DART COURSE

- To write any Dart program, be it a script or a Flutter app, you must define a function called main. void main() { **print('Hello, Dart');** } This function tells Dart where the program starts, and it must be in the file that is considered the "entry point" for you program. By convention, this will be in a file called main.dart.
- What we write inside print() function will be executed (as an output) throw the **Debug Console.**
- Every line must end with a semicolon.

Example:

```
void main() {
  print("Hello");
}
```

Variables

Variables is an identifier used to refer memory location in computer memory that holds a value for that variable, this value can be changed during the execution of the program. When you create a variable in Dart, this means you are allocating some space in the memory for that variable. The size of memory block allocated and type of the value it holds is completely dependent upon the type of variable.

Rules For Naming a Variable in Dart

- Variable name can consist of letter and alphabets.
- Keywords are not allowed to use as a variable name.
- Blank spaces are not allowed in variable name.
- First character of variable should always be alphabet and cannot be digit.

- Variable name is case sensitive i.e. UPPER and lower case are significant.
- Special characters like #, $ are not allowed except the underscore (_) and the dollar ($) sign.
- Recommendation :- Variable name must be readable and should be relative to its purpose.

Declaring Variables in Dart

- In Dart, a variables must be declared before they are used. Variables are declared using the var keyword followed by variable name that you want to declare. Dart is a type inferred language, which allows compiler automatically infer(know) the type of data we want to store based on the initial value we assign.

Example:

```
void main() {
  var count;
}
```

Type Annotations

- Although Dart is a type inferred language, you can optionally provide a type annotation while declaring a variable to suggest type of the value variable can hold. In Dart, by prefixing the variable name with the data type ensures that a variable holds only data specific to a data type.

Example:

```
void main() {
  String name;}
```

Declaring Multiple Variables

- In Dart, it is possible to declare multiple variables of same type in a single statement separated by commas, with a single type annotation.

Example:

```
void main() {
  String fname, lname;
}
```

Variable Assignment in Dart

- The assignment operator (=) is used to assign values to a variable, the operand in the left side of the assignment operator (=) indicates the name of the variable and the operand in the right side of the assignment operator (=) indicates the value to be stored in that variable.

Example:

```
void main() {
  String name;
  name = "Ahmed";
}
```

Initializing a Variable in Dart

- In Dart, it is possible to declare and assign some initial value to a variable in single statement.

Example:

```
void main() {
  String name = "Ahmed";
}
```

Default Value

- In Dart, uninitialized variables are provided with an initial value of null. Even variables with numeric types are initially assigned with null value, because numbers like everything else in Dart are objects.

Example:

```
void main() {
  int ctr;
  assert(ctr == null);
}
```

Note:

```
void main() {
  String name = "ahmed";
  print("$name"); // ahmed
}
```

Datatypes

- Data types help you to categorize all the different types of data you use in your code. For e.g. numbers, texts, symbols, etc. The data type specifies what type of value will be stored by the variable. Each variable has its data type. Dart supports the following built-in datatypes: **Numbers, Strings, Booleans, Lists, Sets, Maps, Runes, Symbols, Null**

Built-in Types

- **Numbers (int, double, num):** It represents numeric values. When you need to store numeric value on dart, you can use either int or double. Both int and

double are subtype of num. You can use num to store both int or double value.

Example:

```
void main() {
  // Declaring Variables
  int num1 = 100; // without decimal point.
  double num2 = 130.2; // with decimal point.
  num num3 = 50;
  num num4 = 50.4;

// For Sum
  num sum = num1 + num2 + num3 + num4;

// Printing Info
  print("Num 1 is $num1");
  print("Num 2 is $num2");
  print("Num 3 is $num3");
  print("Num 4 is $num4");
  print("Sum is $sum");
}
```

Example – Round Double Value to 2 Decimal Places

```
void main() {
// Declaring Variables
  double prize = 1130.2232323233233; // valid.
  print(prize.toStringAsFixed(2));
}
```

- **String:** It represents a sequence of characters. String helps you to store text data. You can store values like **I love dart, New York 2140** in String. You can use single or double quotes to store string in dart.

Example:

```
void main() {
// Declaring Values
  String schoolName = "Diamond School";
```

```
  String address = "New York 2140";

// Printing Values
  print("School name is $schoolName and address is $address");
}
```

- If you want to create a multi-line String in dart, then you can use triple quote with either single or double quotation marks.

Example:

```
void main() {
// Multi Line Using Single Quotes
  String multiLineText = '''
This is Multi Line Text
with 3 single quote
I am also writing here.
''';

// Multi Line Using Double Quotes
  String otherMultiLineText = """
This is Multi Line Text
with 3 double quote
I am also writing here.
""";

// Printing Information
  print("Multiline text is $multiLineText");
  print("Other multiline text is $otherMultiLineText");
}
```

- You can also create raw string in dart. Special characters won't work here. You must write r after equal sign.

Example:

```
void main() {
// Set prize value
  num prize = 10;
  String withoutRawString = "The value of prize is \t $prize"; // regular String
  String withRawString = r"The value of prize is \t $prize"; // raw String
```

```
  print("Without Raw: $withoutRawString"); // regular result
  print("With Raw: $withRawString"); // with raw result
}
```

- **Boolean:** In Dart, boolean holds either true or false value. You can write the bool keyword to define the boolean data type. You can use boolean if the answer is true or false. Consider the answer to the following questions:
    - Are you married?
    - Is the door open?
    - Does a cat fly?
    - Is the traffic light green?
    - Are you older than your father?

These all are yes/no questions. Its a good idea to store them in boolean.

Example:

```
void main() {
  bool isMarried = true;
  print("Married Status: $isMarried");
}
```

- **List:** The list holds multiple values in a single variable. It is also called arrays. If you want to store multiple values without creating multiple variables, you can use a list.

Example:

```
void main() {
  List<String> names = ["Raj", "John", "Max"];
  print("Value of names is $names");
  print("Value of names[0] is ${names[0]}"); // index 0
  print("Value of names[1] is ${names[1]}"); // index 1
  print("Value of names[2] is ${names[2]}"); // index 2
```

```
  // Finding Length of List
  int length = names.length;
  print("The Length of names is $length");
}
```

Note:

- List index always starts with 0. Here names[0] is Raj, names[1] is John and names[2] is Max.
- **Set:** An unordered collection of unique items is called set in dart. You can store unique data in sets.
- Set doesn't print duplicate items.

Example:

```
void main() {
  Set<String> weekday = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
  print(weekday);
}
```

- **Map:** In dart, a map is an object where you can store data in key-value pairs. Each key occurs only once, but you can use same value multiple times.

Example:

```
void main() {
  Map<String, String> myDetails = {
    'name': 'John Doe',
    'address': 'USA',
    'fathername': 'Soe Doe'
  };

  print(myDetails['name']);
}
```

- **Var** keyword: In dart, **var** automatically finds a data type. In simple terms, var says if you don't want to specify a data type, I will find a data type for you.

