OOP In Dart

Object-oriented programming (OOP) is a programming method that uses objects and their interactions to design and program applications

In **OOP**, an object can be anything, such as a person, a bank account, a car, or a house. Each object has its attributes (or properties) and behavior (or methods). For example, a person object may have the attributes **name**, **age** and **height**, and the behavior **walk** and **talk**.

Features Of OOP

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

Note: The main purpose of OOP is to break complex problems into smaller objects.

Class In Dart

In object-oriented programming, a class is a blueprint for creating objects. A class defines the properties and methods that an object will have. For example, a class called **Dog** might have properties like **breed**, **color** and methods like **bark**, **run**.

Declaring Class In Dart

You can declare a class in dart using the **class** keyword followed by class name and braces {}. It's a good habit to write class name in **PascalCase**. For example, **Employee**, **Student**, **QuizBrain**, etc.

Syntax

```
class ClassName {
// properties or fields
// methods or functions
}
```

In the above syntax:

- The **class** keyword is used for defining the class.
- **ClassName** is the name of the class and must start with capital letter.
- Body of the class consists of **properties** and **functions**.
- **Properties** are used to store the data. It is also known as **fields** or **attributes**.
- **Functions** are used to perform the operations. It is also known as **methods**.

Example : Declaring A Person Class In Dart

In this example below, there is class **Person** with four properties: **name**, **phone**, **isMarried**, and **age**. The class also has a method called **displayInfo**, which prints out the values of the four properties.

```
class Person {
  String? name;
  String? phone;
  bool? isMarried;
  int? age;

  void displayInfo() {
    print("Person name: $name.");
    print("Phone number: $phone.");
    print("Married: $isMarried.");
    print("Age: $age.");
  }
}
```

Object In Dart

In object-oriented programming, an object is a self-contained unit of code and data. Objects are created from templates called classes. An object is made up of properties(variables) and methods(functions). An object is an instance of a class.

For example, a bicycle object might have attributes like color, size, and current speed. It might have methods like changeGear, pedalFaster, and brake.

Info

Note: To create an object, you must create a class first. It's a good practice to declare the object name in lower case.

Instantiation

In object-oriented programming, instantiation is the process of creating an instance of a class. In other words, you can say that instantiation is the process of creating an object of a class. For example, if you have a class called **Bicycle**, then you can create an object of the class called **bicycle**.

Declaring Object In Dart

Once you have created a class, it's time to declare the object. You can declare an object by the following syntax:

Syntax

```
ClassName objectName = ClassName();
```

Example Declaring An Object In Dart

In this example below, there is class **Bycycle** with three properties: **color**, **size**, and **currentSpeed**. The class has two methods. One is **changeGear**, which changes the gear of the bicycle, and **display** method prints out the values of the three properties. We also have an object of the class **Bycycle** called **bicycle**.

```
class Bicycle {
  String? color;
  int? size;
  int? currentSpeed;

  void changeGear(int newValue) {
    currentSpeed = newValue;
  }

  void display() {
    print("Color: $color");
    print("Size: $size");
    print("Current Speed: $currentSpeed");
  }
}

void main(){
    // Here bicycle is object of class Bicycle.
    Bicycle bicycle = Bicycle();
    bicycle.color = "Red";
    bicycle.size = 26;
    bicycle.currentSpeed = 0;
    bicycle.changeGear(5);
    bicycle.display();
}
```

Show Output

```
Color: Red
Size: 26
Current Speed: 5
```

Here **!** is used to tell the compiler that the variable is not null. If you don't use **!**, then you will get an error. You will learn more about it in null safety later.

Example 3: Find Simple Interest Using Class and Objects

In this example below there is class **SimpleInterest** with three properties: **principal**, **rate**, and **time**. The class also has a method called **interest**, which calculates the simple interest.

```
class SimpleInterest{
  //properties of simple interest
  double? principal;
  double? rate;
  double? time;

  //functions of simple interest
  double interest(){
    return (principal! * rate! * time!)/100;
  }
}
void main(){
  //object of simple interest created
  SimpleInterest simpleInterest = SimpleInterest();

  //setting properties for simple interest
  simpleInterest.principal=1000;
  simpleInterest.rate=10;
  simpleInterest.time=2;

  //functions of simple interest called
  print("Simple Interest is ${simpleInterest.interest()}.");
}
```

Show Output

```
Simple Interest is 200.
```

Constructor In Dart

A constructor is a special method used to initialize an object. It is called automatically when an object is created, and it can be used to set the initial values for the object's properties. For example, the following code creates a **Person** class object and sets the initial values for the **name** and **age** properties.

```
Person person = Person("John", 30);
```

Without Constructor

If you don't define a constructor for class, then you need to set the values of the properties manually. For example, the following code creates a **Person** class object and sets the values for the **name** and **age** properties.

```
Person person = Person():
person.name = "John";
person.age = 30;
```

Things To Remember

- The constructor's name should be the same as the class name.
- Constructor doesn't have any return type.

Syntax

```
class ClassName {
  // Constructor declaration: Same as class name
  ClassName() {
    // body of the constructor
  }
}
```

Info

Note: When you create a object of a class, the constructor is called automatically. It is used to initialize the values when an object is created.

Example 1: How To Declare Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also created an object of the class **Student** called **student**.

```
class Student {
  String? name;
  int? age;
  int? rollNumber;

  // Constructor
  Student(String name, int age, int rollNumber) {
    print(
        "Constructor called"); // this is for checking the constructor is
called or not.
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}

void main() {
  // Here student is object of class Student.
  Student student = Student("John", 20, 1);
  print("Name: ${student.name}");
```

```
  print("Age: ${student.age}");
  print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Constructor called
Name: John
Age: 20
Roll Number: 1
```

Note: The **this** keyword is used to refer to the current instance of the class. It is used to access the current class properties. In the example above, parameter names and class properties of constructor **Student** are the same. Hence to avoid confusion, we use the **this** keyword.

Example 5: Write Constructor Single Line

In the avobe section, you have written the constructor in long form. You can also write the constructor in short form. You can directly assign the values to the properties. For example, the following code is the short form of the constructor in one line.

```
class Person{
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor in short form
  Person(this.name, this.age, this.subject, this.salary);

  // display method
  void display(){
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}");
  }
}

void main(){
  Person person = Person("John", 30, "Maths", 50000.0);
  person.display();
}
```

Show Output

```
Name: John
Age: 30
```

```
Subject: Maths
Salary: 50000
```

Constructor With Optional Parameters

In the example below, we have created a class **Employee** with four properties: **name**, **age**, **subject**, and **salary**. Class has one constructor for initializing the all properties values. For **subject** and **salary**, we have used optional parameters. It means we can pass or not pass the values of **subject** and **salary**. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Employee** called **employee**.

```
class Employee {
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor
  Employee(this.name, this.age, [this.subject = "N/A", this.salary=0]);

  // Method
  void display() {
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}");
  }
}

void main(){
  Employee employee = Employee("John", 30);
  employee.display();
}
```

Show Output

```
Name: John
Age: 30
Subject: N/A
Salary: 0
```

Default Constructor

The constructor which is automatically created by the dart compiler if you don't create a constructor is called a default constructor. A default constructor has no parameters. A default constructor is declared using the class name followed by parentheses ().

Example 1: Default Constructor In Dart

In this example below, there is a class **Laptop** with two properties: **brand**, and **prize**. Lets create constructor with no parameter and print something from the constructor. We also have an object of the class **Laptop** called **laptop**.

```
class Laptop {
  String? brand;
  int? prize;

  // Constructor
  Laptop() {
    print("This is a default constructor");
  }
}

void main() {
  // Here laptop is object of class Laptop.
  Laptop laptop = Laptop();
}
```

Show Output

```
This is a default constructor
```

Note: The default constructor is called automatically when you create an object of the class. It is used to initialize the instance variables of the class.

Parameterized Constructor

Parameterized constructor is used to initialize the instance variables of the class. Parameterized constructor is the constructor that takes parameters. It is used to pass the values to the constructor at the time of object creation.

Syntax

```
class ClassName {
  // Instance Variables
  int? number;
  String? name;
  // Parameterized Constructor
  ClassName(this.number, this.name);
}
```

Example 1: Parameterized Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```
class Student {
  String? name;
  int? age;
  int? rollNumber;
  // Constructor
  Student(this.name, this.age, this.rollNumber);
}

void main(){
    // Here student is object of class Student.
    Student student = Student("John", 20, 1);
    print("Name: ${student.name}");
    print("Age: ${student.age}");
    print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Name: John
Age: 20
Roll Number: 1
```

Example 2: Parameterized Constructor With Named Parameters In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```
class Student {
  String? name;
  int? age;
  int? rollNumber;

  // Constructor
  Student({String? name, int? age, int? rollNumber}) {
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}

void main(){
    // Here student is object of class Student.
    Student student = Student(name: "John", age: 20, rollNumber: 1);
    print("Name: ${student.name}");
    print("Age: ${student.age}");
    print("Roll Number: ${student.rollNumber}");
}
```

Example 3: Parameterized Constructor With Default Values In Dart

In this example below, there is class **Student** with two properties: **name**, and **age**. The class has parameterized constructor with default values. The constructor is used to initialize the values of the two properties. We also have an object of the class **Student** called **student**.

```
    class Student {
      String? name;
      int? age;

      // Constructor
      Student({String? name = "John", int? age = 0}) {
        this.name = name;
        this.age = age;
      }
    }

    void main(){
        // Here student is object of class Student.
        Student student = Student();
        print("Name: ${student.name}");
        print("Age: ${student.age}");
    }
```

Note: In parameterized constructor, at the time of object creation, you must pass the parameters through the constructor which initialize the variables value, avoiding the null values.

