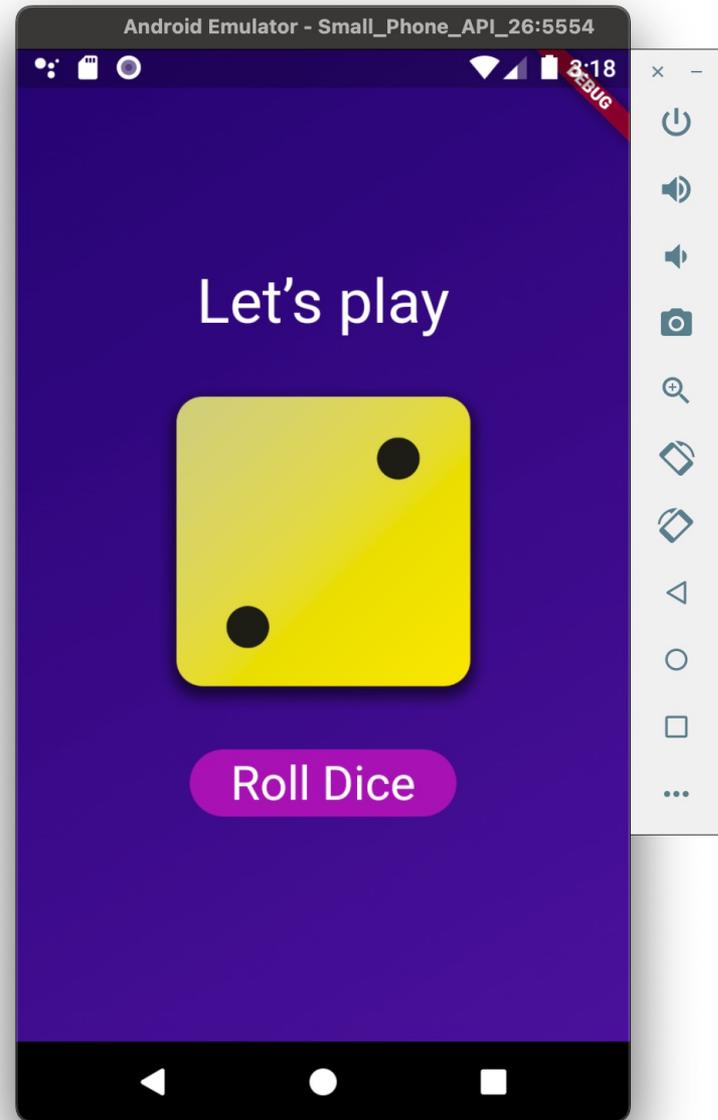


# Basics of Dart & Flutter: Part II

Shan-Hung Wu  
CS, NTHU

Questions?



# Final vs. Const

```
void main() {  
  final date = DateTime.now(); // OK  
  const date = DateTime.now(); // Error  
  
  const pi = 3.14; // OK  
  final gravity; // Error: must be initialized  
  final gravity = Gravity();  
  gravity.setForce(...) // OK  
}
```

- A `final` variable refers to only one object
  - Referenced object can still be mutable
- A `const` variable refers to a compile-time constant object
  - **Canonicalized**: only one copy in memory
  - Const widgets can be rebuilt faster in Flutter

# Immutable Classes/Widgets

```
class ImmutablePoint {
    final double x; // or any immutable type
    final double y;

    // Const constructor
    const ImmutablePoint(this.x, this.y);
}
```

- Try make **const** Widget yourself

```
void main() {
    const point1 = ImmutablePoint(2, 3);
    const point2 = ImmutablePoint(2, 3);
    print(point1 == point2); // Outputs: true
    print(identical(point1, point2)); // Outputs: true

    final point3 = ImmutablePoint(2, 3);
    final point4 = ImmutablePoint(2, 3);
    print(point1 == point2); // Depends on if "==" is overridden
    print(identical(point1, point2)); // Outputs: false
}
```

