All Dart Keywords

The words that the Dart language reserves for its own use

Flutter Developer

@ Ahmed Hussein
01114484229

Dart Keywords

In this file, we will discuss all the keywords in Dart. Some of these keywords can be used as identifiers, while others cannot. It is recommended not to use any of these keywords as identifiers to avoid conflicts between your code and Dart's original code, ensuring clarity and organization.

I have categorized these keywords to make them clear and easy to understand. Each keyword includes explanations and examples.

@ Ahmed Hussein
01114484229

1: Control Flow and Logic Keywords

1- break → Exits a loop or switch case immediately.

```dart
for (int i = 0; i < 5; i++) {
  if (i == 3) break;
  print(i); // Prints 0, 1, 2
}
```

2- case → Used in switch statements to match a value.

3- switch → Used to select one of many code blocks to be executed.

4- default → Used in switch statements to provide a default case.

```dart
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");
}
```

5- continue → Skips the current iteration of a loop and moves to next .

```dart
for (int i = 0; i < 5; i++) {
  if (i == 3) continue;
  print(i); // Prints 0, 1, 2, 4
}
```


@ Ahmed Hussein
01114484229

6- for → Used to loop through a block of code a number of times.

```dart
for (int i = 0; i < 5; i++) {
  print(i); // Prints 0, 1, 2, 3, 4
}
```

7- while → Used to loop through a block of code while a specified condition is true.

```dart
int i = 0;
while (i < 5) {
  print(i);
  i++;
}
```

8- do →Executes a loop at least once before checking the condition.

```dart
int x = 0;
do {
  print(x);
  x++;
} while (x < 5);
```

9- if → Used to execute some code if a specified condition is true.

10- else → Used to execute some code if the condition is false.

11- else if → Used to specify a new condition if the first condition is false.

```dart
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");
}
```


@ Ahmed Hussein
01114484229

12- **rethrow** → Used to rethrow an exception.

13- **return** → Exits a function and returns a value.

14- **try** → Used to test a block of code for errors.

15- **catch** → Used to handle the error.

16- **finally** → Used to execute code, after try and catch, regardless of the result.

17- **throw** → Used to throw an exception.

18- **assert** → Used for debugging purposes.

19- **on** → Used to specify the type of exception to catch.

20- **Exception** → Used to define custom exceptions.

```dart
int depositMoney(int amount) {
  try {
    if (amount < 0) {
      assert(amount >= 0); // Throws an AssertionError
      throw Exception('Amount cannot be negative'); // Throws an exception
    } else {
      return amount; // Returns the amount
    }
  } on Exception catch (e) {
    // Handles the exception
    print(e);
    rethrow; // Rethrows the exception
  } finally {
    // Executes after try and catch
    print('Transaction completed');
  }
}
```


@ Ahmed Hussein
01114484229

2: Object-Oriented Programming Keywords

1- class → Used to define a new class.

2- extends → Inherits from another class.

3- super → Used to call the constructor of the parent class.

```dart
// parent class
class Animal {
  void makeSound() => print(object: "Animal sound");
}

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

4.1- abstract → Defines a class that cannot be instantiated (Cannot be instantiated, Can be inherited, Can be implemented).

4.2- interface → Defines a contract for a class to follow (Can be instantiated, Cannot be inherited, Can be implemented).

5- implements → Forces a class to follow an interface.

```dart
abstract class Car {
  void drive(); // Abstract method (does not have a body)
}

interface class Vehicle2 {
  void honk() {} // known implementation of the method (must have a body)
}

class ElectricCar implements Car {
  @override // Overriding the drive method of the Car interface
  void drive() {
    print(object: "Electric car is driving");
  }
}
```


@ Ahmed Hussein
01114484229

6- mixin → Adds shared functionality to multiple classes.

7- with → Used to add a mixin to a class.

8- on (in mixin) → Used to restrict the mixin to a specific class.

```dart
mixin Swim {
  // Define a method in the mixin
  void swim() => print(object: "Swimming");
}

// Use the mixin in a class with the with keyword
// normal class extends from mixin class (multiple inhertance)
class Duck extends Animal with Swim {
  @override // Override the makeSound method
  void swim() => print(object: "Swimming");

  void quack() => print(object: "Quack!"); // Define a new method
}

// Use the mixin in a class with the on keyword
mixin Fly on Duck {
  //  mixin class extends from normal class
  void fly() => print(object: "Flying");
}
```


@ Ahmed Hussein
01114484229

9.1- factory → A factory constructor can return a cached instance, a subtype, or perform pre-construction logic

9.2- static → Used to create a class-level variable or method.