Example:

```
void main() {
  var name = "John Doe"; // String
  var age = 20; // int

  print(name);
  print(age);
}
```

- **Runes:** With runes, you can find Unicode values of String. The Unicode value of a is 97, so runes give 97 as output.
- As we know that, the strings are the sequence of Unicode UTF-16 code units. Unicode is a technique which is used to describe a unique numeric value for each digit, letter, and symbol. Since Dart Runes are the special string of Unicode UTF-32 units. It is used to represent the special syntax.
- For example - The special heart character ♥ is equivalent to Unicode code \u2665, where \u means Unicode, and the numbers are hexadecimal integer. If the hex value is less or greater than 4 digits, it places in a curly bracket ({}). For example - An emoji 😀 is represented as \u{1f600}.

Example:

```
void main() {
  String value = "a";
  print(value.runes);
}
```

Example:

```
void main() {
  var heart_symbol = '\u2665';
  var laugh_symbol = '\u{1f600}';
  print(heart_symbol);
  print(laugh_symbol);
}
```

Type Conversion in Dart

- In dart, type conversion allows you to convert one data type to other type. For e.g. to convert String to int, int to String or String to bool, etc.

Convert String to Int in Dart

- You can convert String to int using int.parse() method. The method takes String as an argument and converts it into an integer.

Example:

```
void main() {
  String strvalue = "1";
  print("Type of strvalue is ${strvalue.runtimeType}");
  int intvalue = int.parse(strvalue);
  print("Value of intvalue is $intvalue");
// this will print data type
  print("Type of intvalue is ${intvalue.runtimeType}");
}
```

radix

- In computing, a radix is the base of a number system. When we write numbers in standard decimal notation, we use a radix of 10. This means that each digit in a number represents a power of 10. For example, in the number 123, the "1" represents 100, the "2" represents 20, and the "3" represents 3.

- When working with other number systems, such as hexadecimal (base 16), binary (base 2), or octal (base 8), we use a different radix. In these number systems, each digit represents a power of the radix. For example, in hexadecimal notation, the digits 0 through 9 represent values 0 through 9, and the letters A through F represent values 10 through 15.
- The int.parse method in Dart provides an optional named parameter called radix which specifies the base of the number system being used. When converting a string representation of a number to an integer using int.parse, we can set the radix parameter to the appropriate value for the number system we are working with. For example, to convert a hexadecimal string to an integer, we would set radix: 16.

Example:

```
int hexToDecimal(String hexString) {
  // Use int.parse with radix 16 to convert the hex string to decimal
  int decimalNum = int.parse(hexString, radix: 16);

  return decimalNum;
}
```

Convert Int to String in Dart

- You can convert int to String using toString() method.

Example:

```
void main() {
  int one = 1;
  print("Type of one is ${one.runtimeType}");
  String oneInString = one.toString();
  print("Value of oneInString is $oneInString");
```

```
// this will print data type
  print("Type of oneInString is ${oneInString.runtimeType}");
}
```

Convert Double to Int in Dart

- You can convert double to int using toInt() method.

Example:

```
void main() {
  double num1 = 10.01;
  int num2 = num1.toInt(); // converting double to int

  print("The value of num1 is $num1. Its type is ${num1.runtimeType}");
  print("The value of num2 is $num2. Its type is ${num2.runtimeType}");
}
```

How to Check Runtime Type

- You can check runtime type in dart with .runtimeType after variable name.

Example:

```
void main() {
  var a = 10;
  print(a.runtimeType);
  print(a is int); // true
}
```

Optionally Type Language

- You may heard of the statically-typed language. It means the data type of variables is known at compile time. Similarly, dynamically-typed language means data types of variables are known at run time. Dart supports dynamic and static types, so it is called optionally-typed language.

- **Statically typed:** A language is statically typed if the data type of variables is known at compile time. Its main advantage is that the compiler can quickly check the issues and detect bugs.

Example:

```
void main() {
  var myVariable = 50; // You can also use int instead of var
  myVariable = "Hello"; // this will give error
  print(myVariable);
}
```

- **Dynamically typed:** A language is dynamically typed if the data type of variables is known at run time.

Example:

```
void main() {
  dynamic myVariable = 50;
  myVariable = "Hello";
  print(myVariable);
}
```

Note:

- Using static type helps you to prevent writing silly mistakes in code. It's a good habit to use static type in dart.

Null Safety

```
void main() {
  int n;
  print(n); // Error
}
```

The previous example will throw an error because in dart, it is not allowed to not initialize a variable to a null value unless you use the question mark (?) next to the type of the variable like the following example:

```
void main() {
  int? n;
  print(n); // null
}
```

Arithmetic Operators

- Arithmetic operators in dart provide arithmetic operations such as add, division, subtraction, and division multiplication operators.
- Arithmetic operators are + , - , * , / , %, ~/ and more else.
- ~/ this is for integer division. In Dart, any simple division with / results in a double value. To get only the integer part, you would need to make some kind of transformation (that is, type cast) in other programming languages; however, here, the integer division operator does this task.
- Note that if the operator is (/), then the **result** variable must be **double.** if we didn't, then the compiler will throw an error.

Example:

```
void main() {
  int x = 10;
  int y = 2;
  int result = x / y;
  print(result); // Error
}
```

Example:

```
void main() {
  int x = 10;
  int y = 2;
  double result = x / y;
  print(result); // 5.0
}
```

Example:

```
void main() {
  int x = 10;
  int y = 2;
  int result = x ~/ y;
  print(result); // 5
}
```

The previous example will return 5 without using **double** keyword.

Example:

```
void main() {
  int x = 10;
  int y = 2;
  int result = x % y;
  print(result); // 0
}
```

Test Operators

Example:

```
void main() {
  int x = 1;
  print(x is int); // true
}
```

Example:

```
void main() {
  int x = 1;
  print(x is! int); // false
}
```

Assignment Operators

=

Example:

```
void main() {
  int x = 10; // Simple Assignment
}
```

??=

- This is used to deal with values that might be null. One is the ??= assignment operator, which assigns a value to a variable only if that variable is currently null.

Example:

```
void main() {
  int? x;
  x ??= 20;
  print(x); // 20
}
```

Example:

```
void main() {
  int? a; // = null
```

```
  a ??= 3;
  print(a); // <-- Prints 3.

  a ??= 5;
  print(a); // <-- Still prints 3.
}
```

Example:

```
void main() {
  print(1 ?? 3); // <-- Prints 1.
  print(null ?? 12); // <-- Prints 12.
}
```

Comment

There are two types of comments in Dart

Example:

```
void main() {
  // This is a single line comment

  /*
  This is
  a multiple
  Line Comment
  */
}
```

if – else if – else statement

- if-else statements are control flow statements. Using these statements, you can control the flow of a dart program. Basically, the if block comes before else. if block checks for a condition. If the condition is true it executes the code written in that block. Else, it executes the code written in the else block.

