Dart Learning

▼ Day 1 Introduction

1. Comments in Dart

   Single line comments : //

   Multi line comments : /* */

2. Keywords

   Reserved words in a programming language.

   a. print()

   ```
   1. print(”Hello Word”);
   2. print(variable_name);
   ```

   b. var : to declare a variable

3. Don’t forget ; like as in C and C++.

4. Variables

   Those values that can change during programming.

   Defining variables by var. For examples:

   ```
   var name = “Sulabh”
   ```

   Private Variables : To make same line private variable we can use.

   ```
   var _name = “Sulabh”
   ```

   Public Variables :

5. Main Function

   ```
   void main() {

   }
   ```

6. Print in Dart

   ```
   print(variable_name);
   ```

7. Hello Word Program

   ```
   void main() {

     print(”Hello World”)

   }
   ```

   The output will be:

```
Hello World
```

8. Datatypes in Dart
    a. Boolean

        Either true or false

        ```
        bool isLoggedIn = true;

        bool isNotLoggedIn = false;
        ```

    b. Numerical

        Integer

        ```
        int serial = 1;
        ```

        Double

        ```
        double price = 125.25;
        ```

    c. Sting

        ```
        String firstName = “Ram”;
        ```

        i. Single Line

        “ “

        ii. Multiple Line

        “”” “””””

        String Interpolation

        String Concatenation

        String Properties and Methods

        Simple Programs

    d. If we don’t know what data type are we using then we can define it by using var as follows :

        ```
        void main(){
        bool isLoggedIn = true;
        int sn = 1;
        double price = 125.25;

        var name2;

        print(isLoggedIn);
        name2 = false;
        print(name2);
        print(sn);
        name2 = 2;
        print(name2);
        print(price);
        name2 = 12.35;
        ```

```
print(name2);

}
```

The output will be:

```
true
false
1
2
125.25
12.35
```

String and all methods and properties learned about it.

```
void main(){

// String Datatypes

String firstName = "Sulabh";
String lastName  = "Ghimire";
int number = 45;
var newNum;

//String Concatenation
print("My name is $firstName $lastName.");

//String Interpolation
String fullName = firstName +" " + lastName;
print("My name is $fullName.");

//String Properties
print(fullName.isEmpty); //check if string is empty returns true or fasle
print(fullName.isNotEmpty); // checks if string is not empty returns true or false
print(fullName.length); // finds the length of the given string

//Methods

print(fullName.toLowerCase()); //to change string to all lower case
print(fullName.toUpperCase()); //to change string to all upper case
newNum = number.toString(); // converts numerical to string
print(newNum);
print(newNum.runtimeType); // gives the datatype during runtime

}
```

The output will be:

```
My name is Sulabh Ghimire.
My name is Sulabh Ghimire.
false
true
14
sulabh ghimire
SULABH GHIMIRE
45
String
```

Dart Operators

a. Arithmetic Operators
b. Logical Operators
c. Type Test Operators

d. Bitwise Operators

Examples:

```
void main(){

/* Arithmetic OPERATORS

- greater then : returns boolean
- < less than : returns boolean
- = greater then or equals to : return boolean
- <= less than or equals to : returns boolean
- == equals to : returns boolean
- != not equal to : returns boolean value
*/

/* Logical OPERATORS

- && and operator : use to add tow conditions and returns true if both are ture
- || or operator : use to add two conditions and if even one of then is true return true
- ! not operator : use to reverse the result
*/

/* Type Test OPERATOR

- Use to check type
- is : returns true if the two data types matches with each other
- is! : returns false if the two data types doesn't match with each other
*/

}
```

▼ Day-2 Dart Control Flow

1. **If statement**
2. **IF .. ELSE statement**

```
void main()
{
    // if and else statement
    bool isEnrolled = false;

    if (isEnrolled==true){
       print("You are enrolled.");
    }
    else{
       print("You aren't enrolled.");
    }
}
```

The output will be:

```
You aren't enrolled.
```

3. **IF .. ELSE .. ELSEIF statement**

```
void main()
{
    // if statement
    String isEnrolled = "Sulabh";

    if (isEnrolled== "Rabin"){
      print("$isEnrolled is enrolled.");
    }
```

```
    else if (isEnrolled=="Hema"){
      print("$isEnrolled is enrolled.");
   }
   else if (isEnrolled=="Suresh"){
     print("$isEnrolled is enrolled.");
   }
   else {
     print("Noone is enrolled.");
   }
}
```

The output will be:

```
Noone is enrolled.
```

4. **Ternary Operators**

   Syntax:

```
Condition 1? Expression 1:
Expression 2;
```

   ? acts as if and : acts as else

```
void main()
{
    String isEnrolled = "Sabin";

    isEnrolled == "Sulabh" ? print("Hello $isEnrolled") :
    print("You aren't registered");
}
```

   The output will be:

```
You aren't registered
```

5. **JUMP Statements**

   Break statement breaks and jumps to end of a loop.

