What is Dart:

Dart is an open-source programming language originally developed by Google. It is meant for both the server side as well as the user side. Dart is an Object-oriented language and is quite similar to that C, C++, and Java Programming.

main( ) function In Dart:

In dart main() function is a predefined method and acts as the entry point to the application. A dart script needs the main() method for the execution of the code.

⇒ **Simple Dart Program Example:**

```
main() {
        print("Welcome to SkillQode");
}
```

Variables and Data type in Dart:

A variable name is a name assigned to the memory location where the user stores the data and that data can be fetched when required with the help of the variable by calling its variable name. There are various data types of variables that are used to store the data. The type which will be used to store data depends upon the type of data to be stored.

⇒ **Syntax to declare a variable:**

```
datatype variableName;
```

⇒ **Syntax: To declare multiple variables of the same type:**

```
datatype variableNameOne, variableNameTwo, variableNameThree, ....variableName;
```

⇒ **Conditions to write variable names or identifiers are as follows:**

- Variable names or identifiers can't be the keyword.
- Variable names or identifiers can contain alphabets and numbers.
- Variable names or identifiers can't contain spaces and special characters, except the dollar($) sign.
- Variable names or identifiers can't begin with numbers.

⇒ **Variable and Data type Example:**

```
void main(){

        // Declaring and initialising a variable
        int varOne = 10;
        double varTwo = 0.2; // must declare double a value or it will throw an error
        bool varThree = false; // must declare boolean a value or it will throw an error
        String varFour = "0", varFive = "Hello Dart";

        print(varOne); // Print 10
        print(varTwo); // Print default double value
        print(varThree); // Print default string value
        print(varFour); // Print default bool value
        print(varFive); // Print Hello Dart
    }
```

Dynamic data type in Dart:

This is a special variable initialised with keyword dynamic. The variable declared with this data type can store implicitly any value during running the program. It is quite similar to the var datatype in Dart, but the difference between them is the moment you assign the data to a variable with the var keyword it is replaced with the appropriate data type.

⇒ **Syntax to declare a Dynamic data type:**

```
    dynamic variableName;
```

⇒ **Dynamic data type Example:**

```
    void main(){
        // Assigning value to name variable
        dynamic name = "Mohil thummar";

        // Printing variable name
        print(name);

        // Reassigning the data to the variable and printing it
        name = 3.14157;
        print(name);
    }
```

Dart Conditional Operators:

The operators are special symbols that are used to carry out certain operations on the operands. The Dart has numerous built-in operators which can be used to carry out different functions, for example, '+' is used to add two operands. Operators are meant to carry operations on one or two operands.

⇒ **Different types of operators in Dart:**

- Arithmetic Operators
- Relational Operators
- Type Test Operators
- Bitwise Operators
- Assignment Operators
- Logical Operators
- Conditional Operator
- Cascade Notation Operator

Dart Object-Oriented Concepts:

Dart is an object-oriented programming language, and it supports all the concepts of object-oriented programming such as classes, objects, inheritance, mixin, and abstract classes. As the name suggests, it focuses on the object and objects are real-life entities. The Object-oriented programming approach is used to implement concepts like polymorphism, data hiding, etc. The main goal of oops is to reduce programming complexity and do several tasks simultaneously. The oops concepts are given below.

- **Class**
- **Object**
- **Inheritance**
- **Polymorphism**
- **Interfaces**
- **Abstract class**

1. Class:

Dart classes are defined as the blueprint of the associated objects. A Class is a user-defined data type that describes its characteristics and behavior. To get all properties of the class, we must create an object of that class.

⇒ Syntax to declare a Class:

```
class ClassName {
   <fields>
   <getter/setter>
   <constructor>
   <functions>
}
```

The **class** keyword is followed by the class name. The rules for identifiers must be considered while naming a class.

⇒ A class definition can include the following:

- **Fields −** A field is any variable declared in a class. Fields represent data pertaining to objects.
- **Setters and Getters −** Allows the program to initialise and retrieve the values of the fields of a class. A default getter/ setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/ getter.
- **Constructors −** responsible for allocating memory for the objects of the class.
- **Functions −** Functions represent actions an object can take. They are also at times referred to as methods.

Class Example:

```
class Car {
  // field
  String engine = "E1001";

  // function
  void disp() {
    print(engine);
  }
}
```

2. Object:

We can access the class properties by creating an object of that class. In Dart.

An object is a real-life entity such as a table, human, car, etc. The object has two characteristics - state and behaviour. Let's take an example of a car that has a name, model name, price, and behaviour moving, stopping, etc. Object-oriented programming offers to identify the state and behaviour of the object.

⇒ **Object Syntax:**

```
ClassName objectName = ClassName([ arguments ]);
```

⇒ **Syntax Declaration:**

- new is the keyword used to declare the instance of the class
- object_name is the name of the object and its naming is similar to the variable name in dart.
- class_name is the name of the class whose instance variable has been created.
- arguments are the input that is needed to be passed if we are willing to call a constructor.
- After the object is created, there will be the need to access the fields which we will create. We use the dot(.) operator for that purpose.

Object Example:

```
Car obj = Car("Engine 1")
```

To access class property syntax:

```
 // For accessing the property
objectName.propertyName;

// For accessing the method
objectName.methodName();
```

Class And Object Example:

```
// Creating a Class named School
class School {

   // Creating Field inside the class
   String stdName;

   // Creating Function inside class
   void getStuName()
   {
      print("Student Name: $stdName");
   }
}

void main()
{
   // Creating Instance of class
   School schoolData = School();
```

```
    // Calling field name stdName and assigning value
    // to it using the object of the class School
    schoolData.stdName = 'mohil';

    // Calling function name school data using object of the class School
    schoolData.getStuName();
}
```

Standard Input in Dart:

In Dart programming language, you can take standard input from the user through the console via the use of the **.readLineSync()** function. To take input from the console you need to import a library, named dart:io from libraries of Dart.