# Object or Dynamic?

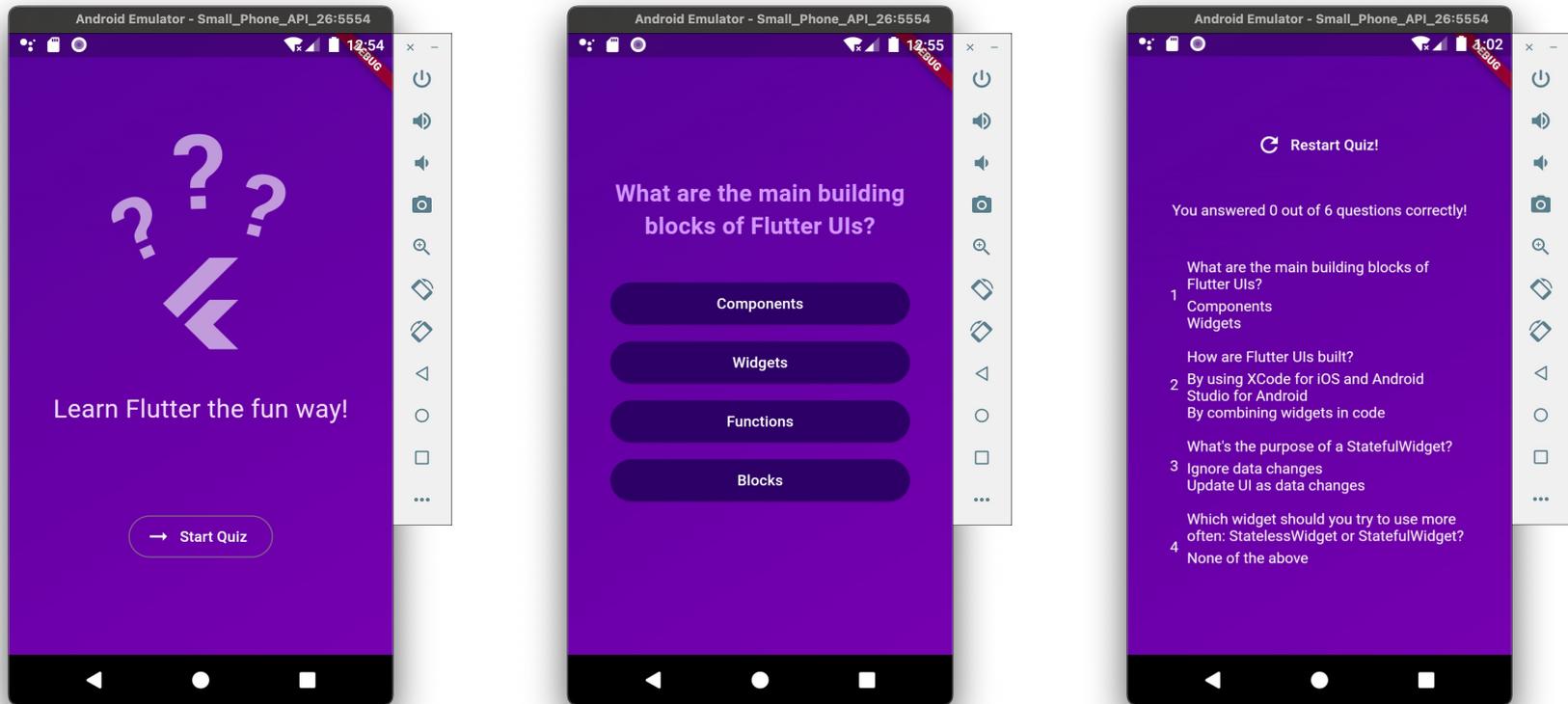
```
// Map<String, dynamic> or Map<String, Object>?

void main() {
    Object obj = 'Hello';
    print(obj.length); // Compile-time error
    print((obj as String).length); // OK

    dynamic dyn = 'Hello';
    print(dyn.length); // OK
    print(dyn.isEven); // Runtime error
}
```

- `Object` offers compile-time type safety
- `dynamic` bypasses compile-time type checks, risking runtime errors

# Meet the Quiz App



# Today's Topics

- More Dart language
  - Operators & null safety
  - More OOP
  - Control flows
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# Today's Topics

- More Dart language
  - Operators & null safety
  - More OOP
  - Control flows
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# Operators

- Most ops exist in C:

+ , ++ , % , += , == , && , etc.