   Continues to the loop after skipping current execution

6. **CONST vs FINAL Statements**

   CONST are the values that doesn't change from start. Compile time constant.

   Doesn’t change if one initialized. Run time constant.

7. **SWITCH .. CASE Statements**

```
void main()
{
  //Swith Case
  String name = "Sulabh";

  switch(name){

    case "Rabin":
      print("You are $name.");
      break;
```

```
    case "Sabin":
      print("You are $name.");
      break;

    case "Prabin":
      print("You are $name.");
      return;

    case "Kalpana":
      print("You are $name.");
      break;

    default:
      print("You are unknown.");
  }
}
```

The output will be:

```
You are unknown.
```

▼ Day-3 Loops in Dart, Lists

1. **FOR loops in Dart**

   Syntax :

   ```
   for (initialization; condition; increment/decrement){

   //code to be executed

   }
   ```

   Example:

```
// Factorail of a number using for loop
void main()
{
  int num = 5;
  int factorial = 1;

  for (int i = num; i >=1; i--){
    factorial *= i;
  }

  print("The factorial of number $num is $factorial.");
}
```

The output will be:

```
The factorial of number 5 is 120.
```

2. **WHILE Loops**

   Syntax :

   ```
   initialization;

   while (condition) {

   //code to be executed
   ```

```
}
```

We need to initialize the values outside not like in for loops. First we check condition in while loops then only we run the block of codes.

Example:

```
void main() {
    int i = 0;

    while(i<=10){

      print("This is $i loop.");

      i++;
    }
}
```

The output will be:

```
This is 0 loop.
This is 1 loop.
This is 2 loop.
This is 3 loop.
This is 4 loop.
This is 5 loop.
This is 6 loop.
This is 7 loop.
This is 8 loop.
This is 9 loop.
This is 10 loop.
```

3. **DO .. WHILE Loops**

Syntax :

```
initialization;

do {

//code to be executed

} while(condition);
```

Unlike in while loops for do while loops once the code block will get executed even if the condition is false. Then only the code will be checked.

Example:

```
void main() {
    int i = 0;

    do {

      print("This is $i loop.");
      i++;

    }while(i <= 10);
}
```

The output will be:

```
This is 0 loop.
This is 1 loop.
This is 2 loop.
This is 3 loop.
This is 4 loop.
This is 5 loop.
This is 6 loop.
This is 7 loop.
This is 8 loop.
This is 9 loop.
This is 10 loop.
```

4. **FOR .. IN Loops**

   For in loop we must have collection data type like List.

   Syntax:

   ```
   for (data_type_decleration var in collection_data_type){
       // Code to be executed
   }
   ```

   Example:

   ```
   void main() {

     List<String> names= ['sulabh', 'ghimire', 'sajan', 'preety'];

     for (String name in names){
       print(name);
     }

   }
   ```

   Output:

   ```
   sulabh
   ghimire
   sajan
   preety
   ```

5. **FOR .. EACH Loops**

   Example:

   ```
   void main() {

     List<String> names= ['sulabh', 'ghimire', 'sajan', 'preety'];

     names.forEach((name){
       print('Hello $name');
     });

   }
   ```

   Output:

   ```
   Hello sulabh
   Hello ghimire
   ```

```
Hello sajan
Hello preety
```

6. **List Datatype in Dart**

   Syntax to declare list datatype.

```
List var_name = [ //values ];
Example:
List student_names = ['Rabin', 'Sabin', 'Prabin']
```

   List is a dynamic datatype. So, it can contain any type of data within it. Like

```
List values = ['Sulabh', 1, bool];
```

   The above example is valid datatype in dart.

   If we want our list to contain same type of data values then we can do that as follows:

```
List<data_type> name = [ //specidied datatypes seperated by commas];

for example:
Li<String> names = ['Sulabh', 'Nischal', 'Priya'];
```

   Indexing:

   We can use indexing in list. The indexing values in list starts from 0 and ends to length of the string-1.