⇒ About Stdin Class:
This class allows the user to read data from standard input in both synchronous and asynchronous ways. The method readLineSync() is one of the methods used to take input from the user.

Stander Input Example:

```
1). // importing dart:io file
import 'dart:io';

void main()
{
    print("Enter your name?");
    // Reading name of the user
    String? name = stdin.readLineSync();

    // Printing the name
    print("Hello, $name! \nWelcome to Skill Qode!!");
}
```

```
2).// Importing dart:io file
import 'dart:io';

void main()
{
    // Asking for favourite number
    print("Enter your favourite number:");

   // Scanning number
    int? n = int.parse(stdin.readLineSync()!);
   // Here ? and ! are for null safety

    // Printing that number
    print("Your favourite number is $n");
}
```

3).// Importing dart:io file
```
import 'dart:io';

void main()
{
    print("<<< ----------- Skill Qode ----------- >>>");
    print("Enter first number");
    int? n1 = int.parse(stdin.readLineSync()!);

    print("Enter second number");
    int? n2 = int.parse(stdin.readLineSync()!);

    // Adding them and printing them
    int sum = n1 + n2;
    print("Sum is $sum");
}
```

Dart – Null Safety:

Null Safety in simple words means a variable cannot contain a 'null' value unless you initialised with null to that variable. With null safety, all the runtime null-dereference errors will now be shown in compile time.

```
String name = null ; // This means the variable name has a null value.
```

Example 1:

```
class Car {
  String carName = "Aston Martin";
}

void main() {
  Car cars;
  print(cars.carName);
}
```

Output: **Error: Non-nullable variable 'cars' must be assigned before it can be used.**

Example 2:

```
class Car {
  String carName = "Aston Martin";
}

void main() {
  Car cars = Car();
  print(cars.carName);
}
```

Output: **Aston Martin**

In the above simple code it is easy to identify which variable is not initialised, but in a huge codebase it might be very difficult and time-consuming to identify such errors. This issue is solved by null safety.

⇒ Null Safety Principle:

- **Non-nullable By Default:**
  By default, Dart variables aren't nullable unless you explicitly specify that they can be null. This is because non-null was by far the most common option in API research.

- **Incrementally Adoptable:**
  Migration to null safety is completely up to you. You can choose what to migrate to null safety, and when. You can migrate incrementally, combining null-safe and non-null-safe code within the same project.

- **Fully Sound:**
  As a result of Dart's sound null safety, compiler optimizations are possible. If the type system determines that something isn't null, then it cannot be null. Null safety leads to fewer bugs, smaller binaries, and faster execution once your entire project and its dependencies are migrated to null safety.

⇒ Non-Nullable Types:

When we use null safety, all the types are non-nullable by default. For example, when we declare a variable of type int, the variable will contain some integer value.

```
void main() {
  int marks;

  // The value of type `null` cannot be
  // assigned to a variable of type 'int'
  marks = null;
}
```

Output: **Non-nullable variables must always be initialised with non-null values.**

⇒ Nullable Types:

To specify if the variable can be null, then you can use the nullable type " ? " operator.

Example:

```
String? carName;  // initialised to null by default
int? marks = 36;  // initialised to non-null
marks = null; // can be re-assigned to null
```

Note: You don't need to initialise a nullable variable before using it. It is initialised to null by default.

⇒ The Assertion Operator (!):

Use the null assertion operator (!) to make Dart treat a nullable expression as non-nullable if you're certain it isn't null.

Example:

```
int? someValue = 30;
int data = someValue!; // This is valid as value is non-nullable
```

In the above example, we are telling Dart that the variable “someValue” is null, and it is safe to assign it to a non-nullable variable i.e. data

⇒ Type Promotion :
Dart’s analyzer, which tells you what compile-time errors and warnings are, is intelligent enough to determine whether a nullable variable is guaranteed to have values that are not null. Dart uses Flow Analysis at runtime for type promotion (flow analysis is a mechanism that determines the control flow of a program).

Example:

```
int checkValue(int? someValue) {
  if (someValue == null) {
    return 0;
  }
  // At this point the value is not null.
  return someValue.abs();
}

void main(){
  print(checkValue(5));
  print(checkValue(null));
}
```

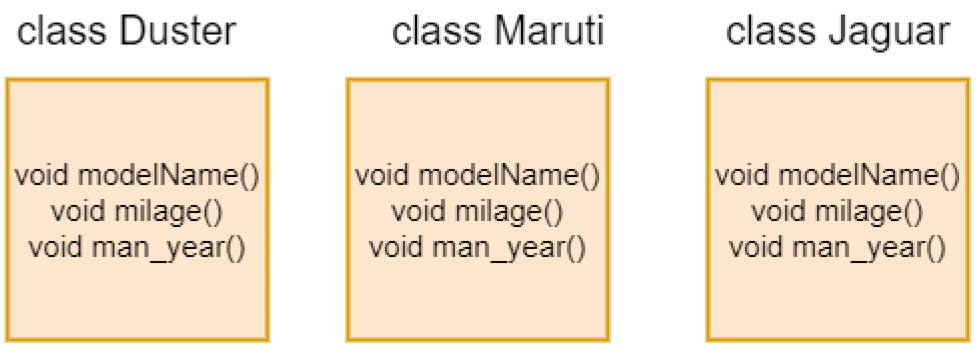
In the above code, the if statement checks if the value is null or not. After the if statement value cannot be null and is treated (promoted) as a non-nullable value. This allows us to safely use “someValue.abs()” instead of “someValue?.abs()” (with the null-aware operator). Here the ".abs()” function will return an absolute value.

3. Inheritance:

Dart inheritance is defined as the process of deriving the properties and characteristics of another class. It provides the ability to create a new class from an existing class. It is the most essential concept of the 00oops(Object-Oriented programming approach). We can reuse all the behaviour and characteristics of the previous class in the new class.