Example:

```
void main() {
  int redmiPrice = 2900;
  int iphonePrice = 20100;
  int myMoney = 5000;

  if (myMoney >= redmiPrice) {
    int rest = myMoney - redmiPrice;
    print("You can buy the Redmi Phone and you still have $rest");
  } else if (myMoney >= iphonePrice) {
    int rest = myMoney - iphonePrice;
    print("You can buy Iphone 11 and you still have $rest");
  } else {
    print("Sorry, You don't have enough money to buy anything.");
  }
}
```

switch case

- In Dart, switch case statement is simplified form of the Nested if else statement , it helps to avoid long chain of if..else if..else statements. A switch case statement evaluates an expression against multiple cases in order to identify the block of code to be executed.

Example:

```
void main() {
  int day = 5;

  switch (day) {
    case 1:
      {
        print("Today is Saturday.");
      }
      break;

    case 2:
      {
        print("Today is Sunday.");
```

```
      }
      break;

    case 3:
      {
        print("Today is Monday.");
      }
      break;

    case 4:
      {
        print("Today is Tuesday.");
      }
      break;

    case 5:
      {
        print("Today is Wednsday.");
      }
      break;

    case 6:
      {
        print("Today is Thursday.");
      }
      break;

    case 7:
      {
        print("Today is Firday.");
      }
      break;

    default:
      {
        print("Invalid Day.");
      }
      break;
  }
}
```

for loop

- For loop in Dart for loop is a type of definite loop (it can also iterate for infinite times, but in practical cases, we use it when the number of iterations is known). The for loop executes a block of code for a specified number of times.

Example:

```
void main() {
  for (int num = 1; num <= 10; num++) {
    print(num);
  }
}
```

Example:

- In this example shows that we can give a label for a **for loop**.

```
void main() {
  var avengers = ["Iron man", "Hulk", "Capitain America"];

  avengersLoop:
  for (var i = 0; i < avengers.length; i++) {
    switch (avengers[i]) {
      case "Iron man":
        print("Sometimes you gotta run before you can walk.");
        break;
      case "Capitain America":
        print("I can do this all day.");
        break;
      case "Hulk":
        print("Smaaaaaaaaaash!");
        break avengersLoop;
    }
  }
}

/*
```

```
Outpur:

Sometimes you gotta run before you can walk.
Smaaaaaaaaaash!+
 */
```

while loop

- While loop is one of the control flow statements in Dart. While loop is used for evaluating a condition before loop executes.

Example:

```
void main() {
  int num = 1;
  while (num <= 10) {
    print(num);
    num++;
  }
}
```

do while

- Do while is one of the control flow statements in Dart. Do while loop evaluates the condition after loop has executed at least one time. After condition is evaluated for the first time then condition is evaluated for subsequent execution as well. When a do while loop is used, code will be executed at least once.

Example:

```
void main() {
  int num1 = 1;
  int num2 = 1;
```

```
  do {
    print(num1);
    num1++;
  } while (num1 >= 10); // the result = 1

  print("====================\n");

  do {
    print(num2);
    num2++;
  } while (num2 <= 10);
}
```

break

- The break statement in Dart inside any loop gives you a way to break or terminate the execution of the loop containing it, and hence transfers the execution to the next statement following the loop. It is always used with the if-else construct.

Example:

```
void main() {
  for (int i = 1; i <= 10; i++) {
    print(i);
    if (i == 5) break;
  }
}
```

continue

- Dart continue statement is used to skip the execution of subsequent statements inside loop after continue statement, and continue with the next iteration. If continue statement is used in nested loops, only the immediate

loop is continued with. continue statement can be used inside a For loop, While loop and Do-while loop statements.

Example:

```
void main() {
  for (int i = 1; i <= 10; i++) {
    if (i == 5) continue;
    print(i);
  }
}
```

Note:

- In the previous example if we write the line **print(i);** before the condition, then there will be 5 in the result because it is printed before the condition skip it.

Numbers

.isFinite

- This is used to check if the number is finite or not. It returns Boolean value.

Example:

```
void main() {
  int x = 5;
  print(x.isFinite); // true
}
```

.isInfinite

- This is used to check if the number is infinite or not. It returns Boolean value.

Example:

```
void main() {
  int x = 5;
  double y = (10 / 0);
  print(x.isInfinite); // false
  print(y.isInfinite); // true
}
```

.isNegative

- This is used to check if the number is negative or not. It returns Boolean value.

Example:

```
void main() {
  int x = 5;
  print(x.isNegative); // false
}
```

.sign

This is used to return 0 if the number is zero, return 1 if the number is positive and return -1 if the number is negative.

Example:

```
void main() {
  int x = 5;
  int y = -200;
  int z = 0;
  print(x.sign); // 1
  print(y.sign); // -1
  print(z.sign); // 0
}
```

.isEven

This is used to check if the number is even or not. It returns Boolean value.

Example:

```
void main() {
  int x = 7;
  int y = 12;
  print(x.isEven); // false
  print(y.isEven); // true
}
```

.isOdd

This is used to check if the number is odd or not. It returns Boolean value.

Example:

```
void main() {
  int x = 7;
  int y = 12;
  print(x.isOdd); // true
  print(y.isOdd); // false
}
```

abs()

- This is used to return the absolute value of the integer.

Example:

```
void main() {
  int x = -10;
  print(x.abs());
}
```

ceil()

- This is used to return the least integer that's not smaller than this number.

Example:

```
void main() {
  double x = 10.9;
  double y = 10.4;
  print(x.ceil()); // 11
  print(y.ceil()); // 11
}
```

compareTo()

- This returns -1 if the number is less than the other number you enter to the function, 0 if they are equal, and 1 if the number is greater than the number you enter to the function.

Example:

```
void main() {
  int x = 5;
  print(x.compareTo(5)); // 0
  print(x.compareTo(7)); // -1
  print(x.compareTo(2)); // 1
}
```

floor()

- This is used to return the greatest integer no greater than this number, i.e. It rounds the number towards the number before it.

Example:

```
void main() {
  double x = 4.966;
  double y = 4.3;
  print(x.floor()); // 4
  print(y.floor()); // 4
}
```

round()

- This is used to return the integer closest to this number.

Example:

```
void main() {
  double x = 4.7;
  double y = 4.3;
  print(x.round()); // 5
  print(y.round()); // 4
}
```

toInt()

- This is used to convert double to integer.

Example:

```
void main() {
  double x = 4.761365;
  print(x.toInt()); // 4
}
```

toDouble()

- This is used to convert integer to double.

Example:

```
void main() {
  int x = 235;
  print(x.toDouble()); // 235.0
}
```

num.parse()

- This is used to parse a string containing a number literal into a number.

Example:

```
void main() {
  String x = "100";
  print(num.parse(x)); // 100
  print(num.parse(x) is int); // true
}
```

String Methods

Concatenation

- To append two strings in Dart, use concatenation operator +. + accepts two strings as operands and returns concatenated string.
- We can't concatenate a string with an integer.