   Example:

```
void main(){

  // List datatypes

  List<String> names = ['Rabin', 'Sabin', 'Prabin', 'Sulabh', 'Priya', 'Sagar', 'Kalpana'];

  print(names[2]);

}
```

   Output:

```
Prabin
```

   Properties in List

```
void main(){

  List datatypes

  List<String> names = ['Rabin', 'Sabin', 'Prabin', 'Sulabh', 'Priya', 'Sagar', 'Kalpana'];

  print(names.length); // Gives length of string
  print(names.first); // Gives first element of list
  print(names.last); // Gives last velement of string
  print(names.isEmpty); // Returns True if list is empty
  print(names.isNotEmpty); // Returns True if list isn't empty
  print(names.reversed); // It returns an iterable object containing the lists value sin reversed order
  print(names.Single); //It is used to check if list had one emelent and returns it
}
```

The output will be:

```
7
Rabin
Kalpana
false
true
('Kalpana'. 'Sagar', 'Priya', 'Sulabh', 'Prabin', 'Sabin', 'Rabin')
```

Methods in List

```
void main(){

  // List datatypes

  List names = ['Rabin', 'Sabin', 'Prabin', 'Sulabh'];

  //Ads elements to last of the list
  names.add(14.5);
  print(names);

  List names2 = ['Priya', 'Sandhya'];

  //Add one list to another at the last of the list that current is being
  //added to
  names.addAll(names2);
  print(names);

  //insert() is used to insert element to specified poistiion.
  //Synatx: list_name.insert(index, value);
  names.insert(2, true);
  print(names);

  //insertaLL() is used to insert a list of multiple values to a given list
  //at specified position
  //Syntax: list_name(index, list_to_be_inserted)
  List<int> names3 = [45, 12, 65];
  names.insertAll(1, names3);
  print(names);

  //Updating in list
  //Syntax: list_name[index] = new_value;
  names[2] = true;
  print(names);

}
```

The output is:

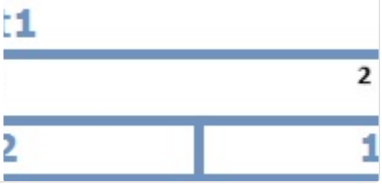
```
[Rabin, Sabin, Prabin, Sulabh, 14.5]
[Rabin, Sabin, Prabin, Sulabh, 14.5, Priya, Sandhya]
[Rabin, Sabin, true, Prabin, Sulabh, 14.5, Priya, Sandhya]
[Rabin, 45, 12, 65, Sabin, true, Prabin, Sulabh, 14.5, Priya, Sandhya]
[Rabin, 45, true, 65, Sabin, true, Prabin, Sulabh, 14.5, Priya, Sandhya]
```

Can be learned about it more at:

Dart Lists - W3Schools | W3Adda

Dart Lists In Dart, list data type is used to represent a collection of objects. A List is an ordered group of objects. The List data type is actually similar to the concept of an array in other programming languages. An array is used to hold multiple values in

https://www.w3adda.com/dart-tutorial/dart-lists



▼ Day-4 Maps, Enumes and Runes

1. **Maps**

   Maps are like dictionary from python. It has key and values. Generally map is a dynamic data type which can be described as:

   Map <dynamic, dynamic> names

   We have data-type for both key and values. The first dynamic determines data-type for key and second dynamic determines data-type for values.

   Syntax:

   ```
   Map map_name = {'key1' :value, 'key2' : value, ......., 'keyn': value};
   ```

   Now, we can change the data-types of both key and values as follows:

   ```
   Map<data_type_for_key, data_type_for_value> = {'key1' :value, 'key2' : value, ......., 'keyn': value};
   ```

   We generally use string for key and other data-types for values:

   Example:

   ```
   void main(){

     Map<int, String> info = {1:'Sulabh Ghimre', 2:'Bijen Poudel'};
     print(info[1]);

     Map<String, dynamic> info2 = {'Bool': true, 'Age':22, 'Nationality': 'nepalese'};
     print(info2['Nationality']);

     Map<String, int> info3 = {'OS':80, 'ES':90};
     print(info3);

   }
   ```

   Output:

   ```
   Sulabh Ghimre
   nepalese
   {OS: 80, ES: 90}
   ```

   Accessing the values

   We can use key to access a value.