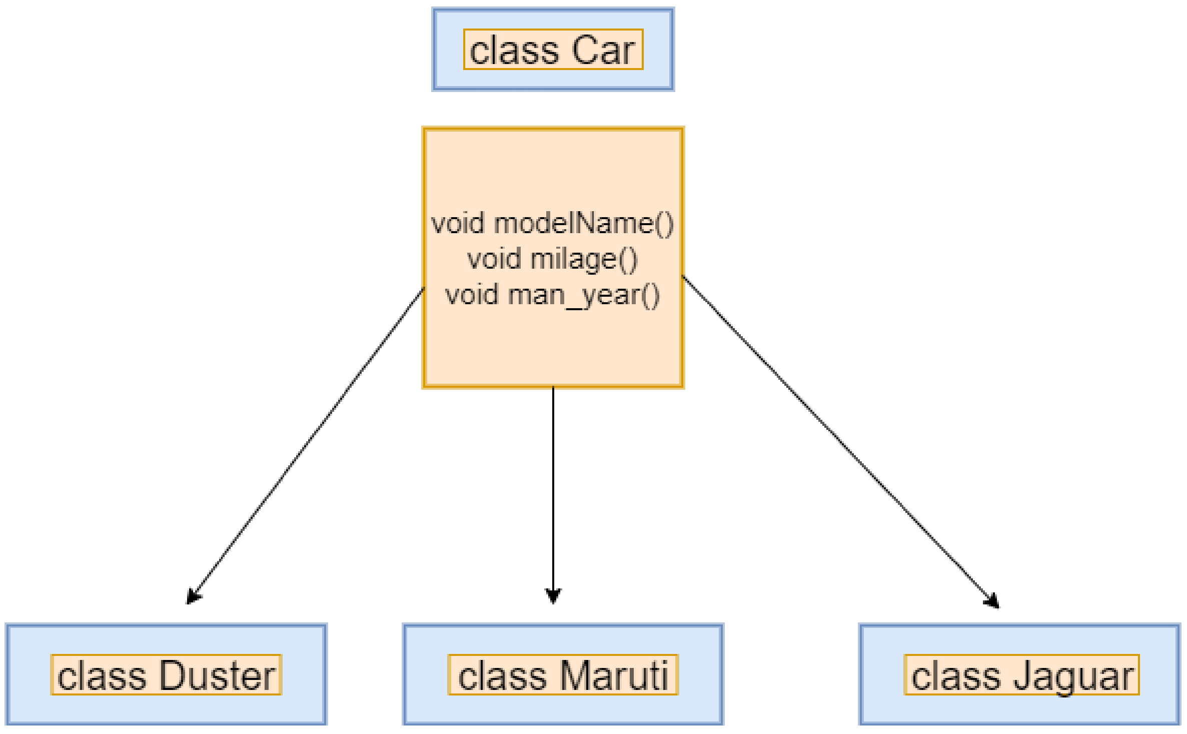
- **Parent Class -** A class which is inherited by the other class is called **superclass** or **parent class**. It is also known as a **base class**.
- **Child Class -** A class which inherits properties from other classes is called the child class. It is also known as the **derived class** or **subclass**.

Suppose we have a fleet of cars, and we create three classes: Duster, Maruti, and Jaguar. The methods modelName(), milage(), and man_year() will be the same for all of the three classes. By using inheritance, we don't need to write these functions in each of the three classes.



As you can see in the above figure, if we create class Car and write the common function in each of the classes. Then, it will increase duplication and data redundancy in the program. The inheritance is used to avoid this type of situation.

We can avoid data redundancy by defining the class Car with these functions in it and inheriting in the other classes from the Car class. It enhances the re-usability of code. We just need to write functions one time instead of multiple times. Let's have a look at the following image.



Syntax:

```
class childClass extends parentClass {
   //body of child class
}
```

The child class inherits functions and variables, or properties of the parent class using the extends keyword. It cannot inherit the parent class constructor; we will discuss this concept later.

⇒ Types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

1. Single Inheritance:

In the single inheritance, a class is inherited by a single class or subclass is inherited by one parent class. In the following example, we create a Person which inherits Human class.

Example -

```
class Bird{
    void fly(){
        print("The bird can fly");
      }
  }
 // Inherits the super class
class Parrot extends Bird{
      //child class function
      void speak(){
          print("The parrot can speak");
        }
}
void main() {
    // Creating object of the child class
    Parrot p = Parrot();
    p.speak();
    p.fly();
}
```

Output:

```
The parrot can speak
The bird can fly
```

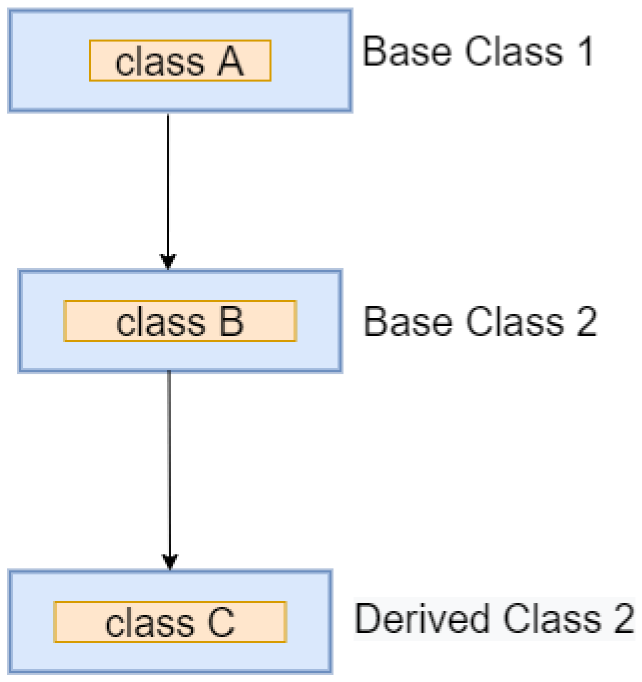
⇒ Explanation:

In the above code, we created the parent class Bird and declared the fly() function in it. Then, we created the child class called Parrot, which inherited the parent class's property using the extends keyword. The child class has its own function speak().