: Named Constructor In Dart

In this example below, there is a class **Animal** with two properties **name** and **age**. The class has three constructors. The first constructor is a default constructor. The second and third constructors are named constructors. The second constructor is used to initialize the values of name and age, and the third constructor is used to initialize the value of name only. We also have an object of the class **Animal** called **animal**.

```
class Animal {
  String? name;
  int? age;

  // Default Constructor
  Animal() {
    print("This is a default constructor");
  }

  // Named Constructor
```

```
  Animal.namedConstructor(String name, int age) {
    this.name = name;
    this.age = age;
  }

  // Named Constructor
  Animal.namedConstructor2(String name) {
    this.name = name;
  }
}
void main(){
  // Here animal is object of class Animal.
  Animal animal = Animal.namedConstructor("Dog", 5);
  print("Name: ${animal.name}");
  print("Age: ${animal.age}");

  Animal animal2 = Animal.namedConstructor2("Cat");
  print("Name: ${animal2.name}");
}
```

Show Output

```
Name: Dog
Age: 5
Name: Cat
```

Constant Constructor In Dart

Constant constructor is a constructor that creates a constant object. A constant object is an object whose value cannot be changed. A constant constructor is declared using the keyword **const**.

Info

Note: **Constant Constructor** is used to create a object whose value cannot be changed. It Improves the performance of the program.

Rule For Declaring Constant Constructor In Dart

- All properties of the class must be final.
- It does not have any body.
- Only class containing **const** constructor is initialized using the **const** keyword.

Example 1: Constant Constructor In Dart

In this example below, there is a class **Point** with two final properties: **x** and **y**. The class also has a constant constructor that initializes the two properties. The class also has a method called **display**, which prints out the values of the two properties.

```dart
class Point {
  final int x;
  final int y;

  const Point(this.x, this.y);
}

void main() {
  // p1 and p2 has the same hash code.
  Point p1 = const Point(1, 2);
  print("The p1 hash code is: ${p1.hashCode}");

  Point p2 = const Point(1, 2);
  print("The p2 hash code is: ${p2.hashCode}");
  // without using const
  // this has different hash code.
  Point p3 = Point(2, 2);
  print("The p3 hash code is: ${p3.hashCode}");

  Point p4 = Point(2, 2);
  print("The p4 hash code is: ${p4.hashCode}");
}
```

Show Output

```
The p1 hash code is: 918939239
The p2 hash code is: 918939239
The p3 hash code is: 745146896
The p4 hash code is: 225789186
```

Factory Constructor In Dart

All of the constructors that you have learned until now are **generative constructors**. Dart also provides a special type of constructor called a **factory constructor**.

A **factory constructor** gives more flexibility to create an object. Generative constructors only create an instance of the class. But, the factory constructor can return an instance of the **class or even subclass**. It is also used to return the **cached instance** of the class.

Syntax

```dart
class ClassName {
  factory ClassName() {
    // TODO: return ClassName instance
  }

  factory ClassName.namedConstructor() {
    // TODO: return ClassName instance
  }
```

```
}
```

Rules For Factory Constructors

- Factory constructor must return an instance of the **class** or **sub-class**.
- You can't use **this** keyword inside factory constructor.
- It can be **named** or **unnamed** and called like normal constructor.
- It can't access **instance members** of the class.

Example 1: Without Factory Constructor

In this example below, there is a class named **Area** with final properties **length** and **breadth**, and **area**. When you pass the **length** and **breadth** to the constructor, it calculates the **area** and stores it in the **area** property.

Info

Note: An initializer list allows you to assign properties to a new instance variable before the constructor body runs, but after creation.

```
class Area {
  final int length;
  final int breadth;
  final int area;

  // Initializer list
 const Area(this.length, this.breadth) : area = length * breadth;
}

void main() {
  Area area = Area(10, 20);
  print("Area is: ${area.area}");

  // notice that here is a negative value
  Area area2 = Area(-10, 20);
  print("Area is: ${area2.area}");
}
```

Show Output

```
Area is: 200
Area is: -200
```

Here **area2** object has a negative value. This is because we are not validating the input. Let's create a factory constructor to validate the input.

Example 2: With Factory Constructor

In this example below, **factory constructor** is used to validate the input. If the input is valid, it will return a new class instance. If the input is invalid, then it will throw an exception.

```
class Area {
  final int length;
  final int breadth;
  final int area;

  // private constructor
  const Area._internal(this.length, this.breadth) : area = length *
breadth;

  // Factory constructor
  factory Area(int length, int breadth) {
    if (length < 0 || breadth < 0) {
      throw Exception("Length and breadth must be positive");
    }
    // redirect to private constructor
    return Area._internal(length, breadth);
  }
}

void main() {
  // This works
  Area area = Area(10, 20);
  print("Area is: ${area.area}");

  // notice that here is negative valu
  Area area2 = Area(-10, 20);
  print("Area is: ${area2.area}");
}
```

Show Output

```
Area is: 200
Unhandled exception:
Exception: Length and breadth must be positive
```

Encapsulation In Dart

In Dart, **Encapsulation** means **hiding data** within a library, preventing it from outside factors. It helps you control your program and prevent it from becoming too complicated.

What Is Library In Dart?

By default, every **.dart** file is a library. A library is a collection of functions and classes. A library can be imported into another library using the **import** keyword.

How To Achieve Encapsulation In Dart?

Encapsulation can be achieved by:

- Declaring the class properties as **private** by using **underscore(_)**.
- Providing public **getter** and **setter** methods to access and update the value of private property.

Info

Note: Dart doesn't support keywords like **public**, **private**, and **protected**. Dart uses _ (underscore) to make a property or method private. The encapsulation happens at library level, not at class level.

Getter and Setter Methods

Getter and **setter** methods are used to access and update the value of private property. **Getter** methods are used to access the value of private property. **Setter** methods are used to update the value of private property.

Example 1: Encapsulation In Dart

In this example, we will create a class named **Employee**. The class will have two private properties **_id** and **_name**. We will also create two public methods **getId()** and **getName()** to access the private properties. We will also create two public methods **setId()** and **setName()** to update the private properties.

```
class Employee {
  // Private properties
  int? _id;
  String? _name;

// Getter method to access private property _id
  int getId() {
    return _id!;
  }
// Getter method to access private property _name
  String getName() {
    return _name!;
  }
// Setter method to update private property _id
  void setId(int id) {
```

```
    this._id = id;
  }
// Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }

}

void main() {
  // Create an object of Employee class
  Employee emp = new Employee();
  // setting values to the object using setter
  emp.setId(1);
  emp.setName("John");

  // Retrieve the values of the object using getter
  print("Id: ${emp.getId()}");
  print("Name: ${emp.getName()}");
}
```

Show Output

```
Id: 1
Name: John
```

Private Properties

Private property is a property that can only be accessed from same **library**. Dart does not have any keywords like **private** to define a private property. You can define it by prefixing an **underscore (_)** to its name.

Example 2: Private Properties In Dart

In this example, we will create a class named **Employee**. The class has one private property **_name**. We will also create a public method **getName()** to access the private property.

```
class Employee {
  // Private property
  var _name;

  // Getter method to access private property _name
  String getName() {
    return _name;
  }

  // Setter method to update private property _name
  void setName(String name) {
```

```
    this._name = name;
  }
}

void main() {
  var employee = Employee();
  employee.setName("Jack");
  print(employee.getName());
}
```

Show Output

```
Jack
```

Why Aren't Private Properties Private?

In the main method, if you write the following code, it will compile and run without any error. Let's see why it is happening.

```
class Employee {
  // Private property
  var _name;

  // Getter method to access private property _name
  String getName() {
    return _name;
  }

  // Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }
}

void main() {
  var employee = Employee();
  employee._name = "John"; // It is working, but why?
  print(employee.getName());
}
```

Show Output

```
John
```

Reason

The reason is that using **underscore (_)** before a variable or method name makes it **library private** not **class private**. It means that the variable or method is only visible to the library in which it is declared. It is not visible to any other library. In simple words, library is one file. If you write the main method in a separate file, this will not work.

Solution

To see private properties in action, you must create a separate file for the class and import it into the main file.

