Dart Abstract Classes

Abstract classes are the classes in Dart that has one or more abstract method. Abstraction is a part of the data encapsulation where the actual internal working of the function hides from the users. They interact only with external functionality. We can declare the abstract class by using the abstract keyword. There is a possibility that an abstract class may or may not have abstract methods.

Rules for Abstract classes:

1.An abstract class can have an abstract method (method without implementation), or not.

2. If there is at least one abstract method, then the class must be declared abstract.

3.The object of the abstract class cannot be created, but it can be extended.

4.An abstract keyword is used to declare the abstract class.

5.An abstract class can also include normal or concrete (method with the body) methods.

6.All abstract methods of parent class must be implemented in the subclass.

Declaring Abstract Class

An abstract keyword followed by a class name is used to declare the abstract class. An abstract class mostly used to offer a base for the subclass to extends and implement the abstract method.

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

we have a class **Person** that has method **displayInfo()**, and we have two sub classes of it **Boy** and **Girl**. Each of the person information varies from the other person, so there is no benefit to implementing the **displayInfo()** in the parent class. Because every subclass must override the parent class method by provides its own implementation. Thus, we can force the subclass to provide implementation to that method, so that is the benefit to make method abstract. We don't require the give implementation in the parent class.

```dart
abstract class Person {

  void displayInfo(); //abstract method
}

class Boy extends Person {
  void displayInfo() {
    print("My name is Mike");
  }
}
class Girl extends Person {
// Overriding method
  void displayInfo() {
    print("My name is Eve");
  }
}
void main() {
  Boy b = new Boy(); // Creating Object of Boy class
  Girl g = new Girl(); // Creating Object of Girl class

  b.displayInfo();
  g.displayInfo();
}
```

Dart Interfaces

An interface defines the syntax that any entity must adhere to. Dart does not have any separate syntax to define interfaces. An Interface defines the same as the class where any set of methods can be accessed by an object. The Class declaration can interface itself.

The keyword **implement** is needed to be writing, followed by class name to be able to use the interface. Implementing class must provide a complete definition of all the functions of the implemented interface. We can say that a class must define every function with the body in the interface that we want to achieve.

Declaring an Interface

Dart doesn't provide syntax for declaring interface directly. Implicitly, a class declaration itself an interface containing the entire instance member of the class and of any interfaces it implements.

Implementing an Interface

To work with interface methods, the interface must be implemented by another class using the **implements** keyword. A class which is implemented the interface must provide a full implementation of all the methods that belongs to the interface. Following is the syntax of the implementing interface.

class ClassName **implements** InterfaceName

```
class Employee {
  void display() {
    print("I am working as an engineer");
  }
}
class Engineer implements Employee {
  void display() {
    print("I am an engineer in this company");
  }
}
void main() {
  Engineer eng = new Engineer();
  eng.display();
}
```

Implementing Multiple Inheritance

We have discussed previously that the multiple inheritance is not supported by the Dart, but we can apply the multiple interfaces. We can say that, using multiple interfaces, we can achieve multiple inheritance in Dart. The syntax is given below.

Syntax:

```
class ClassName implements interface1, interface2,…interface n
```

```dart
class Student {
  String? name;
  int? age;
  void displayName() {
    print("I am ${name}");   }
  void displayAge() {
    print("My age is ${age}");    } }
class Faculty {
  String dep_name = '';
  int salary = 0;
  void displayDepartment() {
  print("I am a professor of ${dep_name}"); }
  void displaySalary() {
    print("My salary is ${salary}");   } }

void main() {
  College cg = new College();
  cg.name = "Ahmed";
  cg.age = 25;
  cg.dep_name = "Data Structure";
  cg.salary = 50000;
  cg.displayName();
  cg.displayAge();
  cg.displayDepartment();
  cg.displaySalary();
}
```

```dart
class College implements Student, Faculty {
  // Overriding Student class members
  String? name;
  int? age;
  void displayName() {
    print("I am ${name}");   }
  void displayAge() {
    print("My age is ${age}");   }
//Overriding each data member of Faculty class
  late String dep_name;
  late int salary;
  void displayDepartment() {
    print("I am a professor of ${dep_name}"); }
  void displaySalary() {
    print("My salary is ${salary}");  }
}
```

Rules for Implementing Interfaces

1. A class that implements the interface must override every method and instance variable of an interface.

2. Dart doesn't provide syntax to declare the interface directly. The class declaration can consider as the interface itself.

3. An interface class must provide the full implementation of all the methods belong to the interfaces.

4. We can implement one or more interfaces simultaneously.

5. Using the interface, we can achieve multiple inheritance.

Dart mixins

A **mixin** is a class with methods and properties utilized by other classes in Dart.
It is a way to reuse code and write code clean.

