FormState, TextFormField
- You can (and usually do) mix and match widgets to make more complex ones: the entire UI is basically widgets, composed of widgets, composed of widgets ... it's widgets all the way down
- Flutter's tag line: "everything's a widget"

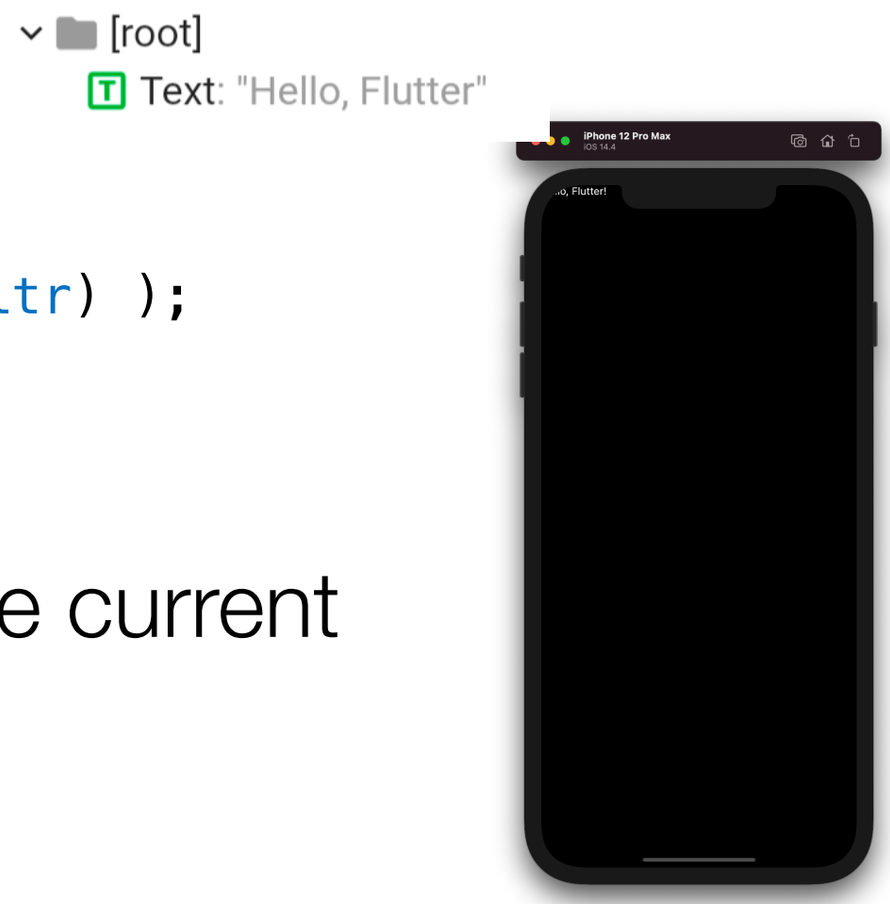
The Simplest Possible Flutter Apps

- Plan: we will demo slides 51-53 (these will work in VSC, but the first 2-3 will not in DartPad)

The Simplest Possible Flutter App

- main() has 1 job - invoke runApp(), passing in ... a widget
- That widget becomes the **root** of the **widget tree**
- We could just use one of Flutter's widgets, like Text(), but then we'd be limited by what Flutter can do

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

void main() {
  runApp( Text('Hello, Flutter!',
                textDirection: TextDirection.ltr) );
}
```

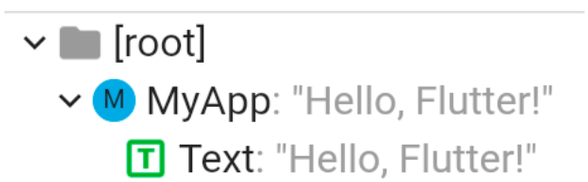


Create this by deleting 99.9% of the current version of main.dart

The Simplest Possible Flutter App

- So, instead, we design our *own* widget by subclassing Widget, overriding build(). It is here where we can now customize the app to do ... whatever we want

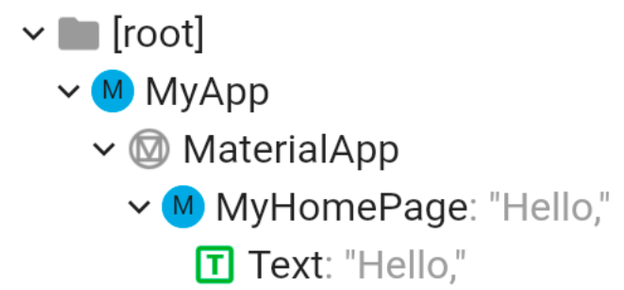
```dart
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('Hello, Flutter!', textDirection: TextDirection.ltr);
  }
}
```



The Simplest Possible Flutter App

- Usually, at the top of the widget tree we place a MaterialApp, to help with navigation and styling

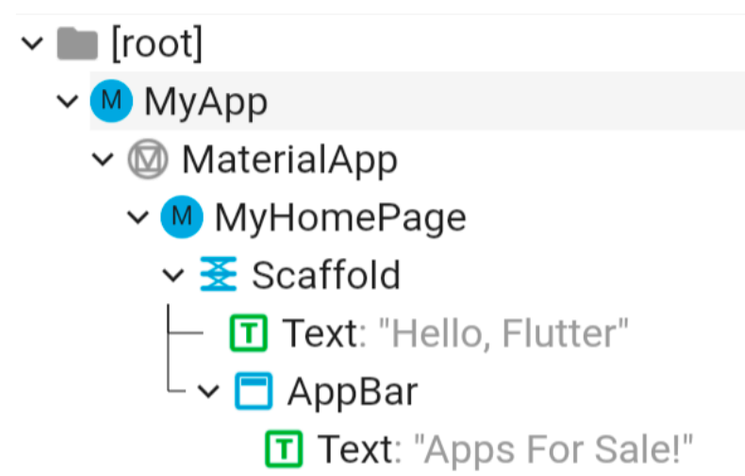
```dart
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: MyHomePage(),
        title: 'Hello, Flutter',
        theme: ThemeData(primarySwatch: Colors.orange));
  }
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('Hello, Flutter');
  }
}
```



The Simplest Possible Flutter App

- A **Scaffold** widget encapsulates an **app bar** - at the top of the screen, a body, drawers, &c.

```
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: MyHomePage(),
        title: 'Hello, Flutter',
        theme: ThemeData(primarySwatch: Colors.orange));
  }
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Apps For Sale!')),
        body: Text('Hello, Flutter'),
        drawer: null);
  }
}
```

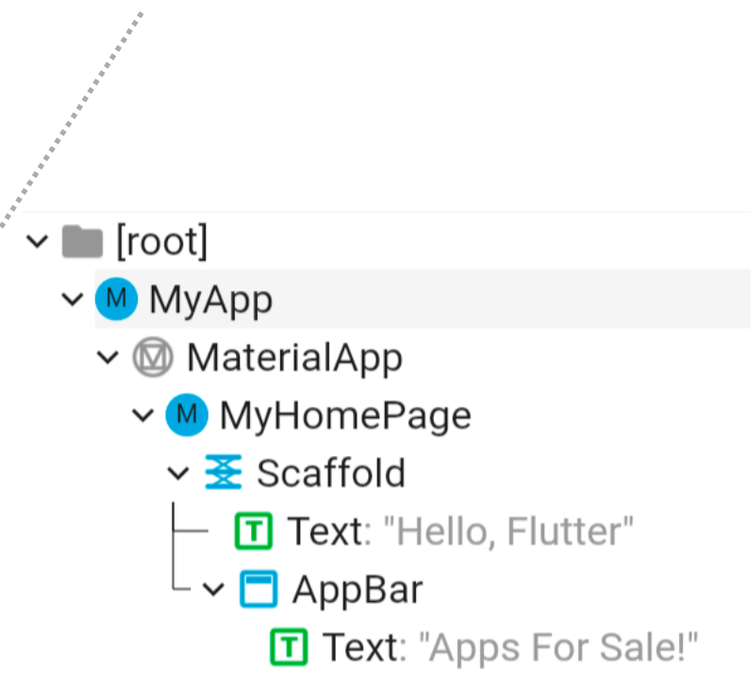


https://dartpad.dev/d29360a2b795c05a39dabd2213febc2f

The Simplest Possible Flutter App

- And now ... the tricky bit: what goes *here?*