Read-only Properties

You can control the properties's access and implement the encapsulation in the dart by using the read-only properties. You can do that by adding the **final** keyword before the properties declaration. Hence, you can only access its value, but you cannot change it.

How To Create Getter and Setter Methods?

You can create getter and setter methods by using the **get** and **set** keywords. In this example below, we have created a class named **Vehicle**. The class has two private properties **_model** and **_year**. We have also created two getter and setter methods for each property. The getter and setter methods are named **model** and **year**. The getter and setter methods are used to access and update the value of the private properties.

```
class Vehicle {
  String _model;
  int _year;

  // Getter method
  String get model => _model;

  // Setter method
  set model(String model) => _model = model;

  // Getter method
  int get year => _year;

  // Setter method
  set year(int year) => _year = year;
}

void main() {
  var vehicle = Vehicle();
  vehicle.model = "Toyota";
  vehicle.year = 2019;
  print(vehicle.model);
  print(vehicle.year);
}
```

Show Output

```
Toyota
2019
```

Note: In dart, any identifier like (class, class properties, top-level function, or variable) that starts with an underscore _ it is private to its library.

Why Encapsulation Is Important?

- **Data Hiding**: Encapsulation hides the data from the outside world. It prevents the data from being accessed by the code outside the class. This is known as data hiding.
- **Testability**: Encapsulation allows you to test the class in isolation. It will enable you to test the class without testing the code outside the class.
- **Flexibility**: Encapsulation allows you to change the implementation of the class without affecting the code outside the class.
- **Security**: Encapsulation allows you to restrict access to the class members. It will enable you to limit access to the class members from the code outside the library

Getter In Dart

Getter is used to get the value of a property. It is mostly used to access a **private property's** value. Getter provide explicit read access to an object properties.

Syntax

```
return_type get property_name {
  // Getter body
}
```

Note: Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

Example 1: Getter In Dart

In this example below, there is a class named **Person**. The class has two properties **firstName** and **lastName**. There is getter **fullName** which is responsible to get full name of person.

```
class Person {
  // Properties
  String? firstName;
  String? lastName;

  // Constructor
  Person(this.firstName, this.lastName);

  // Getter
  String get fullName => "$firstName $lastName";
}
```

```
void main() {
  Person p = Person("John", "Doe");
  print(p.fullName);
}
```

Show Output

```
John Doe
```

Example 2: Getter In Dart

In this example below, there is a class named **NoteBook**. The class has two private properties _**name** and _**prize**. There are two getters **name** and **prize** to access the value of the properties.

```
class NoteBook {
  // Private properties
  String? _name;
  double? _prize;

  // Constructor
  NoteBook(this._name, this._prize);

  // Getter method to access private property _name
  String get name => this._name!;

  // Getter method to access private property _prize
  double get prize => this._prize!;
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook("Dell", 500);
  // Display the values of the object
  print(nb.name);
  print(nb.prize);
}
```

Show Output

```
Name: Dell
Price: 500.0
```

Setter In Dart

Setter is used to set the value of a property. It is mostly used to update a **private property's** value. Setter provide explicit write access to an object properties.

Syntax

```
set property_name (value) {
```

```
  // Setter body
}
```

Note: Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

Example 1: Setter In Dart

In this example below, there is a class named **NoteBook**. The class has two private properties _**name** and _**prize**. There are two setters **name** and **prize** to update the value of the properties. There is also a method **display** to display the value of the properties.

```
class NoteBook {
  // Private Properties
  String? _name;
  double? _prize;

  // Setter to update private property _name
  set name(String name) => this._name = name;

  // Setter to update private property _prize
  set prize(double prize) => this._prize = prize;

  // Method to display the values of the properties
  void display() {
    print("Name: ${_name}");
    print("Price: ${_prize}");
  }
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook();
  // setting values to the object using setter
  nb.name = "Dell";
  nb.prize = 500.00;
  // Display the values of the object
  nb.display();
}
```

Show Output

```
Name: Dell
Price: 500.0
```

Getter And Setter

Getter and **Setter** provide explicit read and write access to an object properties. In dart, **get** and **set** are the keywords used to create getter and setter. Getter read the value of property and act as **accessor**. Setter update the value of property and act as **mutator**.

Info

Note: You can use same name for **getter** and **setter**. But, you can't use same name for **getter**, **setter** and **property name**.

Use Of Getter and Setter

- Validate the data before reading or writing.
- Restrict the read and write access to the properties.
- Making the properties read-only or write-only.
- Perform some action before reading or writing the properties.

Example 1: Getter And Setter In Dart

In this example below, there is a class named **Student** with three private properties **_firstName**, **_lastName** and **_age**. There are two getters **fullName** and **age** to get the value of the properties. There are also three setters **firstName**, **lastName** and **age** to update the value of the properties. If **age** is less than 0, it will throw an error.

```
class Student {
  // Private Properties
  String? _firstName;
  String? _lastName;
  int? _age;

  // Getter to get full name
  String get fullName => this._firstName! + " " + this._lastName!;

  // Getter to read private property _age
  int get age => this._age!;

  // Setter to update private property _firstName
  set firstName(String firstName) => this._firstName = firstName;

  // Setter to update private property _lastName
  set lastName(String lastName) => this._lastName = lastName;

  // Setter to update private property _age
  set age(int age) {
```

```
    if (age < 0) {
      throw new Exception("Age can't be less than 0");
    }
    this._age = age;
  }
}

void main() {
  // Create an object of Student class
  Student st = new Student();
  // setting values to the object using setter
  st.firstName = "John";
  st.lastName = "Doe";
  st.age = 20;
  // Display the values of the object
  print("Full Name: ${st.fullName}");
  print("Age: ${st.age}");
}
```

Show Output

```
Full Name: John Doe
Age: 20
```

Example 2: Getter And Setter In Dart

In this example below, there is a class named **BankAccount** with one private property _**balance**. There is one getter **balance** to read the value of the property. There are methods **deposit** and **withdraw** to update the value of the _**balance**.

```
class BankAccount {
  // Private Property
  double _balance = 0.0;

  // Getter to read private property _balance
  double get balance => this._balance;

  // Method to deposit money
  void deposit(double amount) {
    this._balance += amount;
  }

  // Method to withdraw money
  void withdraw(double amount) {
    if (this._balance >= amount) {
      this._balance -= amount;
    } else {
      throw new Exception("Insufficient Balance");
    }
```

```
  }
}

void main() {
  // Create an object of BankAccount class
  BankAccount account = new BankAccount();
  // Deposit money
  account.deposit(1000);
  // Display the balance
  print("Balance after deposit: ${account.balance}");
  // Withdraw money
  account.withdraw(500);
  // Display the balance
  print("Balance after withdraw: ${account.balance}");
}
```

Show Output

```
Balance after deposit: 1000
Balance after withdraw: 500
```

When To Use Getter And Setter

- Use getter and setter when you want to restrict the access to the properties.
- Use getter and setter when you want to perform some action before reading or writing the properties.
- Use getter and setter when you want to validate the data before reading or writing the properties.
- Don't use getter and setter when you want to make the properties read-only or write-only

Inheritance In Dart

Inheritance is a sharing of behaviour between two classes. It allows you to define a class that extends the functionality of another class. The **extend** keyword is used for inheriting from parent class.

Note: Whenever you use inheritance, it always create a **is-a** relation between the parent and child class like **Student is a Person**, **Truck is a Vehicle**, **Cow is a Animal** etc.

Dart supports single inheritance, which means that a class can only inherit from a single class. Dart does not support multiple inheritance which means that a class cannot inherit from multiple classes.

Syntax

```
class ParentClass {
  // Parent class code
}

class ChildClass extends ParentClass {
```

```
  // Child class code
}
```

In this syntax, **ParentClass** is the super class and **ChildClass** is the sub class. The **ChildClass** inherits the properties and methods of the **ParentClass**.

Terminology

Parent Class: The class whose properties and methods are inherited by another class is called parent class. It is also known as base class or super class.

Child Class: The class that inherits the properties and methods of another class is called child class. It is also known as derived class or sub class.

Example 1: Inheritance In Dart

In this example, we will create a class **Person** and then create a class **Student** that inherits the properties and methods of the **Person** class.

```
class Person {
  // Properties
  String? name;
  int? age;

  // Method
  void display() {
    print("Name: $name");
    print("Age: $age");
  }
}
// Here In student class, we are extending the
// properties and methods of the Person class
class Student extends Person {
  // Fields
  String? schoolName;
  String? schoolAddress;

  // Method
  void displaySchoolInfo() {
    print("School Name: $schoolName");
    print("School Address: $schoolAddress");
  }
}

void main() {
  // Creating an object of the Student class
  var student = Student();
```

```
  student.name = "John";
  student.age = 20;
  student.schoolName = "ABC School";
  student.schoolAddress = "New York";
  student.display();
  student.displaySchoolInfo();
}
```

Show Output

```
Name: John
Age: 20
School Name: ABC School
School Address: New York
```

Advantages Of Inheritance In Dart

- It promotes reusability of the code and reduces redundant code.
- It helps to design a program in a better way.
- It makes code simpler, cleaner and saves time and money on maintenance.
- It facilitates the creation of class libraries.
- It can be used to enforce standard interface to all children classes.

