



# Object- Oriented Principles, Functional Concepts

Module -3

Chapter 2

# Core OOP Concepts

- Dart is an object-oriented programming language.
- OOP in Dart helps create **scalable, maintainable, and reusable software**.

## OOP

- Defining classes and creating objects
- properties, methods, and constructors
- Pillars of OOP: Inheritance, Polymorphism, Abstraction, Encapsulation

# Class and Object

- **Class** is a **blueprint** for creating objects.
- **Object** is an instance of a class — used to access its properties and methods.
  - They have **properties (attributes)** and **methods (actions)**.

## Class Members

- **Fields** – variables inside a class that hold data.
- **Getters/Setters** – control access to private fields.
- **Constructors** – special functions used to initialize objects.
- **Functions (Methods)** – define behaviors of an object.

The screenshot shows a code editor window with a dark theme. At the top, there's a toolbar with 'Run | Debug'. Below it is a code editor area containing the following script:

```
void main()
{
    var n=college();
    n.names();
}

class college
{
    String c='GPTC kkm';
    void names()
    {
        print('college name : $c');
    }
}
```

At the bottom of the editor, there are tabs for 'PROBLEMS' (with a count of 16), 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'DEBUG CONSOLE' tab is currently selected, showing the output:

```
college name : GPTC kkm
Exited.
```

# Functions

Functions are the **building blocks** of readable and reusable code.

## Concepts:

- **Defining a function:** specifies when & how a task is done.
- **Calling a function:** executes it.
- **Returning functions:** send back a value.

Parameterized **functions**: use parameters to pass values.

The screenshot shows a code editor interface with a dark theme. At the top, there's a toolbar with icons for Run, Debug, and other options. Below the toolbar is a status bar with tabs for PROBLEMS (16), OUTPUT, and DEBUG CONSOLE. The main area contains the following C++ code:

```
Run | Debug
void main()
{
    fun();
    print(fun2());
    fun3("function 3");
    fun4();
}
void fun()
{
    print('function 1');
}
string fun2()
{
    return "function 2";
}
void fun3(string st)
{
    print(st);
}
void fun4()
{
    print('function 4');
}
```

The code uses color-coded syntax highlighting: blue for keywords like void, int, and if, purple for identifiers like main, fun, fun2, fun3, fun4, and st, and red for strings like "function 3". There are also some yellow and green highlights on certain characters. In the bottom right corner of the code editor, there's a small circular icon with a question mark.

Below the code editor, the OUTPUT tab is selected, showing the following text:

```
function 1
function 2
function 3
function 4

Exited.
```

The DEBUG CONSOLE tab is also visible but appears to be empty.

# Parameter Function Types

**Optional Positional Parameter:** use [ ]

**Optional Named Parameter:** use { }

- Must specify parameter name when passing value.

**Optional Parameter with Default Value:**

- Assign default values that can be overridden.

```
Run | Debug
1 void main()
2 {
3   position('one', 'two');
4   position('three');
5   print('');
6   Name(two: 'password', one: 'username');
7   print('');
8   def('para 1');
9   def('para 3', two: 'para 4');
10 }
11 void position(String one, [String? two])
12 {
13   print(one);
14   print(two);
15 }
16 void Name({String? one, required String two}){
17   print('parameter1 $one');
18   print('parameter2 $two');
19 }
20 void def(string one, {string two = 'admin'})
21 {
22   print('parameter1 $one');
23   print('parameter2 $two');
24 }
```

```
one
two
three
null

parameter1 username
parameter2 password

parameter1 para 1
parameter2 admin
parameter1 para 3
parameter2 para 4

Exited.
```

```
Run | Debug
1 void main()
2 {
3   position('one','two');
4   position('three');
5   print('');
6   Name(two:'password',one: 'username');
7   print('');
8   def('para 1');
9   def('para 3',two: 'para 4');
10 }
11 void position(String one, [String? two])
12 {
13   print(one);
14   print(two);
15 }
16 void Name({String? one,required String two}){
17   print('parameter1 $one');
18   print('parameter2 $two');
19 }
20 void def(String one, {String two ='admin'})
21 {
22   print('parameter1 $one');
23   print('parameter2 $two');
24 }
```

# Constructors

Special functions with the **same name as the class**.