- Precedence & associativity

- `print(3 + 5 * 7 * (1 - 4) % 2 << 3);`  
`// Prints what?`

- New operators:

- Type checking & conversion ops
- Null-aware ops

# Type Checking

```
Object obj1 = ...;
Object obj2 = ...;

if (obj1 is String) {
    ... // Do something
}
if (obj2 is! int) {
    ... // Do some other thing
}

print(obj1.runtimeType);
```

# Type Conversion

- String **to** int:

```
int one = int.parse('1');
```

- int **to** String:

```
String oneStr = one.toString();
```

- double **to** int:

```
pi.toInt();
```

- Parent to child type:

```
Object obj = 'This is a string';  
String str = obj as String;
```

# Null-awareness & Safety

- By default, variables **cannot** hold null values
  - To minimize the chance of null pointer exception (crashes) at runtime
- Unless with `? type` or `late` keyword

```
int a = 100;    // Non-nullable
int b;         // Compile-time error
int? c;        // OK
late int d;    // OK
print(d);     // Error: must be initialized first
d = 3;
print(d);     // OK
```

# Null-aware Operators

```
String? str;

print(str.length); // Compile-time error

// Prints length if not null; otherwise prints null
print(str?.length);

// Prints string if not null; otherwise 'Default'
print(str ?? 'Default');

// Assign 'Default' to str if str is null
str ??= 'Default';

// Tells compiler str is not null
int length = str!.length;
```

- Use ! with care, as it risks runtime errors

# Sound Null Safety

- ***Soundness***: if compiler determines that a variable is non-nullable, then the variable can ***never*** be null at runtime
- Benefits:
  - Eliminating null checks at runtime
  - Optimized inline caching and type specialization
  - Code size reduction
  - Faster hot reload

# Today's Topics

- More Dart language
  - Operators & null safety
  - **More OOP**
  - Control flows
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# More OOP

- Static class members
- Enums
- Abstract classes
- Annotations
- Mixins
- Extensions

# Static Class Members

```
class Circle {
    static const double pi = 3.14159;
    double radius;

    Circle(this.radius);

    double area() {
        return pi * radius * radius;
    }

    double circumference() {
        return 2 * pi * radius;
    }
}

void main() {
    print(Circle.pi);
    print(Circle(5.0).area());
    print(Circle(8.0).circumference());
}
```

- Static members are shared between all instances

# Enums

```
enum Status { loading, success, error }
```

```
void main() {  
    Status status = .loading; // Shorthand for Status.loading  
  
    print(status.name); // to String  
  
    switch(status) {  
        case .loading:  
            ...  
            break;  
        case .success:  
            ...  
            break;  
        default:  
            ...  
    }  
}
```

# Abstract Classes

- There's no  
interface  
keyword in Dart

```
abstract class GameCharacter {
    String name;

    GameCharacter(this.name);

    // Must be implemented by subclasses
    void performAbility();
}

class Warrior extends GameCharacter {
    Warrior(String name) : super(name);

    @override
    void performAbility() {
        print('$name swings sword!');
    }
}

class Mage extends GameCharacter {
    Mage(String name) : super(name);

    @override
    void performAbility() {
        print('$name casts a spell!');
    }
}
```

# Annotations

- Metadata for compilers and other tools

```
class Vehicle {  
    // not to be overridden  
    @nonVirtual  
    void startEngine() {  
        ...  
    }  
}
```

```
class Car extends Vehicle {  
    // Error  
    @override  
    void startEngine() {  
        ...  
    }  
}
```

```
class Animal {  
    // seen in only this & subclass  
    @protected  
    void breathe() {  
        print('Breathing');  
    }  
}
```

```
class Human extends Animal {  
    void meditate() {  
        breathe(); // OK  
    }  
}
```

```
void main() {  
    Human human = Human();  
    human.meditate(); // OK  
    human.breathe(); // Error  
}
```