```
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: MyHomePage(),
        title: 'Hello, Flutter',
        theme: ThemeData(primarySwatch: Colors.orange));
  }
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Apps For Sale!')),
        body: Text('Hello, Flutter'),
        drawer: null);
  }
}
```



Material Design

- The Flutter SDK has two types of widgets:
    - **Material Widgets** - implements the Material design language - suitable for iOS, Android, web, desktop
    - **Cupertino Widgets** - implements iOS design language - especially well suited for iOS, macOS
- The Flutter Platform Widgets package allegedly gets around this problem
- In this workshop we will use material widgets (which is a benefit/consequence of using MaterialApp)

Single Child Widgets

- Some widgets take a single child, and influence how it is laid out in some way. For example (this is not a definitive list)
  - **Align()** - aligns a child within itself
  - **Center()** - centers a child within itself
  - **Container()** - combines painting, positioning and sizing widgets - highly recommended
  - **Padding()** - puts padding around a child
  - **SizedBox()** - forces its child to have a particular size
  - **Transform()** - applies a transformation to its child
- This is a definitive list -- docs.flutter.dev/development/ui/widgets/layout

Multi-Child Widgets

```
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Apps For Sale!')),
        body: Text('Hello, Flutter'),
        drawer: null);
  }
}
```

- Some layout widgets take multiple children, specified in a children parameter* that takes a List<Widget>

- Row and Column are the most basic

- Row(children: [widget1, widget2])

  - where widget1 and widget2 are widget instances

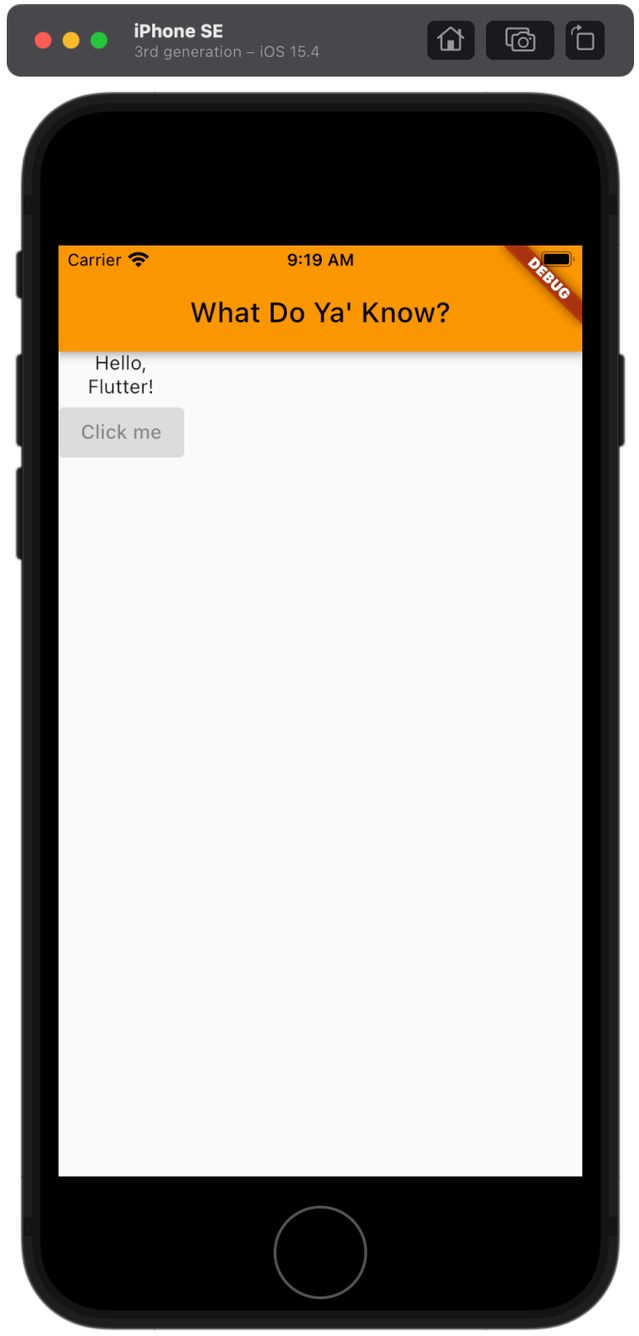
  - you can place as many as you wish here, but at some point you'll run out of room

*in the constructor, as always

Exercise

```
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Apps For Sale!')),
        body: Text('Hello, Flutter'),
        drawer: null);
  }
}
```

- Replace Text() with Column and embed 2 Text widgets ~~and 1 ElevatedButton~~ as children

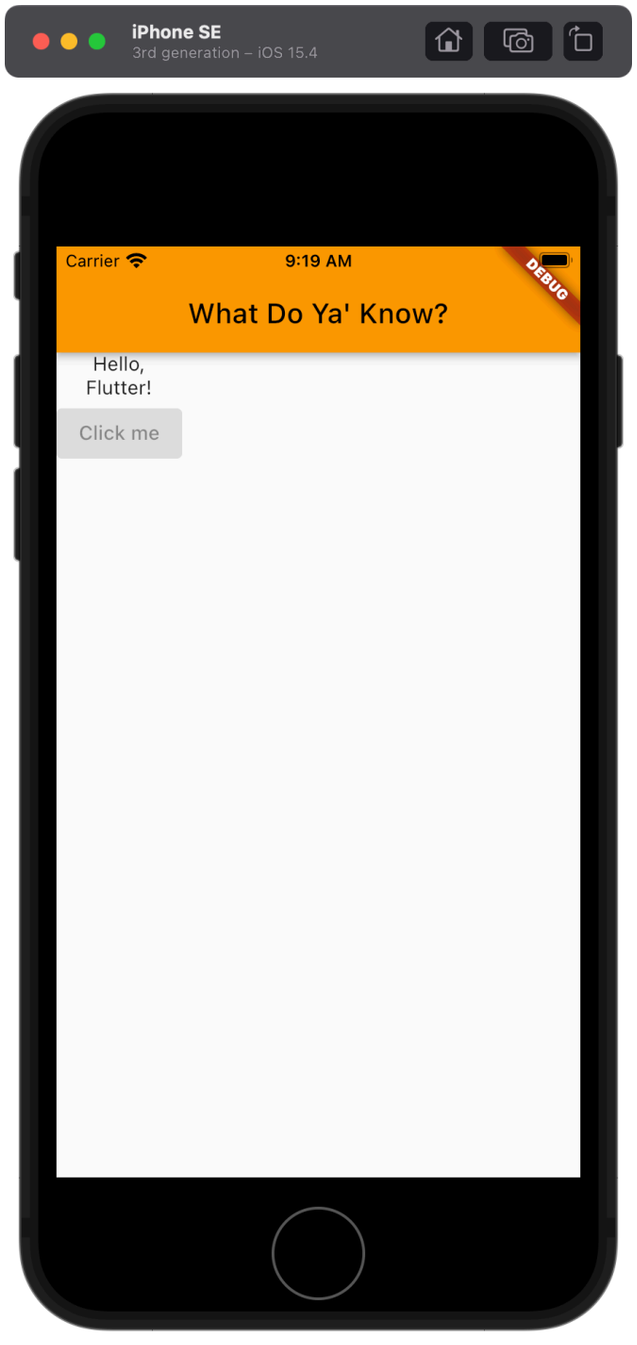


*in the constructor, as always

Exercise Solution

```
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Apps For Sale!')),
        body: Text('Hello, Flutter'),
        drawer: null);
  }
}
```