Execute **automatically** when an object is created.

**Named constructors** allow you to create and call custom constructors.

The screenshot shows a code editor with a Dart file. The code defines a class named 'cars' with a constructor and three methods: 'toyota()', 'suzuki()', and 'bmw()'. The 'main()' function creates two objects, 'obj' and 'obj2', and calls their respective methods. The output window shows the printed results: 'bmw,benz,toyota,suzuki,tata', 'Toyota Supra', 'Suzuki cars:Alto K10, Swift,Baleno', and 'i7'.

```
Run | Debug
void main()
{
  var obj=cars();
  obj.toyota();
  var obj2=cars.suzuki();
  obj2.bmw();
}
class cars
{
  cars()
  {
    print('bmw,benz,toyota,suzuki,tata');
  }
  void toyota()
  {
    print('Toyota Supra');
  }
  cars.suzuki()
  {
    print('Suzuki cars:Alto K10, Swift,Baleno');
  }
  void bmw()
  [
    print('i7');
  ]
}
```

PROBLEMS 16 OUTPUT DEBUG CONSOLE TERMINAL PORTS

bmw,benz,toyota,suzuki,tata  
Toyota Supra  
Suzuki cars:Alto K10, Swift,Baleno  
i7

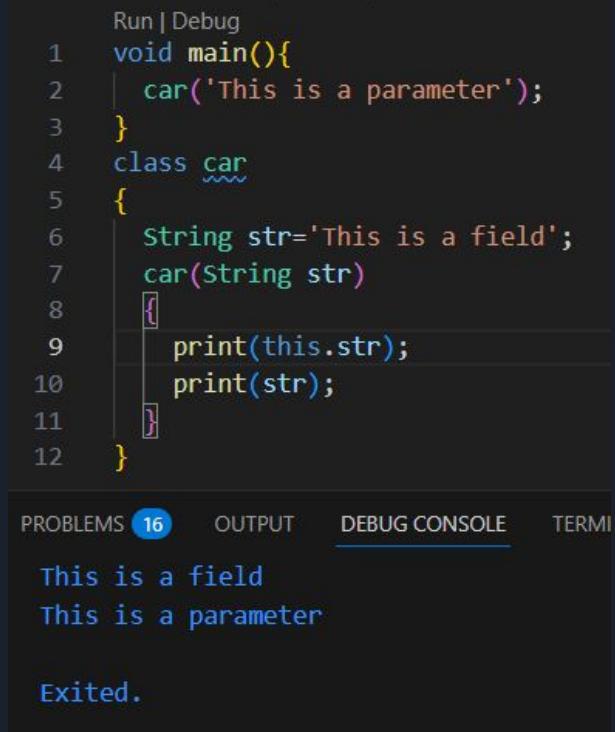
Exited.

# this Keyword

Refers to the **current instance** of a class.

Useful when **instance variables** share names with parameters.

Example: two strings named **str** (one class, one parameter);  
**this.str** accesses the class variable.



The screenshot shows a code editor with Java code. The code defines a class named 'car' with a constructor that takes a string parameter 'str'. Inside the constructor, it prints 'this.str' and the parameter 'str'. The code editor interface includes tabs for Run | Debug, PROBLEMS (16), OUTPUT, DEBUG CONSOLE (underlined), and TERMINAL.

```
Run | Debug
1 void main(){
2     car('This is a parameter');
3 }
4 class car
{
5     String str='This is a field';
6     car(String str)
7     {
8         print(this.str);
9         print(str);
10    }
11 }
```

PROBLEMS 16    OUTPUT    DEBUG CONSOLE    TERMINAL

```
This is a field
This is a parameter

Exited.
```

# Setters / Getters

Defined using `get` and `set` keywords.