# Extensions

```
extension EmailValidator on String {  
    bool get isValidEmail {  
        return RegExp(  
            r'^[a-zA-Z0-9.]+@[a-zA-Z0-9]+\.[a-zA-Z]+',  
        ).hasMatch(this);  
    }  
}
```

```
void main() {  
    var email = 'example@example.com';  
    if (email.isValidEmail) {  
        ...  
    }  
}
```

- Add new functionalities to existing classes without subclassing

# Mixins

```
mixin Logger {  
    void log(String message) {  
        // persist message  
    }  
}  
  
class OrderProcessor extends Processor with Logger, ... {  
    void createOrder(String details) {  
        ...  
        log('Order created: $details');  
    }  
}
```

- A means of implementing multiple inheritance

# Today's Topics

- More Dart language
  - Operators & null safety
  - More OOP
  - **Control flows**
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# Control Flows

- Similar to those in C, C++ and Java
  - `if`, `switch`, `for`, `while`, ... statements
- With some additions

```
for (var e in myList) {  
    ...  
}
```

# Switch Expressions

```
void main() {  
    var char = ';';  
    var token = switch (char) {  
        '+' || '-' || '*' || '/' => 'Operator',  
        ',' || ';' => 'Punctuation',  
        '0' || '1' || '2' || '3' || '4' || '5' || '6' ||  
            '7' || '8' || '9' => 'Number',  
        _ => throw FormatException('Invalid character'),  
    };  
    ...  
}
```

- Evaluated to a single value

# Spread, `if` and `for` in Collection Literals

```
bool includeOddNumbers = true;
var numbers = [1, 2, 3, 4, 5];
var evenNumbers = [2, 4];

var combinedList = [
  // Conditionally add items to list
  if (includeOddNumbers) ...numbers.where((n) => n.isOdd),

  // Iterate and add items
  for (var n in evenNumbers) n,
];

print(combinedList); // Prints what?
```

- Simplifies constructor calling → widget tree construction
- Body needs to be single-line expression

# Today's Topics

- More Dart language
  - Operators & null safety
  - More OOP
  - Control flows
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# Styles & Conventions

- Variable/function names: `lowerCamelCase`
- Class/extension names: `UpperCamelCase`
- Package names: `snake_case`
- Import `dart`: before other imports
  - Sort each section alphabetically
- Leading underscore only for private identifiers: `_age`
- Leading `k` for global variables: `kColorScheme`
- Prefer single-quoted strings: `'...'`
  - Exception: `"... 'single-quoted content' ..."`
- Abbreviations: `HttpConnection`, `DBPort`, `UserId`
- `_` for unused callback parameters:

```
numbers.forEach((_) {  
    return 0;  
});
```

# Composite over Inheritance

```
class StyledText extends Text {  
    ...  
}
```

- Composition allows widgets to remain loosely coupled, promoting reusability



```
class StyledText extends StatelessWidget {  
    final String text;  
    final TextStyle style;  
  
    const StyledText(this.text, {  
        super.key,  
        required this.style,  
    });  
  
    @override  
    Widget build(BuildContext context) {  
        return Text(text, style: style);  
    }  
}
```

# Comments

- **Single-line:** `//` Single-line comment.
- **Multi-line:** `/*` Multi-line comment. `*/`
- **Documentation:**