Example 2: Inheritance In Dart

In this example, here is parent class **Car** and child class **Toyota**. The **Toyota** class inherits the properties and methods of the **Car** class.

```
class Car{
  String color;
  int year;

  void start(){
    print("Car started");
  }
}

class Toyota extends Car{
  String model;
  int prize;

  void showDetails(){
    print("Model: $model");
    print("Prize: $prize");
  }
}

void main(){
  var toyota = Toyota();
  toyota.color = "Red";
```

```
  toyota.year = 2020;
  toyota.model = "Camry";
  toyota.prize = 20000;
  toyota.start();
  toyota.showDetails();
}
```

Show Output

```
Car started
Model: Camry
Prize: 20000
```

Types Of Inheritance In Dart

1. **Single Inheritance** - In this type of inheritance, a class can inherit from only one class. In Dart, we can only extend one class at a time.
2. **Multilevel Inheritance** - In this type of inheritance, a class can inherit from another class and that class can also inherit from another class. In Dart, we can extend a class from another class which is already extended from another class.
3. **Hierarchical Inheritance** - In this type of inheritance, parent class is inherited by multiple subclasses. For example, the **Car** class can be inherited by the **Toyota** class and **Honda** class.
4. **Multiple Inheritance** - In this type of inheritance, a class can inherit from multiple classes. **Dart does not support multiple inheritance.** For e.g. **Class Toyota extends Car, Vehicle {}** is not allowed in Dart.

Example 3: Single Inheritance In Dart

In this example below, there is super class named **Car** with two properties **name** and **prize**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties.

```
class Car {
  // Properties
  String? name;
  double? prize;
}

class Tesla extends Car {
  // Method to display the values of the properties
  void display() {
    print("Name: ${name}");
    print("Prize: ${prize}");
  }
}

void main() {
```

```
  // Create an object of Tesla class
  Tesla t = new Tesla();
  // setting values to the object
  t.name = "Tesla Model 3";
  t.prize = 50000.00;
  // Display the values of the object
  t.display();
}
```

Show Output

```
Name: Tesla Model 3
Prize: 50000.0
```

Example 4: Multilevel Inheritance In Dart

In this example below, there is super class named **Car** with two properties **name** and **prize**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties. There is another sub class named **Model3** which inherits the properties of the sub class **Tesla**. The sub class has a property **color** and a method **display** to display the values of the properties.

```
class Car {
// Properties
String? name;
double? prize;
}

class Tesla extends Car {
// Method to display the values of the properties
void display() {
  print("Name: ${name}");
  print("Prize: ${prize}");
}
}

class Model3 extends Tesla {
// Properties
String? color;

// Method to display the values of the properties
void display() {
  super.display();
  print("Color: ${color}");
}
}

void main() {
```

```
// Create an object of Model3 class
Model3 m = new Model3();
// setting values to the object
m.name = "Tesla Model 3";
m.prize = 50000.00;
m.color = "Red";
// Display the values of the object
m.display();
}
```

Show Output

```
Name: Tesla Model 3
Prize: 50000.0
Color: Red
```

Note: Here super keyword is used to call the method of the parent class.

Example 5: Multilevel Inheritance In Dart

In this example below, there is class named **Person** with two properties **name** and **age**. There is sub class named **Doctor** with properties **listofdegrees** and **hospitalname**. There is another subclass named **Specialist** with property **specialization**. The sub class has a method **display** to display the values of the properties.

```
class Person {
  // Properties
  String? name;
  int? age;
}

class Doctor extends Person {
  // Properties
  List<String>? listofdegrees;
  String? hospitalname;

  // Method to display the values of the properties
  void display() {
    print("Name: ${name}");
    print("Age: ${age}");
    print("List of Degrees: ${listofdegrees}");
    print("Hospital Name: ${hospitalname}");
  }
}

class Specialist extends Doctor {
  // Properties
  String? specialization;
```

```
  // Method to display the values of the properties
  void display() {
    super.display();
    print("Specialization: ${specialization}");
  }
}

void main() {
  // Create an object of Specialist class
  Specialist s = new Specialist();
  // setting values to the object
  s.name = "John";
  s.age = 30;
  s.listofdegrees = ["MBBS", "MD"];
  s.hospitalname = "ABC Hospital";
  s.specialization = "Cardiologist";
  // Display the values of the object
  s.display();
}
```

Show Output

```
Name: John
Age: 30
List of Degrees: [MBBS, MD]
Hospital Name: ABC Hospital
Specialization: Cardiologist
```

Example 6: Hierarchical Inheritance In Dart

In this example below, there is class named **Shape** with two properties **diameter1** and **diameter2**. There is sub class named **Rectangle** with method **area** to calculate the area of the rectangle. There is another subclass named **Triangle** with method **area** to calculate the area of the triangle.

```
class Shape {
  // Properties
  double? diameter1;
  double? diameter2;
}

class Rectangle extends Shape {
  // Method to calculate the area of the rectangle
  double area() {
    return diameter1! * diameter2!;
  }
}
```

```
class Triangle extends Shape {
  // Method to calculate the area of the triangle
  double area() {
    return 0.5 * diameter1! * diameter2!;
  }
}

void main() {
  // Create an object of Rectangle class
  Rectangle r = new Rectangle();
  // setting values to the object
  r.diameter1 = 10.0;
  r.diameter2 = 20.0;
  // Display the area of the rectangle
  print("Area of the rectangle: ${r.area()}");

  // Create an object of Triangle class
  Triangle t = new Triangle();
  // setting values to the object
  t.diameter1 = 10.0;
  t.diameter2 = 20.0;
  // Display the area of the triangle
  print("Area of the triangle: ${t.area()}");
}
```

Show Output

```
Area of the rectangle: 200.0
Area of the triangle: 100.0
```

Key Points

- Inheritance is used to reuse the code.
- Inheritance is a concept which is achieved by using the **extends** keyword.
- Properties and methods of the super class can be accessed by the sub class.
- Class **Dog** extends class **Animal**{} means Dog is sub class and Animal is super class.
- The sub class can have its own properties and methods.

Why Dart Does Not Support Multiple Inheritance?

Dart does not support multiple inheritance because it can lead to ambiguity. For example, if class **Apple** inherits class **Fruit** and class **Vegetable**, then there may be two methods with the same name **eat**. If the method is called, then which method should be called? This is the reason why Dart does not support multiple inheritance.

What's problem Of Copy Paste Instead Of Inheritance?

If you copy the code from one class to another class, then you will have to maintain the code in both the classes. If you make any changes in one class, then you will have to

make the same changes in the other class. This can lead to errors and bugs in the code.

Does Inheritance Finished If I Learned Extending Class?

No, there is a lot more to learn about inheritance. You need to learn about **Constructor Inheritance**, **Method Overriding**, **Abstract Class**, **Interface** and **Mixin** etc. You will learn about these concepts in the next chapters.

What Is Inheritance Of Constructor In Dart?

Inheritance of constructor in Dart is a process of inheriting the constructor of the parent class to the child class. It is a way of reusing the code of the parent class.

Example 1: Inheritance Of Constructor In Dart

In this example below, there is class named **Laptop** with a constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor.

```
class Laptop {
  // Constructor
  Laptop() {
    print("Laptop constructor");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook() {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook();
}
```

Show Output

```
Laptop constructor
MacBook constructoR
```

Note: The constructor of the parent class is called first and then the constructor of the child class is called.

Example 2: Inheritance Of Constructor With Parameters In Dart

In this example below, there is class named **Laptop** with a constructor with parameters.
There is another class named **MacBook** which extends the **Laptop** class.
The **MacBook** class has its own constructor with parameters.

```
class Laptop {
  // Constructor
  Laptop(String name, String color) {
    print("Laptop constructor");
    print("Name: $name");
    print("Color: $color");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook(String name, String color) : super(name, color) {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook("MacBook Pro", "Silver");
}
```

Show Output

```
Laptop constructor
Name: MacBook Pro
Color: Silver
MacBook constructo
```

Example 3: Inheritance Of Constructor

In this example below, there is class named **Person** with properties **name** and **age**.
There is another class named **Student** which extends the **Person** class.
The **Student** class has additional property **rollNumber**. Lets see how to create a constructor for the **Student** class.

```
class Person {
  String name;
  int age;

  // Constructor
  Person(this.name, this.age);
}

class Student extends Person {
```

```
  int rollNumber;

  // Constructor
  Student(String name, int age, this.rollNumber) : super(name, age);
}

void main() {
  var student = Student("John", 20, 1);
  print("Student name: ${student.name}");
  print("Student age: ${student.age}");
  print("Student roll number: ${student.rollNumber}");

}
```

Show Output

```
Student name: John
Student age: 20
Student roll number: 1
```

Example 4: Inheritance Of Constructor With Named Parameters In Dart

In this example below, there is class named **Laptop** with a constructor with named parameters. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters.

```
class Laptop {
  // Constructor
  Laptop({String name, String color}) {
    print("Laptop constructor");
    print("Name: $name");
    print("Color: $color");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook({String name, String color}) : super(name: name, color: color) {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook(name: "MacBook Pro", color: "Silver");
}
```

Show Output

```
Laptop constructor
Name: MacBook Pro
Color: Silver
```

```
MacBook constructor
```

Example 5: Calling Named Constructor Of Parent Class In Dart

In this example below, there is class named **Laptop** with one default constructor and one named constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters. You can call the named constructor of the parent class using the **super** keyword.

```
class Laptop {
  // Default Constructor
  Laptop() {
    print("Laptop constructor");
  }

  // Named Constructor
  Laptop.named() {
    print("Laptop named constructor");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook() : super.named() {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook();
}
```

Show Output

```
Laptop named constructor
MacBook constructor
```

What Is Super In Dart?