**Setter must run first;** otherwise, getter will return `null`.

## Getter

To read data

```
print(obj.name)
```

## Setter

To assign data

```
obj.name = 'value'
```

The screenshot shows a code editor interface with a dark theme. At the top, there's a 'Run | Debug' button. Below it is the code:

```
1 void main(){
2     var obj = car();
3     obj.Name = 'shift';
4     print(obj.Name);
5 }
6 class car{
7     late String suzuki;
8
9     set Name(String s){
10        this.suzuki = s;
11    }
12
13
14     String get Name{
15         return suzuki;
16     }
17 }
18 }
```

A yellow circular icon with the number '16' is located in the bottom left corner of the code editor area. Below the editor, there are tabs for 'PROBLEMS' (with '16'), 'OUTPUT', and 'DEBUG CONSOLE'. The 'DEBUG CONSOLE' tab is currently selected. In the console, the output is:

```
shift
Exited.
```

# Class Inheritance

- Allows one class (child) to access another class (parent).
- Create derived class objects to inherit parent class properties.

## Types of Inheritance:

1. Single
2. Multilevel
3. Hierarchical

```
Run | Debug
1 void main()
2 {
3     var obj= car();
4     print(obj.ca);
5     print(obj.bm);
6     print(obj.i);
7
8 }
9 class car extends bmw
10 [
11     String ca ='CARS';
12 ]
13 class bmw extends i7
14 [
15     String bm ='BMW CAR';
16 ]
17 class i7
18 [
19     String i ='I7 BMW';
20 ]
```

PROBLEMS 16 OUTPUT DEBUG CONSOLE

CARS  
BMW CAR  
I7 BMW

Exited.

# Super Keyword

`super.function()` calls the parent's version of the function.

Determines order of execution based on where it appears in the child function.

```
Run | Debug
1 void main(){
2 var obj = Car();
3 obj.function();
4
5 }
6 class Bike{
7     void function()
8     {
9
10    print('This is from base class');
11 }
12 }
13 class Car extends Bike{
14     void function(){
15     super.function();
16     print('This is from derived class');
17 }
18 }
19
```

PROBLEMS 16 OUTPUT DEBUG CONSOLE ... Filter (e)

This is from base class  
This is from derived class

Exited.

# Abstract Class and Method

- Use **abstract** keyword for abstract class.
- **Abstract method:** use semicolon (;) instead of body.
- When a subclass inherits from the abstract class, it must write the actual code for this method. This process is called overriding.
- The Abstract Class cannot create objects/instances of it. You can only inherit (extend) it with a concrete subclass.

The screenshot shows a code editor interface with a dark theme. At the top, there are buttons for 'Run' and 'Debug'. Below the code area, there are tabs for 'PROBLEMS' (with 16 items), 'OUTPUT', 'DEBUG CONSOLE' (which is underlined, indicating it's active), and '...'. A 'Filter' input field is also present. The code itself is as follows:

```
1 void main(){
2     var x= 20; var y=15;
3     Ans().val(x, y);
4     Ans().add(x, y);
5     Ans().sub(x, y);
6 }
7 abstract class Cal{
8     void val(int x,int y);
9
10    void add(int x,int y){
11        print('add: ${x+y}');
12    }
13    void sub(int x,int y){
14        print('sub: ${x-y}');
15    }
16 }
17 class Ans extends Cal{
18     @override
19     void val(int x,int y){
20         print('number1: $x , number2: $y');
21     }
22 }
```

In the 'OUTPUT' tab, the following logs are displayed:

```
number1: 20 , number2: 15
add: 35
sub: 5
```

At the bottom of the output, the message 'Exited.' is shown.

# Polymorphism

Ability of an object,function or method to take **many forms**.

## Types:

- **Compile-time** (method overloading)

Same method name, different parameters.

