Unit 3

In Flutter (or any programming language), functions can be categorized based on whether they accept arguments and return values. Let's walk through each of these scenarios with examples in Flutter's Dart language.

1. No Arguments and No Return Type

A function that neither accepts arguments nor returns a value.

```
void showMessage() {
  print("Hello, this is a message!");
}

void main() {
  showMessage(); // Calling the function
}
```

- **Explanation**: This function doesn't take any parameters (arguments), and it doesn't return anything (`void` means no return value).

2. With Arguments and No Return Type

A function that takes arguments but doesn't return a value.

```
void greetUser(String name) {
  print("Hello, $name!");
}

void main() {
  greetUser("Alice"); // Passing a name as an argument
}
```

- **Explanation**: The `greetUser` function takes a `String` argument but doesn't return any value. It just prints the greeting.

3. No Arguments and Return Type

A function that doesn't take any arguments but returns a value.

```
int getCurrentYear() {
  return DateTime.now().year;
}

void main() {
  int year = getCurrentYear(); // Assigning the return valu
e to a variable
  print("The current year is $year");
}
```

- **Explanation**: `getCurrentYear` doesn't take any parameters, but it returns an integer value (the current year).

4. With Arguments and With Return Type

A function that takes arguments and returns a value.

```
int addNumbers(int a, int b) {
  return a + b;
}

void main() {
  int result = addNumbers(5, 3); // Passing two numbers and
getting the sum
  print("The sum is $result");
}
```

- **Explanation**: `addNumbers` takes two integer arguments and returns their sum as an integer.

In Flutter, these function types are essential to structuring code efficiently.

INHERITANCE

Inheritance is a core concept in object-oriented programming (OOP), where a class (child or subclass) can inherit properties and behaviors (methods) from another class (parent or superclass). Let's go over examples of the different types of inheritance in Dart:

1. Single Inheritance

In single inheritance, a subclass inherits from a single superclass.

```
// Parent class
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

// Child class
class Dog extends Animal {
  void bark() {
    print("Dog is barking");
  }
}

void main() {
  Dog dog = Dog();
  dog.eat();  // Inherited method from Animal class
  dog.bark(); // Method from Dog class
}
```

- **Explanation**: The `Dog` class inherits from the `Animal` class. It can access the `eat()` method of the `Animal` class along with its own `bark()` method.

2. Multiple Inheritance (Not Supported Directly in Dart)

Dart doesn't support multiple inheritance directly (i.e., a class cannot inherit from more than one class). However, it achieves similar behavior through **mixins**. Mixins allow you to add methods and properties from multiple classes.

```
// Mixin 1
mixin Swimmer {
  void swim() {
    print("Swimming");
  }
}

// Mixin 2
```

```
mixin Runner {
  void run() {
    print("Running");
  }
}

// Class that uses both mixins
class Person with Swimmer, Runner {}

void main() {
  Person person = Person();
  person.swim();  // Method from Swimmer mixin
  person.run();   // Method from Runner mixin
}
```

- **Explanation**: The `Person` class uses two mixins (`Swimmer` and `Runner`). Dart uses the `with` keyword to mix in multiple behaviors, similar to multiple inheritance.

3. Multi-level Inheritance

In multi-level inheritance, a class is derived from a class that is also derived from another class (a chain of inheritance).

```
// Grandparent class
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

// Parent class
class Mammal extends Animal {
  void walk() {
    print("Mammal is walking");
  }
}

// Child class
```

```
class Dog extends Mammal {
  void bark() {
    print("Dog is barking");
  }
}

void main() {
  Dog dog = Dog();
  dog.eat();   // Inherited from Animal class
  dog.walk();  // Inherited from Mammal class
  dog.bark();  // Method from Dog class
}
```

- **Explanation**: The `Dog` class inherits from the `Mammal` class, which in turn inherits from the `Animal` class. Thus, the `Dog` class can access methods from both `Mammal` and `Animal`.