Now the child class has two functions fly() and speak(). So we created the object of the child class and accessed both functions. It printed the result to the console.

2. Multilevel Inheritance:

In the multiple inheritance, a subclass is inherited by another subclass or creates the chaining of inheritance.



Multilevel Inheritance

Example -

```
class Bird{
    void fly(){
        print("The bird can fly");
      }
  }
    // Inherits the super class
class Parrot extends Bird{
      void speak(){
                print("The parrot can speak");
           }

}

// Inherits the Parror base class
class Eagle extends Parrot {
       void vision(){
                print("The eagle has a sharp vision");
           }
}
void main() {
    // Creating object of the child class
    Eagle e = Eagle();
    e.speak();
    e.fly();
```

```
        e.vision();
    }
```

Output:

```
The parrot can speak
The bird can fly
The eagle has a sharp vision
```

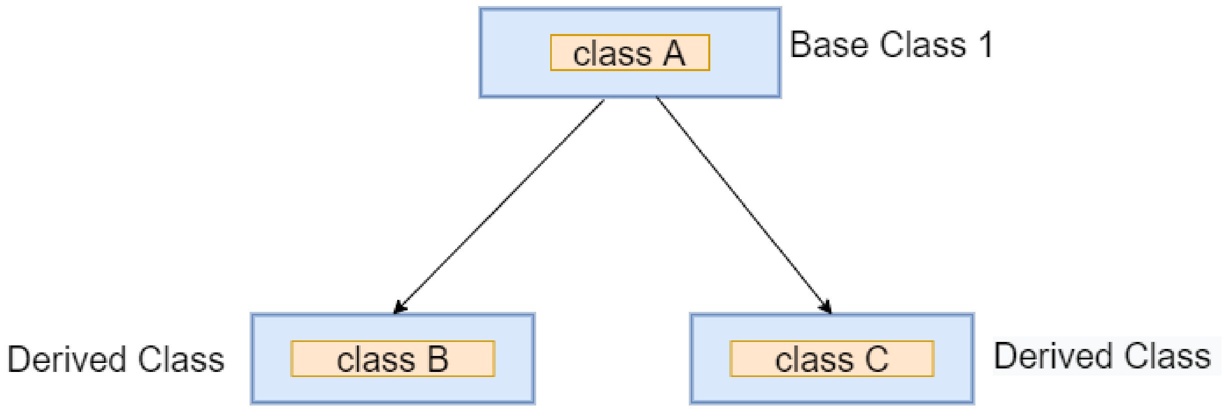
⇒ Explanation:

In the above example, we created another new class Eagle and inherited the Parrot class. Now the parrot is the parent class of Eagle, and class Eagle acquired all functions of both parent classes. We created the object of the child class and accessed all properties. It printed the output to the screen.

Note - Dart doesn't support multiple inheritance because it creates complexity in the program. Hierarchical Inheritance

3. Hierarchical Inheritance:

In hierarchical inheritance, two or more classes inherit a single class. In the following example, the two-child classes Peter and James inherit the Person class.



Hierarichal Inheritance

Example:

```
    // Parent Class
    class Person {
      void dispName(String name) {
        print(name);
      }

      void dispAge(int age) {
        print(age);
      }
    }

    class Peter extends Person {

      void dispBranch(String nationality) {
        print(nationality);
```

```
  }
}
//Derived class created from another derived class.
class James extends Person {
        void result(String result){
           print(result);
         }
}
void main() {
     // Creating Object of James class
     James j = James();
     j.dispName("James");
     j.dispAge(24);
     j.result("Passed");

    // Creating Object of Peter class
     Peter p = Peter();
     p.dispName("Peter");
     p.dispAge(21);
     p.dispBranch("Computer Science");

}
```

Output:

```
James
24
Passed
Peter
21
Computer Science
```

4. Polymorphism:

Polymorphism is a combination of the two Greek words poly, which means many and morph means morphing into different forms or shapes. Together, polymorphism means the same entity can be used in various forms. In the programming aspect, the same method can be used in different classes.

For example - We have a Shape class to define the shape of the object. The shape can be the circle, rectangle, square, straight line, etc. So here the goal is common, but the approach is different.

⇒ **Method Overriding:**

When we declare the same method in the subclass, which is previously defined in the superclass is known as the method overriding. The subclass can define the same method by providing its own implementation, which already exists in the superclass. The method in the superclass is called method overridden, and method in the subclass is called method overriding.

⇒ **Method Overriding Example:**

We define two classes; first, is a subclass called Human, and the second is a superclass Boy. The Boy subclass inherits the Human superclass. The same method void showInfo() in both classes is defined with a different implementation. The subclass has its own definition of the void showInfo(). Let's have a look at the following code snippet.

Example -

```
class Human{
  //Overridden method
   void run(){
     print("Human is running");
  }
}
class Man extends Human{
  //Overriding method
   void run(){
     print("Boy is running");
  }
}
void main(){
     Man m = Man();
     //This will call the child class version of run()
     m.run();
}
```

Output:

```
Boy is running
```

Explanation:

In the above example, we defined a method with the same name in both, subclass and superclass. The purpose of method overriding is to give the implementation of a subclass method. When we created the object of the Boy subclass, it executed the subclass method and printed the Man is running instead of Human is running.

If we create the object of a parent class, then it will always invoke the parent class method.

Example - 2

```
class College{
 // Declaring variables
        String name;
        int rollno;

// Overridden Method
void stuDetails(name,rollno){
       this.name = name;
       this.rollno = rollno;


}

void display(){
       print("The student name:${name}");
```

```
                print("The student rollno: ${rollno}");
                print("The result is passed");

            }

        }

        class Student extends College{
        // Overriding Method
        void stuDetails(name,rollno){
                this.name = name;
                this.rollno = rollno;

        }

        void show(){
                        print("The student name:${name}");
                        print("The student rollno: ${rollno}");
                        print("The result is failed");
                }
        }

        void main(){
        //Creating object of subclass
        Student  st = Student();
        st.stuDetails("Joseph",101);
        st.show();

        // Creating object of superclass
        College cg = College();
        cg.stuDetails("Jason",102);
        cg.display();
        }
```

Output:

```
The student name: Joseph
The student rollno: 101
The result is failed
The student name:Peter
The student rollno: 102
The result is passed
```

Explanation:
We create two classes - College as a parent class and Student as a child class. The method stu_details defined in both classes with the same parameters and same return types.