   ```
   Map info = {'key1' :value, 'key2' : value, ......., 'keyn': value};
   print(info['key1']);
   ```

   Example:

   ```
   void main(){

     Map info = {'first_name': 'Sulabh', 'last_name':'Ghimire', 'Age':22, 'address':'Pokhara', 'married':false};

     //printing the map
     print(info);

     //acessing the certain values from keys
     String fname = info['first_name'];
     String lname = info['last_name'];
   ```

```
  int age      = info['Age'];
  String add   = info['address'];
  bool status  = info['married'];

  print("Your name is $fname $lname.");
  print("You are $age years old.");
  print("You live in $add.");

  if (status==true) {

    print("Marital status : Married");

  }
  else{
    print("Marital staus : Not married");
  }
}
```

The output will be:

```
{first_name: Sulabh, last_name: Ghimire, Age: 22, address: Pokhara, married: false}
Your name is Sulabh Ghimire.
You are 22 years old.
You live in Pokhara.
Marital staus : Not married
```

Adding Values to Map Literals at Runtime

Example:

```
void main() {
   Map<String, String> details = {'Usrname':'tom','Password':'pass@123'};
   details['Uid'] = 'U1oo1';
   print(details);
}
```

Output:

```
{Usrname: tom, Password: pass@123, Uid: U1oo1}
```

Properties in Maps:

S.N.	Properties	Description
1.	keys	Returns an iterable object representing keys
2.	values	Returns an iterable object representing values
3.	length	Returns the size of the Map
4.	isEmpty	Returns true if the Map is an empty Map
5.	isnotEmpty	Returns true if the Map is an not empty Map

Example:

```
void main() {

  Map<String, String> details = {'Usrname':'tom','Password':'pass@123'};

  print(details.length);
  print(details.keys);
  print(details.values);
  print(details.isEmpty);
  print(details.isNotEmpty);

}
```

Output:

```
2
(Usrname, Password)
(tom, pass@123)
false
true
```

Methods in List:

S.N.	Properties	Description
1.	addAll()	Adds all key-value pairs of other to this map.
2.	clear()	Removes all pairs from the map.
3.	remove()	Removes key and its associated value, if present, from the map.
4.	forEach()	Applies f to each key-value pair of the map.

Example:

```
void main() {

  Map m = {'name':'Tom','Id':'E1001'};
  print('Map :$m');

  //Solutin to problem i faced during printing
  print('Map :${m['name']}');

  m.addAll({'dept':'HR','email':'tom@xyz.com'});
  print('Map after adding  entries :$m');

  //using forEach()
  m.forEach((k,v) => print('$k: $v'));

  var email_popped = m.remove('email');
  var phone_popped = m.remove('phone');
  print('Map removeing email and phone :$m');
  print('Popped email : $email_popped');
  print('Popped phone : $phone_popped');

  m.clear();
  print('Final map is $m');

}
```

Output:

```
Map :{name: Tom, Id: E1001}
Map :Tom
Map after adding  entries :{name: Tom, Id: E1001, dept: HR, email: tom@xyz.com}
name: Tom
Id: E1001
dept: HR
email: tom@xyz.com
Map removeing email and phone :{name: Tom, Id: E1001, dept: HR}
Popped email : tom@xyz.com
Popped phone : null
Final map is {}
```

2. **Enumes**

   They are user-defined data-types in dart and are defined outside the void main() function.

   Syntax:

   ```
   enum data_type_name{
   ```

```
};

void main(){

}
```

Examples:

```
enum TrafficLights {

  red,
  green,
  yellow

}

void main(){

  TrafficLights trafficlight = TrafficLights.red;

  if (trafficlight == TrafficLights.red){
    print("Stop");
  }
  else if (trafficlight == TrafficLights.green){
    print("Go");
  }
  else{
    print("Slow Down");
  }

}
```

Output:

```
Stop
```

3. **Runes**

   They are like emojis. Runes are uni-code characters.

   Example:

```
void main(){

  var heart = '\u2665';
  var laugh = '\u{1f600}';

  print("hello $heart");
  print("hello $laugh");
}
```

Output:

```
hello ♥
hello 😀
```

▼

▼ Day-4 Functions

1. **Main Function**

We write everything inside main function. Dart executes main function.

Syntax:

```
void main(){

}
```

2. **General Function**

   We can define the general function both inside and outside the function as well.

   If we want our function to return **dynamic data-type** then we can skip writing **function_return_type** and keep it **empty**.

   The values passed to function are **arguments**.

   The values received in function are **parameters.**

   We can skip data_types of parameters but it is good to provide it.

   Syntax for function definition:

```
function_return_type function_name(data_type par1, data_type par2, ....){
    // our code to be executed
}
```

   Syntax for function call:

```
function_name(arg1, arg2, ....);
```

   Example:

```
 void display(){

    print("hello world!");

}

void main() {

  display();

}
```

   Output:

```
hello world!
```

   Types of Parameters:

   a. Default parameters : What value do we want to pass if we don't pass any arguments. Default parameters must be enclosed with in big brackets [ ]. Always write default parameters later on.