Super is used to refer to the parent class. It is used to call the parent class's properties and methods.

Example 1: Super In Dart

In this example below, the **show()** method of the **MacBook** class calls the **show()** method of the parent class using the **super** keyword.

```
class Laptop {
  // Method
    void show() {
```

```
        print("Laptop show method");
    }
}

class MacBook extends Laptop {
    void show() {
        super.show(); // Calling the show method of the parent class
        print("MacBook show method");
    }
}

void main() {
  // Creating an object of the MacBook class
  MacBook macbook = MacBook();
  macbook.show();
}
```

Show Output

```
Laptop show method
MacBook show method
```

Example 2: Accessing Super Properties In Dart

In this example below, the **display()** method of the **Tesla** class calls the **noOfSeats** property of the parent class using the **super** keyword.

```
class Car {
  int noOfSeats = 4;
}

class Tesla extends Car {
  int noOfSeats = 6;

  void display() {
    print("No of seats in Tesla: $noOfSeats");
    print("No of seats in Car: ${super.noOfSeats}");
  }
}

void main() {
  var tesla = Tesla();
  tesla.display();
}
```

Show Output

```
No of seats in Tesla: 6
No of seats in Car: 4
```

Run Online

Example 3: Super With Constructor In Dart

In this example below, the **Manager** class constructor calls the **Employee** class constructor using the **super** keyword.

```
class Employee {
  // Constructor
  Employee(String name, double salary) {
    print("Employee constructor");
    print("Name: $name");
    print("Salary: $salary");
  }
}

class Manager extends Employee {
  // Constructor
  Manager(String name, double salary) : super(name, salary) {
    print("Manager constructor");
  }
}

void main() {
  Manager manager = Manager("John", 25000.0);
}
```

Show Output

```
Employee constructor
Name: John
Salary: 25000.0
Manager constructor
```

Example 4: Super With Named Constructor In Dart

In this example below, the **Manager** class named constructor calls the **Employee** class named constructor using the **super** keyword.

```
class Employee {
  // Named constructor
  Employee.manager() {
    print("Employee named constructor");
  }
}

class Manager extends Employee {
  // Named constructor
```

```
  Manager.manager() : super.manager() {
    print("Manager named constructor");
  }
}

void main() {
  Manager manager = Manager.manager();
}
```

Show Output

```
Employee named constructor
Manager named constructor
```

Example 5: Super With Multilevel Inheritance In Dart

In this example below, the **MacBookPro** class method **display** calls the **display** method of the parent class **MacBook** using the **super** keyword. The **MacBook** class method **display** calls the **display** method of the parent class **Laptop** using the **super** keyword.

```
class Laptop {
  // Method
  void display() {
    print("Laptop display");
  }
}

class MacBook extends Laptop {
  // Method
  void display() {
    print("MacBook display");
    super.display();
  }
}

class MacBookPro extends MacBook {
  // Method
  void display() {
    print("MacBookPro display");
    super.display();
  }
}

void main() {
  var macbookpro = MacBookPro();
  macbookpro.display();
}
```

Show Output

```
MacBookPro display
MacBook display
Laptop display
```

Run Online

Key Points To Remember

- The **super** keyword is used to access the parent class members.
- The **super** keyword is used to call the method of the parent class.

Polymorphism In Dart

Poly means **many** and morph means **forms**. Polymorphism is the ability of an object to take on many forms. As humans, we have the ability to take on many forms. We can be a student, a teacher, a parent, a friend, and so on. Similarly, in object-oriented programming, polymorphism is the ability of an object to take on many forms.

Info

Note: In the real world, polymorphism is updating or modifying the feature, function, or implementation that already exists in the parent class.

Polymorphism By Method Overriding

Method overriding is a technique in which you can create a method in the child class that has the same name as the method in the parent class. The method in the child class overrides the method in the parent class.

Syntax

```
class ParentClass{
void functionName(){
  }
}
class ChildClass extends ParentClass{
@override
void functionName(){
  }
}
```

Example 1: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Animal** with a method named **eat()**. The **eat()** method is overridden in the child class named **Dog**.

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

```
}

class Dog extends Animal {
  @override
  void eat() {
    print("Dog is eating");
  }
}

void main() {
  Animal animal = Animal();
  animal.eat();

  Dog dog = Dog();
  dog.eat();
}
```

Show Output

```
Animal is eating
Dog is eating
```

Run Online

Example 2: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Vehicle** with a method named **run()**. The **run()** method is overridden in the child class named **Bus**.

```
class Vehicle {
  void run() {
    print("Vehicle is running");
  }
}

class Bus extends Vehicle {
  @override
  void run() {
    print("Bus is running");
  }
}

void main() {
  Vehicle vehicle = Vehicle();
  vehicle.run();

  Bus bus = Bus();
  bus.run();
}
```

Show Output

```
Vehicle is running
Bus is running
```

Run Online

Info

Note: If you don't write @**override**, the program still runs. But, it is a good practice to write @**override**.

Example 3: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Car** with a method named **power()**. The **power()** method is overridden in two child classes named **Honda** and **Tesla**.

```
class Car{
  void power(){
    print("It runs on petrol.");
  }
}

class Honda extends Car{

}
class Tesla extends Car{
  @override
  void power(){
    print("It runs on electricity.");
  }
}

void main(){
  Honda honda=Honda();
  Tesla tesla=Tesla();

  honda.power();
  tesla.power();
}
```

Show Output

```
It runs on petrol.
It runs on electricity.
```

Run Online

Example 4: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Employee** with a method named **salary()**. The **salary()** method is overridden in two child classes named **Manager** and **Developer**.

```
class Employee{
  void salary(){
    print("Employee salary is \$1000.");
  }
}

class Manager extends Employee{
  @override
  void salary(){
    print("Manager salary is \$2000.");
  }
}

class Developer extends Employee{
  @override
  void salary(){
    print("Developer salary is \$3000.");
  }
}

void main(){
  Manager manager=Manager();
  Developer developer=Developer();

  manager.salary();
  developer.salary();
}
```

Show Output

```
Manager salary is $2000.
Developer salary is $3000.
```

Run Online

Advantage Of Polymorphism In Dart

- Subclasses can override the behavior of the parent class.
- It allows us to write code that is more flexible and reusable.

Static In Dart

If you want to define a variable or method that is shared by all instances of a class, you can use the **static** keyword. Static members are accessed using the class name. It is used for **memory management**.

Dart Static Variable

A static variable is a variable that is shared by all instances of a class. It is declared using the static keyword. It is initialized only once when the class is loaded. It is used to store the **class-level data**.

How To Declare A Static Variable In Dart

To declare a static variable in Dart, you must use the static keyword before the variable name.

```
class ClassName {
  static dataType variableName;
}
```

How To Initialize A Static Variable In Dart

To initialize a static variable simply assign a value to it.

```
class ClassName {
  static dataType variableName = value;
  // for e.g
  // static int num = 10;
  // static String name = "Dart";
}
```

How To Access A Static Variable In Dart

You need to use the **ClassName.variableName** to access a static variable in Dart.

```
class ClassName {
  static dataType variableName = value;
  // Accessing the static variable inside same class
  void display() {
    print(variableName);
  }
}

void main() {
  // Accessing static variable outside the class
  dataType value =ClassName.variableName;
}
```

Example 1: Static Variable In Dart

In this example below, there is a class named **Employee**. The class has a static variable **count** to count the number of employees.

```
class Employee {
```

```
  // Static variable
  static int count = 0;
  // Constructor
  Employee() {
    count++;
  }
  // Method to display the value of count
  void totalEmployee() {
    print("Total Employee: $count");
  }
}

void main() {
  // Creating objects of Employee class
  Employee e1 = new Employee();
  e1.totalEmployee();
  Employee e2 = new Employee();
  e2.totalEmployee();
  Employee e3 = new Employee();
  e3.totalEmployee();
}
```

Show Output

```
Total Employee: 1
Total Employee: 2
Total Employee: 3
```

Run Online

Info

Note: While creating the objects of the class, the static variable **count** is incremented by 1. The **totalEmployee()** method displays the value of the static variable **count**.