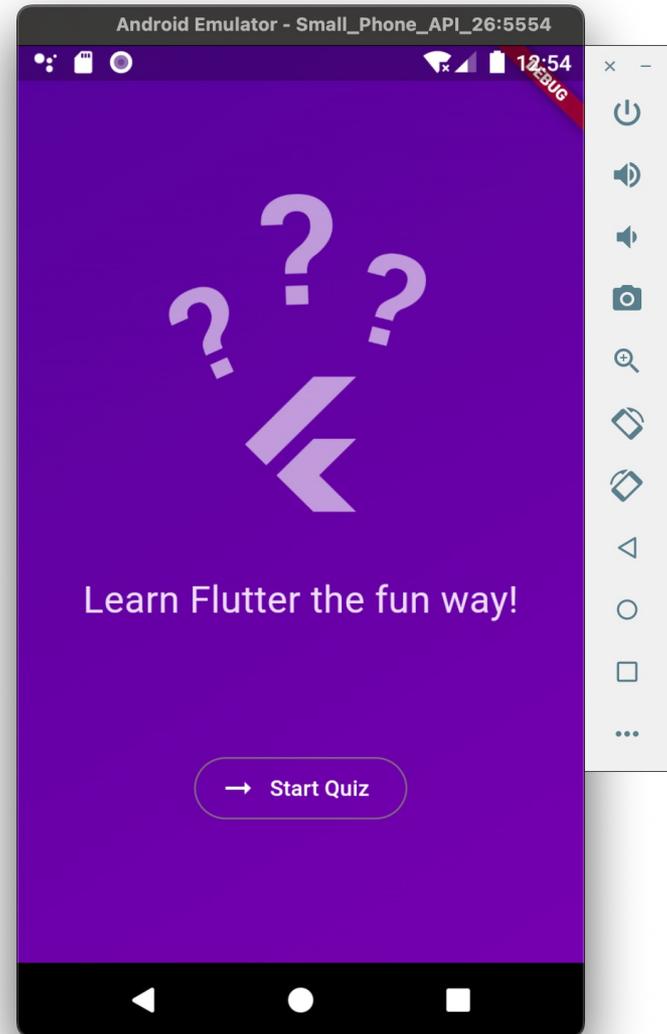
```
/// Calculates the area of a rectangle.
///
/// [width] the width of the rectangle.
/// [height] the height of the rectangle.
///
/// Returns the calculated area as a double.
///
/// Throws an [ArgumentError] if [width] or [height] is negative.
double calculateArea(double width, double height) {
    if (width < 0 || height < 0) {
        throw ArgumentError('Width and height must be non-negative.');
```

# Today's Topics

- More Dart language
  - Operators & null safety
  - More OOP
  - Control flows
  - Coding conventions
- More complex UI
  - Interoperations between multiple widgets
  - Data models
  - Basic styling
  - Scrolling
- Dev tools

# Quiz App

- Dev tool: Inspector
- Image opacity
- Navigation
- Data models
- Styling
- Lists
- Scrolling



# Dev tool: Inspector

- Run the app in debug mode
- In Debug Sessions, open “Flutter Inspector”

# Image Opacity in StartScreen

- Using `Opacity` widget
  - Applies to entire subtree under `child`
  - Subtree first rendered to offscreen buffer
  - Then transformation applied
- Using the `color` property of `Image`
  - Flutter applies color as tint
  - Part of the image rendering process
  - Can be accelerated by GPU



# Navigation & State Lifting

- How does the “Start” button in `StartScreen` triggers screen transition?
  1. **Lift up** navigation state to a shared parent widget  
`Quiz`
  2. Define callback that changes state, and pass it to `StartScreen` **as closure**
- Same strategy is used to pass selected answers from `QuestionsScreen` to `ResultsScreen`

# Data Models

- Plain old Dart objects (PODO) holding app's data structure
- No business logics
  - E.g., persistence, authentication, etc.
- Can be immutable
- Can have **getter** and **setter** methods to control access

```
class User {
  User(this._name, this._age);

  String _name; // private
  String get name => _name; // getter
  set name(String newName) { // setter
    if (newName.isNotEmpty) {
      _name = newName;
    } else {
      print('Name cannot be empty');
    }
  }

  int _age; // private
  int get age => _age; // getter
  set age(int newAge) { // setter
    if (newAge > 0) {
      _age = newAge;
    } else {
      print('Age must be > 0');
    }
  }
}

...
var name = user.name; // call getter
user.age = 20; // call setter
```