To declare a mixin, we use the mixin keyword:

```
mixin Mixin_name{
 }
```

Mixins, in other words, are regular classes from which we can grab methods (or variables) without having to extend them. To accomplish this, we use the with keyword.

```dart
mixin Bark {
  void bark() => print('Barking');
}

mixin Fly {
  void fly() => print('Flying');
}
mixin Crawl {
  void crawl() => print('Crawling');
}
class Animal {
  void breathe() {
    print("Breathing");
  }
}
class Dog extends Animal with Bark {}
class Bat extends Animal with Fly {}
class Snake extends Animal with Crawl {
   void display() {
    print(".....Snake.....");
    breathe();
    crawl();
  }
}
```

```dart
main() {
  var dog = Dog();
  dog.breathe();
  dog.bark();

  var snake = Snake();
  snake.display();
}
```

```
Breathing
Barking
.....Snake.....
 Breathing
Crawling
```

Note: A mixin cannot be instantiated.

Access Modifiers in Dart

One of the key features of Dart is its access modifiers, which allow developers to control the visibility and accessibility of class members. Access modifiers provide a way to protect the internal state of an object and prevent unwanted modifications.

Types of Access Modifiers in Dart

Dart supports three types of access modifiers:

1.public

2.private

3.extension

1. Public Access Modifier

The public access modifier is the default access level in Dart. It is applied to all class members that do not have an explicit access modifier. A public class member can be accessed from any code within the same package or library.

```dart
class Person {
  String name = '';
  int age = 0;
  void greet() {
    print('Hello, my name is $name and I am $age years old.');
  }
}
void main() {
  var person = Person();
  person.name = 'Omer';
  person.age = 30;
  person.greet();
}
```

In this example, the name, age, and greet() members of the Person class are all public. They can be accessed from the main() function, which is also in the same package.

2. Private Access Modifier

The private access modifier is used to restrict access to class members within the same library. A private class member is denoted by prefixing its name with an underscore character (_).

```
class Person {
  String _name = '';
  int _age = 0;
  void _greet() {
  print('Hello, my name is $_name and I am $_age years old.');
  }
  void introduce() {
    _name = 'Omer';
    _age = 30;
    _greet();
  }
}
void main() {
  var person = Person();
  person.introduce();
}
```

In this example, the name, age, and greet() members of the Person class are all private. They can only be accessed from within the Person class. The introduce() method is public and can be accessed from the main() function, which is also in the same library.

3. Extension Access Modifier

The extension access modifier was introduced in Dart 2.6 and is used to add functionality to an existing class without having to modify the class itself. An extension can add new methods and properties to a class, even if the class is final or comes from a third-party library.

```dart
extension IntExtensions on int {
  bool isEven() => this % 2 == 0;
  bool isOdd() => this % 2 == 1;
}

void main() {
  var number = 3;
  print(number.isEven); // false
  print(number.isOdd); // true
}
```

In this example, we create an extension called IntExtensions on the int class. The extension adds two new methods isEven() and isOdd() that can be used on any int value.

