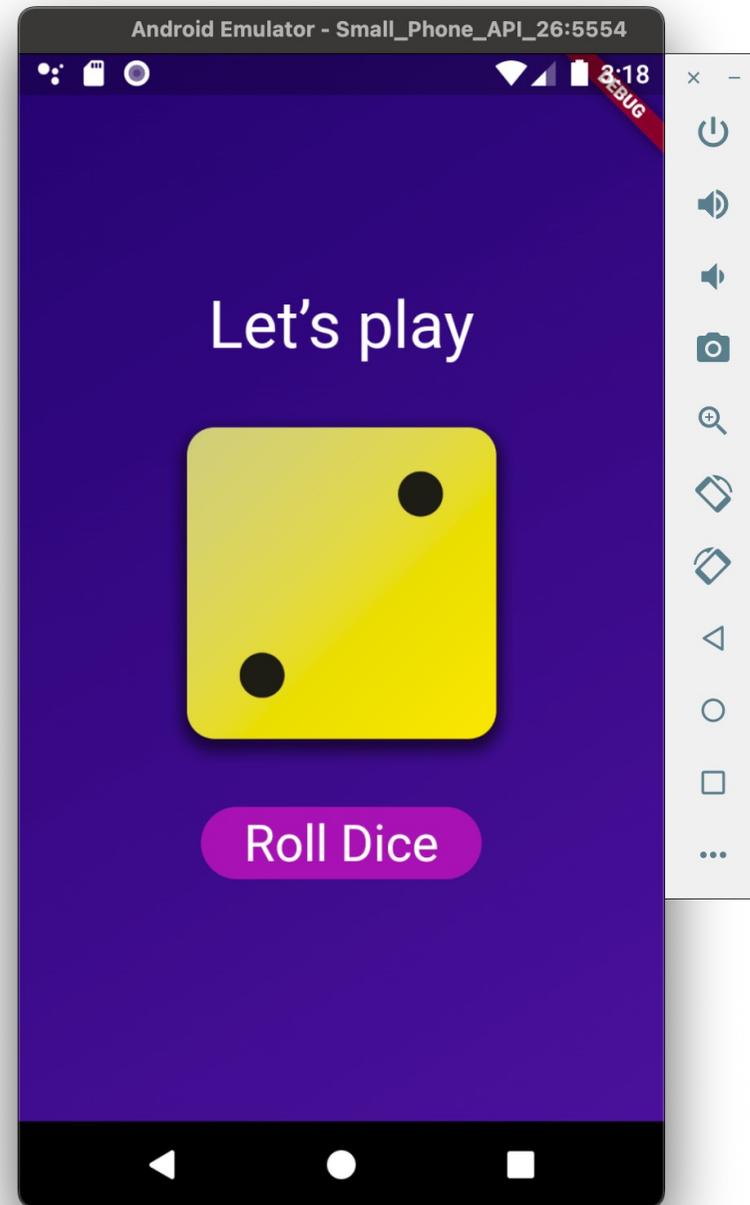


Basics of Dart & Flutter: Part I

Shan-Hung Wu
CS, NTHU

Let's Roll Dice



Today's Topics

- Dart language
- Flutter app overview
- Stateless vs stateful widgets

Today's Topics

- Dart language
- Flutter app overview
- Stateless vs stateful widgets

Evolution of Dart

- Dart 1.0 (Nov 2013)
 - Initially introduced as an alternative to JavaScript
 - [Pub](#) package manager
- Dart 2.0 (Aug 2018)
 - Strong type system
 - Sound null safety
 - Supports Flutter!
- Dart 3.0 (May 2023)
 - Null Safety by default
 - Unified dev workflow across different platforms

Dart Features

- Platform-independent (Windows, Mac, Linux, and Web)
 - Just-In-Time (JIT) compilation in development
 - Runs in VM; offering *hot reload*
 - Ahead-Of-Time (AOT) compilation in production
 - Native code or Web Assembly; high performance
- Auto memory management with *Garbage Collection* (GC)
- *Function as first-class citizen*
- *Sound null safety*
- *Object-oriented*
 - Supports encapsulation, inheritance, polymorphism, interface, extension, etc.
- *Async, await, and concurrency* (Isolates)
- Foreign Function Interface (FFI)
- Free and open-source