Example:

```
void main() {
  String fname = "Ahmed";
  String lname = "Ghaly";
  print(fname + " " + lname); // Ahmed Ghaly
}
```

Example:

```
void main() {
  String fname = "Ahmed";
  String lname = "Ghaly";
  int num = 100;
  print(fname + " " + lname + " $num"); // Ahmed Ghaly 100
}
```

Example:

```
void main() {
  String fname = "Ahmed";
  String lname = "Ghaly";
  int num = 100;
  print(fname + " " + lname + " ${num + 5}"); // Ahmed Ghaly 105
}
```

.isEmpty

- This is used to check if the string is empty or not. It returns Boolean value.

Example:

```
void main() {
  String fname = "Ahmed";
  String name = "";
  String space = " ";
  print(fname.isEmpty); // false
  print(name.isEmpty); // true
  print(space.isEmpty); // false
}
```

Example:

```
void main() {
  String fname;
  print(fname.isEmpty); // Error
}
```

The previous example returns an error because it's a null value.

.length

- This is used to return the length of the string.

Example:

```
void main() {
  String name = "Ahmed Ghaly";
  print(name.length); // 11
}
```

toUpperCase()

- This is used to convert all the characters in the string to upper case.

Example:

```
void main() {
  String name = "ahmed ghaly";
  print(name.toUpperCase()); // AHMED GHALY
}
```

toLowerCase()

- This is used to convert all characters in the string to lower case.

Example:

```
void main() {
  String name = "AHMED";
  print(name.toLowerCase()); // ahmed
}
```

trim()

- This is used to return the string without any leading and trailing whitespaces.

Example:

```
void main() {
  String name = "ahmed ";
  print(name.trim()); // ahmed
  print(name.length); // 6
  print(name.trim().length); // 5
}
```

split()

- This takes a parameter (**pattern**), then search the index for that pattern, and when it's found, the function add the String after that pattern to the list and return that list.

Example:

```
void main() {
  String text = "A#h#m#e#ddd#";
  print(text.split('#'));
  // you'll find a space at the end of the result because of # at the
end of the String
}
```

join()

- This does the inverse of **split()** it takes a separator as a parameter to separate between the elements of a specific list.

```
void main() {
  List text = ['A', 'h', 'm', 'e', 'd', 5, 66, true];
  print(text.join('#'));
}
```

compareTo()

- This is used to return 0 if the two strings are equal, -1 if the other string is grater than this string, and 1 if this string is greater than the other string.

Example:

```
void main() {
  String name = "ahmed";
  print(name.compareTo("ghaly")); // -1
  print(name.compareTo("ahmed")); // 0
  print(name.compareTo("ahmedddd")); // -1
  print(name.compareTo("ah")); // 1
}
```

replaceAll()

- This is used to replace a string with another.

Example:

```
void main() {
  String name = "ahmed ghaly";
  print(name.replaceAll("ghaly", "elsayed")); // ahmed elsayed
}
```

List

- Lists in dart are an indexable collection of objects. These can contain objects of the same type as well as objects of different data type.
- It is also possible that we can create a list of fixed length or a list that is growable in nature.
- Lists in dart are 0 index-based.
- We can make a list as a object of another list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar"];
  print(names[1]); // ali
}
```

Example:

```
void main() {
  List names = [
    "ahmed",
    "ali",
    [1, 2, 3],
    "omar"
  ];
  print(names[2]); // [1, 2, 3]
  print(names[2][0]); // 1
}
```

We can change the value of an object as below:

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print("Names List Before Change: $names");
  names[1] = "Mohamed";
  print("Names List After Change: $names");
}
```

We can use three dots operator (…) to get the content of a list out.

Example:

```
void main() {
  List<int> list1 = [1, 2, 3];
  List<int> list2 = [
    10,
    ...list1,
    ...[20, 30]
```

```
  ];
  print(list2); // [10, 1, 2, 3, 20, 30]
}
```

We can use a condition to check if we'll get the content of a list'

Example:

```
void main() {
  List<int> list1 = [1, 2, 3];
  List<int> list2 = [
    10,
    if (list1.length > 5) ...list1,
    ...[20, 30]
  ];
  print(list2); // [10, 20, 30]
}
```

The next example se use (...?) to check if the list isn't **null,** so if it's not null, then get the content of it and if it's null, then don't get it.

Example:

```
void main() {
  List<int>? list1 = null;
  List<int> list2 = [
    10,
    ...?list1,
    ...[20, 30]
  ];
  print(list2); // [10, 20, 30]
}
```

add()

- This is used to add an element to the end of the list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  names.add("Fairooz");
  print(names);
}
```

.length

- This is used to return the length of the list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.length); // 4
}
```

.forEach()

- **forEach** () method is used to iterate over the elements of a list and execute a block of code for each element.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];

  names.forEach((name) {
    print(name);
  });
}
```

We can do the same using loops but it's better to use .forEach() with lists:

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];

  for (int i = 0; i < names.length; i++) {
    print(names[i]);
  }
}
```

.first

- This is used to return the first element of the list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.first); // ahmed
}
```

.last

- This is used to get the last element of the list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.last); // kamal
}
```

.isEmpty

- This is used to check if the list is empty or not. It returns Boolean values.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.isEmpty); // false
}
```

.isNotEmpty

- This is used to check if the list is empty or not. It returns Boolean values.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.isNotEmpty); // true}
```

.reversed

- This is used to reverse a list.
- It doesn't return a list datatype, so we can use **toList()** function to convert it to a list.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.reversed.toList());
}
```

.single

- This is used to check if the list has one element and returns that element.
- If the list has more than one element, it throws an error.

Example:

```
void main() {
  List names = ["ahmed", "ali", "omar", "kamal"];
  print(names.single); // Error
}
```

Example:

```
void main() {
  List names = ["ahmed"];
  print(names.single); // ahmed
}
```

addAll()

- This is used to add all the objects of a list to the end of another list.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.addAll(["Mohamed", 500, 30]);
  print(names);
}
```

insert()

- This is used to insert an element at position (index) in a list.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.insert(0, "mohamed");
  print(names);
}
```

insertAll()

- This is used to insert all objects of a list at position (index) in another list.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.insertAll(1, ["Kamal", "Kareem"]);
  print(names); // ["ahmed", "Kamal", "Kareem", "ali", 12, 3]
}
```

replaceRange()

- replaces the value of the element (s) within the provided range.
- It takes three arguments, the first is the start index, the second is the end index, and the third is the objects we want to add.
- Note that the end index isn't included, i.e., we stop at the end index but it is not in with us.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.replaceRange(1, 3, ["Mohamed", 500]);
  print(names); // [ahmed, Mohamed, 500, 3]
}
```

replaceFirst()

- This takes two parameters, the first is the **String** you want to replace, and the second is the **String** you want to replace with. It search for that String in the string and if it's duplicated, then it replaces the first one with the string you want.

Example:

```
void main() {
  String name = "ahmed Ghaly";
  String name2 = "New String New String New String";

  print(name.replaceFirst('a', "Mohamed ".toUpperCase()));
  /*

  The result is => MOHAMED hmed Ghaly
  there are two "a" in the string but it only replaces the first one