```
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('What Do Ya\' Know?')),
        body: Column(children: [
          Text('Hello,'),
          Text('Flutter!',),
          ElevatedButton(child: Text('Click me'), onPressed: null),
        ]),
        drawer: null);
  }
}
```



*in the constructor, as always

MainAxis Alignment

- Column (and row) has a **main** axis -- in column the main axis runs vertically
- Using mainAxisAlignment we can position children in various locations

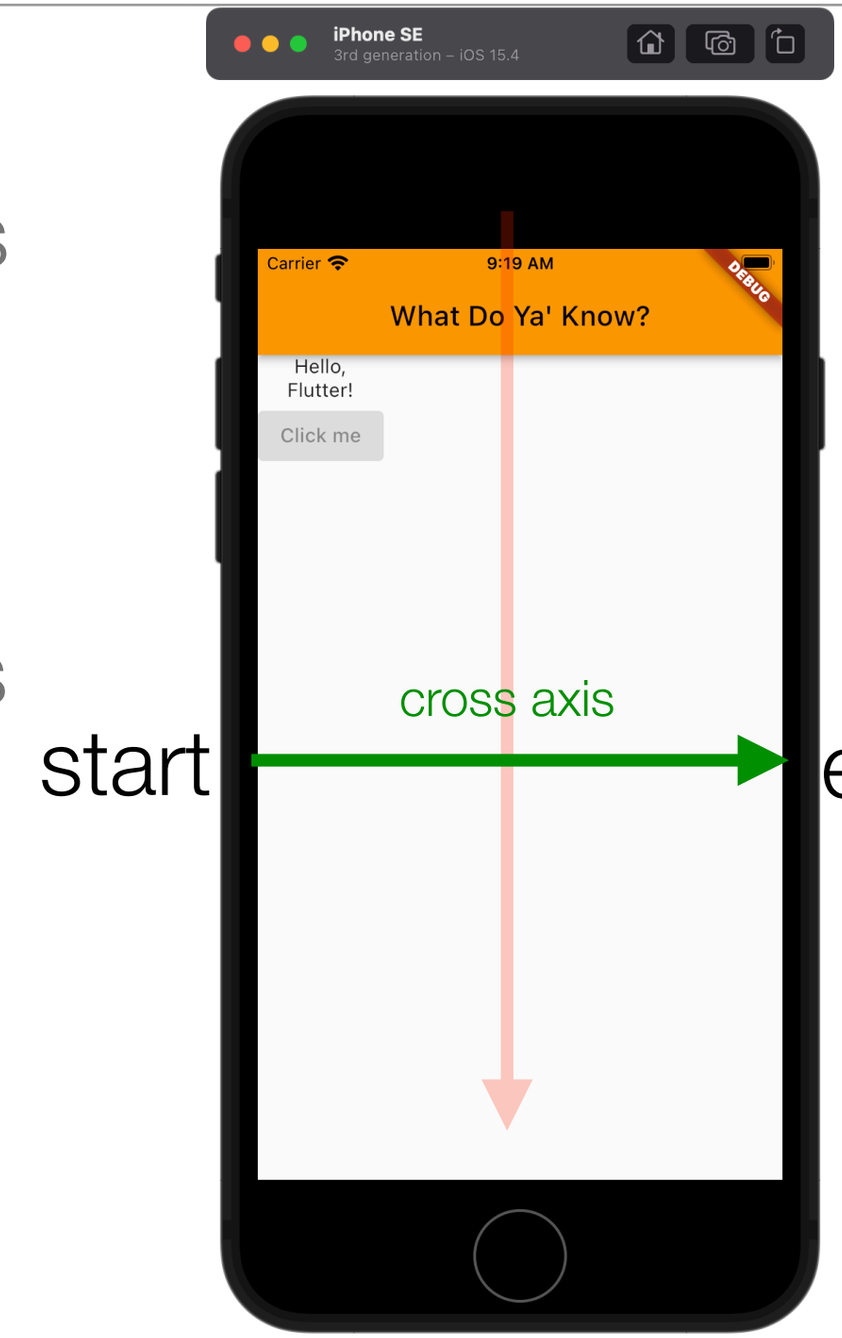
```
class HelloWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.
      children: [
        Text('I am a powerful person'),
        Text('I can do great things!'),
        RaisedButton(child: Text('Tap me'
      ],
    );
  }
}
```

```
values
center
end
spaceAround
spaceBetween
spaceEvenly
start
```

CrossAxis Alignment

- Column (and row) also has a **cross** axis -- in column the cross axis runs horizontally

- Using crossAxisAlignment we can position children in various locations along the cross axis*




```
class HelloWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.
      children: [
        Text('I am a powerful person'),
        Text('I can do great things!'),
        RaisedButton(child: Text('Click m
      ],
    );
  }
}
```

```
values
baseline
center
end
start
stretch
```

a column is only as wide as its widest child, so right now if you choose CrossAxisAlignment.end, it will only move a smidge. CrossAxisAlignment.stretch will stretch the button (the Texts are still only as wide as their content).

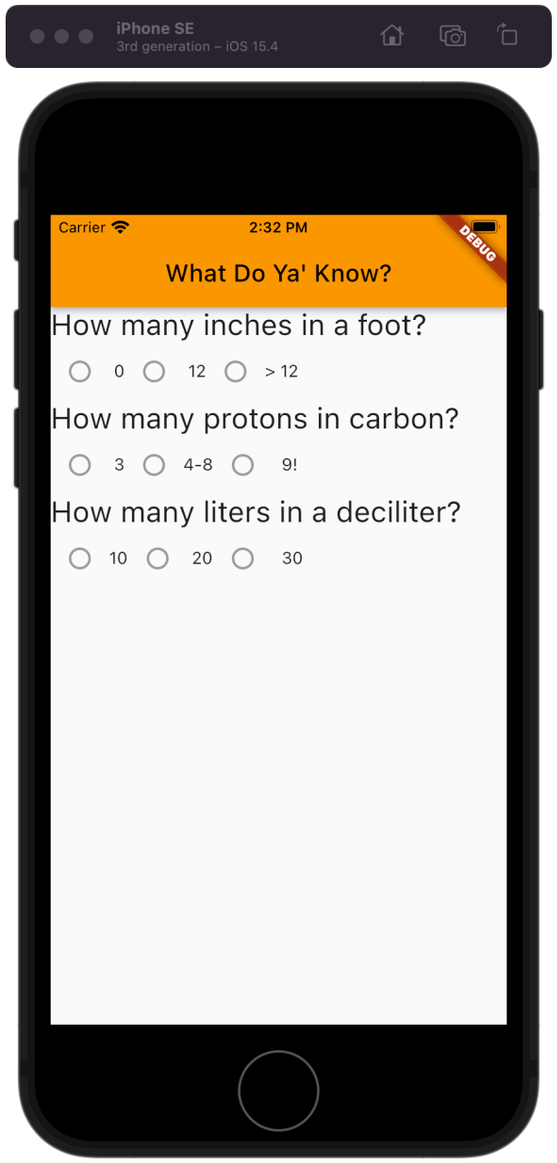
Making a Button Active

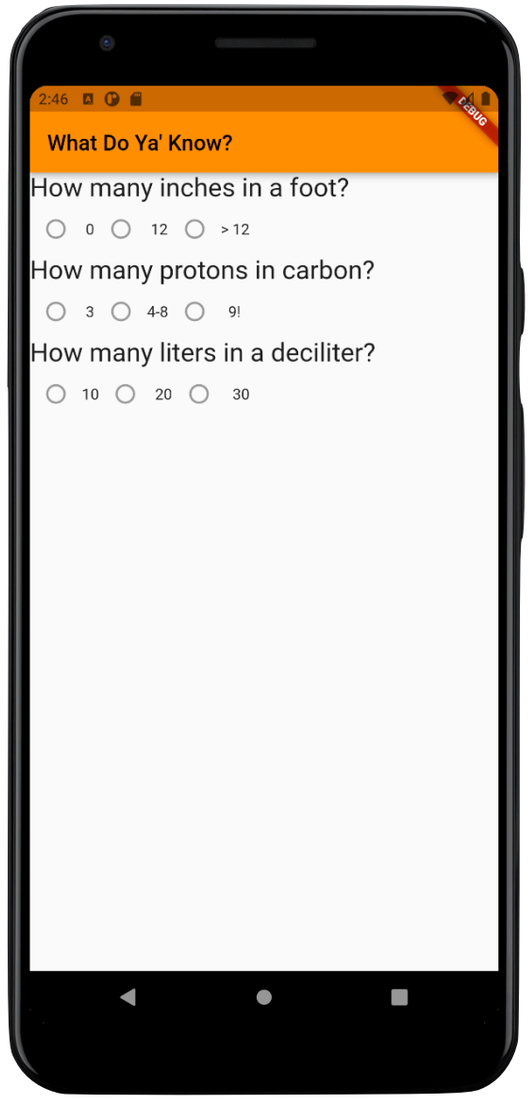
- Setting onPressed to non-null makes it tappable
- $property embeds a property in a String. To embed an *expression* requires ${}

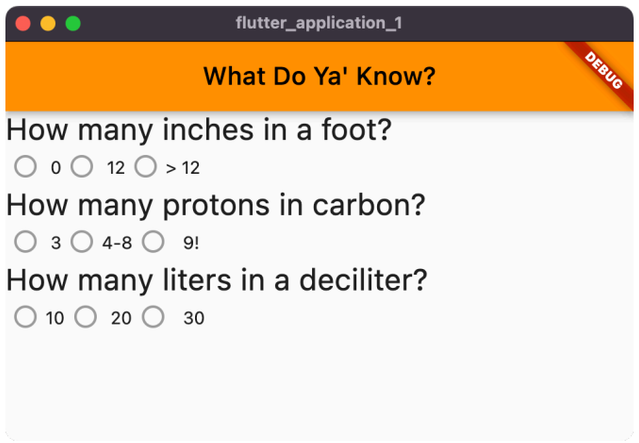
```
class HelloWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: [
        Text('Think of a question and tap the button',
            style: TextStyle(fontSize: 24)),
        Text('Your fortune will be ... '),
        RaisedButton(child: Text('Tell My Fortune'), onPressed: _tellMyFortune),
      ],
    );
  }