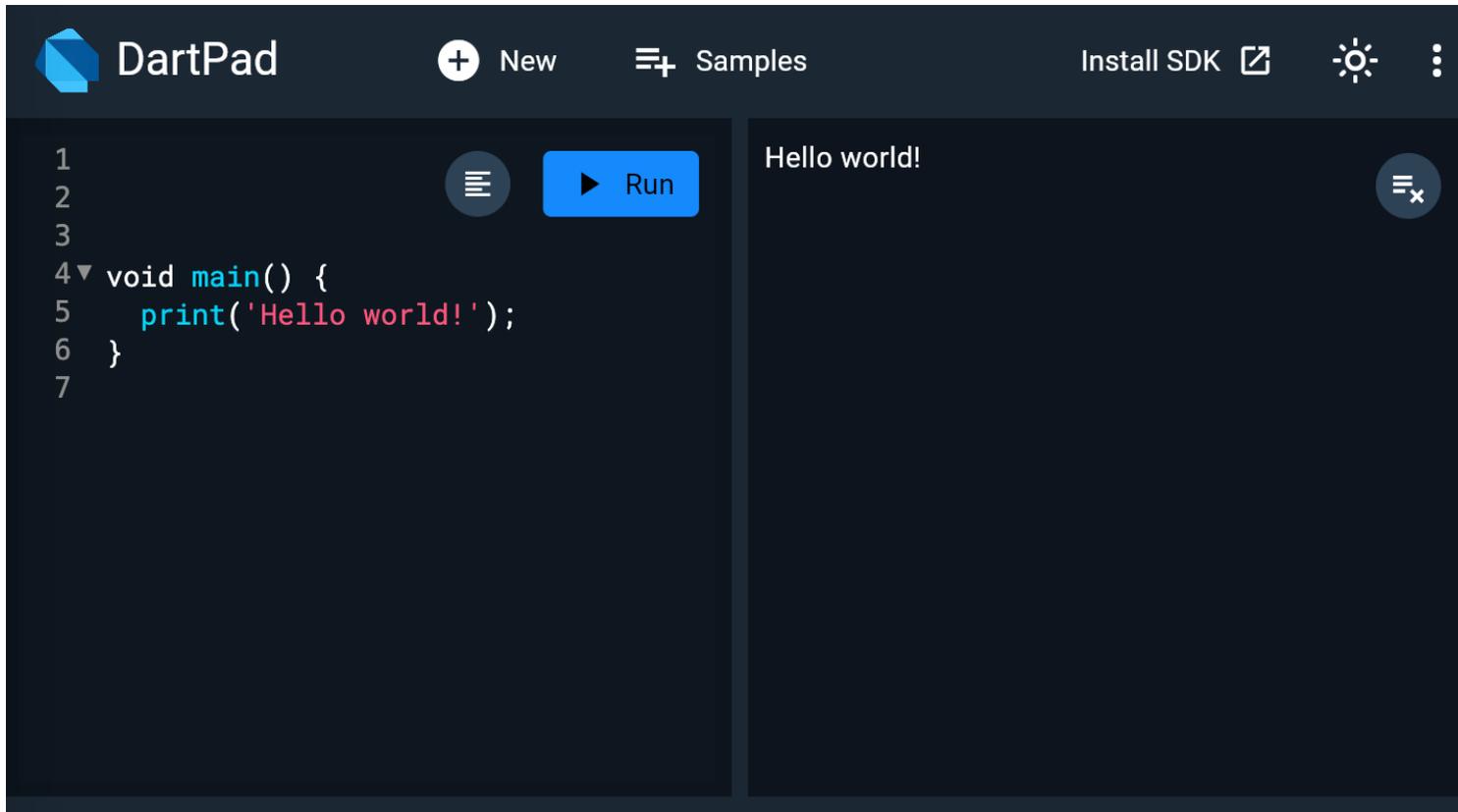
Warm up:

What's the difference between
statements & expressions?