   */

  print("\n================================================\n");

  print(name2.replaceFirst("New", "Old"));
  /*

  The result is => Old String New String
  There are three "New" in the String but it only replaces the first one

   */
}
```

remove()

- This is used to remove an element from the list.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.remove("ali");
  print(names);
}
```

removeAt()

- This is used to remove an element at a specific index.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.removeAt(1);
  print(names);
}
```

removeRange()

- This is used to remove a range of elements from a list.
- It takes two arguments. The first is the start index, and the second is the end index.
- Note that the end index isn't included. i.e., the removal is less than the end index.

Example:

```
void main() {
  List names = ["ahmed", "ali", 12, 3];
  names.removeRange(1, 3);
  print(names); // [ahmed, 3]
}
```

reduce()

- Reduces a collection to a single value by iteratively combining elements of the collection using the provided function.
- The iterable must have at least one element. If it has only one element, that element is returned.

Example:

```
void main() {
  List<int> myList = [1, 2, 3];
  print(myList.reduce((value, element) => value + element)); // output:
6
}
```

2D List

Two-Dimensional List.

Example:

```
void main() {
  var array = [
    [1, 2, 3],
    [4, 5, 6],
  ];

  // print the whole 2D list
  print(array);

  print("\n=======================================\n");

  // print every element of the list
  for (int row = 0; row <= 1; row++) {
    // loop on rows

    for (int column = 0; column < 3; column++) {
      // loop for columns
      print(array[row][column]);
    }
  }

  print("\n=======================================\n");

  // print rows of the array

  for (var i in array) {
    print(i);
  }}
```

Map

- 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([]).
- A Map is a dynamic collection. In other words, Maps can grow and shrink at runtime.
- Maps can be declared in two ways:
    - Using Map literals.
    - Using a Map constructor
- A map value can be any object including NULL.

Declaring a Map using Map Literals

```
void main() {
  var details = {'Username': 'Ahmed', 'Pass': 'ahmed123'};
  print(details); // {Username: Ahmed, Pass: ahmed123}
}
```

Adding values to Map Literals at Runtime

```
void main() {
  var details = {'Username': 'Ahmed', 'Pass': 'ahmed123'};
  details['id'] = '123333';
  print(details); // {Username: Ahmed, Pass: ahmed123, id: 123333}
}
```

Map Constructor

```
void main() {
  var details = new Map();
  details['Username'] = 'admin11';
  details['Pass'] = 'ahmed@123';
  print(details); // {Username: admin11, Pass: ahmed@123}
}
```

To Get the Value of a Key

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info['Name']); // Ahmed Ghaly
}
```

.keys

- This is used to return the keys of the Map.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info.keys); // (Name, Birthday, id)
}
```

containsKey()

- To check if this map contains a specific key or not.

Example:

```
void main() {
  var myMap = {
    1: 'a',
    2: 'b',
    3: 'c',
  };

  print(myMap.containsKey(1));
}
```

.values

- This is used to return the values of the Map.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info.values); // (Ahmed Galy, 1/5/2002, 12333)
}
```

containsValue()

- To check if this map contains a specific value or not.

Example:

```
void main() {
  var myMap = {
    1: 'a',
    2: 'b',
    3: 'c',
  };
```

```
  print(myMap.containsValue('c'));
}
```

.entries

- Returns entries of the map

Example:

```
void main() {
  var myMap = {
    1: 'a',
    2: 'b',
    3: 'c',
  };

  print(myMap.entries);
}
```

forEach()

- This is used to apply an action on each key/value of a Map.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  info.forEach((key, value) {
    print("$key : $value");
  });
}
```

.isEmpty

- This is used to check if a Map is empty or not. It returns a Boolean value.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info.isEmpty); // false
}
```

.isNotEmpty

- This is used to check if a Map is empty or not. It returns a Boolean value.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info.isNotEmpty); // true
}
```

.length

- This is used to return the length of the Map.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  print(info.length); // 3 ==> the number of key/value in a map
}
```

remove()

- This is used to remove a key and its associated value from a map.
- Returns the value that is associated with the key before it was removed.
- Returns **null** if the key is not in the map.

- Note that some maps allow null as a value, so a returned null value doesn't always mean that the key was absent.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  (info.remove('Birthday'));
  print(info); // {Name: Ahmed Galy, id: 12333}
}
```

clear()

- removes all entries from a map.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  info.clear();
  print(info); // {}
}
```

addAll()

- This is used to add all key/value pairs to this map.
- If a key is already in the map, its value is overwritten.
- The operation is equivalent to **this[key] = value;** for each key and associated value in other.

Example:

```
void main() {
  Map info = {'Name': 'Ahmed Galy', 'Birthday': '1/5/2002', 'id': '12333'};
  info.addAll({"age": 20, "tele": 012333});
```

```
  print(info); // {Name: Ahmed Galy, Birthday: 1/5/2002, id: 12333, age: 20,
tele: 12333}
}
```

Note:

- We can make a list inside a map and vice versa, but as a **value.**

Example:

```
void main() {
  Map map1 = {
    1: "ahmed",
    2: [
      "ali",
      "kamal",
      {'age': 20, 'id': 5552}
    ],
    3: "End"
  };
  print(map1);
}
```

Dynamic Variables – var

- In Dart, the var keyword is used to declare a **variable**. The Dart compiler automatically knows the type of data based on the assigned to the variable because Dart is an infer type language.
- If you declare a variable **var x = 2;** and then tried to change it to a string in the next line, this will throw an error because the compiler automatically knows the type of data based on the assigned to the variable.

Example:

```
void main() {
  var x = "ahmed";
```

```
  print(x); // ahmed
  print(x.runtimeType); // String
}
```

Example:

```
void main() {
  var x = "ahmed";
  x = "Ahmed Ghaly";
  print(x); // Ahmed Ghaly
                                                                    }
```

Example:

```
void main() {
  var x = "ahmed";
  x = 2; // Error
  print(x);
}
```

final

- The **final** variable can only be set once.
- If we try to change it, then it will throw an error.

Example:

```
void main() {
  final String name = "Ahmed Ghaly";
  name = "ali"; // Error
  print(name);
}
```

const

- const is used to create compile-time constants. We can declare a value to compile-time constant such as number, string literal, a const variable, etc.

- const keyword is also used to create a constant value that cannot be changed after its creation.
- If we try to change it, then it will throw an error.

Example:

```
void main() {
  const String name = "Ahmed Ghaly";
  name = "ali"; // Error
  print(name);
}
```

What is the difference between const and final?

- The difference has to do with how memory is allocated. Memory is allocated for a final variable at runtime, and for a const variable at compile-time.
- We can give a value for a **final** variable either in **run time** or **compile time**, while **const** variable must be initialized in **compile time**.
- Variable from classes can be final but not constant and if you want a constant at class level make it static const.
- If you have a final variable and a const variable without being initialized in a class, and you try to initialize them using constructor, it works with **final** but doesn't with **const**.