- **Run-time** (method overriding)

Example: animal(class) sub - cat ,dog = breed (function )

```
Run | Debug
void main()
{
    Animal cat=Cat();
    Animal dog=Dog();
    cat.breeds();
    dog.breeds();
}
class Animal
{
    void breeds()
    {
        print('animals');
    }
}
class Cat extends Animal
{
    void breeds()
    {
        print('cat: pershian');
    }
}
class Dog extends Animal
{
    void breeds()
    {
        print('dog : beagle');
    }
}
```

PROBLEMS 16 OUTPUT DEBUG CONSOLE ...

cat: pershian  
dog : beagle

Exited.

# Interface

An interface is like a rulebook or contract between classes.

It defines what methods a class must have, but not how they work.

A **normal class** can serve as an **interface** using **implements**.

Must **override all methods** of the implemented class.

Multiple classes can be implemented, but only one class can be extended.

Interfaces are **important in Flutter** for cross-platform behavior.

```
Run | Debug
1 void main()
2 {
3     Vehicles().bmw();
4     print('');
5     Vehicles().suzuki();
6     print('');
7     Vehicles().hero();
8     print('');
9     Vehicles().tvs();
10 }
11 abstract class car {
12     void bmw();
13     void suzuki();
14 }
15 abstract class bike{
16     void hero();
17     void tvs();
18 }
19 class Vehicles implements car,bike{
20     Vehicles(){
21         print('vehicle.....');
22     }
23     @override
24     void bmw(){
25         print('BMW - CAR');
26     }
27     @override
28     void suzuki(){
29         print('suzuki - CAR');
30     }
31     @override
32     void hero(){
33         print('hero bike');
34     }
35     @override
36     void tvs() {
37         print('tvs bike');
38     }
39 }
```

vehicle.....  
BMW - CAR

vehicle.....  
suzuki - CAR

vehicle.....  
hero bike

vehicle.....  
tvs bike

Exited.

# Exercise 1

## Palindrome Number Check

```
while(num > 0)
```

→ Loop runs until `num` becomes 0.

```
remainder = num % 10;
```

→ Gets the last digit of `num`.

Example: if `num = 121`, then `remainder = 1`.

```
reversed = reversed * 10 + remainder;
```

→ Builds the reversed number digit by digit.

Example:  $0 * 10 + 1 = 1$ , then next loop  $\rightarrow 1 * 10 + 2 = 12$ ,  
 $12 * 10 + 1 = 121$

```
num = num ~/ 10;
```

→ Removes the last digit of `num`.

Example:  $121 ~/ 10 = 12$ .

$12 / 10 = 1$ ,  $1 / 10 = 0$

**After loop ends**

→ `reversed` holds the reversed version of the original number.

```
Run | Debug
void main()
{
    int num= 121;
    int originalnum=num;
    int reversed=0;
    int reminder;
    while(num>0)
    {
        reminder =num % 10;
        reversed=reversed*10+reminder;
        num=num~/10;
    }
    if (originalnum==reversed)
    {
        print('It s a palindrome number');
    }
    else
    {
        print('It s Not a palindrome number');
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

It s a palindrome number

Exited.

# Exercise 2

## Multiple Hierarchy

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
Value form class B: value from B
Value form class C: value from C
Value form class D: value from D
Value form class F: value from F
```

Exited.

```
Run | Debug
1 void main() {
2     A a = A();
3     a.printval();
4 }
5
6 class A with B, C, D, F {
7     void printval() {
8         print('Value from class B: $bstr');
9         print('Value from class C: $cstr');
10        print('Value from class D: $dstr');
11        print('Value from class F: $fstr');
12    }
13 }
14
15 mixin B {
16     String bstr = 'value from B';
17 }
18
19 mixin C {
20     String cstr = 'value from C';
21 }
22
23 mixin D {
24     String dstr = 'value from D';
25 }
26
27 mixin F {
28     String fstr = 'value from F';
29 }
```