Terms Revisited

- **Statement**: command that ends with “;”
 - `print('Hello world!');`
- **Expression**: command evaluated to a single value
 - `'Hello ' + 'world!'`
- **Keyword**: word reserved for compiler
 - `int, String, if, for, static, final, etc.`
- **Identifier**: name of variable, function, class, etc.
 - `int age;`
- **Literal**: value directly written in source code
 - `double pi = 3.14;`

Hello World!



The screenshot shows the DartPad web editor interface. The top navigation bar includes the DartPad logo, a '+ New' button, a 'Samples' menu, an 'Install SDK' link, and settings icons. The main editor area is split into two panels. The left panel contains a code editor with the following Dart code:

```
1
2
3
4 void main() {
5   print('Hello world!');
6 }
7
```

Next to the code is a 'Run' button. The right panel displays the output of the code, which is 'Hello world!'.

- Let's run it on [DartPad](#) (Dart → JavaScript):

Variables

```
void main() {
    String firstName = 'John';
    var lastName = 'Smith';
    String address = 'USA';
    int age = 20;
    double height = 5.9;
    bool isMarried = false;

    // String interpolation
    print('Name is $firstName $lastName');
    print('Address is $address');
    print('Age is $age');
    print('Height is ${height + 20}');
    print('Married status is $isMarried');
}
```

- Use `var` with type inference
- Use `final` or `const` to declare fixed values:
`const pi = 3.14159;`

User Input

```
import 'dart:io';

void main() {
  print('Enter your name:');
  String? name = stdin.readLineSync();
  if (name != null) {
    print('Hello ${name}');
  }
}
```

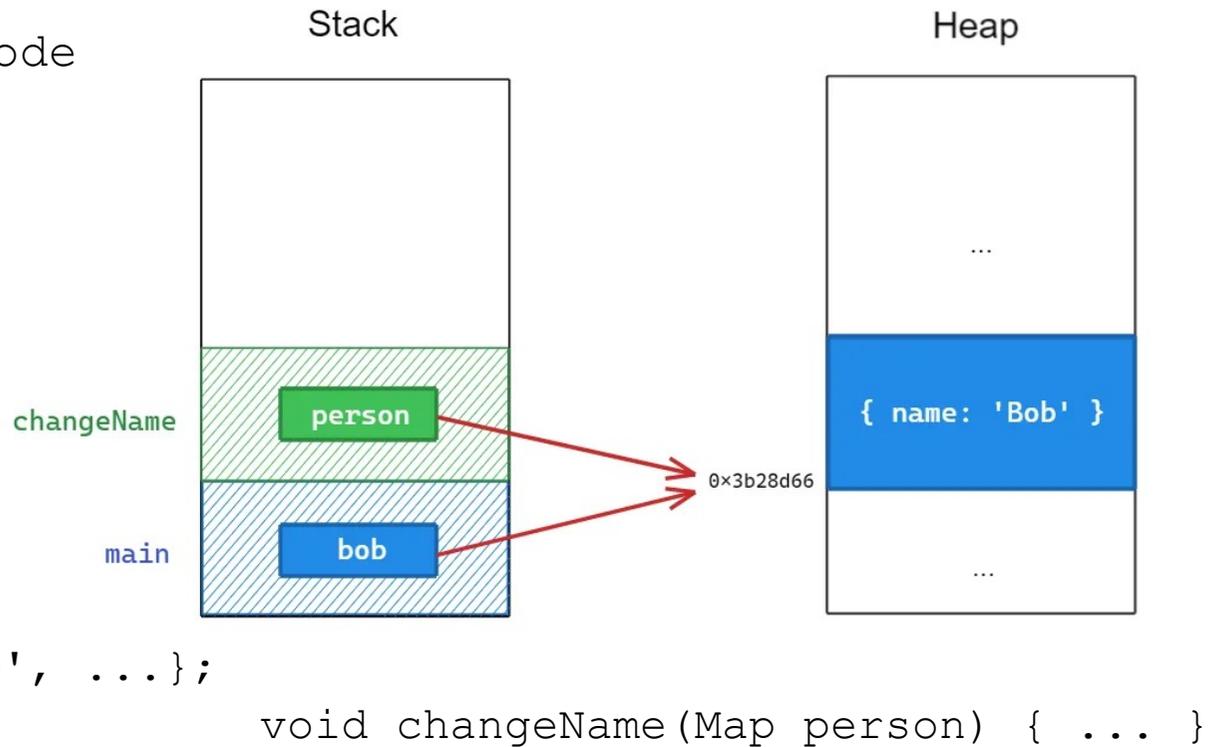
- Package `dart:io` provided by Dart
- Only runs in command line via `dart run`
- `String?` means “name could be null”
 - We will discuss this feature later

