Abstract Classes in Dart

An ***Abstract class*** in Dart is defined as those classes which contain one or more than one abstract method (methods without implementation) in them. Whereas, to declare an abstract class we make use of the ***abstract*** keyword. So, it must be noted that a class declared abstract may or may not include abstract methods but if it includes an abstract method then it must be an abstract class.

Features of Abstract Class:

- A class containing an abstract method must be declared abstract whereas the class declared abstract may or may not have abstract methods i.e. it can have either abstract or concrete methods
- A class can be declared abstract by using **abstract** keyword only.
- A class declared as abstract can't be initialized.
- An abstract class can be extended, but if you inherit an abstract class then you have to make sure that all the abstract methods in it are provided with implementation.

Generally, abstract classes are used to implement the abstract methods in the extended subclasses.

Syntax:

```
abstract class class_name {


    // Body of the abstract class
}
```

Overriding abstract method of an abstract class.

Example:

```
// Understanding Abstract class in Dart

// Creating Abstract Class
abstract class Gfg {
    // Creating Abstract Methods
    void say();
    void write();
}

class Geeksforgeeks extends Gfg{
    @override
    void say()
    {
        print("Yo Geek!!");
    }

    @override
    void write()
    {
        print("Geeks For Geeks");
    }
}

main()
{
    Geeksforgeeks geek = new Geeksforgeeks();
    geek.say();
    geek.write();
}
```

Output:

```
Yo Geek!!

Geeks For Geeks
```

Explanation:

First, we declare an abstract class Gfg and create an abstract method geek_info inside it. After that, we extend the Gfg class to the second class and override the methods say() and write(), which result in their respective output.

***Note:** It is not mandatory to override the method when there is only one class extending the abstract class, because override is used to change the pre-defined code and as in the above case, nothing is defined inside the method so the above code will work just fine without override.*
Overriding abstract method of an abstract class in two different classes

Example

```
// Understanding Abstract Class In Dart
// Creating Abstract Class
abstract class Gfg {
    // Creating Abstract Method
    void geek_info();
}

// Class Geek1 Inheriting Gfg class
class Geek1 extends Gfg {
    // Overriding method
    @override
    void geek_info()
    {
        print("This is Class Geek1.");
    }
}

// Class Geek2 Inheriting Gfg class
class Geek2 extends Gfg {
    // Overriding method again
    @override
    void geek_info()
    {
        print("This is Class Geek2.");
    }
}

void main()
{
    Geek1 g1 = new Geek1();
    g1.geek_info();
    Geek2 g2 = new Geek2();
    g2.geek_info();
}
```

Output

```
This is Class Geek1.

This is Class Geek2.
```

Dart – Static Keyword

The **static** keyword is used for memory management of global data members. The static keyword can be applied to the fields and methods of a class. The static variables and methods are part of the class instead of a specific instance.

- The static keyword is used for a class-level variable and method that is the same for every instance of a class, this means if a data member is static, it can be accessed without creating an object.
- The static keyword allows data members to persist Values between different instances of a class.
- There is no need to create a class object to access a static variable or call a static method: simply put the class name before the static variable or method name to use them.

Dart Static Variables

The static variables belong to the class instead of a specific instance. A static variable is common to all instances of a class: this means only a single copy of the static variable is shared among all the instances of a class. The memory allocation for static variables happens only once in the class area at the time of class loading.

Declaring Static Variables

Static variables can be declared using the static keyword followed by data type then the variable name

Syntax: `static [date_type] [variable_name];`

Accessing Static Variable

The static variable can be accessed directly from the class name itself rather than creating an instance of it.

Syntax: `Classname.staticVariable;`

Dart Static Methods

The static method belongs to a class instead of class instances. A static method is only allowed to access the static variables of class and can invoke only static methods of the class. Usually, utility methods are created as static methods when we want it to be used by other classes without the need of creating an instance.