# Styling

- Google font used in `questions_screen.dart` to make questions stand out
- Rounded corners and centered text for `answer_button.dart`
- Icons in “Start” and “Restart” buttons
  - See [all available material icons](#)

# Filtering & Analyzing Lists

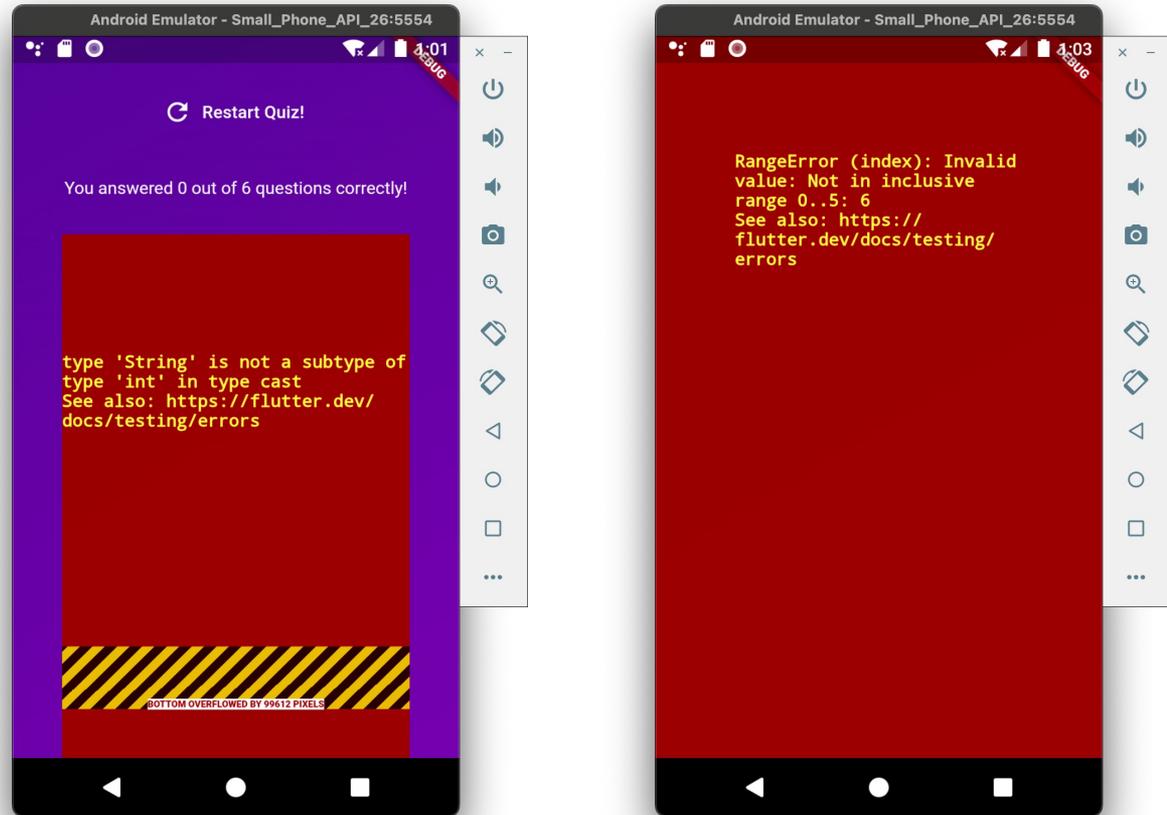
```
// in results_screen.dart
final List<Map<String, Object>> summaryData = ...;
final numTotalQuestions = questions.length;
final numCorrectQuestions = summaryData.where((data) {
  return data['user_answer'] == data['correct_answer'];
}).length;
```

- `where` filters the list and results an `Iterable`

# Scrolling

- Watchout the ***overflow*** issues
- Try removing `Expand` and `SingleChildScrollView` widgets in `questions_summary.dart`
- In Flutter, overflowed widgets are ***not*** wrapped nor scrollable by default

# The Code We Release will be Buggy!



- Learn how to debug in the next lab 🤗

# References

- [Introduction to Dart](#)