Built-in Data Types

```
bool
int
double
num // int or double
runs // String's Unicode
String
List
Set
Map
Function
```

```
void main() {
  var bob = {'name': 'Bob', ...};
  changeName(bob);
}
```

- All types are **object types**
 - Extending Object class
- Passed “by reference”, not “by value”

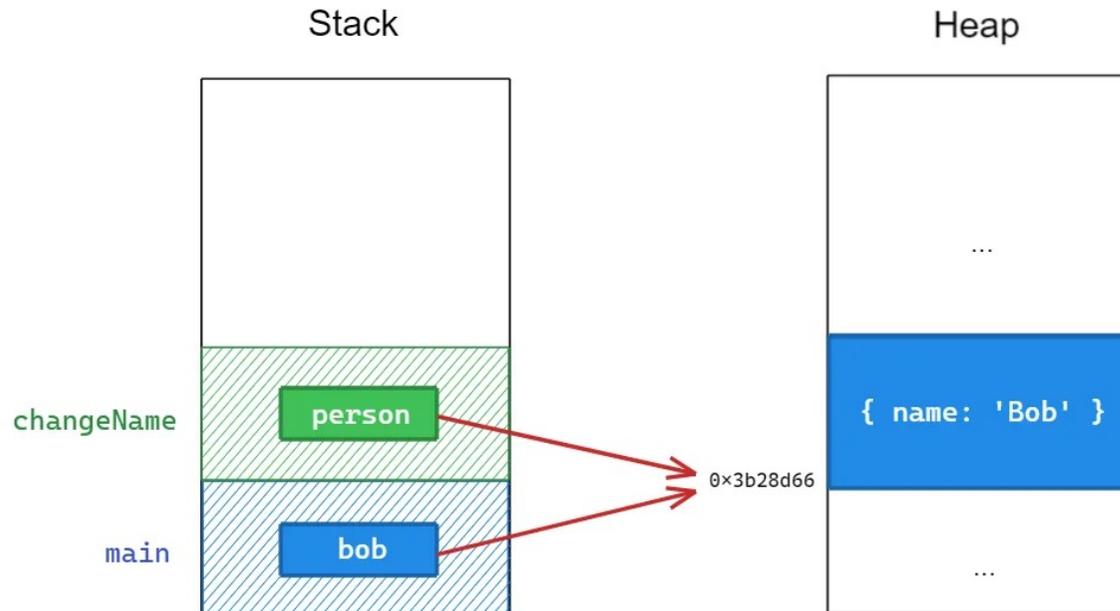


Immutable Types

- `bool`, `int`, `double` and `String` types are *immutable*
- `'Hello ' + 'world!'` creates new `String` object
- In effect, these types *behave like value types* in other languages
- Why not just use value types?
- Object types can have methods:

```
const pi = 3.14159;  
print('Pi is ${pi.toStringAsFixed(2)}');
```