Example:

```
void main() {
  final List myList = [1, 2, 3];
  const List myList2 = [2, 2, 2];

  myList.add(5);

  myList2.add(55); // Run Time Error
```

```
  myList = [5, 5, 5]; // can't be initialized again (constatnt
initialization)
}
```

Note

- From the previous example, we notice that **final** makes a constant initialization for the list, but not its elements, so we can modify the list's elements.
- **const** makes a constant initialization for the list and its elements too, so we can't modify anything.

Set

- set is an unordered collection of data, that means the data stored in sets are unstructured.
- The measure difference between list & set is, in sets **duplicate values can't be stored**, data store in sets are always unique.
- If you have a duplicated element in a set, and you tried to print it, the result will have only one value of them.
- If you tried to add an already existed element in a set, then tried to print it, the result will have only one value.

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem", "ali"};
  print(newSet); // {ahmed, ali, kareem}
}
```

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem"};
  newSet.add("ali");
  print(newSet); // {ahmed, ali, kareem}
}
```

add()

- This is used to add a new element to a set.

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem"};
  newSet.add("Kamal");
  print(newSet); // {ahmed, ali, kareem, Kamal}
}
```

addAll()

- This is used to add all elements to a set.

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem"};
  newSet.addAll(["gamal", "elsayed"]);
  print(newSet); // {ahmed, ali, kareem, gamal, elsayed}
}
```

Note:

- We can use ({}) instead of ([]) to add elements using addAll() function.

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem"};
  newSet.addAll({"gamal", "elsayed"});
  print(newSet); // {ahmed, ali, kareem, gamal, elsayed}
}
```

.signle

- This is used to check if the list has one element and returns that element.
- If the list has more than one element, it throws an error.

Example:

```
void main() {
  Set newSet = {"ahmed"};
  print(newSet.single); // ahmed}
```

Note:

- Most of the functions like **.isEmpty**, **.isNotEmpty, remove(), removeAt(), removeAll()**, and more in List and Map are in Set too.

Convert between List, Map, Set

toList()

- This is used to create a list containing the elements of this iterable.

Example:

```
void main() {
  Set newSet = {"ahmed", "ali", "kareem"};
  print(newSet.toList()); // [ahmed, ali, kareem]}
```

toSet()

- This is used to create a set containing the elements of this iterable.

Example:

```
void main() {
  List newList = [1, 2, 3, 4, 5];
  print(newList.toSet()); // {1, 2, 3, 4, 5}
}
```

Convert a Map to a List or a Set

- A map contains key/value pairs and list or set only cares about values.
- To convert a map to a list or to a set do as below:

Example:

```
void main() {
  List values = [];
  Map newMap = {'name': 'ahmed', 'age': 20};

  newMap.forEach((key, value) {
    values.add(value);
  });

  print(values); // [ahmed, 20]
}
```

- In Case we want to make a list of the keys we just change **value** to **key.**
- If we want to convert a map to a set, we do the same with changing the first line to **Set values = {};**

Function

- Functions in Dart behave like **first-class objects** meaning they can be stored in a variable, passed as an argument or returned like a normal return value of a function. Functions in Dart behave much like function in JavaScript.
- Function is a set of statements that take inputs, do some specific computation, and produces output.
- Functions are created when certain statements are repeatedly occurring in the program and a function is created to replace them.
- Functions make it easy to divide the complex program into smaller sub-groups and increase the code reusability of the program.

Example:

```
void main() {
  String name = "Ahmed";

  void printName() {
    print(name);
  }

  printName(); // Ahmed
}
```

Example:

```
void main() {
  int sum(int num1, int num2) {
    int result = num1 + num2;
    return result;
  }

  print(sum(5, 10));
}
```

Note:

- Return type **void** doesn't return a value, so it can't be stored in a variable.
- If a function returns integer, float, etc. Then the return type of the function must be the same as the type of what it returns.
- Functions can be defined outside the main.

Example:

```
void main() {
  int sum(int num1, int num2) {
    int result = num1 + num2;
    return result;
  }

  print(sum(50, 150));
}
```

Example:

```
void main() {
  int sum(int num1, int num2) {
    int result = num1 + num2;
    return result;
  }

  int res = sum(50, 100);
  print(res);}
```

Function Scope

- We can declare a variable outside the **main()** function.
- We can declare a function outside the **main()** function.
- If a variable is declared outside the main(), then it can be accessed anywhere in the program, either inside the main() or outside it.

- If a variable is declared inside the main(), then it can only be accessed anywhere inside the main().
- If a variable is declared inside a function, then it can only be accessed inside this function.

Example:

```
void main() {
  int x = 5;

  int sumX(int num) {
    int res = x + num;
    return res;
  }

  print(sumX(10)); // 15
}
```

Example:

```
void main() {


  int sumX() {
    int x = 10;
    int res = x + 20;
    return res;
  }

  print(x); // Error: Undefined name 'x'}
```

Example:

```
int x = 10;

void main() {
  int sumX() {
    int res = x + 20;
    return res;
  }
```

```
  print(sumX()); // 30
}
```

import

- In Dart, we can also import a local Dart file using a relative path. Hence import ‘./src/classes.dart’; will import classes.dart from src.
- When you import a Dart file, whatever components defined inside that file will be imported in the current file, like variables, classes, functions, etc.

Example:

- I created a file called sum.dart, this file has a function called sumNums() returns the sum of two integers.
- The sum.dart file contains the code below:

```
int sumNums(int n1, int n2) {
  int res = n1 + n2;
  return res;
}
```

- The file I work on contains the code below:

```
import 'sum.dart';

void main() {
  print(sumNums(10, 20)); // 30
}
```

Note:

- If the file, we want to import is inside a folder next to our project then we write the folder name then slash (/) then the file name.

- If the file is outside the project folder and we want to get one step back we use (../).

Example:

- if the file sum.dart inside a folder called **libraries**, so to call it we do as below:

```
import 'libraries/sum.dart';

void main() {
  print(sumNums(10, 20)); // 30
}
```

Example:

- if the file sum.dart inside a folder called **libraries** outside the project folder.