Dart types of variables

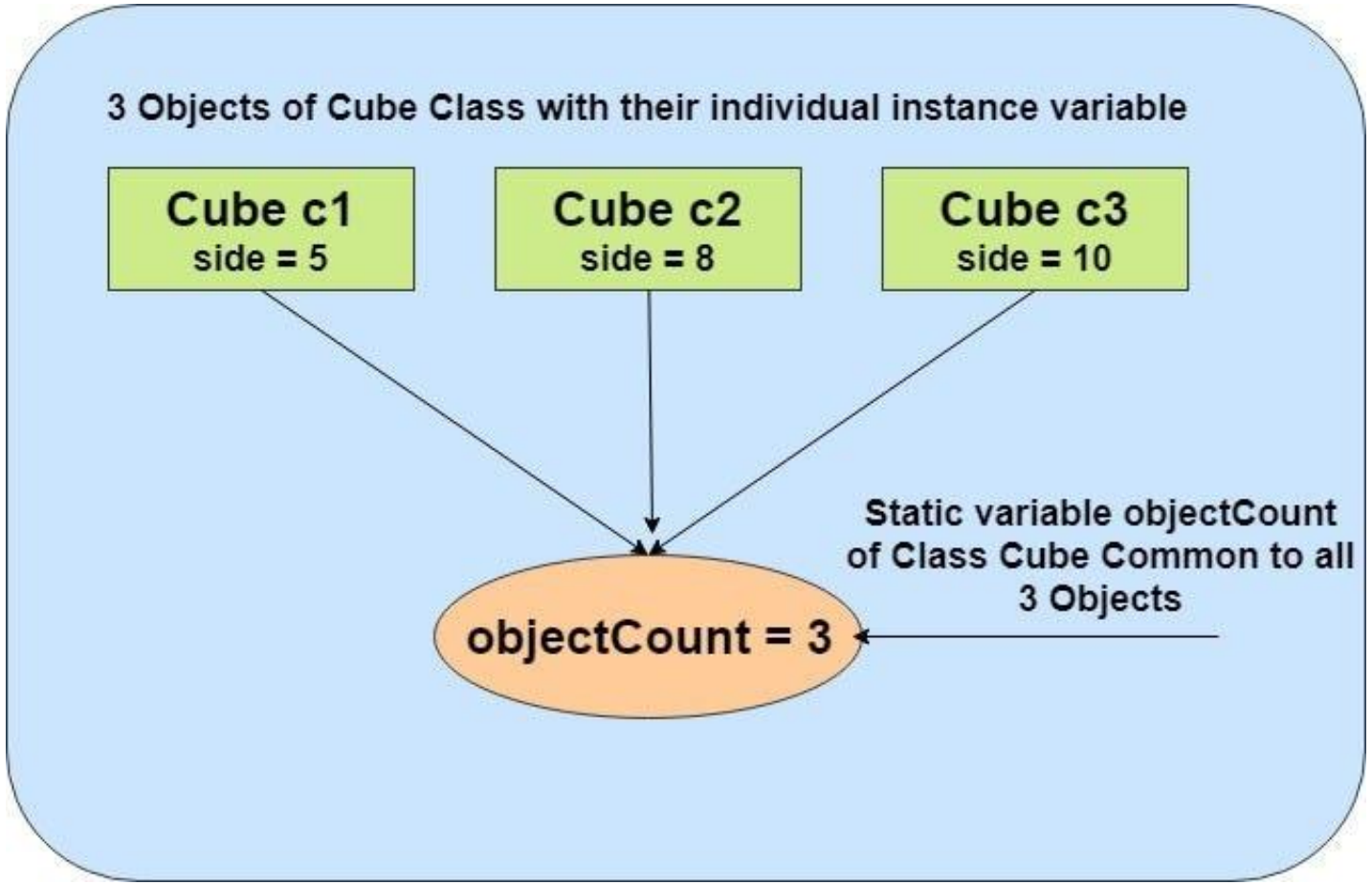
- **Top-level variables**
- **Static variables**
- **Instance variables**
- **Local variables**

Top-Level variables

Top-level variables are variables that are not linked to any class and that can be accessed from anywhere else in your program.

Static variables

Static variables are variables that are related to the class. (mostly called **class variables**). That is a member variable of a given class that is shared across all instances (objects).

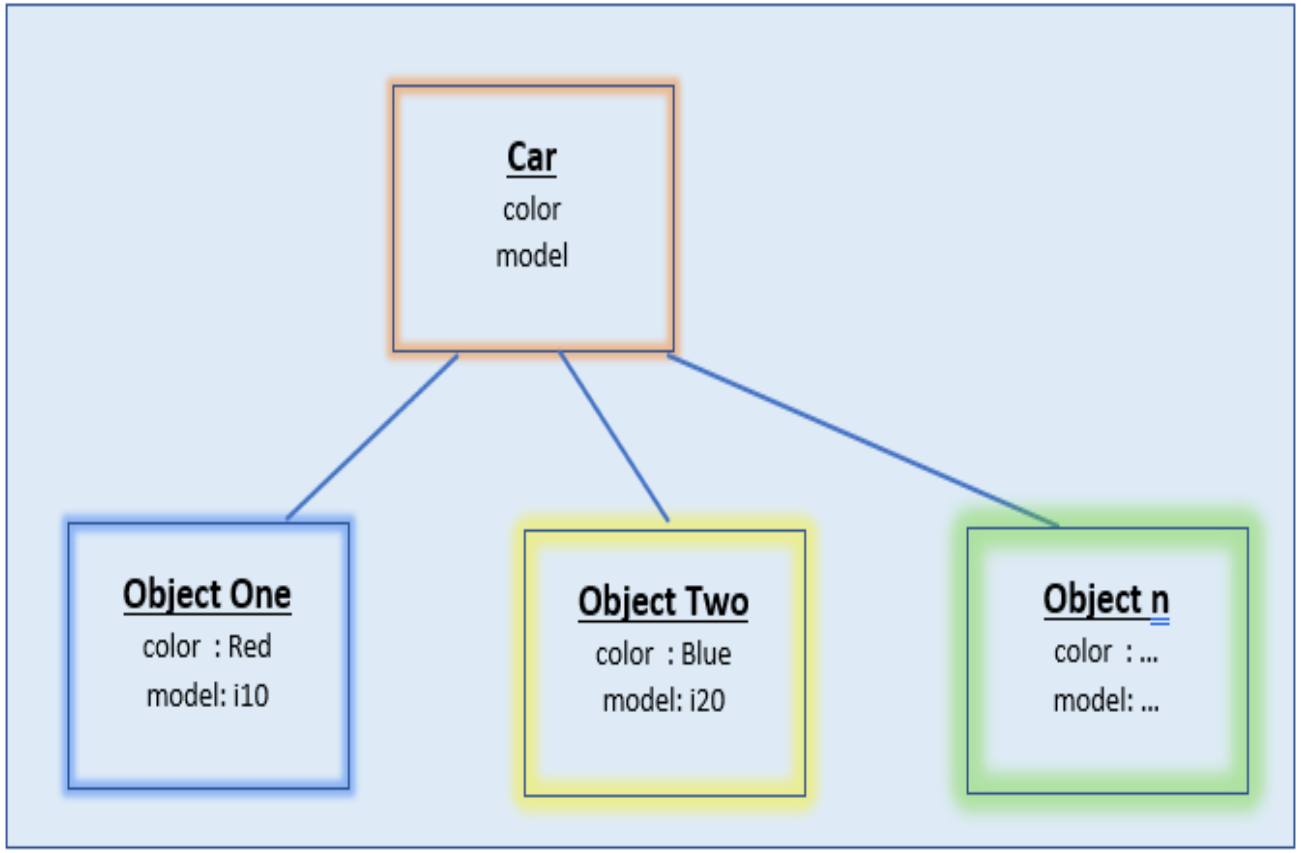



```
void main() {
  print(A.id);
}

class A {
  static int id =
0;
}
```