Memory Management



- In Dart, the **garbage collector** periodically finds “unused” objects in heap and frees their memory
 - No need to call `free()` as in C
- What “unused” objects?
- Those not referenced by variables in stack

Lists, Maps, and Generics

- List (array):

```
List<int> myList = [1, 2, 3];  
myList.add(4); // Adds 4 to the end  
myList[0] = 10;  
myList.removeAt(0);  
print(myList[0]); // Prints what?
```

- Map:

```
Map<String, String> myMap = {  
  'lang': 'Dart',  
  'client': 'Flutter',  
  'server': 'Firebase'  
};  
myMap['store'] = 'Google Play';  
myMap['lang'] = 'Dart language';  
myMap.remove('server');  
print(myMap['lang']); // Prints what?
```

Iterating Lists & Maps

- **List:**

```
for (var e in myList) {  
    print(e);  
}
```
- **Map:**

```
for (var e in myMap.entries) {  
    print('${e.key}: ${e.value}');  
}  
  
for (var k in myMap.keys) {  
    print(k);  
}  
  
for (var v in myMap.values) {  
    print(v);  
}
```

Functions

```
void add(num num1, num num2) {  
    // num1 and num2 are parameters  
    num sum = num1 + num2;  
    print('The sum is $sum');  
}
```

```
void main() {  
    // 10 and 20 are arguments  
    add(10, 20);  
}
```

Parameters & Default Values

```
// Optional positional parameters
void sayMessage(String message, [String? author]) {
  print("$message by ${author ?? 'Anonymous user'}");
}

// Named parameters & default values
void setDimensions({int width = 10, int height = 10}) {
  print("Width: $width, Height: $height");
}

void main() {
  sayMessage('Hello, Dart!');
  setDimensions(height: 20, width: 30);
}
```

- Named param is optional by default
 - Use `required` to make it mandatory

Generics in Functions

```
// Returns the first element of any list
T firstElement<T>(List<T> list) {
    if (list.isEmpty) {
        throw Exception('The list is empty');
    }
    return list.first;
}
```

```
// Returns the larger argument
T maxValue<T extends Comparable>(T a, T b) {
    return a.compareTo(b) > 0 ? a : b;
}
```

- Helps ensure type safety

Multiple Return Values

```
final json = <String, dynamic>{ // "dynamic" means "any type"  
  'name': 'Dash',  
  'age': 10,  
  'color': 'blue',  
};
```

```
// Returns multiple values in a record:  
(String, int) userInfo(Map<String, dynamic> json) {  
  return (json['name'] as String, json['age'] as int);  
}
```

```
// Destructures using a record pattern:  
var (name, age) = userInfo(json);  
print(name);  
// Or access by index  
print(userInfo(json) .$1);
```

Arrow Functions

- If a function consists of just one line of code that returns a value, it has simpler syntax:

```
// Normal function  
int add(int a, int b) { return a + b; }
```

```
// Arrow function  
int add(int a, int b) => a + b;
```

- Don't use arrow if function doesn't return anything

Functions as First-Class Citizens

```
// Basic operations
int add(int a, int b) => a + b;
int multiply(int a, int b) => a * b;

// Higher-order function that accepts functions as params
void printOPResult(int a, int b, int Function(int, int) op) {
    var result = op(a, b);
    print('Result: $result');
}

void main() {
    // Assigning functions to variables
    int Function(int, int) op = add;
    printOPResult(4, 2, op);
    op = multiply;
    printOPResult(4, 2, op);
}
```

- Functions can be assigned to variables and passed around

Anonymous Functions

- E.g., list manipulation:

```
for (int e in myList) {  
    print(e);  
}
```

```
myList.forEach((e) {  
    print(e);  
});
```

```
// Multiply each el by 2  
var myList2 = myList.map(  
    (e) => e * 2  
) .toList();
```

```
// Filter list  
var myList3 = myList.where(  
    (e) => e > 0  
) .toList();
```