Now, the College superclass is inherited by the Student subclass and the stu_details() method is overridden in the subclass.

We created the object of Student and to invoke the stu_details() with suitable arguments. It executed the subclass method, and then it printed the result.

Same as we created the object of College superclass, invoked its methods and printed the different results.

⇒ Method Overriding using super Keyword:
We can invoke the parent class method without creating its object. It can be done by using the super keyword in the subclass. The parent class data member can be accessed in the subclass by using the super keyword. Let's understand the following example.

Example -
```
class Human{
  //Overridden method
   void run(){
     print("Human is running");
  }
}
class Man extends Human{
  //Overriding method
   void run(){
      // Accessing Parent class run() method in child class
      super.run();
     print("Boy is running");
  }
}
void main(){
     Man m = Man();
     //This will call the child class version of eat()
     m.run();
}
```

Output:
```
Human is running
Boy is running
```

Explanation:
In the above program, we accessed the Human class method in child class using the super keyword. Now, we don't need to instantiate the parent class. We only created the object of subclass, which invoked the run() method of child class and parent class.

Note - When we created the child class object and invoked the method, it executed the parent class (if accessed by super keyword) method first, then the child class method.

⇒ Advantage of method overriding:
The main benefit of the method overriding is that the subclass can provide its own implementation to the same method as per requirement without making any changes in the superclass method. This technique is used when we want to subclass method to behave differently also with the same name.

⇒ Rules of Method overriding in Dart:

The few rules of method overriding are given below. These points must be kept in mind while declaring the same method in subclass.

- The overriding method (the child class method) must be declared with the same configuration as the overridden method (the superclass method). The return type, list of arguments and its sequence must be the same as the parent class method.
- The overriding method must be defined in the subclass, not in the same class.
- The static and final method cannot be inherited in the subclass as they are accessible in their own class
- The constructor of the superclass cannot be inherited in a subclass.
- A method that cannot be inherited, then it cannot be overridden.

5. Interface:

The interface in the dart provides the user with the blueprint of the class, that any class should follow if it interfaces that class i.e. if a class inherits another it should redefine each function present inside an interface class in its way. They are nothing but a set of methods defined for an object. Dart doesn't have any direct way to create an inherited class, we have to make use of the "**implements**" keyword to do so.

Syntax:

```
class InterfaceClassName{
  ...
}

class ClassName implements InterfaceClassName {
  ...
}
```

Example 1:

```
void main(){
  // Creating Object
  // of the class ShowRoom
  ShowRoom showRoom= ShowRoom();

  // Calling method
  // (After Implementation )
  showRoom.printdata();
}

// Class Car (Interface)
class Car {
  void printdata() {
    print("Audi");
  }
}

// Class ShowRoom implementing Car
class ShowRoom implements Car {
  void printdata() {
    print("audi a4");
```

```
        }
    }
```

Output:

Welcome to audi a4

Note: Class should use the implements keyword, instead of extending to be able to use an interface method.

⇒ **Multiple Inheritance:**

Multiple inheritances are achieved by the use of implements. Although practically dart doesn't support multiple inheritance, it supports multiple interfaces.

Syntax:

```
class interfaceClassOne {
   ...
}
class interfaceClassTwo {
   ...
}
.
.
.
.
class interfaceClassThree {
   ...
}

class className implements InterfaceClassOne, InterfaceClassTwo, ...., InterfaceClassThree {
   ...
}
```

Example:

```
// Dart Program to show Multiple Inheritance
void main(){

  // Creating Object of
  // the class ShowRoom
  ShowRoom showRoom = ShowRoom();

  // Calling method (After Implementation )
  showRoom.printdata1();
  showRoom.printdata2();
  showRoom.printdata3();
}

// Class Audi (Interface1)
class Audi {
  void printdata1() {
    print("Hello Audi a4 !!");
```

```
     }
   }
    // Class Mercedes (Interface2)
   class Mercedes {
     void printdata2() {
       print("Hello Mercedes s class !!");
     }
   }
    // Class Volkswagen (Interface3)
   class Volkswagen {
     void printdata3() {
       print("Hello Volkswagen Polo !!");
     }
   }

   // Class ShowRoom implementing Audi, Mercedes, Volkswagen.
   class ShowRoom implements Audi, Mercedes, Volkswagen {
     void printdata1() {
       print("Howdy Audi,\nWelcome to ShowRoom");
     }

    void printdata2() {
       print("Howdy Mercedes,\nWelcome to ShowRoom");
     }

    void printdata3() {
       print("Howdy Volkswagen,\nWelcome to ShowRoom");
     }
   }
```

Output:

```
Howdy Audi,
Welcome to ShowRoom
Howdy Mercedes,
Welcome to ShowRoom
Howdy Volkswagen,
Welcome to ShowRoom
```

⇒ **Importance of Interface:**

- Used to achieve abstraction in Dart.
- It is a way to achieve multiple inheritances in Dart.

⇒ **Important Points:**

- If a class has been implemented then all of its method and instance variables must be overridden during the interface.
- In Dart, there are no direct means to declare an interface, so a declaration of a class is itself considered as a declaration on the interface.
- A class can extend only one class but can implement as many as you want.