Declaring Static Methods

A static method can be declared using static keyword followed by return type, followed by method name

```
Syntax:

static return_type method_name()
{
    // Statement(s)
}
```

Calling Static Method

Static methods can be invoked directly from the class name itself rather than creating an instance of it.

```
Syntax: ClassName.staticMethod();
```

Example 1:

```
// Dart Program to show
// Static methods in Dart
class Employee {
  static var emp_dept;
  var emp_name;
  int emp_salary;

  // Function to show details
  // of the Employee
  showDetails() {
    print("Name of the Employee is: ${emp_name}");
    print("Salary of the Employee is: ${emp_salary}");
    print("Dept. of the Employee is: ${emp_dept}");
  }
}

// Main function
void main() {
  Employee e1 = new Employee();
  Employee e2 = new Employee();
  Employee.emp_dept = "MIS";

  print("GeeksforGeeks Dart static Keyword Example");
  e1.emp_name = 'Rahul';
  e1.emp_salary = 50000;
  e1.showDetails();

  e2.emp_name = 'Tina';
  e2.emp_salary = 55000;
  e2.showDetails();
}
```

Output:

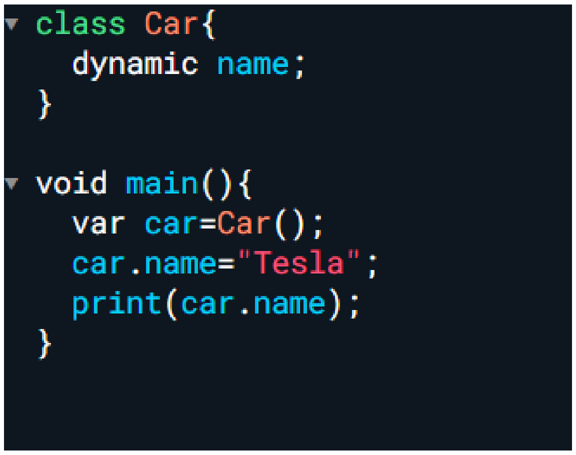
```
GeeksforGeeks Dart static Keyword Example

Name of the Employee is: Rahul

Salary of the Employee is: 50000

Dept. of the Employee is: MIS

Name of the Employee is: Tina

Salary of the Employee is: 5

Dept. of the Employee is: MIS
```

Encapsulation in Dart

Encapsulation is a mechanism for hiding important and sensitive data from users. To use encapsulation, you make the field private and use the public getters and setters to access and set the value of that field. In Dart, encapsulation is done at the library level, not at the class level. To provide default getters and setters, you can get and set values directly using field names. We can say encapsulation is building code into a single unit where you can determine the scope of each piece of data.

Example: A college can have different departments like Student Service, Account Department, and CS Department all these departments makes a college.

Default Encapsulation



```
class Car{
  dynamic name;
}

void main(){
  var car=Car();
  car.name="Tesla";
  print(car.name);
}
```

Output:

```
Tesla
```

In the above example, we have created a class **Car** and the object is stored in the **variable Car**. Then you need to access the fields from within that class using the default getter and setter methods. The **car. name** is used to set the value “Tesla” and then **car. name** is used to get the value.

Private Fields:

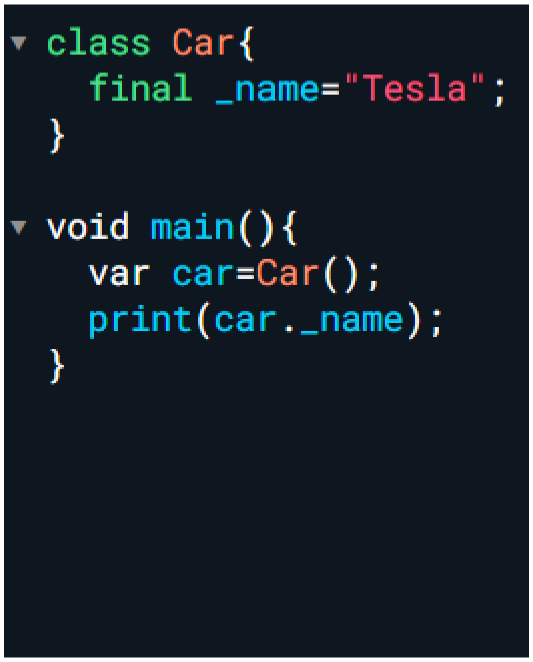
```
class Car{
  dynamic _name;
}

void main(){
  var car=Car();
  car._name="Tesla";
  print(car._name);
}
```