```
import '../libraries/sum.dart';

void main() {
  print(sumNums(10, 20)); // 30
}
```

Examples

Example – calculate the average of numbers inside a list.

First solution

```
void main() {
  List<int> nums = [10, 20, 30, 40, 50, 60];
  int sum = 0;
  int count = 0;

  for (int i = 0; i < nums.length; i++) {
    sum += nums[i];
    count++;
  }
  double avg = sum / count;
```

```
  print(avg);
}
```

Second solution

```
void main() {
  List<int> nums = [10, 20, 30, 40, 50, 60];
  int sum = 0;

  for (int i = 0; i < nums.length; i++) {
    sum += nums[i];

    if (i + 1 == nums.length) { // i + 1 ==>  5 + 1
      print(sum / nums.length);
    }
  }
```

i + I ➔5 + 1 = 6 nums.length = 6

Third solution

```
int sum = 0;

void main() {
  List<int> nums = [10, 20, 30, 40, 50, 60];
  avg(nums);
}

avg(List<int> list1) {
  for (int i = 0; i < list1.length; i++) {
    sum += list1[i];

    if (i + 1 == list1.length) {
      print(sum / list1.length);
    }
  }
}
```

Example – Showing the elements of a list

First solution

```
void main() {
  List users = [
    {'name': 'Ahmed', 'age': 20, 'tele': 0125423, 'Pass': 124433},
    {'name': 'Ali', 'age': 18, 'tele': 65651, 'Pass': 556213}
  ];

  users.forEach((element) {
    print("Name: ${element['name']}");
    print("Age: ${element['age']}");
    print("Tele: ${element['tele']}");
    print("Password: ${element['Pass']}");
  });
}
```

Second solution

```
void main() {
  List users = [
    {'name': 'Ahmed', 'age': 20, 'tele': 0125423, 'Pass': 124433},
    {'name': 'Ali', 'age': 18, 'tele': 65651, 'Pass': 556213}
  ];

  for (int i = 0; i < users.length; i++) {
    print(
        "Name: ${users[i]['name']} || Age: ${users[i]['age']} || Tele:
${users[i]['tele']} || Password: ${users[i]['Pass']}");}}
```

Runes

- In Dart language, strings are simply a sequence of UTF-16 (16-bit Unicode Transformation Format) code units. The Unicode format maps a unique numeric character to represent every digit, letter, or symbol.
- A rune can be defined as an integer used to represent any Unicode code point. As a Dart string is a simple sequence of UTF-16 code units, 32-bit

Unicode values in a string are represented using a special syntax. The String class in the dart:core library gives ways to access runes. Runes can be accessed in the following ways :

- Using .codeUnits property
- Using .runes property
- Using .codeUnitAt() function

Using .codeUnits property

- This property returns an unchangeable list of the 16-bit UTF-16 code units of the given string.

Example:

```
void main() {
  String x = "Ahmed";
  print(x.codeUnits); // [65, 104, 109, 101, 100]
}
```

Using .runes property

- This property returns an iterable of Unicode code-points of the specified string.

Example:

```
void main() {
  String x = "Ahmed";
  print(x.runes); // (65, 104, 109, 101, 100)
}
```

```
void main() {
  String x = "Ahmed";
  x.runes.forEach((element) {
    print(element);
```

```
  });}
```

Example:

```
void main() {
  String x = "Ahmed";
  x.runes.forEach((int s) {
    var ch = new String.fromCharCode(s);
    print(ch);
  });
}
```

Note:

- We can return the character using its Unit-code by **fromCharCode().**

Using .codeUnitAt function

- It is used to return the UTF-16 code unit at the specified index of this string.

Example:

```
void main() {
  String x = "Ahmed";
  print(x.codeUnitAt(2)); // 109
}
```

Note:

- We can use the Uni-code of emojis to print them.

Example:

```
void main() {
  String emo = "\u{1F923}";
  print(emo);
}
```

Assert

- assert statements are useful for debugging a dart project. It is used mainly in development mode.
- assert takes two arguments, the first is the condition and it's required, and the second is the error message and it's optional.
- assert takes a condition and checks if it is true or false. If it is true, the program runs normally and if it is false, it stops the execution and throws one error called AssertionError.
- Flutter enables it by default for debug mode.

Example:

```
void main() {
  int x = 10;
  assert(x == null, "X must not equal NULL");
  print(x);
}
```

max() & min()

- They are functions of **dart:math** library, so to use them we must import this library.
- max() is used to return the maximum number of two numbers.
- min() is used to return the minimum number of two numbers.

Example:

```
import 'dart:math';

void main() {
  print(max(10, 250)); // 250
```

```
  print(min(-100, 3)); // -100
}
```

Note:

- We can make functions have the same functionality of max() and min() functions, but it's better to use max(), min() because this saves your time.

Defining functions like max() & min()

```
void main() {
  print(larger(100, 530)); // 530
  print(lesser(6, 50)); // 6
}

int larger(int n1, int n2) {
  if (n1 > n2)
    return n1;
  else
    return n2;
}

int lesser(int n1, int n2) {
  if (n1 < n2)
    return n1;
  else
    return n2;
}
```

Datatype (List, Set, Map)

- It's better to define the datatype of the variable instead of making it **var** because this saves time for compiling your program and make the performance better.
- We can define the datatype of the elements of a list using (**<>**), and this **dynamic** way is the best practice to get better performance.

Example:

```
void main() {
  List<String> names = ["ahmed", 20, "ali"]; // Error
  print(names);
}
```

- The previous example throws an error because its datatype is **String** and there's an integer element inside the list, so to solve this problem, you must remove this integer.

Example:

```
void main() {
  List<Map<String, String>> users = [
    {'fname': "ahmed", 'lname': 'ghaly'},
    {'fname': 'ali', 'lname': 'mohamed'}
  ];
}
```

- In the previous example we defined that the elements of the list must be of **Map** type, and we also defined that the key and the value must be of **String** type.
- If we enter elements with different types, this throws an error.

Example:

```
void main() {
  Map<String, int> info = {'age': 20, 'id': 1332};
  print(info);
}
```

Example:

```
void main() {
  Set<String> names = {"Kamal", "Alaa", "Salah"};
  print(names);}
```

sublist()

- This is used to return a new list between start and end from this list.
- It takes two arguments, the first is the start index and it's required, and the second is the end index and it's optional.

Example:

```
void main() {
  List<String> names = ["Ahmed", "Amin", "Mohamed", "Alaa", "Ali"];
  List subList = names.sublist(1);
  print(subList); // [Amin, Mohamed, Alaa, Ali]
}
```

shuffle()

- This is used to shuffle the elements of this list randomly.

Example:

```
void main() {
  List<String> names = ["Ahmed", "Amin", "Mohamed", "Alaa", "Ali"];
  names.shuffle();
  print(names);
}
```

asMap()

- This is used to return an unmodifiable (Map) view of this list.

Example:

```
void main() {
  List<String> names = ["Ahmed", "Amin", "Mohamed", "Alaa", "Ali"];
  Map namesMap = names.asMap();
  print(namesMap);
}
```

whereType()

- This is used to return an iterable with all elements that have the defined type.

Example:

```
void main() {
  List names = ["Ahmed", 10, "Amin", "Mohamed", 5, 33, "Alaa", "Ali", 97];
  var newNames = names.whereType<String>().toList();
  print(newNames); // [Ahmed, Amin, Mohamed, Alaa, Ali]
}
```

firstWhere()

- This is used to return the first element that satisfies the given predict.
- It iterates through the elements and returns the first to satisfy the test.
- ==> is a symbol instead of writing **return.**

Example:

```
void main() {
  List<int> names = [3, 1, 7, 10, 50];
  var newNames = names.firstWhere((element) => element > 5);
  print(newNames); // 7
}
```

Note:

- The result is **7,** although there are other numbers that are greater than 5, but 7 is the first element to satisfy the test.

Another solution

```
void main() {
  List<int> names = [3, 1, 7, 10, 50];
  var newNames = names.firstWhere((element) {
```