Summary:

- **Single Inheritance**: A class inherits from a single parent class.
- **Multiple Inheritance**: Dart does not support this directly, but it can be mimicked with **mixins**.
- **Multi-level Inheritance**: A class inherits from a class that is also a subclass.

These examples show how inheritance works in Dart and how mixins can be used to simulate multiple inheritance.

STATIC KEYWORD

In Dart, the `static` keyword is used to declare class-level variables and methods, meaning they belong to the class itself rather than to instances of the class. Static members can be accessed without creating an instance of the class.

Simple Example of `static` Keyword

```
class MathOperations {
  // Static variable
  static double pi = 3.14159;
```

```
  // Static method
  static double circleArea(double radius) {
    return pi * radius * radius;
  }
}

void main() {
  // Accessing static variable and method without creating
an object
  print("The value of pi is: ${MathOperations.pi}");
  print("The area of the circle is: ${MathOperations.circle
Area(5)}");
}
```

Explanation:

- **Static Variable (`pi`)**: The variable `pi` is marked as `static`, so it belongs to the class `MathOperations` and can be accessed using `MathOperations.pi` without creating an object.
- **Static Method (`circleArea`)**: The method `circleArea()` is also marked as `static`, meaning it can be called using `MathOperations.circleArea()` without creating an object.

Key Points:

- **Static variables and methods** are shared across all instances of the class.
- **Static methods** cannot access non-static instance variables or methods directly. They can only work with other static members.

SUPER KEYWORD

The `super` keyword in Dart is used to refer to the superclass (parent class) and is primarily used to:

1. Call a method or access a property from the parent class.
2. Call the parent class's constructor.

Example of `super` Keyword

```
// Parent class
class Animal {
  void makeSound() {
    print("Animal makes a sound");
  }
}

// Child class
class Dog extends Animal {
  @override
  void makeSound() {
    super.makeSound(); // Call the parent class method
    print("Dog barks");
  }
}

void main() {
  Dog dog = Dog();
  dog.makeSound(); // Output: Animal makes a sound
                   //         Dog barks
}
```

Explanation:

- `super.makeSound()`: Calls the `makeSound` method from the `Animal` class.
- The `Dog` class adds its own behavior by printing "Dog barks".

This example demonstrates how `super` can be used to call the parent class method in an overridden method.

MIXIN

In Dart, **mixins** are a way of reusing a class's code in multiple class hierarchies. Mixins allow a class to share functionality with other classes without requiring inheritance from a specific superclass. This is helpful when you want to add common behavior to multiple classes without needing to inherit from a single parent class.

Key Points About Mixins:

- A mixin is a class that can be reused by other classes.
- Mixins are used with the `with` keyword.
- A mixin can be applied to multiple classes.
- A mixin cannot have a constructor.

Simple Example of Mixins in Dart

```
// Defining a mixin
mixin Swimmer {
  void swim() {
    print("Swimming");
  }
}

// Another mixin
mixin Runner {
  void run() {
    print("Running");
  }
}

// Class that uses both mixins
class Athlete with Swimmer, Runner {}

void main() {
  Athlete athlete = Athlete();
  athlete.swim();  // Output: Swimming
  athlete.run();   // Output: Running
}
```

Explanation:

1. **Mixin Declaration**: The `Swimmer` and `Runner` classes are defined as mixins (without any special keyword).
2. **Using Mixins**: The `Athlete` class uses both `Swimmer` and `Runner` mixins by using the `with` keyword.

3. **Functionality**: The `Athlete` class can now call methods from both mixins (`swim()` and `run()`), even though it doesn't inherit from them in a traditional sense.

When to Use Mixins:

- When you want to share behavior between multiple classes without using inheritance.
- When you want to avoid a deep inheritance chain or tightly coupled classes.