6. Abstract Classes:

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 keywords only.
- A class declared as abstract can't be initialised.
- 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 className {
   // Body of the abstract class
}
```

Overriding abstract method of an abstract class.
Example:

```
// Understanding Abstract class in Dart

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

class Boy extends Man{
   @override
   void say(){
      print("Yo Boy!!");
   }

   @override
   void write(){
      print("Boy is writing!!");
   }
}

main(){
   Boy boy = Boy();
   boy.say();
   boy.write();
}
```

Output:

```
Yo Boy!!
Boy is writing!!
```

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 MainClass {
   // Creating Abstract Method
   void classInfo();
}

// Class ClassOne Inheriting MainClass class
class ClassOne extends MainClass {
   // Overriding method
   @override
   void classInfo(){
      print("This is Class1.");
   }
}

// Class ClassTwo Inheriting ClassOne
class ClassTwo extends MainClass {
   // Overriding method again
   @override
   void classInfo(){
      print("This is Class2.");
   }
}

void main(){
   ClassOne class1 = ClassOne();
   classTwo.classInfo();
   ClassTwo classTwo = ClassTwo();
   classTwo.classInfo();
}
```

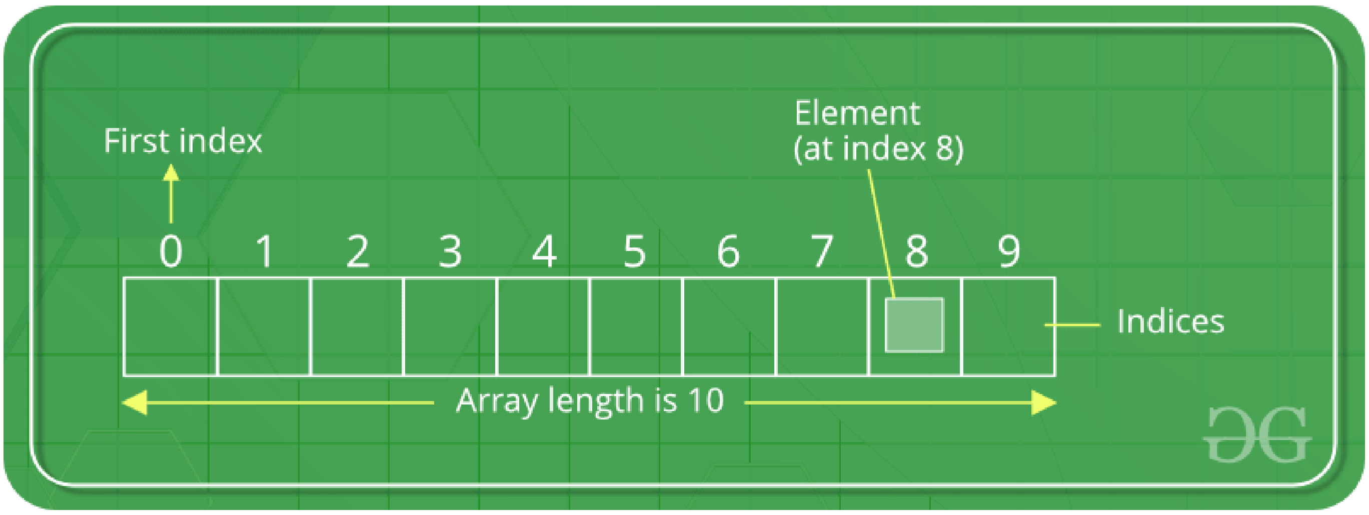
Output:

This is Class1.
This is Class2.

List:

List data type is similar to arrays in other programming languages. List is used to represent a collection of objects. It is an ordered group of objects. The core libraries in Dart are responsible for the existence of the List class, its creation, and manipulation.

Logical Representation of List



⇒ **Types of Lists:**
The Dart list can be categorised into two types -

- **Fixed Length List**
- **Growable List**

⇒ **Fixed Length List:**
The fixed-length lists are defined with the specified length. We cannot change the size at runtime. The syntax is given below.

Syntax - Create the list of fixed-size

```
List listName = List.filled(5,"12345");
```

The above syntax is used to create the list of the fixed size. We cannot add or delete an element at runtime. It will throw an exception if any try to modify its size.

The syntax of initialising the fixed-size list element is given below.

Syntax - Initialise the fixed size list element

```
listName[index] = value;
```

Let's understand the following example.

Example -

```
void main() {
  List list1 = List.filled(5,"12345");
  list1[0] = 10;
  list1[1] = 11;
  list1[2] = 12;
  list1[3] = 13;
  list1[4] = 14;
  print(list1);
}
```

Output:

```
[10, 11, 12, 13, 14]
```

Explanation -
In the above example, we have created a variable list1 that refers to the list of fixed size. The size of the list is five and we inserted the elements corresponding to its index position where 0th index holds 10, 1st index holds 12, and so on.

⇒ **Growable List:**
The list is declared without specifying size and is known as a Growable list. The size of the Growable list can be modified at the runtime. The syntax of the declaring Growable list is given below.

Syntax - Declaring a List

```
// creates a list with values
List listName = [val1, val2, val3];
Or
// creates a list of the size zero
List listName = [];
```

Syntax - Initialising a List

```
listName[index] = value;
```

Example - 1

```
void main() {
  List listOne = [10,11,12,13,14,15];
  print(listOne);
}
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

In the following example, we are creating a list using the empty list or List() constructor. The add() method is used to add elements dynamically in the given list.

Example - 2

```
void main() {
  var listOne = [];
  listOne.add(10);
  listOne.add(11);
  listOne.add(12);
```

```
    listOne.add(13);
    print(listOne);
  }
```

Output:
```
[10, 11, 12, 13]
```

⇒ List Properties:

Property	Description
first	It returns the first element case.
isEmpty	It returns true if the list is empty.
isNotEmpty	It returns true if the list has at least one element.
length	It returns the length of the list.
last	It returns the last element of the list.
reversed	It returns a list in reverse order.
Single	It checks if the list has only one element and returns it.

⇒ Inserting Element into List:
Dart provides four methods which are used to insert the elements into the lists. These methods are given below.

- add()
- addAll()
- insert()
- insertAll()

⇒ The add() Method:
This method is used to insert the specified value at the end of the list. It can add one element at a time and returns the modified list object. Let's understand the following example -

Syntax -
```
listName.add(element);
```

Example -
```
void main() {
   var oddList = [1,3,5,7,9];
   print(oddList);
   oddList.add(11);
   print(oddList);
}
```

Output:
```
[1, 3, 5, 7, 9]
```