  void _tellMyFortune() {
    print('you will become rich ${DateTime.now()}');
  }
}
```

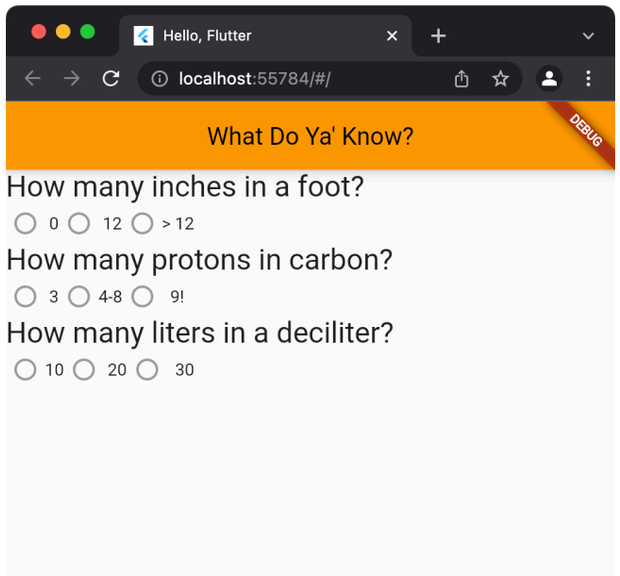
Exercise

- How would you produce this sort of layout?
- For fun we are showing iOS, Android, macOS and Chrome versions?



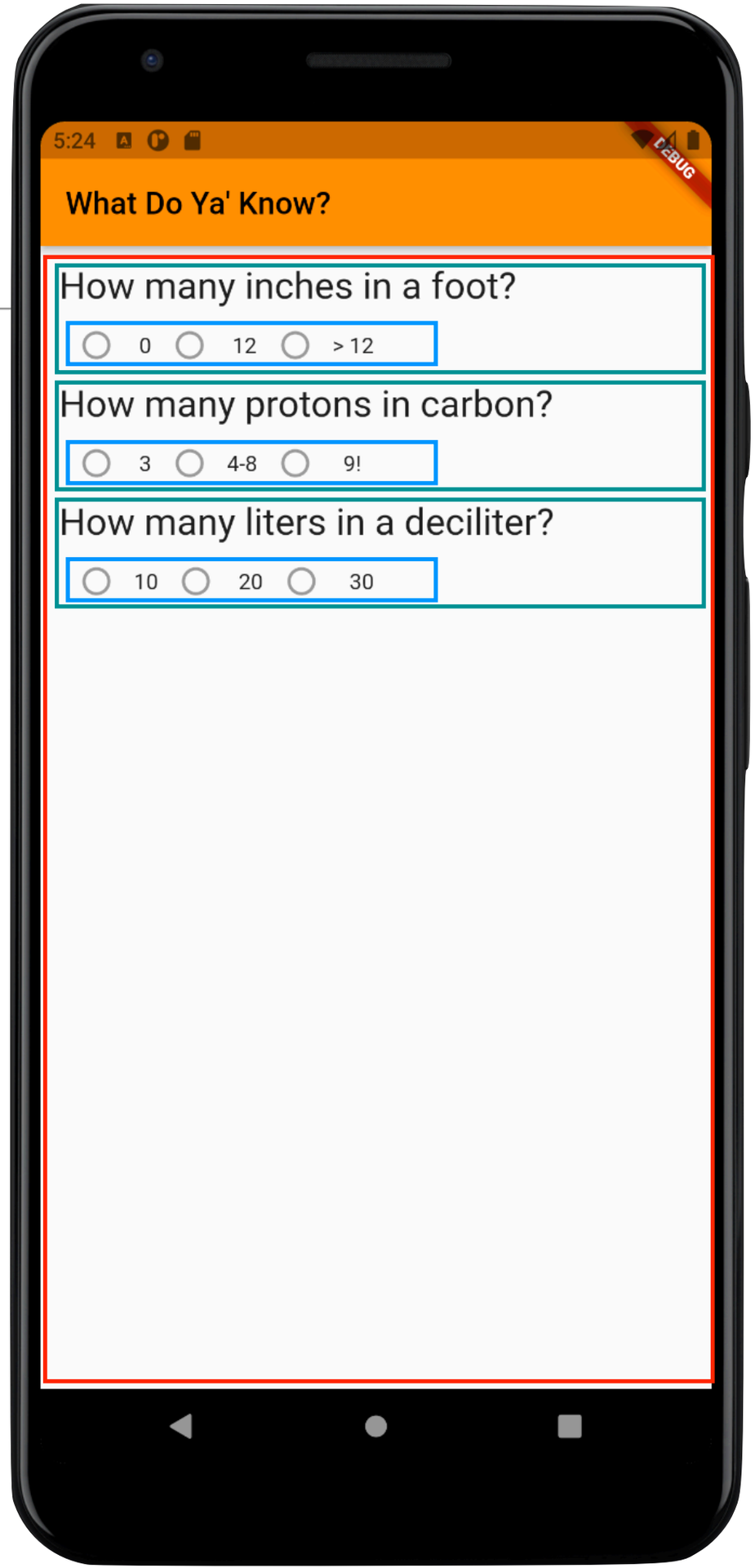






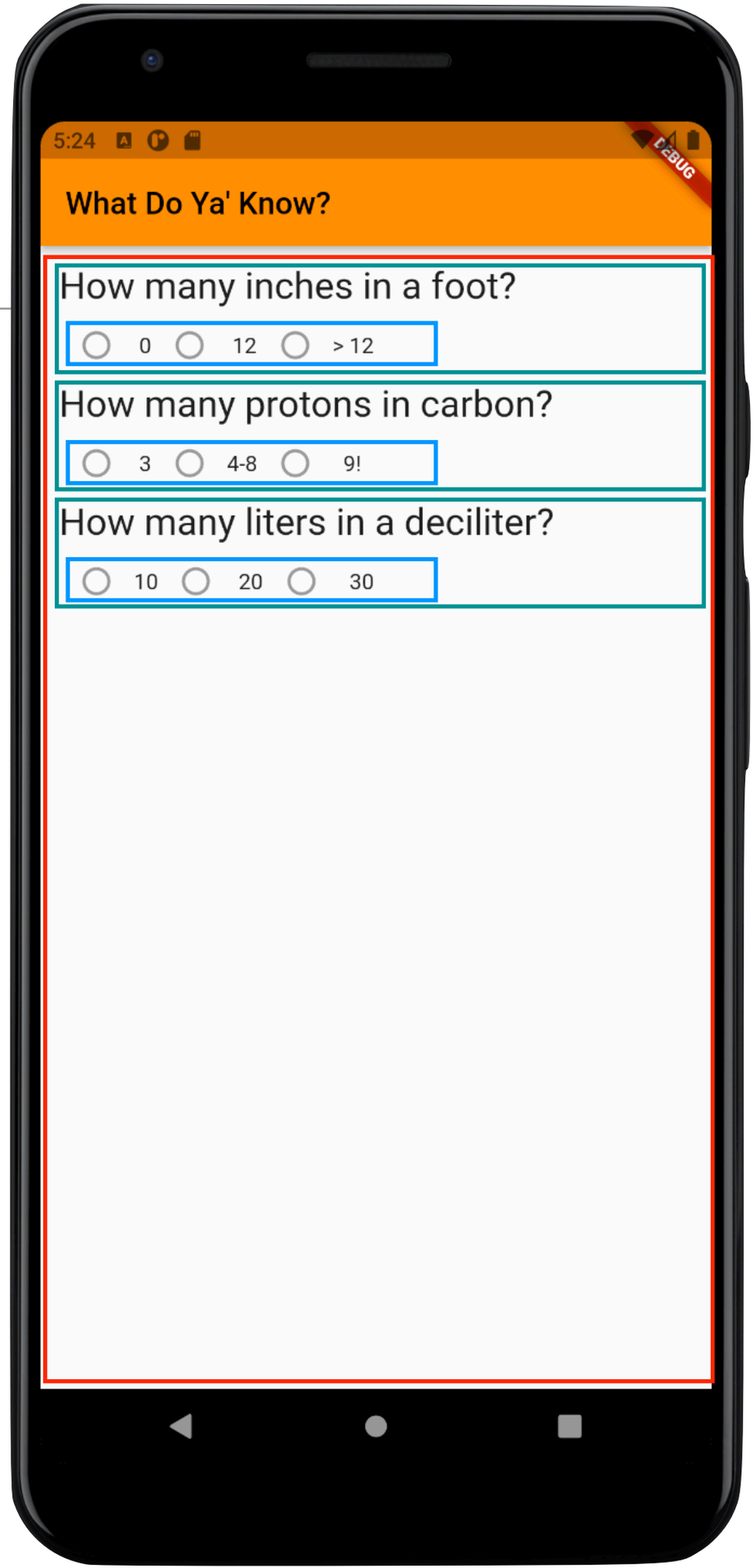
Exercise Solution (Conceptual)

- A column of questions
- Each question consists of a column consisting of
  - a Text widget
  - a row of 6 widgets (Radio widgets + Texts) beneath



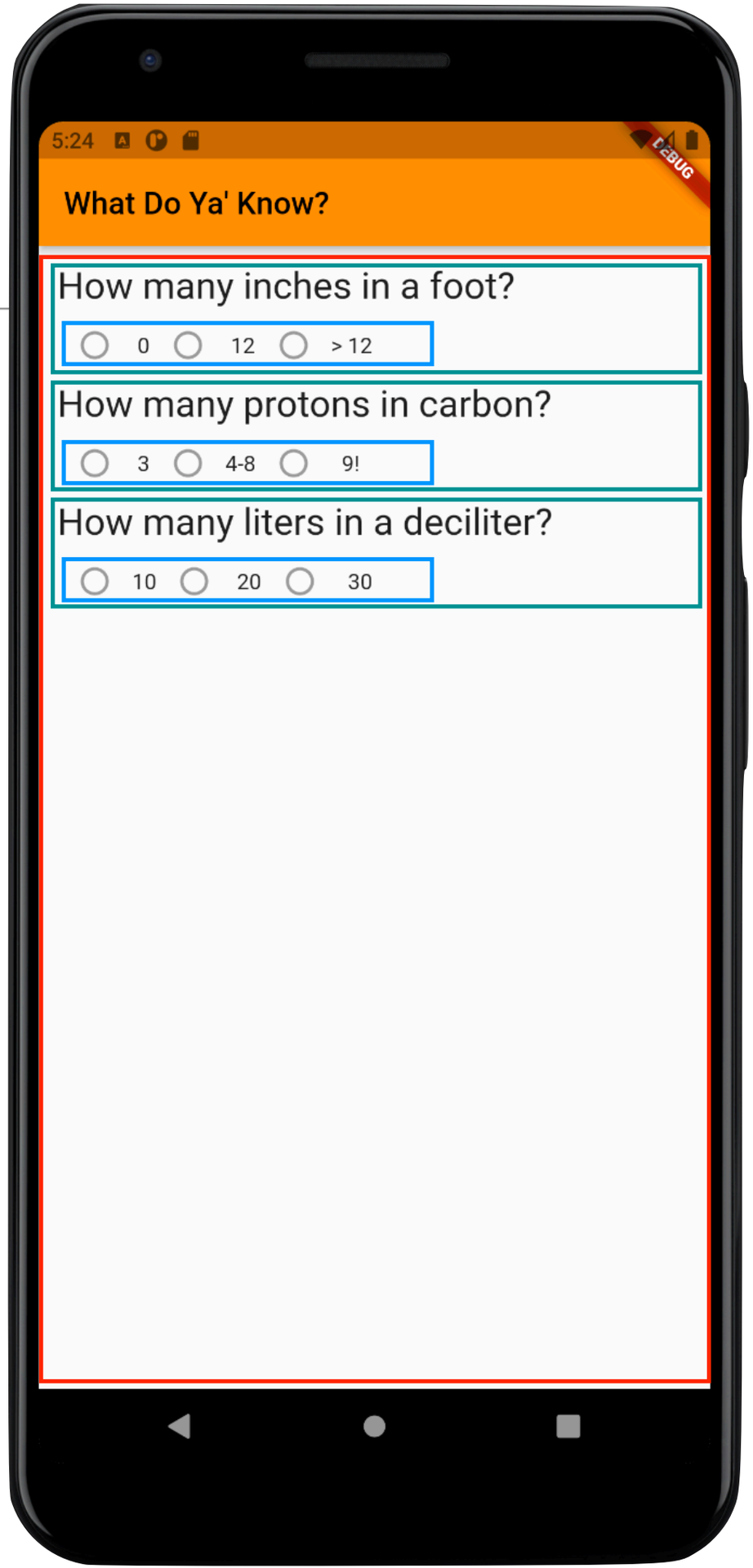
Exercise Solution (Code)

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

  @override
  Widget build(BuildContext context) {
    return Column(children: [
      MultipleChoiceQuestion(
          question: "How many inches in a foot?",
          answers: [' 0', ' 12', '> 12']),
      MultipleChoiceQuestion(
          question: "How many protons in carbon?",
          answers: [' 3', '4-8', '  9!']),
      MultipleChoiceQuestion(
          question: "How many liters in a deciliter?",
          answers: ['10', ' 20', '  30'])
    ]);
  }
}
```