Example with an Existing Class and a Mixin

```
class Animal {
  void breathe() {
    print("Animal is breathing");
  }
}

// Defining a mixin
mixin Flyer {
  void fly() {
    print("Flying");
  }
}

// Class that extends Animal and uses a mixin
class Bird extends Animal with Flyer {}

void main() {
  Bird bird = Bird();
  bird.breathe();  // Output: Animal is breathing
  bird.fly();      // Output: Flying
}
```

- **Bird**: Extends the `Animal` class and also uses the `Flyer` mixin to gain flying ability.

Mixins are a powerful feature in Dart to add reusable functionality across multiple classes.

STRUCTURING WIDGETS

Structuring widgets in Flutter involves organizing your UI components (widgets) in a modular, reusable, and maintainable way. As your Flutter app grows, properly structuring your widgets will help you manage the complexity and improve the readability and maintainability of your code. This is especially important in larger apps with many screens and UI elements.

Recommended Widget Structure for Flutter Projects

Here is a common way to structure widgets in a Flutter app:

```
lib/
|
├── main.dart                // Entry point of the applicat
ion
|
├── src/
|   ├── screens/             // Screens or pages of your ap
p
|   |   ├── home_screen.dart
|   |   └── profile_screen.dart
|   |
|   ├── widgets/             // Reusable widgets used acros
s the app
|   |   ├── custom_button.dart
|   |   └── custom_card.dart
|   |
|   ├── models/              // Data models representing ap
p data
|   |   └── user.dart
|   |
|   ├── services/            // Business logic or API servi
ces
|   |   └── api_service.dart
```

```
|   |
|   └── utils/              // Utility functions or classe
s
|       └── date_utils.dart
```

Example of Widget Structure in Flutter

1. Main Entry Point (`main.dart`)

The `main.dart` file is the entry point of the application. It often sets up the app's basic structure (e.g., MaterialApp, routes).

```
import 'package:flutter/material.dart';
import 'src/screens/home_screen.dart';

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

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

2. Screens (`screens/home_screen.dart`)

Screens are pages in your app. Each screen might use several smaller widgets to construct its UI.

```
import 'package:flutter/material.dart';
import '../widgets/custom_button.dart';  // Import reusable
widget

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Home Screen')),
      body: Center(
        child: CustomButton(
          text: 'Click Me',
          onPressed: () {
            print('Button clicked');
          },
        ),
      ),
    );
  }
}
```

3. Reusable Widgets (`widgets/custom_button.dart`)

Reusable widgets are UI components that can be used on different screens, helping you avoid repetition and making the code more maintainable.

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

class CustomButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;

  CustomButton({required this.text, required this.onPresse
d});
```

```
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(text),
    );
  }
}
```

4. Another Widget Example (`widgets/custom_card.dart`)

You can also create custom widgets like cards to reuse across multiple screens.

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

class CustomCard extends StatelessWidget {
  final String title;
  final String description;

  CustomCard({required this.title, required this.descriptio
n});

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            Text(
              title,
              style: TextStyle(fontSize: 20, fontWeight: Fo
ntWeight.bold),
            ),
```

```
            SizedBox(height: 10),
            Text(description),
          ],
        ),
      ),
    );
  }
}
```

Breakdown:

1. `screens/`: Contains each screen or page of the app. Every screen is typically a `StatefulWidget` or `StatelessWidget` and can consist of multiple widgets.
2. `widgets/`: This folder contains reusable UI components or custom widgets, which can be used across multiple screens. For example, custom buttons, cards, or form fields. It helps to reduce redundancy.
3. `models/`: Stores data models representing the app's data structure.
4. `services/`: Handles business logic, API calls, and other services.
5. `utils/`: Contains helper functions or utility classes for things like formatting dates, handling validation, etc.

Structuring Widgets by Type and Reusability

When to create a new custom widget:

- **Reusability**: If you find a certain UI pattern repeated in multiple places (e.g., a custom button, card, or form field), it's best to extract that part into a separate widget in the `widgets/` directory.
- **Complexity**: If a widget's code becomes too large and hard to manage, you should break it into smaller, more manageable widgets.