Control Flow and Logic Keywords

```dart
// break → Exits a loop or switch case immediately
for (int i = 0; i < 5; i++) {
  if (i == 3) break;
  print(i); // Prints 0, 1, 2
}

// case → Used in switch statements to match a value
// switch → Selects one of many code blocks to execute
// default → Provides a default case in switch statements
int number = 2;
// Checks the value of number
switch (number) {
  case 1: // Skips this case
    print("One");
    break;
  case 2:
    print("Two"); // This runs
    break;
  default: // If number is not 1 or 2
    print("Unknown number");
}

// continue → Skips the current loop iteration and moves to the next
for (int i = 0; i < 5; i++) {
  if (i == 3) continue;
  print(i); // Prints 0, 1, 2, 4
}

// for → Loops through a block of code a set number of times
for (int i = 0; i < 5; i++) {
  print(i); // Prints 0, 1, 2, 3, 4
}

// while → Loops through a block of code while a condition is true
int i = 0;
while (i < 5) {
  print(i);
  i++;
}

// do → Executes a loop at least once before checking the condition
int x = 0;
do {
  print(x);
  x++;
} while (x < 5);

// if → Executes code if a condition is true
// else → Executes code if the condition is false
// else if → Specifies a new condition if the first one is false
int age = 18;
if (age < 18) {
  print("You are a minor");
} else if (age >= 18 && age < 60) {
  print("You are an adult");
} else {
  print("You are a senior citizen");
}

// rethrow → Rethrows an exception
// return → Exits a function and returns a value
// try → Tests a block of code for errors
// catch → Handles the error
// finally → Executes code after try and catch, regardless of the result
// throw → Throws an exception
// assert → Used for debugging to check conditions
// on → Specifies the type of exception to catch
// Exception → Defines custom exceptions
int depositMoney(int amount) {
  try {
    if (amount < 0) {
      assert(amount >= 0); // Throws an AssertionError if condition fails
      throw Exception('Amount cannot be negative'); // Throws a custom exception
    } else {
      return amount; // Returns the amount if no error
    }
  } on Exception catch (e) {
    // Handles the thrown exception
    print(e);
    rethrow; // Rethrows the exception to the caller
  } finally {
    // Always executes after try and catch
    print('Transaction completed');
  }
}
```

Asynchronous and Synchronous programming

```dart
// Future → Represents a value or error that will be available in the future
// async → Marks a function as asynchronous, allowing the use of await
// await → Pauses an async function until a Future completes
Future<void> fetchData() async {
  print('Fetching data...');
  // Simulate network request
  await Future.delayed(const Duration(seconds: 2));
  print('Data fetched!');
}

// Stream → Represents a sequence of asynchronous events
// yield → Emits a value from a generator function
// async* → Marks a function as an asynchronous generator
Stream<int> countStream(int max) async* {
  for (int i = 1; i <= max; i++) {
    yield i;
  }
}

// sync* → Marks a function as a synchronous generator that returns an Iterable
Iterable<int> count(int max) sync* {
  for (int i = 1; i <= max; i++) {
    yield i;
  }
}
```

characters Keywords

```dart
// . → Accesses properties and methods of an object
// .. → Cascades a sequence of operations on the same object
// ... → Spreads elements of a list or map
final dog = Dog();
final name3 = dog.makeSound();

// Cascading operations
final person3 = Person2('John')..printName();

// Spreading elements
final list1 = [1, 2, 3];
final list2 = [0, ...list1, 4, 5]; // [0, 1, 2, 3, 4, 5]

// ? → Checks if an object is null before accessing its properties or methods
// ?? → Provides a default value if an object is null
// ! → Asserts that an expression is not null
final String? name4 = null;
// Checking for null
final nameLength = name4?.length;

// Providing a default value
final name5 = name4 ?? 'Guest';

// Asserting non-null
final nameLength2 = name4!.length;
```

Variables and Type Keywords

```dart
// const → For compile-time constants with fixed values known at compile time
// final → For variables set once, with values determined at runtime
// var → Infers the type at compile time and keeps it consistent
// dynamic → For variables that can hold different types and change at runtime
// late → For non-nullable variables initialized after declaration
// null → Represents an uninitialized object
final int y = 10;
const int x = 5;
var name = 'John';
dynamic value = 100;
late String city;
String? name2 = null;

// void → Indicates a function returns nothing
void printName() {
  print("John Doe");
}

// as → For type casting to treat an object as a specific type
dynamic someValue = 'Hello, Dart!';
// Casting dynamic to String
String stringValue = someValue as String;

// is → Checks the type of an object at runtime
void main2() {
  var name = 'John';
  // Check if 'name' is a String
  if (name is String) {
    print('name is a String');
  }
}

// covariant → Allows a parameter in a method to be overridden
with a more specific type in a subclass
class Animal5 {
  void makeSound(covariant Animal5 animal) {
    print("Animal makes a sound");
  }
}

class Dog5 extends Animal5 {
  @override
  void makeSound(Dog5 dog) {
    print("Dog barks");
  }
}

void mainFunction() {
  Animal5 animal = Dog5();
  animal.makeSound(Dog5()); // Dog barks
}

// enum → Defines a group of named constants
enum Status { none, running, stopped, paused }

// operator → Overloads operators for custom behavior in classes
class Point {
  int x, y;
  Point(this.x, this.y);
  // Overloading the + operator
  Point operator +(Point point) {
    return Point(x + point.x, y + point.y); // Adds two Point objects
  }

  // Overloading the == operator
  @override
  bool operator ==(Object other) {
    return other is Point && x == other.x && y == other.y;
  }

  // Overriding the hashCode getter
  @override
  int get hashCode => x.hashCode ^ y.hashCode;
}

// typedef → Defines a function type
typedef Compare = int Function(int a, int b);
```