Exercise Solution (Code)

```
class MultipleChoiceQuestion extends StatelessWidget {
  final String question;
  final List<String> answers;
  int? _choice = 0;
  const MultipleChoiceQuestion(
      {Key? key, required this.question, required this.answers})
      : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Column(crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Text(
        question,
        style: TextStyle(
          fontSize: 24.0,
        ),
        textAlign: TextAlign.start,
      ),
      Row(children: [
        Radio(value: 0, groupValue: _choice, onChanged: null),
        Text(widget.answers[0]),
        Radio(value: 1, groupValue: _choice, onChanged: null),
        Text(widget.answers[1]),
        Radio(value: 2, groupValue: _choice, onChanged: null),
        Text(widget.answers[2]),
      ]),

    ]);
  }
}
```



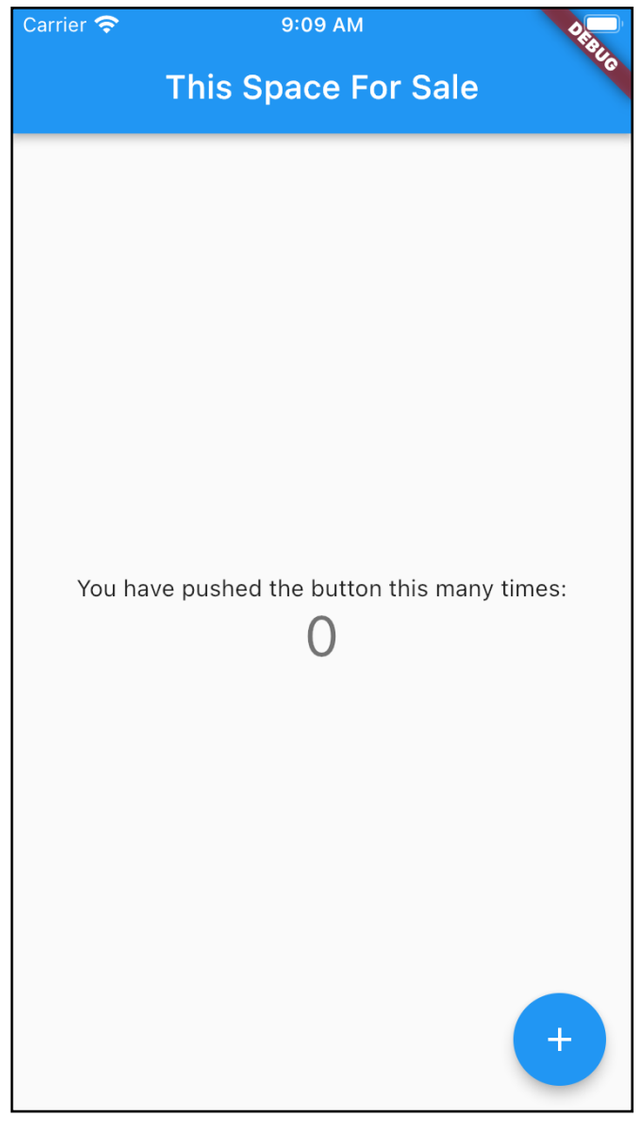
Stateful Widgets

Exercise

- Run this code and see what happens (or ... doesn't 😉)

```dart
class MyHomePage extends StatelessWidget {
  MyHomePage({Key? key}) : super(key: key);

  int _counter = 0;

  void _incrementCounter() {
    _counter++;
    print('Behold, _counter is $_counter');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('This Space For Sale'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

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

void main() {
  runApp(const MyApp());
}

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
              primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

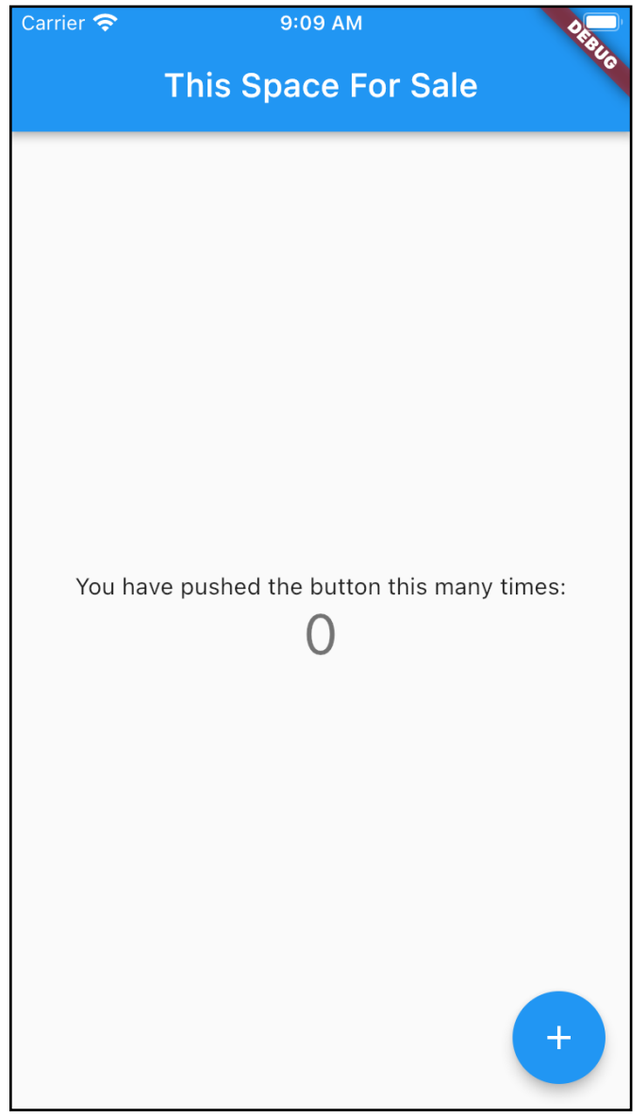


https://dartpad.dev/a0096ccdb6425ee1c2734b48a83d594e

Exercise Solution

- Tapping the button incremented _counter: the widget still showed 0
- That is because widgets are inherently immutable and cannot change
- Q: How can you build a UI out of elements that don't change? 🤔

```
Restarted application in 305ms.
flutter: Behold, _counter is 1
flutter: Behold, _counter is 2
flutter: Behold, _counter is 3
flutter: Behold, _counter is 4
flutter: Behold, _counter is 5
flutter: Behold, _counter is 6
```



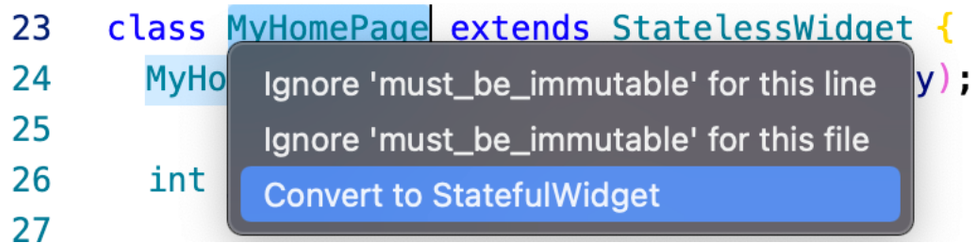
Stateful Widgets 🧐

- A: Recreate the widgets when necessary
- A **stateful** widget is one whose appearance is based on **state** -- data that can change, part of the model that the widget / app depends on
- Whenever state changes, a **brand new widget** is created to reflect new values
- This is called a **declarative UI**, because all we do is *declare* what the UI (widgets) should look like for given values of state, and let the framework handle the rest
- State can be changed by something explicit that the user does, or by a multitude of other events (e.g., incoming data from the network)
- This eliminates a huge source of bugs in apps, because managing state in an app of any complexity is, well, complex. It is better to let the system handle it.

Exercise

- VSC: Starting with the stateless widget we just looked at,

  1.select MyHomePage



  2.type command/ctrl .

  3.choose Convert to Stateful Widget

- DartPad: There is no way to do this easily in DartPad, so the gist below shows a Stateful version of the Counting widget

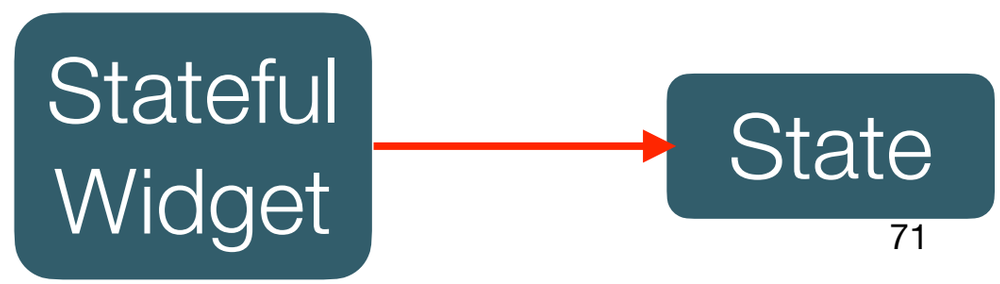
https://dartpad.dev/36eb87bb35387d05a46347c64f414fcc

A Tour of a Stateful Widget

- The Stateful Widget has to create its own state, via a method, createState(), that returns State

- A StatefulWidget's State must extend State<>, and the generic parameter must match the StatefulWidget's type

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return ... ;
  }
}
```



A Tour of a Stateful Widget, Continued

- Properties that represent a widget's state are declared and stored in the State objects
- The build() method has moved into State
- When those properties change, build() will be invoked to create a new widget, showing those updated values

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    _counter++;
    print('Behold, _counter is $_counter');
  }

  @override
  Widget build(BuildContext context) {
     // uses
```

Exercise & Solution

- Click on the button ...
- Nothing happened ☹️
- Need to wrap the _counter++ in a call to **setState()**

- Click on 

then on  again
- 😀

```
void _incrementCounter() {
  _counter++;
  print('Behold, _counter is $_counter');
}
```

```
void _incrementCounter() {
  setState(() {
    _counter++;
  });
  print('Behold, _counter is $_counter');
}
```

Exercise - The Ol' Magic8Ball

- Study and run the code at right
- Clicking on the Tell My Fortune button doesn't do anything. Why?
- Make the necessary changes so that it will.
- Hint: just look at _tellMyFortune()

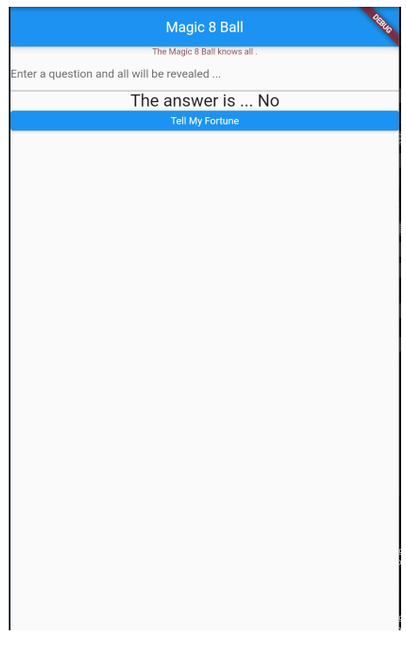
```dart
// ignore_for_file: avoid_print

import 'dart:math';

import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Magic8Ball(),
    );
  }
}

class Magic8Ball extends StatelessWidget {
  var fortunes = ['Yes', 'No', 'Future cloudy', 'Definitely yes'];
  var fortuneIndex = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Magic 8 Ball')),
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.stretch,
        // ignore: prefer_const_literals_to_create_immutables
        children: [
          const Text('The Magic 8 Ball knows all . ',
              style: TextStyle(
                  color: Color.fromARGB(255, 146, 83, 83), fontSize: 12),
              textAlign: TextAlign.center),
          const TextField(
              decoration: InputDecoration(
                  hintText: 'Enter a question and all will be revealed ...')),
          Text('The answer is ... ${fortunes[fortuneIndex]} ',
              style: TextStyle(fontSize: 24), textAlign: TextAlign.center),
          ElevatedButton(
              child: const Text('Tell My Fortune'), onPressed: _tellMyFortune),
        ],
      ),
    );
  }

  void _tellMyFortune() {
    fortuneIndex = Random().nextInt(fortunes.length);
    print('Now fortuneIndex is $fortuneIndex');
  }
}
```

https://dartpad.dev/36eb87bb35387d05a46347c64f414fcc

Exercise Solution