```
[1, 3, 5, 7, 9, 11]
```

Explanation -
In the above example, we have a list named odd_list, which holds odd numbers. We inserted a new element 11 using add() function. The add() function appended the element at the end of the list and returned the modified list.

⇒ The addAll() Method:
This method is used to insert the multiple values to the given list. Each value is separated by the commas and enclosed with a square bracket ([]). The syntax is given below.

Syntax -

```
listName.addAll([valOne,valTwo,valThree,?..val]);
```

Example -

```
void main() {
    var oddList = [1,3,5,7,9]
     print(oddList);
      oddList.addAll([11,13,14]);
      print(oddList);
}
```

Output:

```
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9, 11, 13, 14]
```

Explaination -
In the above example, we don't need to call the add() function multiple times. The addAll() appended the multiple values at once and returned the modified list object.

⇒ The insert() Method:
The insert() method provides the facility to insert an element at specified index position. We can specify the index position for the value to be inserted in the list. The syntax is given below.

Syntax -

```
listName.insert(index,value);
```

Example -

```
void main(){
    List lst = [3,4,2,5];
    print(lst);
    lst.insert(2,10);
    print(lst);
}
```

Output:

```
[3, 4, 2, 5]
[3, 4, 10, 2, 5]
```

Explanation -

In the above example, we have a list of the random numbers. We called the insert() function and passed the index 2nd value 10 as an argument. It appended the value at the 2nd index and returned the modified list object.

⇒ The insertAll() Method:

The insertAll() function is used to insert the multiple value at the specified index position. It accepts index position and list of values as an argument. The syntax is given below.

Syntax -

```
listName.insertAll(index, iterable_list_of_value)
```

Example -

```
void main(){
    List lst = [3,4,2,5];
     print(lst);
     lst.insertAll(0,[6,7,10,9]);
     print(lst);
}
```

Output:

```
[3, 4, 2, 5]
[6, 7, 10, 9, 3, 4, 2, 5]
```

Explanation -

In the above example, we have appended the list of values at the 0th index position using the insertAll() function. It returned the modified list object.

⇒ Updating List:

The Dart provides the facility to update the list and we can modify the list by simply accessing its element and assign it a new value. The syntax is given below.

Syntax -

```
listName[index] = new value;
```

Example -

```
void main(){
      var listOne = [10,15,20,25,30];
      print("List before updation: ${listOne}");
      listOne[3] = 55;
      print("List after updation:${listOne}");
}
```

Output:

```
List before updation: [10, 15, 20, 25, 30]
List after updation: [10, 15, 20, 55, 30]
```

Explanation -

In the above example, we have accessed the 3rd index and assigned the new value 55 then printed the result. The previous list updated with the new value 55.

replaceRange() - The Dart provides **replaceRange()** function which is used to update within the given range of list items. It updates the value of the elements with the specified range. The syntax is given below.

Syntax -

```
listName.replaceRange(int start_val, int end_val, iterable);
```

Example -

```
void main(){
    List listOne = [10,15,20,25,30];
    print("List before updation: ${listOne}");
    listOne.replaceRange(0,4,[1,2,3,4]) ;
    print("List after updation using replaceAll() function : ${listOne}");
}
```

Output:

```
List before updation: [10, 15, 20, 25, 30]
List after updation using replaceAll() function : [1, 2, 3, 4, 30]
```

Explanation -

In the above example, we called the replaceRange() to the list which accepts the three arguments. We passed the starting index 0th, end index 4 and the list of the elements to be replaced as a third arguments. It returned the new list with the replaced element from the given range.

⇒ **Removing List Elements:**

The Dart provides following functions to remove the list elements.

- **remove()**
- **removeAt()**
- **removeLast()**
- **removeRange()**

⇒ **The remove() Method:**

It removes one element at a time from the given list. It accepts element as an argument. It removes the first occurrence of the specified element in the list if there are multiple same elements. The syntax is given below.

Syntax -

```
listName.remove(value)
```

Example -

```
void main(){
    var list1 = [10,15,20,25,30];
    print("List before remove element : ${list1}");
    list1.remove(20) ;
    print("List after removing element : ${list1}");
}
```

Output:

```
List before remove element : [10, 15, 20, 25, 30]
```

List after removing element : [10, 15, 25, 30]

Explanation -
In the above example, we called the remove() function to the list and passed the value 20 as an argument. It removed the 20 from the given list and returned the new modified list.

⇒ The removeAt() Method:
It removes an element from the specified index position and returns it. The syntax is given below.

Syntax -
```
listName.removeAt(int index)
```

Example -
```
void main(){
    var list1 = [10,11,12,13,14];
    print("List before remove element : ${list1}");
    list1.removeAt(3) ;
    print("List after removing element : ${list1}");
}
```

Output:
```
List before remove element : [10, 11, 12, 13, 14]
List after removing element : [10, 11, 12, 14]
```

Explanation -
In the above example, we passed the 3rd index position as an argument to the removeAt() function and it removed the element 13 from the list.

⇒ The removeLast() Method:
The removeLast() method is used to remove the last element from the given list. The syntax is given below.

Syntax-
```
listName.removeLast()
```

Example -
```
void main(){
    List listOne = [12,34,65,76,80];
    print("List before removing element:${listOne}");
    listOne.removeLast();
    print("List after removed element:${listOne}");

}
```

Output:
```
List before removing element:[12, 34, 65, 76, 80]
List after removed element:[12, 34, 65, 76]
```

In the above example, we called the removeLast() method, which removed and returned the last element 80 from the given list.

⇒ The removeRange() Method:
This method removes the item within the specified range. It accepts two arguments - start index and end index. It eliminates all element which lie in between the specified range. The syntax is given below.

Syntax -