Functions Keywords

```dart
// Function → Defines a function
int add(int a, int b) {
  return a + b;
}

Function addFunction = add;

// => → Defines a function that returns a single expression
// return → Exits a function and returns a value
int multiply(int a, int b) => a * b;

int subtract(int a, int b) {
  return a - b;
}

// typedef → Defines a function type alias for cleaner code and better
// readability
typedef MathOperation = int Function(int, int);

int multiply2(int a, int b) => a * b;

void calculate(MathOperation operation) {
  print(operation(4, 5)); // Output: 20
}

// required → Marks a named parameter as required
void greet({required String name}) {
  print('Hello, $name!');
}

// external → Declares a function implemented outside of Dart
external void performNativeTask();

// extension → Adds new functionality to existing classes without modifying
// them
extension StringExtension on String {
  String capitalize() {
    return '${this[0].toUpperCase()}${substring(1)}';
  }
}

// extension type → Defines custom types that extend existing types for new ///
// behaviors and type safety
extension type IdNumber(int id) {
  int get maskedId => id % 1000; // Only last 3 digits
}

void main3() {
  IdNumber myId = IdNumber(123456);
  print(myId.maskedId); // Output: 456
}
```

characters Keywords

```dart
// import → Imports a library into the current file
// as → Gives a library a prefix to avoid naming conflicts
// show → Imports only specific parts of a library
// hide → Hides specific parts of a library
// deferred → Loads a library only when needed
// export → Exports a library to other files
// part → Splits a library across multiple files for better management
// part of → Links a file to its main library
// library → Defines a Dart library to group related code
import 'dart:math' as math; // Imports with a prefix
import 'dart:math' deferred as deferredMath; // Imports lazily
import 'dart:math' hide min, max; // Imports everything except min and max
import 'dart:math' show min, max; // Imports only min and max
import 'dart:math'; // Imports a core Dart library
export 'dart:math'; // Exports a core Dart library
part 'my_part.dart'; // Links a part of a library
part of my_library; // Links to the main library
library my_library; // Defines a Dart library
```

Object-Oriented Programming Keywords

```dart
// class → Defines a new class
// extends → Inherits from another class
// super → Calls the constructor or methods of the parent class
class Animal {
  void makeSound() => print("Animal sound");
}

class Dog extends Animal {
  void bark() {
    super.makeSound(); // Calls the parent class method
    print("Woof!");
  }
}

// abstract → Defines a class that cannot be instantiated, only inherited or
implemented
// interface → Defines a contract for a class to follow with a known
implementation
// implements → Forces a class to follow an interface
abstract class Car {
  void drive(); // Abstract method (no body)
}

interface class Vehicle2 {
  void honk() {} // Method with implementation
}

class ElectricCar implements Car {
  @override
  void drive() {
    print("Electric car is driving");
  }
}

// mixin → Adds shared functionality to multiple classes
// with → Adds a mixin to a class
// on in mixin → Restricts the mixin to a specific class
mixin Swim {
  void swim() => print("Swimming");
}

class Duck extends Animal with Swim {
  @override
  void swim() => print("Swimming");

  void quack() => print("Quack!");
}

mixin Fly on Duck {
  void fly() => print("Flying");
}

// factory → Returns a cached instance, subtype, or performs pre-construction
logic
// static → Creates class-level variables or methods
class Singleton {
  static final Map<String, Singleton> _cache = {};

  final String name;

  factory Singleton(String name) {
    return _cache.putIfAbsent(name, () => Singleton._internal(name));
  }
  Singleton._internal(this.name);
}

// get → Defines a getter method
// set → Defines a setter method
class Person {
  int _age = 30;
  int get age => _age;
  set age(int value) {
    if (value > 0) {
      _age = value;
    }
  }
}

// new → Creates a new instance of a class (optional in Dart 2.1.0+)
void main() {
  var person = new Person();
  person.age = 25;
  print(person.age); // 25
}

// this → Refers to the current instance of the class
class Person2 {
  String name;
  Person2(this.name);
  void printName() {
    print(name);
  }
}

// sealed → Defines a class that cannot be instantiated or extended outside its
library
sealed class Animal3 {
  void makeSound() => print("Animal sound");
}
// Cannot instantiate a sealed class
// Animal3 myAnimal3 = Animal3();

// base → Enforces inheritance within the same library, can be constructed
base class Animal2 {
  void makeSound() => print("Animal sound");
}

Animal2 myAnimal = Animal2(); // Can be constructed

// Must be base, final, or sealed to extend a base class
base class Dog3 extends Animal2 {
  void bark() {
    super.makeSound();
    print("Woof!");
  }
}

// final class → Can be constructed but cannot be extended or implemented
final class Vehicle {
  void moveForward(int meters) => print("Vehicle is moving forward $meters
meters");
}

// Error: Can't be inherited.
// class Car extends Vehicle {
//   int passengers = 4;
// }

// Error: Can't be implemented.
// class MockVehicle implements Vehicle {
//   @override
//   void moveForward(int meters) {}
// }
```


@ Ahmed Hussein
01114484229