```dart
class Magic8Ball extends StatefulWidget {
  @override
  State<Magic8Ball> createState() => _Magic8BallState();
}

class _Magic8BallState extends State<Magic8Ball> {
  var fortunes = ['Yes', 'No', 'Future cloudy', 'Definitely yes'];

  var fortuneIndex = 0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Magic 8 Ball')),
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.stretch,
        // ignore: prefer_const_literals_to_create_immutables
        children: [
          const Text('The Magic 8 Ball knows all . ',
              style: TextStyle(
                  color: Color.fromARGB(255, 146, 83, 83), fontSize: 12),
              textAlign: TextAlign.center),
          const TextField(
              decoration: InputDecoration(
                  hintText: 'Enter a question and all will be revealed ...')),
          Text('The answer is ... ${fortunes[fortuneIndex]} ',
              style: TextStyle(fontSize: 24), textAlign: TextAlign.center),
          ElevatedButton(
              child: const Text('Tell My Fortune'), onPressed: _tellMyFortune),
        ],
      ),
    );
  }

  void _tellMyFortune() {
    setState(() {
      fortuneIndex = Random().nextInt(fortunes.length);
    });
    print('Now fortuneIndex is $fortuneIndex');
  }
}
```

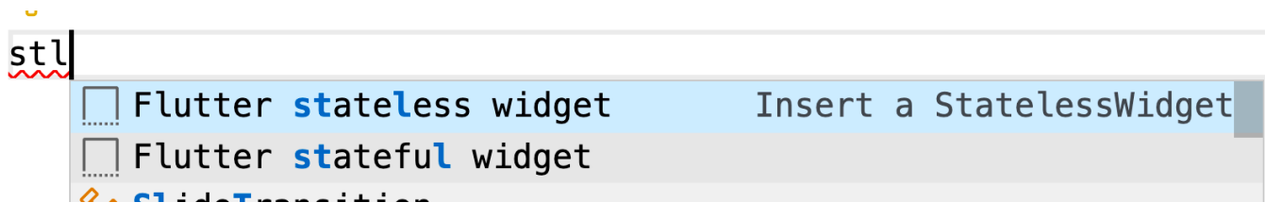
```dart
// ignore_for_file: avoid_print

import 'dart:math';