      Example:

```
void main() {

  print(add(20));

}
```

```
int add(int b, [int a = 20]){

  return a+b;

}
```

Output:

```
40
```

b. Positioned parameters : Arguments and Parameters should be in same position.

c. Optional parameters : You can pass or ignore them.

d. Named Parameters : Arguments will be passed with name of parameters.

   for named parameters we define them in { } brackets and while calling the function we give their name with colon sign :

   Example:

```
void main() {

  print(add(20, a:12));

}

int add(int b, {int a = 20}){

  return a+b;

}
```

   Output:

```
32
```

3. **Arrow Function**

   Arrow functions are used if we need to perform a single task in the function.

   Examples:

```
int addNumbers(int a ,int b) => a+b;

void main() {

  print(addNumbers(12,15));

}
```

   Output:

```
27
```

4. **Anonymous Function**

   The function that doesn't have any name is called anonymous function.

   Example:

```
void main() {

  Function sum = () => 2+3;

  print(sum());

}
```

Output:

```
5
```

▼ Day-5 OOP

1. **Classes**

   Classes are generally group. We write our classes outside of void main() function.

   Class is collection of variables and methods that go together.

   We will be able to access static variables without creating objects and will be same for all objects of that class.

   Syntax of class:

```
class class_name {

}

void main(){


}
```

   Creating an object:

```
var std = new class_name(<constructor_arguments>);
```

   Example:

```
class Student {

  //properties of a class
  //this underscore makes gpa a private variable
  //something is private if it can be acessed only insde class
  double _gpa = 0.0;
  bool _isfailing = false;
  String university = "TU";

  //method-- a function inside of a class
  void showInfo(){
    print("Your gpa is $_gpa and university is $university.");
  }

  //seter --a method which allows us to change a variable inside class
  void setGPA(double gpa){
    if (gpa < 0){
      throw Exception("GPA cann't be negative");
    }
    if (gpa < 2.0){
      _isfailing = true;
    }
    _gpa = gpa;
  }
```

```
  //getter -- to show a private variable
  double get gpa => _gpa;

  bool get isfailing => _isfailing;
}
//Encapsulation -- we are sending the methods to work on some data, together with the class
void main(){

  Student newStudent = Student();
  newStudent.setGPA(4.0);
  newStudent.university = "PU";
  print(newStudent.gpa);
  print(newStudent.isfailing);
  newStudent.showInfo();

}
```

Output:

```
4
false
Your gpa is 4 and university is PU.
```

Example 2:

```
//Parameterized Costructoes

class Student {

  String _name ;

  //paramaterized constructors
  Student(this._name);


  //getter for student name
  String get name => _name;
}

void main(){

  Student std = Student("Sulabh"); //Making object ofparameterized constructors
  print(std.name);
}
```

Output:

```
Sulabh
```

We can’t leave the initialization of class properties empty but we can do something like above to remove the error. We can also mark it late before its data type. If the property is marked late then you can't use it if it isn’t initalized.

```
late int value;
```

Default Constructors:

Code that runs when the class is made. The name of constructor should be same as of the class. There should only be one constructor for a class.

Example:

```
// A constructor is a method that lets us easily set values for a new class
// when we create a new class

class Student {

  //Constructors in darts
  static int data_no = 0;
  String _name = "Rabin";

  //Setting up a constructor of class Student
  Student(){
    print("I am a default constructor.");
    data_no++;
  } //Default constructors

  //static function
  static void fun(){
    print ("I can be acessed by class and am common.");
    data_no++;
    print(data_no);
  }

  //setting up setter for rabin
  void changeName(String name){
    print("We will be changing name from $_name to $name");
    _name = name;
  }

  //setting up a getter for private variables
  String get name => _name;

}

void main(){

  print(Student.data_no);
  var std = Student();
  std.changeName("Sulabh");
  print(std.name);
  print(Student.data_no);
  var std2 = Student();
  print(Student.data_no);
  Student.fun();
}
```

Output:

```
0
I am a default constructor.
We will be changing name from Rabin to Sulabh
Sulabh
1
I am a default constructor.
2
I can be acessed by class and am common.
3
```

Parameterized Constructors:

Example:

```
//Parameterized Costructoes

class Student {

  late String _name ;

  Student(String name){
    this._name = name;
  }

  //getter for student name
```

```
  String get name => _name;
}

void main(){

  Student std = Student("Sulabh"); //Making object ofparameterized constructors
  print(std.name);
}
```