Example 2: Static Variable In Dart

In this example below, there is a class named **Student**. The class has a static variable **schoolName** to store the name of the school. If every student belongs to the same school, then it is better to use a static variable.

```
class Student {
  int id;
  String name;
  static String schoolName = "ABC School";
  Student(this.id, this.name);
  void display() {
    print("Id: ${this.id}");
    print("Name: ${this.name}");
    print("School Name: ${Student.schoolName}");
```

```
  }
}

void main() {
  Student s1 = new Student(1, "John");
  s1.display();
  Student s2 = new Student(2, "Smith");
  s2.display();
}
```

Show Output

```
Id: 1
Name: John
School Name: ABC School
Id: 2
Name: Smith
School Name: ABC School
```

Run Online

Dart Static Method

A static method is shared by all instances of a class. It is declared using the static keyword. You can access a static method without creating an object of the class.

Syntax

```
class ClassName{
static returnType methodName(){
  //statements
}
}
```

Example 3: Static Method In Dart

In this example, we will create a static method **calculateInterest()** which calculates the simple interest. You can call **SimpleInterest.calculateInterest()** anytime without creating an instance of the class.

```
class SimpleInterest {
  static double calculateInterest(double principal, double rate, double
time) {
    return (principal * rate * time) / 100;
  }
}

void main() {
  print(
      "The simple interest is ${SimpleInterest.calculateInterest(1000, 2,
2)}");
```

```
}
```

Show Output

```
The simple interest is 40.0
```

Run Online

Example 4: Static Method In Dart

In this example below, there is static method **generateRandomPassword()** which generates a random password. You can call **PasswordGenerator.generateRandomPassword()** anytime without creating an instance of the class.

```
import 'dart:math';

class PasswordGenerator {
  static String generateRandomPassword() {
    List<String> allalphabets = 'abcdefghijklmnopqrstuvwxyz'.split('');
    List<int> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    List<String> specialCharacters = ["@", "#", "%", "&", "*"];
    List<String> password = [];
    for (int i = 0; i < 5; i++) {
      password.add(allalphabets[Random().nextInt(allalphabets.length)]);
      password.add(numbers[Random().nextInt(numbers.length)].toString());
      password

.add(specialCharacters[Random().nextInt(specialCharacters.length)]);
    }
    return password.join();
  }
}

void main() {
  print(PasswordGenerator.generateRandomPassword());
}
```

Show Output

```
v5*p4*o2&c7%k1@
```

Run Online

Info

Note: You don't need to create an instance of a class to call a static method.

Key Points To Remember

- Static members are accessed using the class name.
- All instances of a class share static members.

Enum In Dart

An enum is a special type that represents a fixed number of constant values. An enum is declared using the keyword **enum** followed by the enum's name.

Syntax Of Enum In Dart

```
enum enumName {
  constantName1,
  constantName2,
  constantName3,
  ...
  constantNameN
}
```

Example 1: Enum In Dart

In this example below, there is enum type named **days**. It contains seven constants days. The **days** enum type is used in the **main()** function.

```
enum days {
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thrusday,
  Friday,
  Saturday
}

void main() {
  var today = days.Friday;
  switch (today) {
    case days.Sunday:
      print("Today is Sunday.");
      break;
    case days.Monday:
      print("Today is Monday.");
      break;
    case days.Tuesday:
      print("Today is Tuesday.");
      break;
    case days.Wednesday:
      print("Today is Wednesday.");
      break;
    case days.Thursday:
      print("Today is Thursday.");
      break;
    case days.Friday:
```

```
      print("Today is Friday.");
      break;
    case days.Saturday:
      print("Today is Saturday.");
      break;
  }
}
```

Show Output

```
Today is Friday.
```

Example 2: Enum In Dart

In this example, there is an enum type named **Gender**. It contains three constants **Male**, **Female**, and **Other**. The **Gender** enum type is used in the **Person** class.

```
enum Gender { Male, Female, Other }

class Person {
  // Properties
  String? firstName;
  String? lastName;
  Gender? gender;

  // Constructor
  Person(this.firstName, this.lastName, this.gender);

  // display() method
  void display() {
    print("First Name: $firstName");
    print("Last Name: $lastName");
    print("Gender: $gender");
  }
}

void main() {
  Person p1 = Person("John", "Doe", Gender.Male);
  p1.display();

  Person p2 = Person("Menuka", "Sharma", Gender.Female);
  p2.display();
}
```

Show Output

```
First Name: John
Last Name: Doe
Gender: Gender.Male
First Name: Menuka
```

```
Last Name: Sharma
Gender: Gender.Female
```

How to Print All Enum Values

In this example, there is enum type named **Days**. It contain 7 days. The for loop iterates through all the enum values.

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

void main() {
 // Days.values: It returns all the values of the enum.
  for (Days day in Days.values) {
    print(day);
  }
}
```

Show Output

```
Days.Sunday
Days.Monday
Days.Tuesday
Days.Wednesday
Days.Thursday
Days.Friday
Days.Saturday
```

Advantages Of Enum In Dart

- It is used to define a set of named constants.
- Makes your code more readable and maintainable.
- It makes the code more reusable and makes it easier for developers.

Characteristics Of Enum

- It must contain at least one constant value.
- Enums are declared outside the class.
- Used to store a large number of constant values.

Enhanced Enum In Dart

In dart, you can declare enums with members. For example, for your accounting software you can store company types like **Sole Proprietorship**, **Partnership**, **Corporation**, and **Limited Liability Company**. You can declare an enum with members as shown below.

```
enum CompanyType {
  soleProprietorship("Sole Proprietorship"),
  partnership("Partnership"),
```

```
  corporation("Corporation"),
  limitedLiabilityCompany("Limited Liability Company");

  // Members
  final String text;
  const CompanyType(this.text);
}

void main() {
  CompanyType soleProprietorship = CompanyType.soleProprietorship;
  print(soleProprietorship.text);
}
```

Show Output

```
Sole Proprietorship
```

Abstract Class

Abstract classes are classes that cannot be initialized. It is used to define the behavior of a class that can be inherited by other classes. An abstract class is declared using the keyword **abstract**.

Syntax

```
abstract class ClassName {
  //Body of abstract class

  method1();
  method2();
}
```

Abstract Method

An abstract method is a method that is declared without an implementation. It is declared with a semicolon (;) instead of a method body.

Syntax

```
abstract class ClassName {
  //Body of abstract class
  method1();
  method2();
}
```

Why We Need Abstract Class

Subclasses of an abstract class must implement all the abstract methods of the abstract class. It is used to achieve abstraction in the Dart programming language.

Example 1: Abstract Class In Dart

In this example below, there is an abstract class **Vehicle** with two abstract methods **start()** and **stop()**. The subclasses **Car** and **Bike** implement the abstract methods and override them to print the message.

```
abstract class Vehicle {
  // Abstract method
  void start();
  // Abstract method
  void stop();
}

class Car extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Car started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Car stopped');
  }
}

class Bike extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Bike started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Bike stopped');
  }
}

void main() {
  Car car = Car();
  car.start();
  car.stop();

  Bike bike = Bike();
```

```
  bike.start();
  bike.stop();
}
```

Show Output

```
Car started
Car stopped
Bike started
Bike stopped
```

Run Online

Info

Note: The abstract class is used to define the behavior of a class that can be inherited by other classes. You can define an abstract method inside an abstract class.

Example 2: Abstract Class In Dart

In this example below, there is an abstract class **Shape** with one abstract method **area()** and two subclasses **Rectangle** and **Triangle**. The subclasses implement the **area()** method and override it to calculate the area of the rectangle and triangle, respectively.

```
abstract class Shape {
  int dim1, dim2;
  // Constructor
  Shape(this.dim1, this.dim2);
  // Abstract method
  void area();
}

class Rectangle extends Shape {
  // Constructor
  Rectangle(int dim1, int dim2) : super(dim1, dim2);

  // Implementation of area()
  @override
  void area() {
    print('The area of the rectangle is ${dim1 * dim2}');
  }
}

class Triangle extends Shape {
  // Constructor
  Triangle(int dim1, int dim2) : super(dim1, dim2);

  // Implementation of area()
  @override
```

```
  void area() {
    print('The area of the triangle is ${0.5 * dim1 * dim2}');
  }
}

void main() {
  Rectangle rectangle = Rectangle(10, 20);
  rectangle.area();

  Triangle triangle = Triangle(10, 20);
  triangle.area();
}
```

Show Output

```
The area of the rectangle is 200
The area of the triangle is 100.0
```

Run Online

Constructor In Abstract Class

You can't create an object of an abstract class. However, you can define a constructor in an abstract class. The constructor of an abstract class is called when an object of a subclass is created.

Example 3: Constructor In Abstract Class

In this example below, there is an abstract class **Bank** with a constructor which takes two parameters **name** and **rate**. There is an abstract method **interest()**. The subclasses **SBI** and **ICICI** implement the abstract method and override it to print the interest rate.