- E.g., map manipulation:

```
for (var k in myMap.keys) {  
    print(k);  
}
```

```
myMap.keys.forEach((k) {  
    print(k);  
});
```

```
var entries = myMap.map(  
    (k, v) =>  
        MapEntry(..., ...)  
) ;  
var myMap2 = Map<..., ...>  
    .fromEntries(entries);
```

Variables are Block-scoped

```
void outerFunction() {  
    var outerStr = "I'm outside!";  
  
    void innerFunction() {  
        var innerStr = "I'm inside!";  
        print(outerStr); // Accessible in nested func!  
    }  
  
    // print(innerStr); // Error: undefined 'innerStr'  
}
```

- Lexical scoping: variable's scope is determined at compile time, not at runtime (dynamic scoping)

Closures & Captured Variables

```
Function makeAdder(int base) {  
    return (int i) => base + i; // return func  
}
```

```
void main() {  
    var addFrom2 = makeAdder(2);  
    var addFrom3 = makeAdder(3);  
    print(addFrom2(10)); // Output: 12  
    print(addFrom3(10)); // Output: 13  
}
```

- **Closures** are functions that capture and retain variables from their lexical scope
- **Captured variables** stay in memory even after their lexical scope ends!

No Bad Closures in Loops*

```
List<Function> createCounters() {
    var counters = <Function>[];
    for (var i = 0; i < 3; i++) {
        counters.add(() => i);
    }
    return counters;
}

void main() {
    var counters = createCounters();
    for (var counter in counters) {
        print(counter()); // Prints 0, 1, 2
    }
}
```

- JavaScript prints “3, 3, 3” (bad closure)
- Dart prints “0, 1, 2”
 - Each closure in loop captures its own copy of **i**

Classes & Custom Types

```
class Person {
    // Instance variables
    String name;
    int age;

    // Constructor
    Person({required this.name, required this.age});

    // Method
    void displayInfo() {
        print('Name: $name, Age: $age');
    }
}

void main() {
    var alice = Person('name': 'Alice', age': 20); // Instance
    alice.displayInfo(); // Output: Name: Alice, Age: 20
    var bob = Person('name': 'Bob', age': 21); // Instance
    bob.displayInfo(); // Output: Name: Bob, Age: 21

    assert(alice is Person); // No AssertionError thrown
}
```

- Class as the blueprint (type) of your custom objects

Named Constructors

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

    // Constructor
    Person({required this.name, required this.age});

    // Named constructor
    Person.fromMap(Map<String, dynamic> data)
        : name = data['name'],
          age = data['age'];
    ...
}

void main() {
    var map = {'name': 'Dave', 'age': 25};
    var person = Person.fromMap(map);
    person.displayInfo();
}
```

Inheritance

(1/3)

```
class Student extends Person {
    String school;

    Student({
        required super.name,
        required super.age,
        required this.school
    });

    Student.fromMap(Map<String, dynamic> data)
        : school = data['school'],
          super.fromMap(data);

    @override
    void displayInfo() {
        super.displayInfo();
        print('School: $school');
    }

    void study() { ... } // custom method
}
```

Inheritance

(2/3)

```
class Employee extends Person {
    String company;

    Student({
        required super.name,
        required super.age,
        required this.company
    });

    Student.fromMap(Map<String, dynamic> data)
        : company = data['company'],
          super.fromMap(data);

    @override
    void displayInfo() {
        super.displayInfo();
        print(Company: $company);
    }

    void work() { ... } // custom method
}
```

Inheritance (3/3)

```
void main() {
    Person alice = Student(..., school: ...);
    alice.displayInfo(); // with "School: ..." (polymorphism)

    assert(alice is Person);
    assert(alice is Student);
    assert(alice is! Employee);

    print(alice.runtimeType); // Output: 'Student'