import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Magic8Ball(),
    );
  }
}
```



https://dartpad.dev/75db5ad9c0365c3291da51c9c998ceef

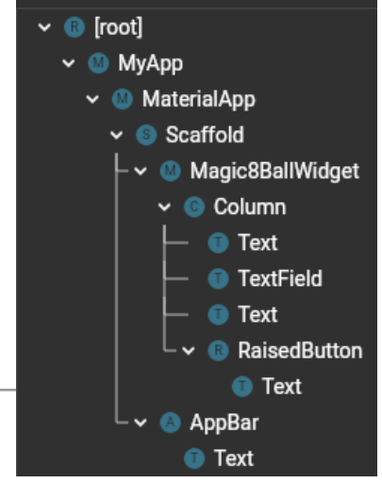
Creating a Stateful Widget from Scratch

- One way to create a stateful widget is to let VSC do the work.
- Type stl



- Choose Flutter stateful widget
- Type the name of your widget, e.g., LotteryTicket

Where To Place State?



- Right now, state is only visible in the Magic8Ball widget, and limited: it is just one int, representing a single answer to a single question
- But what if we want to keep a collection of questions and answers, and allow other parts of the app to see it?
- e.g., Suppose we wanted to create a 2-tabbed application
  - Tab 1: Magic8BallWidget gathers information (questions asked, answers given)
  - Tab 2: ResultsWidget displays *all* the questions asked and answers given
- We would need to put the state higher up in the widget tree, above the Magic8BallWidget and ResultsWidget, so that both could access it

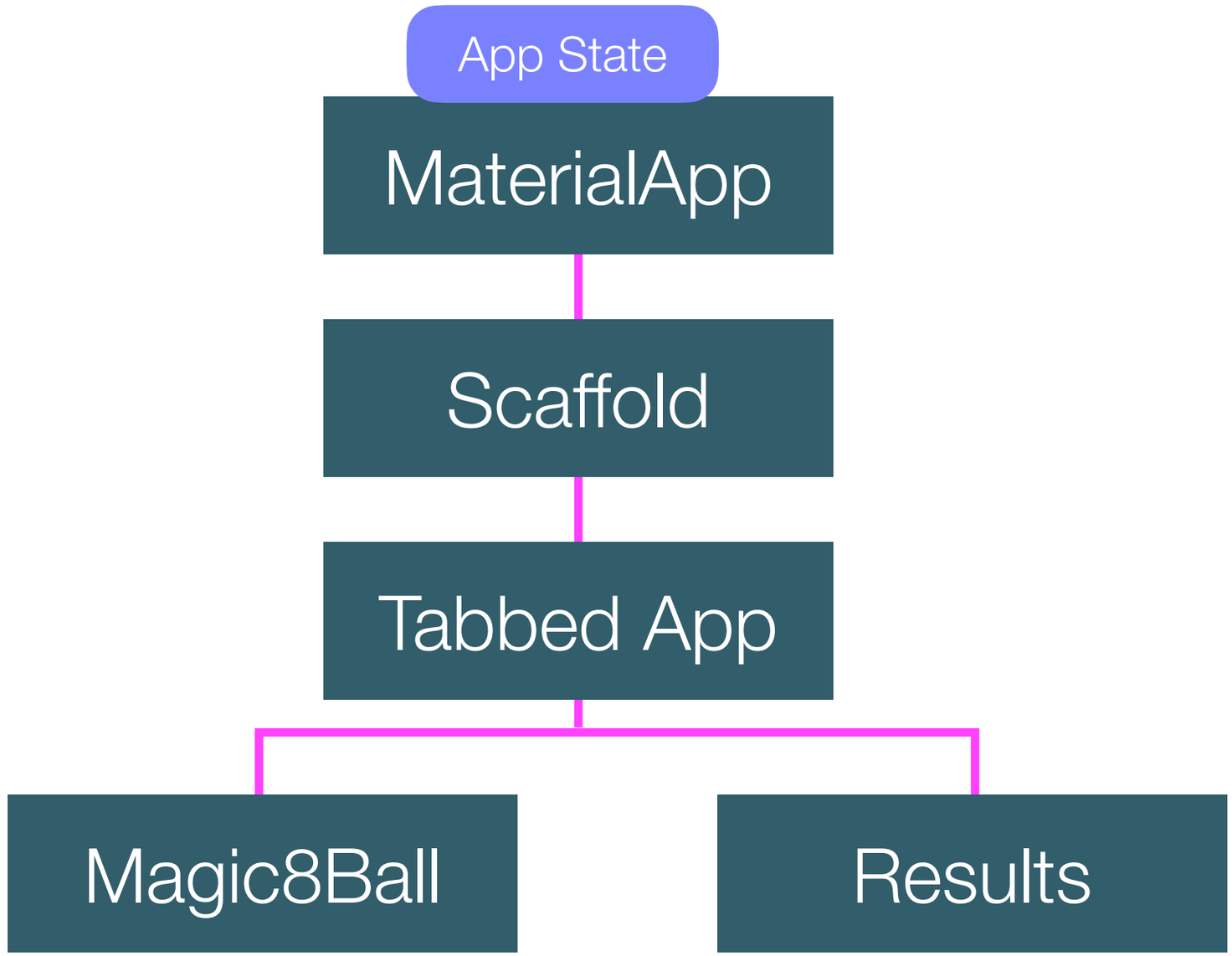
Elevating State

- Define app state -- data of interest to the entire app, not just one widget -- somewhere higher up in the tree

- The Magic8BallWidget could send information up the tree to add to it

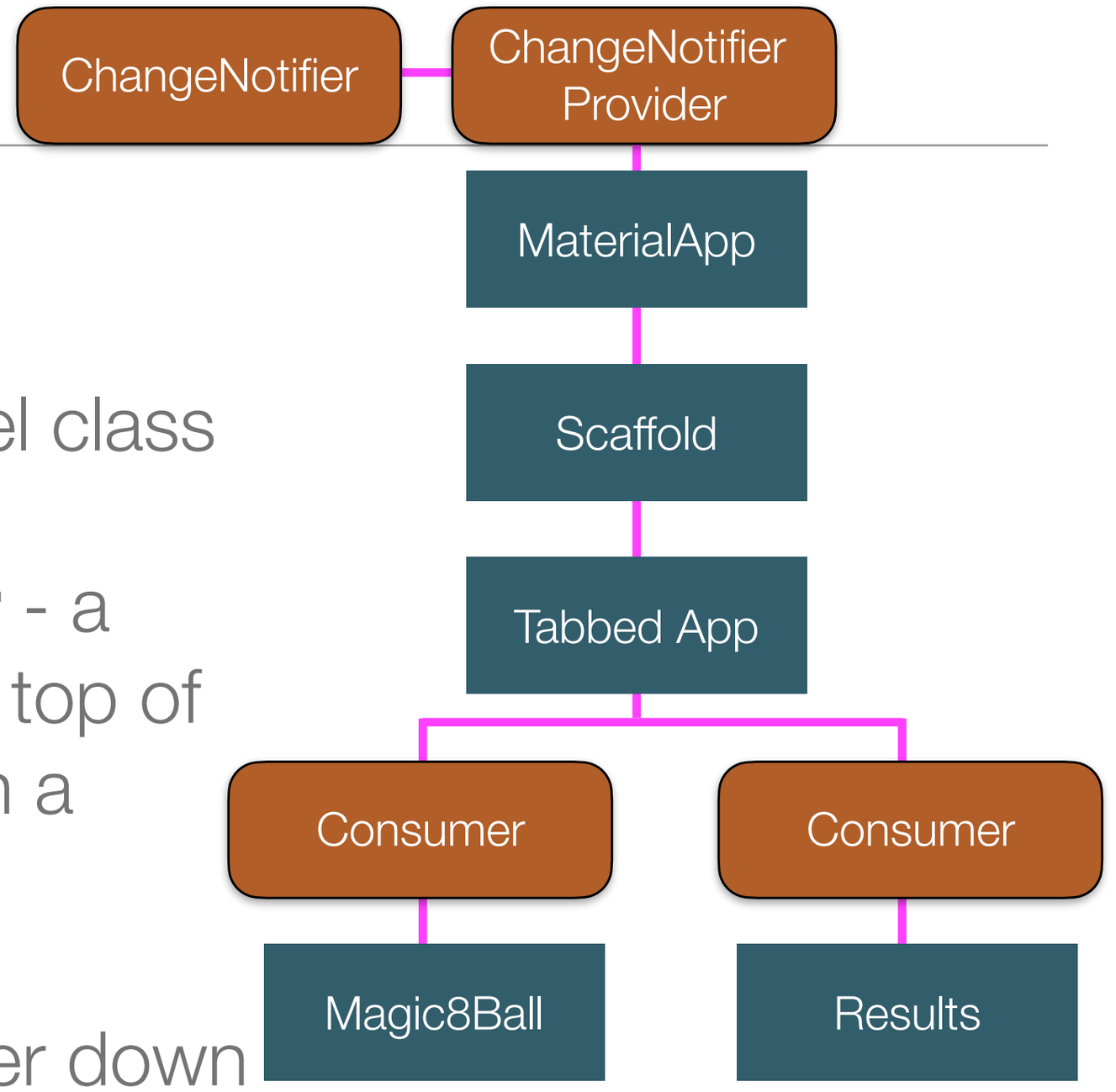
- The ResultsWidget could reach up to it to get its contents for display

- Flutter makes this relatively easy to do



State Management

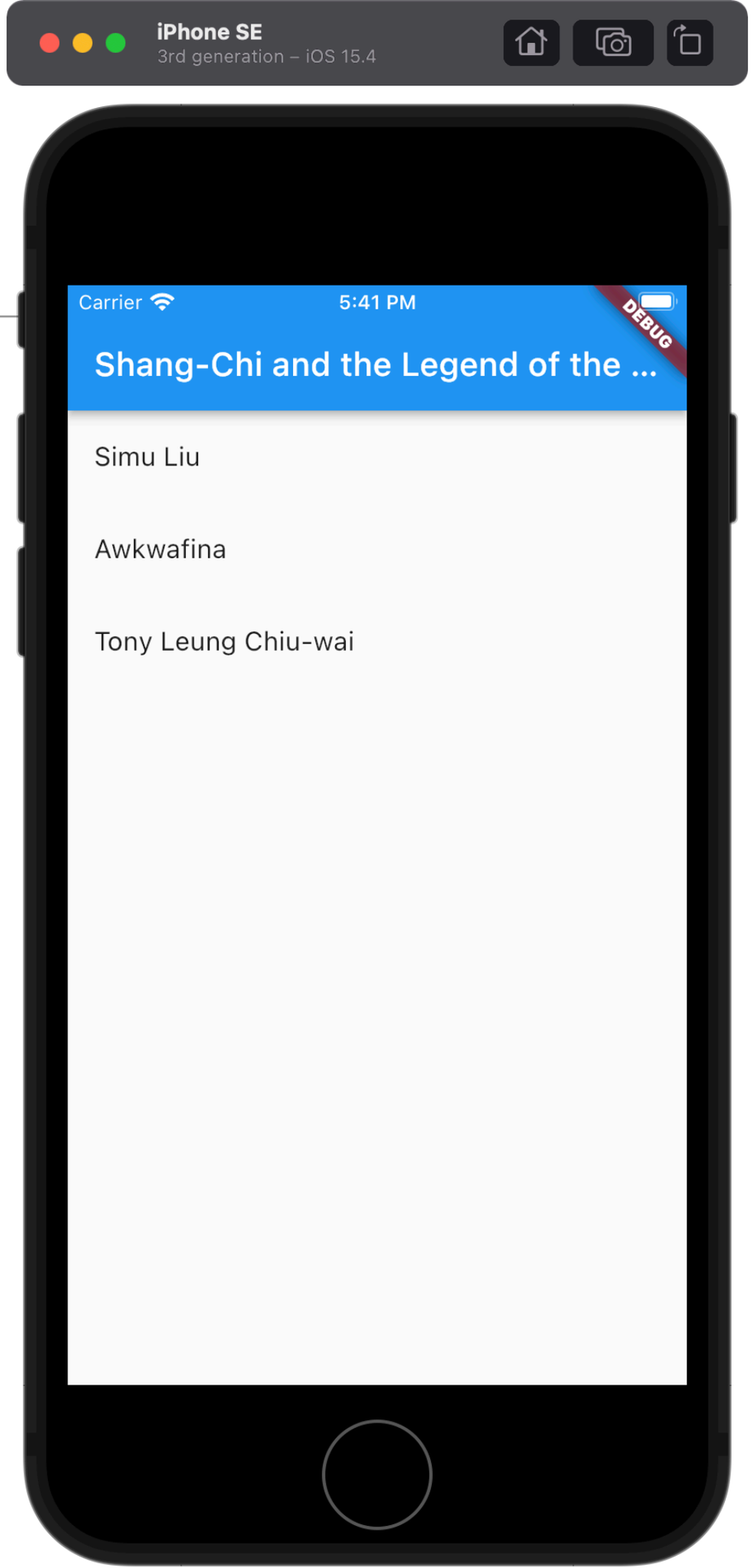
- Flutter provides 3 classes:
- **ChangeNotifier** - a model class
- **ChangeNotifierProvider** - a widget, placed at the very top of the tree, and supplied with a ChangeNotifier
- **Consumer** - widgets lower down in the widget tree that are notified, by the change notifier



ListViews - Static Values

- ListViews / Tables are ubiquitous in mobile apps
- ListViews with static content are simple:

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Shang-Chi and the Legend of the Ten Rings')),
      body: ListView(children: [