```
abstract class Bank {
  String name;
  double rate;

  // Constructor
  Bank(this.name, this.rate);

  // Abstract method
  void interest();

  //Non-Abstract method: It have an implementation
  void display() {
    print('Bank Name: $name');
  }
}

class SBI extends Bank {
```

```
  // Constructor
  SBI(String name, double rate) : super(name, rate);

  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of SBI is $rate');
  }
}

class ICICI extends Bank {
  // Constructor
  ICICI(String name, double rate) : super(name, rate);

  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of ICICI is $rate');
  }
}

void main() {
  SBI sbi = SBI('SBI', 8.4);
  ICICI icici = ICICI('ICICI', 7.3);

  sbi.interest();
  icici.interest();
  icici.display();
}
```

Show Output

```
The rate of interest of SBI is 8.4
The rate of interest of ICICI is 7.3
Bank Name: ICICI
```

Key Points To Remember

- You can't create an object of an abstract class.
- It can have both abstract and non-abstract methods.
- It is used to define the behavior of a class that other classes can inherit.
- Abstract method only has a signature and no implementation.

Interface In Dart

An interface defines a syntax that a class must follow. It is a contract that defines the capabilities of a class. It is used to achieve abstraction in the Dart programming language. When you implement an interface, you must implement all the properties and methods defined in the interface. Keyword **implements** is used to implement an interface.

Syntax Of Interface In Dart

```
class InterfaceName {
  // code
}

class ClassName implements InterfaceName {
  // code
}
```

Declaring Interface In Dart

In dart there is no keyword **interface** but you can use **class** or **abstract class** to declare an interface. All classes implicitly define an interface. Mostly **abstract class** is used to declare an interface.

```
// creating an interface using abstract class
abstract class Person {
  canWalk();
  canRun();
}
```

Implementing Interface In Dart

You must use the **implements** keyword to implement an interface. The class that implements an interface must implement all the methods and properties of the interface.

```
class Student implements Person {
 // implementation of canWalk()
  @override
  canWalk() {
    print('Student can walk');
  }

// implementation of canRun()
  @override
  canRun() {
    print('Student can run');
  }
}
```

Example 1: Interface In Dart

In this example below, there is an interface **Laptop** with two methods **turnOn()** and **turnOff()**. The class **MacBook** implements the interface and overrides the methods to print the message.

```
// creating an interface using concrete class
class Laptop {
    // method
```

```
  turnOn() {
    print('Laptop turned on');
  }
    // method
  turnOff() {
    print('Laptop turned off');
  }
}

class MacBook implements Laptop {
  // implementation of turnOn()
  @override
  turnOn() {
    print('MacBook turned on');
  }

  // implementation of turnOff()
  @override
  turnOff() {
    print('MacBook turned off');
  }
}

void main() {
  var macBook = MacBook();
  macBook.turnOn();
  macBook.turnOff();
}
```

Show Output

```
MacBook turned on
MacBook turned off
```

Run Online

Info

Note: Most of the time, **abstract class** is used instead of **concrete class** to declare an interface.

Example 2: Interface In Dart

In this example below, there is an abstract class named **Vehicle**. The **Vehicle** class has two abstract methods **start()** and **stop()**. The **Car** class implements the **Vehicle** interface. The **Car** class has to implement the **start()** and **stop()** methods.

```
// abstract class as interface
abstract class Vehicle {
  void start();
```

```
  void stop();
}
// implements interface
class Car implements Vehicle {
  @override
  void start() {
    print('Car started');
  }

  @override
  void stop() {
    print('Car stopped');
  }
}

void main() {
  var car = Car();
  car.start();
  car.stop();
}
```

Show Output

```
Car started
Car stopped
```

Run Online

Multiple Inheritance In Dart

Multiple inheritance means a class can inherit from more than one class. In dart, you can't inherit from more than one class. But you can implement multiple interfaces in a class.

Syntax For Implementing Multiple Interfaces In Dart

```
class ClassName implements Interface1, Interface2, Interface3 {
  // code
}
```

Example 3: Interface In Dart With Multiple Interfaces

In this example below, two abstract classes are named **Area** and **Perimeter**.
The **Area** class has an abstract method **area()** and the **Perimeter** class has an abstract method **perimeter()**. The **Shape** class implements both the **Area** and **Perimeter** classes. The **Shape** class has to implement the **area()** and **perimeter()** methods.

```
// abstract class as interface
abstract class Area {
  void area();
```

```
}
// abstract class as interface
abstract class Perimeter {
  void perimeter();
}
// implements multiple interfaces
class Rectangle implements Area, Perimeter {
    // properties
  int length, breadth;

 // constructor
  Rectangle(this.length, this.breadth);

// implementation of area()
  @override
  void area() {
    print('The area of the rectangle is ${length * breadth}');
  }
// implementation of perimeter()
  @override
  void perimeter() {
    print('The perimeter of the rectangle is ${2 * (length + breadth)}');
  }
}

void main() {
  Rectangle rectangle = Rectangle(10, 20);
  rectangle.area();
  rectangle.perimeter();
}
```

Show Output

```
The area of the rectangle is 200
The perimeter of the rectangle is 60
```

Run Online

Example 4: Interface In Dart

In this example below, there is an abstract class named **Person**. The **Person** class has one property **name** and two abstract methods **run** and **walk**. The **Student** class implements the **Person** interface. The **Student** class has to implement the **run** and **walk** methods.

```
// abstract class as interface
abstract class Person {
    // properties
  String? name;
  // abstract method
```

```
  void run();
  void walk();
}

class Student implements Person {
    // properties
  String? name;

 // implementation of run()
 @override
  void run() {
    print('Student is running');
  }
  // implementation of walk()
  @override
  void walk() {
    print('Student is walking');
  }
}

void main() {
  var student = Student();
  student.name = 'John';
  print(student.name);
  student.run();
  student.walk();
}
```

Show Output

```
John
Student is running
Student is walking
```

Run Online

Example 5: Interface In Dart

In this example below, there is abstract class named **CalculateTotal** and **CalculateAverage**. The **CalculateTotal** class has an abstract method **total()** and the **CalculateAverage** class has an abstract method **average()**. The **Student** class implements both the **CalculateTotal** and **CalculateAverage** classes. The **Student** class has to implement the **total()** and **average()** methods.

```
// abstract class as interface
abstract class CalculateTotal {
  int total();
}
// abstract class as interface
```

```dart
abstract class CalculateAverage {
  double average();
}
// implements multiple interfaces
class Student implements CalculateTotal, CalculateAverage {
// properties
  int marks1, marks2, marks3;
// constructor
  Student(this.marks1, this.marks2, this.marks3);
// implementation of average()
  @override
  double average() {
    return total() / 3;
  }
// implementation of total()
  @override
  int total() {
    return marks1 + marks2 + marks3;
  }
}

void main() {
  Student student = Student(90, 80, 70);
  print('Total marks: ${student.total()}');
  print('Average marks: ${student.average()}');
}
```

Show Output

```
Total marks: 240
Average marks: 80.0
```

Run Online

Difference Between Extends & Implements

extends	implements
Used to inherit a class in another class.	Used to inherit a class as an interface in an
Gives complete method definition to sub-class.	Gives abstract method definition to sub-cla
Only one class can be extended.	Multiple classes can be implemented.
It is optional to override the methods.	Concrete class must override the methods
Constructors of the superclass is called before the sub-class constructor.	Constructors of the superclass is not called constructor.
The super keyword is used to access the members of the superclass.	Interface members can't be accessed using

extends	implements
Sub-class need not to override the fields of the superclass.	Subclass must override the fields of the inte

Key Points To Remember

- An interface is a contract that defines the capabilities of a class.
- Dart has no keyword interface, but you can use class or abstract class to declare an interface.
- Use abstract class to declare an interface.
- A class can extend only one class but can implement multiple interfaces.
- Using the interface, you can achieve multiple inheritance in Dart.
- It is used to achieve abstraction.

Mixin In Dart

Mixins are a way of reusing the code in multiple classes. Mixins are declared using the keyword **mixin** followed by the mixin name. Three keywords are used while working with mixins: **mixin**, **with**, and **on**. It is possible to use multiple mixins in a class.

Info

Note: The **with** keyword is used to apply the mixin to the class. It promotes DRY(Don't Repeat Yourself) principle.

Rules For Mixin

- **Mixin** can't be instantiated. You can't create object of mixin.
- Use the **mixin** to share the code between multiple classes.
- **Mixin** has no constructor and cannot be extended.
- It is possible to use multiple **mixins** in a class.

Syntax

```
mixin Mixin1{
  // code
}

mixin Mixin2{
  // code
}

class ClassName with Mixin1, Mixin2{
  // code
}
```

Example 1: Mixin In Dart

In this example below, there are two mixins named **ElectricVariant** and **PetrolVariant**. The **ElectricVariant** mixin has a method **electricVariant()** and the **PetrolVariant** mixin

has a method **petrolVariant()**. The **Car** class uses both the **ElectricVariant** and **PetrolVariant** mixins.