Output:

```
Tesla
```

In dart, you can use the private keyword to make a variable private using an underscore (_). I have used (_) before the **name** which makes the name variable private.

Read-only Fields



```
class Car{
  final _name="Tesla";
}

void main(){
  var car=Car();
  print(car._name);
}
```

Dart allows you to declare read-only fields by using the final keyword in front of the field. Read-only fields let you control access to fields. I can only access the value, not set a new value.

Polymorphism

Polymorphism in Dart refers to the ability of a class or data type to take on multiple forms or behave differently depending on the context. Dart is an object-oriented programming language, and polymorphism is one of the fundamental principles of object-oriented programming.

There are two main types of polymorphism in Dart:

1. **Compile-time Polymorphism (Static Polymorphism):**

- Operator Overloading: Dart allows you to redefine the behavior of operators for custom classes. By implementing specific methods like operator +, operator -, etc., you can define how the operator should behave for objects of that class.

```
class Vector {
          int x, y;

          Vector(this.x, this.y);

          Vector operator +(Vector other) {
            return Vector(x + other.x, y + other.y);
          }
        }

        void main() {
          Vector v1 = Vector(2, 3);
          Vector v2 = Vector(5, 7);

          Vector result = v1 + v2;

          print("Resultant Vector: (${result.x}, ${result.y})");  //
Output: Resultant Vector: (7, 10)
```

In this example, we have defined the operator for the Vector class. The operator + method allows us to add two Vector objects using the + operator, providing a custom implementation for vector addition.

2.Run-time Polymorphism (Dynamic Polymorphism):

Method Overriding: Inheritance in Dart enables a subclass to provide its own implementation of a method that is already defined in the superclass. When a method is called on an object of the subclass, the overridden method in the subclass is executed instead of the one in the superclass. Example

```


class Animal {
          void makeSound() {
            print("Animal makes a generic sound");
          }
        }

        class Dog extends Animal {
          @override
          void makeSound() {
            print("Dog barks");
          }
        }

        class Cat extends Animal {
          @override
          void makeSound() {
            print("Cat meows");
          }
        }

        void main() {
          Animal animal1 = Dog();
          Animal animal2 = Cat();

          animal1.makeSound(); // Output: "Dog barks"
          animal2.makeSound(); // Output: "Cat meows"
        }
```

Output

```
Dog barks
Cat meows
```

In this example, we have a base class Animal with a method makeSound(). Both Dog and Cat are subclasses of Animal and override the makeSound() method. During runtime, the method of the specific object type is called, demonstrating run-time polymorphism.

Polymorphism allows developers to write more flexible and reusable code. By using polymorphism, you can write code that works with a variety of different objects without needing to know their specific types, which enhances code modularity and maintainability.

Here's a simple example of polymorphism in Dart:

```dart
class Shape {
  void draw() {
    print("Drawing a shape");
  }
}

class Circle extends Shape {
  @override
  void draw() {
    print("Drawing a circle");
  }
}

class Square extends Shape {
  @override
  void draw() {
    print("Drawing a square");
  }
}

void main() {
  Shape shape1 = Circle();
  Shape shape2 = Square();

  shape1.draw(); // Output: "Drawing a circle"
  shape2.draw(); // Output: "Drawing a square"
}
```

Output

```
Drawing a circle
Drawing a square
```

In this example, the Shape class is the base class, and Circle and Square are subclasses that override the draw() method. During runtime, the correct draw() method is invoked based on the type of object, demonstrating the concept of run-time polymorphism.