Output:

```
Sulabh
```

Parameterized Named Constructors:

Example

```
//Parameterized named Costructoes

class Student {

  late String _name ;

  Student({String name = "Default name",}){
    this._name = name;
  }

  //getter for student name
  String get name => _name;
}

void main(){

  Student std = Student(); //Making object of name paramaterized
  print(std.name);
  Student std2 = Student(name : "Sulabh"); //Making object of name paramaterized
  print(std2.name);
}
```

Output:

```
Default name
Sulabh
```

Named Constructors

Example:

```
//Named Constructors

class Student {

  late String _name ;

  Student({String name = "Default name",}){
    this._name = name;
  }

  Student.initilization(){
    print("Initilization");
  }

  //getter for student name
  String get name => _name;
}

void main(){
```

```
  //Default constructors calls itself but named constructors need to be called.
  Student std = Student();
  print(std.name);
  Student std2 = Student(name : "Sulabh");
  print(std2.name);
  //We don't need to construct as like above
  Student.initilization();
}
```

Output:

```
Default name
Sulabh
Initilization
```

2. **Inheritance in Dart**

Single Inheritance:

Example:

```
//Inheritance in Dart

class A{

  String ?_name; // we have initilized it to null an dmade it private

  //setter for _name
  changeName(String name){
    _name = name;
  }

  //getter for _name
  get name => _name;

}

class B extends A{

  String address = "Address from class B";

}

void main(){

  A aobj = A();  // created an pbject A
  aobj.changeName("Sulabh"); // Chnaged the object property name to Sulabh
  B bobj = B(); // created and object B
  print(bobj.name); // inherited the name from class A by B and printed null
  bobj.changeName("Sajan"); // chnaged the name of object that B inherited
  print(bobj.name); // printed the name after change
  print(bobj.address); // property of A

}
```

Output

```
null
Sajan
Address from class B
```

Multiple Inheritance:

Example:

```
//Multilevel inheritance
//Dart doesn't support multiple inheritance
```

```
class A{
  String name = "Name from A";
}

class B extends A{
  String address = "Address from class B";
}

class C extends B{
  String phone = "Phone from C";
}

class D extends C{
  String friend = "Friend name from C.";
}

void main(){

  D classDobj = D();
  print(classDobj.friend);  //Own property
  print(classDobj.phone);   //Inherit from C
  print(classDobj.address); //Inherit from B
  print(classDobj.name);    //Inherit from A


}
```

Output:

```
Friend name from C.
Phone from C
Address from class B
Name from A
```

Hierarchical Inheritance

Example:

```
//Hierarchical inheritance
//Dart doesn't support multiple inheritance

class A{
  String name = "Name from A";
}

class B extends A{
  String address = "Address from class B";
}

class C extends A{
  String phone = "Phone from C";
}

void main(){

  C classCobj = C();
  print(classCobj.phone); //Own property
  print(classCobj.name); //Inherit from A
  B classBobj = B();
  print(classBobj.address); //Own property
  print(classBobj.name); //Inherit from A


}
```

Output:

```
Phone from C
Name from A
Address from class B
Name from A
```

3. **Constructors in Inheritance**

Empty/Default Constructors

Example:

```
void main(){

  B bobj = B();

}

class A{
  A(){
    print("A constructor");
  }
}

class B extends A{
  B(){
    print("B constructor");
  }
}

//first the constructor of parent class is called then only of
//child class.
```

Output:

```
A constructor
B constructor
```

Non-Zero Arguments Constructors

Example:

```
void main(){

  B bobj = B();

}

class A{
  A(String name){
    print("A constructor $name");
  }
}

class B extends A{
  B():super(""){
    print("B constructor");
  }
}
```

Output:

```
A constructor
B constructor
```

Example 2

```
//Constructors in Inheritance
//If there is zero ragument constructor then the parent class
//constructors get called automatically
```

```
//If there are arguments in the constructor of the parent class
//then we have use super()

class Person{

  String ?name;
  int ?age;

  Person(String name , int age){

    this.name = name;
    this.age = age;

  }

  showPersonInfo(){

    print("Name is $name");
    print("Age is $age");

  }
}

class Employee extends Person{

  int ?empSalary;

  Employee(int empSalary  , String name , int age):super(name, age){

    this.empSalary = empSalary;

  }

  showEmpInfo(){
    print("Employee Name Is : $name");
    print("Employee Age Is : $age");
    print("Employee Salary Is : $empSalary");
  }

}

void main(){

  print("Making a person class.");
  Person p = Person("Sandip", 22);
  print("Using its own method to show data.");
  p.showPersonInfo();
  print("");
  print("Making a Employee class.");
  Employee e = Employee(200000, "Sulabh", 22);
  print("Using its own method to show data.");
  e.showEmpInfo();
  print("Using its inherited method to show data.");
  e.showPersonInfo();
}
```