        ListTile(title: Text('Simu Liu')),
        ListTile(title: Text('Awkwafina')),
        ListTile(title: Text('Tony Leung Chiu-wai')),
      ]),
    );
  }
}
```



ListViews - Slightly Less Static Values

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

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        title: 'Flutter Demo',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: MovieCastB(cast: ['Simu Liu', 'Awkwafina', 'Tony Leung Chiu-wai']));
  }
}

class MovieCastB extends StatelessWidget {
  List<String>? cast;
  MovieCastB({required List<String> cast, Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Shang-Chi and the Legend of the Ten Rings')),
        body: ListView.separated(
          itemCount: cast!.length,
          itemBuilder: (context, index) => ListTile(title: Text(cast[index])),
          separatorBuilder: (context, index) => Divider(color: Colors.red[500], thickness: 1.0, height: 1.0),
        ));
  }
}
```

Flutter Strengths

- Awesome documentation: entertaining, well-documented, unsurprisingly easy to search 😀
- Google is putting serious resources behind this
- The language is magnificent: Dart just "gets out of the way"
- Developing in VSC using the Flutter extension is a sublime experience
- Apps are truly fast
- Integration with Firebase is tight and straightforward
- All the other advantages listed previously

Flutter Weaknesses

- Flutter leverages the Android and iOS SDKs, and issues with the latter may be difficult to diagnose

- The initial installation -- of three different SDKs -- is onerous

- iOS developers must be on Macs

- Material v. Cupertino Widgets

- The layout system can be difficult to grok, and you can't drag-and-drop the UI (applies to all declarative UI systems)

Flutter in the Classroom

- Taught as part of a special topics course
- Taught as a full Mobile Application Development course
- Went well in both cases
- Contented and only occasionally confused students (the same applies to the instructor)