```
mixin ElectricVariant {
  void electricVariant() {
    print('This is an electric variant');
  }
}

mixin PetrolVariant {
  void petrolVariant() {
    print('This is a petrol variant');
  }
}
// with is used to apply the mixin to the class
class Car with ElectricVariant, PetrolVariant {
  // here we have access of electricVariant() and petrolVariant() methods
}

void main() {
  var car = Car();
  car.electricVariant();
  car.petrolVariant();
}
```

Show Output

```
This is an electric variant
This is a petrol variant
```

Run Online

Example 2: Mixin In Dart

In this example below, there are two mixins named **CanFly** and **CanWalk**. The **CanFly** mixin has a method **fly()** and the **CanWalk** mixin has a method **walk()**. The **Bird** class uses both the **CanFly** and **CanWalk** mixins. The **Human** class uses the **CanWalk** mixin.

```
mixin CanFly {
  void fly() {
    print('I can fly');
  }
}

mixin CanWalk {
  void walk() {
    print('I can walk');
  }
}
```

```
class Bird with CanFly, CanWalk {

}

class Human with CanWalk {

}

void main() {
  var bird = Bird();
  bird.fly();
  bird.walk();

  var human = Human();
  human.walk();
}
```

Show Output

```
I can fly
I can walk
I can walk
```

Run Online

On Keyword

Sometimes, you want to use a mixin only with a specific class. In this case, you can use the **on** keyword.

Syntax Of On Keyword

```
mixin Mixin1 on Class1{
  // code
}
```

Example 3: On Keyword In Mixin In Dart

In this example below, there is abstract class named **Animal** with properties **name** and **speed**. The **Animal** class has an abstract method **run()**. The **CanRun** mixin is only used by class that extends **Animal**. The **Dog** class extends the **Animal** class and uses the **CanRun** mixin. The **Bird** class cannot use the **CanRun** mixin because it does not extend the **Animal** class.

```
abstract class Animal {
  // properties
  String name;
  double speed;

  // constructor
```

```
  Animal(this.name, this.speed);

  // abstract method
  void run();
}

// mixin CanRun is only used by class that extends Animal
mixin CanRun on Animal {
  // implementation of abstract method
  @override
  void run() => print('$name is Running at speed $speed');
}

class Dog extends Animal with CanRun {
  // constructor
  Dog(String name, double speed) : super(name, speed);
}

void main() {
  var dog = Dog('My Dog', 25);
  dog.run();
}

// Not Possible
// class Bird with Animal { }
```

Show Output

```
My Dog is Running at speed 25.0
```

Run Online

What Is Allowed For Mixin

- You can add properties and static variables.
- You can add regular, abstract, and static methods.
- You can use one or more mixins in a class.

What Is Not Allowed For Mixin

- You can't define a constructor.
- You can't extend a mixin.
- You can't create an object of mixin.

Generics In Dart

Generics is a way to create a class, or function that can work with different types of data **(objects)**. If you look at the internal implementation of **List** class, it is a generic class. It can work with different data types like int, String, double, etc. For

example, **List<int>** is a list of integers, **List<String>** is a list of strings, and **List<double>** is a list of double values.

Syntax

```
class ClassName<T> {
  // code
}
```

Example 1: Without Using Generics

Suppose, you need to create a class that can work with both **int** and **double** data types. You can create two classes, one for **int** and another for **double** like this:

```
// Without Generics
// Creating a class for int
class IntData {
  int data;
  IntData(this.data);
}
// Creating a class for double
class DoubleData {
  double data;
  DoubleData(this.data);
}

void main() {
  // Create an object of IntData class
  IntData intData = IntData(10);
  DoubleData doubleData = DoubleData(10.5);
  // Print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

Run Online

This is not a good practice because both class contain same code. You can create one **Generics** class that can work with different data types. See the example below.

Example 2: Using Generics

In this example below, there is single class that can work with **int**, **double**, and any other data types using **Generics**.

```
// Using Generics
```

```
class Data<T> {
  T data;
  Data(this.data);
}

void main() {
  // create an object of type int and double
  Data<int> intData = Data<int>(10);
  Data<double> doubleData = Data<double>(10.5);

  // print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

Run Online

Generics Type Variable

Generics type variables are used to define the type of data that can be used with the class. In the above example, **T** is a type variable. You can use any name for the type variable. A few typical names are **T**, **E**, **K**, and **V**.

Name	Work
T	Type
E	Element
K	Key
V	Value

Dart Map Class

Like **List**, internal implementation of **Map** work with different types of data like int, String, double, etc. This is because Map is a generic class.

```
// Dart implementation of Map class
abstract class Map<K, V> {
  // code
  external factory Map();
}
```

This simply means that the Map class can work with different types of data.

```
void main() {
  final info = {
    "name": "John",
    "age": 20,
    "height": 5.5,
  }
}
```

Generics Methods

You can also create a generic method. For this, you need to use the **<T>** keyword before the method's return type. See the example below.

```
// Define generic method
T genericMethod<T>(T value) {
  return value;
}

void main() {
  // call the generic method
  print("Int: ${genericMethod<int>(10)}");
  print("Double: ${genericMethod<double>(10.5)}");
  print("String: ${genericMethod<String>("Hello")}");
}
```

Show Output

```
Int: 10
Double: 10.5
String: Hello
```

Run Online

Example 3: Generic Method With Multiple Parameters

In this example below, you will learn to create a generic method with multiple parameters.

```
// Define generic method
T genericMethod<T, U>(T value1, U value2) {
  return value1;
}

void main() {
  // call the generic method
  print(genericMethod<int, String>(10, "Hello"));
  print(genericMethod<String, int>("Hello", 10));
}
```

Show Output

```
10
```

```
Hello
```

Run Online

Restricting the Type of Data

While implementing generics, you can restrict the type of data that can be used with the class or method. This is done by using the **extends** keyword. See the example below.

Example 4: Generic Class With Restriction

In this example below, there is a **Data** class that works only with **int** and **double** types. It will not work with other types..

```
// Define generic class with bounded type
class Data<T extends num> {
  T data;
  Data(this.data);
}

void main() {
  // create an object of type int and double
  Data<int> intData = Data<int>(10);
  Data<double> doubleData = Data<double>(10.5);
  // print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
  // Not Possible
  // Data<String> stringData = Data<String>("Hello");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

Run Online

Example 5: Generic Method With Restriction

In this example below, a generic method **getAverage** takes two parameters of Type **T**, which is considered a **num**. The method returns the average of the two parameters.

```
// Define generic method
double getAverage<T extends num>(T value1, T value2) {
  return (value1 + value2) / 2;
}

void main() {
  // call the generic method
  print("Average of int: ${getAverage<int>(10, 20)}");
```

```
  print("Average of double: ${getAverage<double>(10.5, 20.5)}");
}
```

Show Output

```
Average of int: 15
Average of double: 15.5
```

Run Online

Example 6: Generic Class In Dart

In this example below, there is an abstract class **Shape** with one abstract method called area which returns a double. Also there are two classes that implement Shape, **Circle** and **Rectangle**. There is class **Region** which takes a list of Shape objects and has a method called totalArea which returns the sum of the areas of all the shapes in the list.

```
// abstract class Shape
abstract class Shape {
  // abstract method area
  double get area;
}

// class Circle which implements Shape
class Circle implements Shape {
  // field radius
  final double radius;
  // constructor
  Circle(this.radius);

  // implementation of area method
  @override
  double get area => 3.14 * radius * radius;
}
// class Rectangle which implements Shape
class Rectangle implements Shape {
  // fields width and height
  final double width;
  final double height;
  // constructor
  Rectangle(this.width, this.height);

  // implementation of area method
  @override
  double get area => width * height;
}

// Generic class Region
class Region<T extends Shape> {
```

```
  // field shapes
  List<T> shapes;
  // constructor
  Region({required this.shapes});

  // method totalArea
  double get totalArea {
    double total = 0;
    shapes.forEach((shape) {
      total += shape.area;
    });
    return total;
  }
}

void main() {
  // create objects of Circle and Rectangle
  var circle = Circle(10);
  var rectangle = Rectangle(10, 20);
  // create a list of Shape objects
  var region = Region(shapes: [circle, rectangle]);
  // print the total area
  print("Total Area of Region: ${region.totalArea}");
}
```

Show Output

```
Total Area of Region: 514
```

Run Online

Advantages of Generics

- It solve the problem of type safety.
- It helps to reuse our code.