Instance variables

Instance variables are variables that are defined in the class, for which each instantiated object of that class has a separate copy or instance of the variables.




```dart
void main() {
  Car toyota = Car(color: 'Red',
model: 'RAV4');
  Car nissan = Car(color:
'Yellow', model: 'Altima');
  print(toyota.model); // RAV4
  print(nissan.model); // Altima
}

class Car {
  String? color;
  String? model;
  Car({this.color, this.model});
}
```

Local variables

Local variables are variables that only exist within a local scope meaning they can only exist only within the local context of a function or method.

```
void main(List<String> args) {
  print(calculate(4));
}

int calculate(int number) {
  int x = 5;
  return x + number;
}
```

In the above example, the variables number and x are local variables, which means they can only with the local scope of the function **calculate**. After the calculate function executes, the local variables will no longer exist

Late variables

Non-nullable variables must be initialized. This scenario can be walked around with the **late** keyword. This permits non-nullable variables to hold a null value until it is initialized. This may throw a runtime error if you fail to initialize before it is used.

```
late String name;

void main() {
  name= 'Hello Mr.Arc';
  print(name);
}
```

The above line of code is perfectly correct, the late keyword lets us initialize the non-nullable variable after but before it is been used, else it will throw an exception.

In dart, type annotations are options, that is;

int number= 5;

can be written as;

var number = 5 or dynamic number = 5;

This is because the type can be inferred at compile time using the **var** keyword or at runtime using the **dynamic** keyword.

var

With var, after the variable is set, can not be changed to a different type.

Once the variable age is set, can no longer hold any data type except with any value of the type it was initially set with.

var age = 34;

age =50;

age =80;

if you try changing the type of age, in the program, it will lead to an error. say we tried to change it to a value of type string (“Hello”)

Dynamic

With **dynamic**, after the variable is set, it can be changed to a different type. The value is only the exact value of the variable that is only known at runtime. Example;

the following is all correct using the **dynamic** keyword.

dynamic desc = 5;

desc =”Hello”;

desc = 6.7;

desc = [1,2,3,4,5,6];

desc = {4:”yes”, 7:”Arc”,9:”we can”}

All the above initialization and assignment are all accepted.

Final and const

These keywords are used to make the value of a variable fixed through out the code base. That means once the variable is set the state can not be altered or changed.

Final

The final keyword is used to make the value of a variable fixed and it can not be altered in the future.

Final variable will have known at runtime

Example:

final age = 29;

final String name = “MrArc”;

The program will through a runtime error if you try to change the value of the variable example;

age = 100; //error

Const

The const keyword work exactly like the final keyword. the only difference between final and const is that the variable compile time constants only, which means the value needs to be know before the code is converted to executable code.

Const variables only take constant values or literals.

Example:

const currentYear= 2022;

const int years = currentYear — 1996;

The above code is correct and will cause no error.

Note the value of the variable maybe known at compile time.