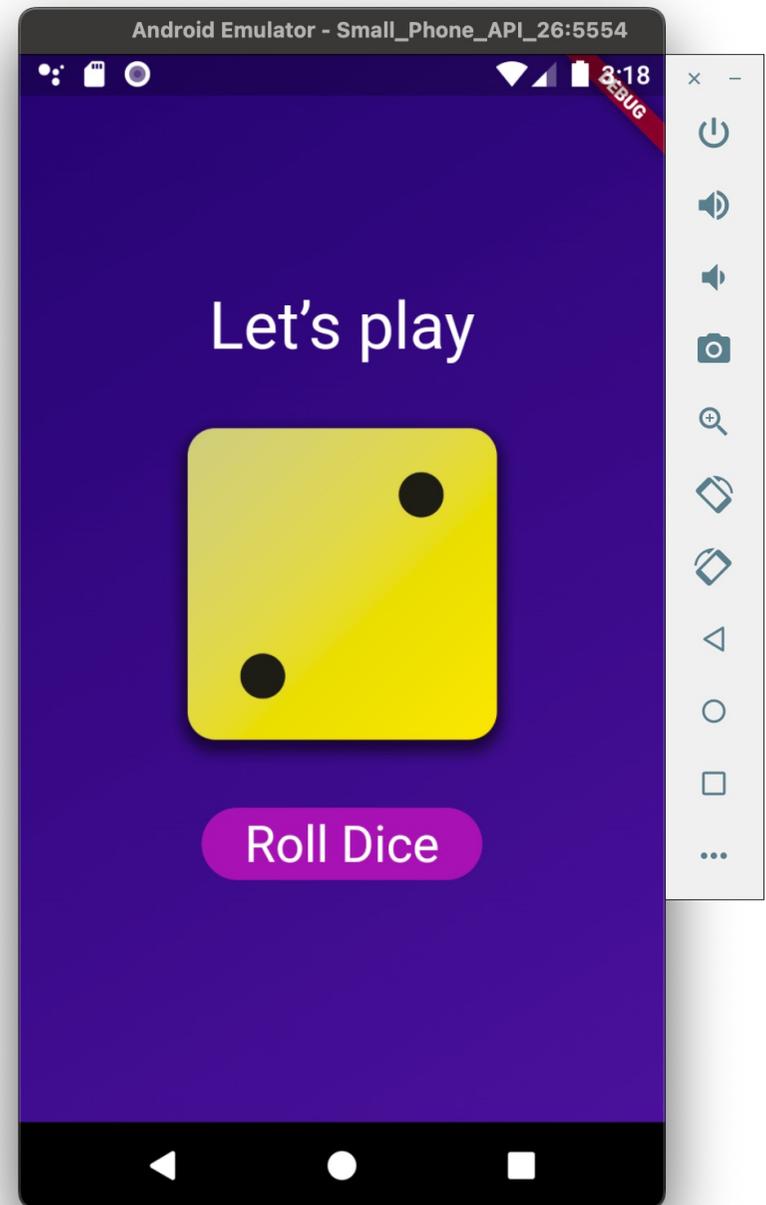
    var studentAlice = alice as Student;
    studentAlice.study();
}
```

Today's Topics

- Dart language
- Flutter app overview
- Stateless vs stateful widgets

Entering Flutter

- Project structure
- `lib/main.dart`
- `MaterialApp` widget
- `build()` in widgets
 - Defines widget hierarchy
- `Button` and `rollDice()` callback in `dice_roller.dart`
- `Random()` and dynamic image file name



Wait, the App Doesn't Work

- When button pressed, dice image doesn't change as expected
- The `rollDice()` callback is indeed invoked
 - We can verify this by inserting `print(currentDiceRoll)` into its body
- What's wrong?

Today's Topics

- Dart language
- Flutter app overview
- Stateless vs stateful widgets

Stateless vs. Stateful Widgets

```
class CounterWidget extends StatefulWidget {
  const CounterWidget({super.key});

  @override
  _CounterWidgetState createState() => _CounterWidgetState();
}
class _CounterWidgetState extends State<CounterWidget> {
  int _counter = 0; // Initial counter value

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Text('Counter: $_counter');
  }
}
```

- `setState()` tells Flutter to rerun `build()` and updates UI

References

- [Introduction to Dart](#)