Output:

```
Making a person class.
Using its own method to show data.
Name is Sandip
Age is 22

Making a Employee class.
Using its own method to show data.
Employee Name Is : Sulabh
Employee Age Is : 22
Employee Salary Is : 200000
Using its inherited method to show data.
Name is Sulabh
Age is 22
```

▼ Day-6 OOP

1. **Mixins In Dart**

   Mixins in Dart are a way of using the code of a class again in multiple class hierarchies. We make use of the with keyword followed by one or more mixins names.

   Mixins can be used in two ways, the first case is when we want to make use of class code in such a way that the class doesn't have any constructor and the object of the class is extended. In such a case, we use the with keyword.

   Another case is when we want our mixin to be usable as a regular class, then we make use of the **mixin** keyword instead of class.

   Example:

   Imagine you are building a wildlife simulation app which needs to have a **Mosquito** class. As an experienced and foreseeing developer, you **extract** the things mosquitoes have in common with other similar insects into **abstract classes.**

```
abstract class Insect {
  void crawl() {
    print('crawling');
  }
}

abstract class AirborneInsect extends Insect {
  void flutter() {
    print('fluttering');
  }

  void buzz() {
    print('buzzing annoyingly')
  }
}

class Mosquito extends AirborneInsect {
  void doMosquitoThing() {
    crawl();
    flutter();
    buzz();
    print('sucking blood');
  }
}
```

   Awesome! You've done it! Adding new insects will go like a breeze without any code duplication... until you realize you also want to have a **Swallow** class (something has to eat all the mosquitoes, after all).

   Again, there are some actions which all birds have in common, so you create an abstract **Bird** class. Then it strikes you - *birds flutter their wings too*! But you ***cannot*** extract the **flutter** method out of the **AirborneInsect** class into a new class called **Fluttering**.

   Why? While **Bird** *could* extend **Fluttering**, this *wouldn't be possible* with **AirborneInsect** because it already extends **Insect**. Dart simply doesn't support multiple inheritance (which is a good thing).

   Oh no, you need to add the **flutter** method into the **Bird** class too! Code duplication awaits.

```
abstract class Insect {
  void crawl() {
    print('crawling');
  }
}

abstract class AirborneInsect extends Insect {
  void flutter() {
    print('fluttering');
  }

  void buzz() {
```

```
    print('buzzing annoyingly')
  }
}

class Mosquito extends AirborneInsect {
  void doMosquitoThing() {
    crawl();
    flutter();
    buzz();
    print('sucking blood');
  }
}
abstract class Bird {
  void chirp() {
    print('chirp chirp');
  }

  // Duplicate method
  void flutter() {
    print('fluttering');
  }
}

class Swallow extends Bird {
  void doSwallowThing() {
    chirp();
    flutter();
    print('eating a mosquito');
  }
}
```

Using Mixins

Mixins are defined in the Dart documentation as *"a way of reusing a class's code in multiple class hierarchies".* Simply said, mixins allow you to plug in blocks of code *without* needing to create sub classes. Declaring a mixin is very simple:

```
mixin Fluttering {
  void flutter() {
    print('fluttering');
  }
}
```

This **mixin** can be *mixed into* regular classes and abstract classes using the **with** keyword. In the wildlife simulation app's case, you'd probably use it on the abstract classes.

```
mixin Fluttering {
  void flutter() {
    print('fluttering');
  }
}

abstract class Insect {
  void crawl() {
    print('crawling');
  }
}

abstract class AirborneInsect extends Insect with Fluttering {
  void buzz() {
    print('buzzing annoyingly');
  }
}

class Mosquito extends AirborneInsect {
  void doMosquitoThing() {
    crawl();
    flutter();
    buzz();
    print('sucking blood');
  }
}
```

```
abstract class Bird with Fluttering {
  void chirp() {
    print('chirp chirp');
  }
}

class Swallow extends Bird {
  void doSwallowThing() {
    chirp();
    flutter();
    print('eating a mosquito');
  }
}
```

3. **Interface in Dart**

   to achieve interface we use implements keyword

   Note:

   Study again later

4. Null AWARE OPERATOR

```
void main(){

  String name = "Rabin";
  String ?email;

  print(email!);
}
```

5. Exception Handling

   a. Default

```
try{

}
catch(e){

}
```

   Example:

```
void main(){

  String name = 'Rabin';
  int num=2;

  try{
    print("Helloi");
    var value = name + num;
  }
  catch(e){
    print(e.toString())
      //what happens when error occura
  }
}
```

6. **Abstract Classes and Methods**

   An ***Abstract class*** in Dart is defined for those classes which contain one or more than one abstract method (methods without implementation) in them. Whereas, to declare 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 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 to the extended subclasses.

```
abstract class class_name {

    // Body of the abstract class
}
```

Example:

```
abstract class AbstractClass {
    // Creating Abstract Methods
    void say();
    void write();
    void playing(){
      print("We all play games together");
    }
}

//abstract methods must be overidden others no need
//we cann't create the object of Abstract classes

class DerivedClass extends AbstractClass{
    @override
    void say()
    {
        print("K cha sathi haru");
    }

    @override
    void write()
    {
        print("Lekhdina");
    }
}

void main()
{
    DerivedClass fromDerivedObject = new DerivedClass();
    fromDerivedObject.say();
    fromDerivedObject.write();
  fromDerivedObject.playing();
}
```

Output:

```
K cha sathi haru
Lekhdina
We all play games together
```

▼ Day-6 Advanced Topics

1. **Lambda Expression**

No name function. Same as anonymous function.

Example:

```
void main(){

  var addNumbers = (a, b) => a*b;


  print("MULTIPLICATION is ${addNumbers(4.5, 6.5)}");

}
```

Code:

```
MULTIPLICATION is 29.25
```

2. **Lexical Scope**

   Example:

```
void main(){

  String msg = "Parent Scope";
  print(msg);
  //print(name); CAnnot be used as name has scopr only inside modify

  void modify(){
    String name = "Sulavbh";
    print("------------------------------");
    print(msg);
    msg = "Functional Scope";
    print(msg);
    print("------------------------------");
  }
  //function modifies the msg so this is lexical scope
  modify();
  print(msg);
  //print(name); CAnnot be used as name has scopr only inside modify
}
```

   Output:

```
Parent Scope
------------------------------
Parent Scope
Functional Scope
------------------------------
Functional Scope
```

3. **Higher Order Function**

   The function that takes function as an argument or returns function is called higher order function.

```
void main(){

  void someFunction(String message, Function myFunc){
    print("Message");
    myFunc(2,3);
  }

  Function otherFunction(String msg){

  Function multi = (number) => number * 5;
    return multi;
```

```
  }

  Function addNumber = (a,b) => print(a+b);
  someFunction("Message", addNumber);



}
```

Output

```
Message
5
```

4. **Recursion**

```
void main(){

  int num = 5;
  print("The factorial of number $num is ${factorial(num)}");

}

int factorial(int num){

  if (num == 0 || num == 1){
    return 1;
  }
  else{
    return num * factorial(num-1);
  }

}
```

Output:

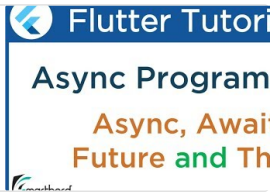
```
The factorial of number 5 is 120
```

5. **Future in Dart**

#4.4 Dart & Flutter Asynchronous Tutorial using Future API, Await, Async and Then functions

Access 7000+ courses for 60 days FREE: https://pluralsight.pxf.io/c/1291657/424552/7490 Dart and Flutter are single-threaded. How to perform long-running ope...

https://www.youtube.com/watch?v=g9Uk1Xou0m4




```
void main(){

  print("Our main thread starts here:");

  printFile();

  print("Our main thread ends here");

}

printFile(){

  Future<String> fileContent = downloadFile();
  print("The content of file is $fileContent.");

}
```

```
Future<String> downloadFile(){

  Future<String> result = Future.delayed(Duration(seconds: 6), (){
    return 'My file content.';
  });

  return result;

}
```

Output:

```
Our main thread starts here:
The content of file is Instance of '_Future<String>'.
Our main thread ends here
```

After we use async and await keywords

```
void main(){

  print("Our main thread starts here:");

  printFile();

  print("Our main thread ends here");

}

printFile() async{

  String fileContent = await downloadFile();
  print("The content of file is $fileContent.");

}

Future<String> downloadFile(){

  Future<String> result = Future.delayed(Duration(seconds: 6), (){
    return 'My file content.';
  });

  return result;

}
```

Output:

```
Our main thread starts here:
Our main thread ends here
The content of file is My file content..
```