```dart
class Singleton {
  static final Map<String, Singleton> _cache = <String, Singleton>{};

  final String name;

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

10- get → Used to define a getter method.

11- set → Used to define a setter method.

```dart
class Person {
  int _age = 30;
  // getter
  int get age => _age;
  // setter
  set void age(int value) {
    if (value > 0) {
      _age = value;
    }
  }
}
```


@ Ahmed Hussein
01114484229

12- new → Used to create a new instance of a class.

```dart
void main() {
  // unnecessary_new keyword is not required in Dart 2.1.0 and later
  var Person person = new Person(); //
  person.age = 25;
  print(object: person.age); // 25
}
```

13- this → Refers to the current instance of the class.

```dart
class Person2 {
  String name;
  Person2(this.name);
  void printName() {
    print(object: name);
  }
}
```

14- sealed → used to define a class that cannot be instantiated directly and cannot be extended outside its own library (file).

```dart
sealed class Animal3 {
  void makeSound() => print(object: "Animal sound");
}
// Error: The class 'Animal3' is sealed and can't be instantiated.
Animal3 myAnimal3 = Animal3();    Abstract classes can't be instantiated.↵Try cre
```


@ Ahmed Hussein
01114484229

15 - base → base → To enforce inheritance of a class or mixin's implementation, use the base modifier and it can only be extended by other classes within the same library.

```dart
base class Animal2 {
  void makeSound() => print(object: "Animal sound");
}
// error : The type 'Dog2' must be 'base', 'final' or 'sealed'
//because the 'Animal2' is 'base'.
class Dog2 extends Animal2 {}    The type 'Dog2' must be 'base', 'final' or 'seale

// No error because the 'Dog3' is 'base'.
base class Dog3 extends Animal2 {
  void bark() {
    super.makeSound(); // Calls the makeSound method of the parent class
    print(object: "Woof!");
  }
}
```

16- final class → A class that Can be constructed, cannot be extended and cannot be implemented.

```dart
final class Vehicle {
  void moveForward(int meters) => print(object: "Vehicle is moving $meters meters");
}

Vehicle myVehicle = Vehicle(); // Can be constructed.

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

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


@ Ahmed Hussein
01114484229

3: Variables and Type Keywords

1- const → Use const for variables that are compile-time constants and whose values are known and fixed at compile time

2- final → Use final for variables that are set once and whose values are determined at runtime

3- var → Use var when you want the type to be inferred at compile time and remain consistent

4- dynamic → Use dynamic when you need a variable that can hold values of different types and change at runtime.

5- late → Use late when you need to declare a non-nullable variable that is initialized after its declaration.

6- null → Represents an object that is not initialized.

```
final int y = 10;
const int x = 5;
var String name = 'John';
dynamic value = 100;
late String city;
String? name2 = null;
```

7- void → Indicates a function returns nothing.

```
void printName() {
  print(object: "John Doe");
}
```

8- enum → Used to define a group of named constants.

```
enum Status { none, running, stopped, paused }
```

9- typedef → Used to define a function type.

```
typedef Compare = int Function(int a, int b);
```


@ Ahmed Hussein
01114484229

10- covariant → Use covariant to allow a parameter in a method or setter to be overridden with a more specific type in a subclass.

```dart
class Animal5 {
  void makeSound(covariant Animal5 animal) {
    print(object: "Animal makes a sound");
  }
}

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

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

11- as → Use as for type casting to ensure an object is treated as a specific type.

```dart
dynamic someValue = 'Hello, Dart!';
// Casting dynamic to String
String stringValue = someValue as String;
```

12- is → Use is to check the type of an object at runtime.

```dart
void main2() {
  var String name = 'John';
  // Check if 'name' is of type String
  if (name is String) {
    print(object: 'name is a String');
  }
}
```


@ Ahmed Hussein
01114484229

13 - Operator → The operator keyword in Dart is used to overload operators for custom behavior in user-defined classes. This allows objects to use operators like +, -, ==, etc.

```dart
class Point {
  int x, y;
  Point(this.x, this.y);
  // Overloading the + operator
  Point operator +(Point point) {
    return Point(x: x + point.x, y: 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;
}
```


@ Ahmed Hussein
01114484229

4: Libraries & imports Keywords

1- import → Used to import a library into the current file.