```
listName. removeRange();
```

Example -

```
void main(){
    var listOne = [12,34,65,76,80];
    print("List before removing element:${listOne}");
    listOne.removeRange(1,3);
    print("List after removed element:${listOne}");
}
```

Output:

```
List before removing element:[12, 34, 65, 76, 80]
List after removed element:[12, 76, 80]
```

Explanation -
In the above example, we called the removeRange() method and passed start index position 1 and end index position 3 as an arguments. It removed all elements which were belonging in between the specified position.

⇒ Dart Iterating List elements:
The Dart List can be iterated using the forEach method. Let's have a look at the following example.

Example -

```
void main(){
    var listOne = ["Smith","Peter","Handscomb","Devansh","Cruise"];
    print("Iterating the List Element");
    listOne.forEach((item){
    print("${listOne.indexOf(item)}: $item");
 });
}
```

Output:

```
Iterating the List Element
0: Smith
1: Peter
2: Handscomb
3: Devansh
4: Cruise
```

Map:

Dart Map is an object that stores data in the form of a key-value pair. Each value is associated with its key, and it is used to access its corresponding value. Both keys and values can be any type. In Dart Map, each key must be unique, but the same value can occur multiple times. The Map representation is quite similar to Python Dictionary. The Map can be declared by using curly braces {} ,and each key-value pair is separated by the commas(,). The value of the key can be accessed by using a square bracket([]).

⇒ Declaring a Dart Map:
Dart Map can be defined in two methods.

- Using Map Literal
- Using Map Constructor

⇒ Using Map Literals:
To declare a Map using map literal, the key-value pairs are enclosed within the curly braces "{}" and separated by the commas. The syntax is given below.

Syntax -

```
Map mapName = {keyFist:value, keySecond:value [.......,key: value]}
```

Example - 1:

```
void main() {
  Map student = {'name':'Tom','age':'23'};
  print(student);
}
```

Output:**{name: Tom, age: 23}**

Example - 2: Adding value at runtime

```
void main() {
  Map student = {'name':' tom', 'age':23};
  student['course'] = 'B.tech';
  print(student);
}
```

Output:
{name: tom, age: 23, course: B.tech}

Explanation -
In the above example, we declared a Map of a student name. We added the value at runtime by using a square bracket and passed the new key as a course associated with its value.

⇒ Using Map Constructor:
To declare the Dart Map using map constructor can be done in two ways. First, declare a map using map() constructor. Second, initialize the map. The syntax is given below.

Syntax -

```
Map mapName = {}

// After that, initialize the values.
mapName[key] = value
```

Example - 1: Map constructor

```
void main() {
  Map student = {};
```

```
    student['name'] = 'Tom';
    student['age'] = 23;
    student['course'] = 'B.tech';
    student['Branch'] = 'Computer Science';
    print(student);
  }
```

Output:

```
{name: Tom, age: 23, course: B.tech, Branch: Computer Science}
```

Note - A map value can be any object including NULL.

Map Properties

The dart:core:package has Map class which defines following properties.

Properties	Explanation
Keys	It is used to get all keys as an iterable object.
values	It is used to get all values as an iterable object.
Length	It returns the length of the Map object.
isEmpty	If the Map object contains no value, it returns true.
isNotEmpty	If the Map object contains at least one value, it returns true.

Example -

```
void main() {
  var student = Map();
  student['name'] = 'Tom';
  student['age'] = 23;
  student['course'] = 'B.tech';
  student['Branch'] = 'Computer Science';
  print(student);

 // Get all Keys
 print("The keys are : ${student.keys}");

 // Get all values
 print("The values are : ${student.values}");

 // Length of Map
 print("The length is : ${student.length}");

//isEmpty function
print(student.isEmpty);

//isNotEmpty function
print(student.isNotEmpty);
}
```

Output:
```
{name: Tom, age: 23, course: B.tech, Branch: Computer Science}
The keys are : (name, age, course, Branch)
The values are : (Tom, 23, B.tech, Computer Science)
The length is : 4
false
true
```

⇒ **Map Methods:**
The commonly used methods are given below.

addAll() - It adds multiple key-value pairs of other. The syntax is given below.

Syntax -
```
Map.addAll(Map<Key, Value> other)
```

⇒ **Parameter:**
- **other -** It denotes a key-value pair. It returns a void type.

Example -
```
void main() {
  Map student = {'name':'Tom','age': 23};
  print('Map :${student}');

  student.addAll({'dept':'Civil','email':'tom@xyz.com'});
  print('Map after adding  key-values :${student}');
}
```

Output:
```
Map :{name: Tom, age: 23}
Map after adding  key-values :{name: Tom, age: 23, dept: Civil, email: tom@xyz.com}
```

remove() - It eliminates all pairs from the map. The syntax is given below.

Syntax -
```
Map.clear()
```

Example -
```
void main() {
  Map student = {'name':'Tom','age': 23};
  print('Map :${student}');

  student.clear();
  print('Map after removing all key-values :${student}');

}
```

Output:
```
Map :{name: Tom, age: 23}
Map after removing all key-values :{}
```

remove() - It removes the key and its associated value if it exists in the given map. The syntax is given below.

Syntax -

```
Map.remove(Object key)
```

Parameter -

- **Keys -** It deletes the given entries. It returns the value associated with the specified key.

Let's understand the following example.

Example -

```
void main() {
  Map student = {'name':'Tom','age': 23};
  print('Map :${student}');

  student.remove('age');
  print('Map after removing given key :${student}');
}
```

Output:

```
Map :{name: Tom, age: 23}
Map after removing given key :{name: Tom}
```

forEach() - It is used to iterate the Map's entries. The syntax is given below.

Syntax -

```
Map.forEach(void f(K key, V value));
```

Parameter -

It denotes the key-value pair of the map.

Example -

```
void main() {
  Map student = {'name':'Tom','age': 23};
  print('Map :${student}');
  student.forEach((k,v) => print('${k}: ${v}'));

}
```

Output:

```
Map :{name: Tom, age: 23}
name: Tom
age: 23
```