2- as → Used to give a library a prefix to avoid naming conflicts.

3- show → Used to show only specific parts of a library.

4- hide → Used to hide specific parts of a library.

5- deferred → Used to load a library only when it is needed.

6- export → Used to export a library to other files.

7- part → Used to split a single library across multiple files, making large codebases easier to manage.

8- part of → Used to split large Dart libraries into multiple files while keeping them unified.

9- library → used to define a Dart library, allowing you to group related code across multiple files while keeping a clear module structure.

```dart
import 'dart:math'; // Importing a core Dart library

import 'dart:math' as math; // Importing a core Dart library with a prefix

import 'dart:math' show min, max; // Importing only the min and max functions

import 'dart:math' hide min, max; // Importing everything except the min and max func

import 'dart:math' deferred as deferredMath; // Importing a core Dart library lazily

export 'dart:math'; // Exporting a core Dart library

part 'my_part.dart'; // Specifying a part of a library    Target of URI doesn't exist

part of my_library; // Specifying the main library file    The part-of directive must

library my_library; // Defining a Dart library    The library directive must appear b
```


@ Ahmed Hussein
01114484229

5: Asynchronous and Synchronous programming Keywords

1- **Future** → Represents a potential value or error that will be available at some time in the future.

2- **async** → Used to mark a function as asynchronous, allowing it to use the await keyword.

3- **await** → Used to pause the execution of an asynchronous function until a Future is complete.

```dart
Future<void> fetchData() async {
  print(object: 'Fetching data...');
  // Simulate network request
  await Future<dynamic>.delayed(duration: const Duration(seconds: 2));
  print(object: 'Data fetched!');
}
```

4- **Stream** → Represents a sequence of asynchronous events.

5- **yield** → Used to emit a value from a generator function .

6- **async*** → Used to mark a function as asynchronous and a generator.

```dart
Stream<int> countStream(int max) async* {
  for (int i = 1; i <= max; i++) {
    yield i;
  }
}
```

8- **sync*** → Used to mark a function as synchronous that returns an Iterable.

```dart
Iterable<int> count(int max) sync* {
  for (int i = 1; i <= max; i++) {
    yield i;
  }
}
```


@ Ahmed Hussein
01114484229

6: Functions Keywords

1- Function → Used to define a function.

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

Function addFunction = add;
```

2- => → Used to define a function that returns a single expression.

3- 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;
}
```

4- typedef → Used to define a function type alias, Useful for clean code and better readability.

```
typedef MathOperation = int Function(int, int);

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

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

5- required → Used to mark a named parameter as required.

```
void greet({required String name}) {
  print(object: 'Hello, $name!');
}
```


@ Ahmed Hussein
01114484229

6- **external** → Used to declare an external function that is implemented outside of Dart.

```
external void performNativeTask();
```

7- **extension** → Used to add new functionality to existing classes without modifying their source code.

```
extension StringExtension on String {
  String capitalize() {
    return '${this[0].toUpperCase()}${substring(start: 1)}';
  }
}
```

8- **extension type** → defining custom types that extend existing types without modifying their original source code. It is useful for adding new behaviors, type safety, and encapsulation.

```
extension type IdNumber(int id) {
  int get maskedId => id % 1000; // Only last 3 digits
}

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


@ Ahmed Hussein
01114484229

7: characters

1- . → Used to access properties and methods of an object.

2- .. → Used to cascade a sequence of operations on the same object.

3- ... → Used to spread the elements of a list or map.

```dart
// Accessing a property .
final Dog dog = Dog();
final void name3 = dog.makeSound();

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

// Spreading elements ...
final List<int> list1 = <int>[1, 2, 3];
final List<int> list2 = <int>[0, ...list1, 4, 5]; // [0, 1, 2, 3, 4, 5]
```

4- ? → Used to check if an object is null before accessing its properties or methods.

5- ?? → Used to provide a default value if an object is null.

6- ! → Used to tell the analyzer that an expression can't be null.

```dart
// Checking for null ?
final String? name4 = null;
final int? nameLength = name4?.length;

// Providing a default value ??
final String name5 = name4 ?? 'Guest';
💡
// Asserting non-null !
final int nameLength2 = name4!.length;
```


@ Ahmed Hussein
01114484229

Ahmed Hussein
Flutter Developer

THANK YOU

Was This Helpful?

Like, Share and Follow
for more Content

Like Comment Share Save