```
    return element > 5;
  });
  print(newNames);
}
```

Example – the first element that has more than 4 letters

```
void main() {
  List<String> names = ["Ali", "Fady", "Ahmed", "Omar"];
  var newNames = names.firstWhere((element) => element.length > 4);
  print(newNames); // Ahmed
}
```

where()

- This is used to return all element that satisfies the given predict.
- The matching elements have the same order in the returned iterable as they have in iterator.

Example:

```
void main() {
  List<String> names = ["Ali", "Fady", "Ahmed", "Omar", "Kamel"];
  var newNames = names.where((element) => element.length > 4).toList();
  print(newNames); // [Ahmed, Kamel]
}
```

retainWhere()

- Removes all the objects that don't satisfy the condition.

Example:

```
void main() {
  List<int> list1 = [11, 2, 3, 4, 8, 20, 10];
  list1.retainWhere((element) => element % 2 == 0);
  print(list1); // [2, 4, 8, 20, 10]
}
```

singleWhere()

- Returns the single element that satisfies the test.
- If there are many elements satisfy the condition, the compiler throws an error.

Example:

```
void main() {
  List<int> list1 = [11, 2, 3];
  int x = list1.singleWhere((element) => element % 2 == 0);
  print(x); // 2
}
```

any()

- Checks whether any element of this iterable satisfies test. Checks every element in iteration order and returns true if any of them make test return true, otherwise returns false.

Example:

```
void main() {
  List<String> names = ["Ali", "Fady", "Ahmed", "Omar"];
  var newNames = names.any((element) => element.length > 6);
  print(newNames); // false
}
```

every()

- Checks whether every element of this iterable satisfies test.
- Checks every element in iteration order and returns false if any of them make test return false, otherwise returns true.

Example:

```
void main() {
  List<String> names = ["Ali", "Fady", "Ahmed", "Omar"];
  var newNames = names.every((element) => element.length > 2);
  print(newNames); // true
}
```

take()

- Returns a lazy iterable of the count first elements of this iterable.
- The returned Iterable may contain fewer than count elements if this contains fewer than count elements.
- The elements can be computed by stepping through iterator until count elements have been seen.
- The count must not be negative.

Example:

```
void main() {
  List names = ["Ali", "Fady", 55, 2, "Ahmed", "Omar", 20, "Kamel"];
  var newNames = names.take(5).toList();
  print(newNames); // [Ali, Fady, 55, 2, Ahmed]
}
```

indexWhere()

- This is used to return the first index in the list that satisfies the provided test.
- It searches the list from the start index to the end index.

Example:

```
void main() {
  List<String> names = ["Ali", "Fady", "Ahmed", "Omar", "Kamel"];
  var newNames = names.indexWhere((element) => element.length > 4);
```

```
  print(newNames); // 2 ==> 'Ahmed'
}
```

startsWith()

- This is used to check if this string starts with a match of pattern.

Example:

```
void main() {
  String name = "Ahmed";
  print(name.startsWith("Ah")); // true
}
```

endsWith()

- This is used to check if this string ends with a match of pattern.

Example:

```
void main() {
  String name = "Ahmed";
  print(name.endsWith("d")); // true
}
```

contains()

- This is used to check if this string contains a match of pattern.

Example:

```
void main() {
  String name = "Ahmed";
  print(name.contains("med")); // true
}
```

indexOf()

- This is used to return the position of the first match of pattern in this string.
- Returns -1 if no match is found.
- It takes two parameters, the first is the pattern and it's required, the second is the start index and it's optional.

Example:

```
void main() {
  String name = "Ahmed";
  print(name.indexOf("m")); // 2
  print(name.indexOf("m", 3)); // -1
}
```

Examples

Example – return the names that starts with letter 'm'.

First solution

```
void main() {
  List<String> names = ["ali", "ahmed", "kamel", "mohamed", "mido", "mahmoud"];
  List<String> namesStartsM = names.where((element) => element[0] ==
'm').toList();
  print(namesStartsM);
}
```

Second solution

```
void main() {
  List<String> names = ["ali", "ahmed", "kamel", "mohamed", "mido", "mahmoud"];
  List<String> namesStartsM = names.where((element) =>
element.startsWith('m')).toList();
  print(namesStartsM);
}
```

Example – return the elements that contain the letter 'm'.

```
void main() {
  List<String> names = ["ali", "ahmed", "kamel", "mohamed", "mido", "mahmoud"];
  List<String> namesStartsM =
      names.where((element) => element.contains('m')).toList();
  print(namesStartsM);
}
```

Iterable Vs. Iterator

What's an iterable?

- An Iterable is a collection of elements that can be accessed sequentially.
- In Dart, an Iterable is an abstract class, meaning that you can't instantiate it directly. However, you can create a new Iterable by creating a new List or Set.
- Both List and Set are Iterable, so they have the same methods and properties as the Iterable class.
- A Map uses a different data structure internally, depending on its implementation. For example, HashMap uses a hash table in which the elements (also called values) are obtained using a key. Elements of a Map can also be read as Iterable objects by using the map's entries or values property.

This example shows a List of int, which is also an Iterable of int:

```
void main() {
  Iterable<int> iterable = [1, 2, 3];
  print(iterable.runtimeType); // List<int>
}
```

- The difference with a List is that with the Iterable, you can't guarantee that reading elements by index will be efficient. Iterable, as opposed to List, doesn't have the [] operator.

For example, consider the following code, which is invalid:

```
void main() {
  Iterable<int> iterable = [1, 2, 3];
  int value = iterable[1]; // Error
}
```

- If you read elements with [], the compiler tells you that the operator '[]' isn't defined for the class Iterable, which means that you can't use [index] in this case.
- You can instead read elements with **elementAt()**, which steps through the elements of the iterable until it reaches that position.

```
void main() {
  Iterable<int> iterable = [1, 2, 3];
  int value = iterable.elementAt(1);
}
```

What's an iterator?

- An iterator is an object used to iterate over iterable using MoveNext, return one element at a time.
- You can generate iterable from iterator method.
- Loop already calls iterator.
- Gives StopIteration if there's no move next element.

Example:

```
void main() {
  List names = ["ahmed", "omar", "ali"];
  Iterator iter = names.iterator;
  while (iter.moveNext()) {
    print(iter.current);
  }
}
```

map()

- returns a new iterable with elements that are created by calling toElement on each element of this Iterable in iteration order.

Example:

```
void main() {
  List<String> names = ["ahmed", "omar", "ali"];
  Iterable lengthElement = names.map((e) => e.length);
  print(lengthElement); // (5, 4, 3)
  Iterable printElement = names.map((e) => e);
  print(printElement); // (ahmed, omar, ali)
}
```

Example:

```
void main() {
  List<String> names = ["ahmed", "omar", "ali"];
  Iterable lengthElement = names.map((e) {
    if (e == "ahmed") {
      return "Yes \"ahmed\" exists";
    }
  });
  print(lengthElement); // (Yes "ahmed" exists, null, null)
}
```

Example:

```
void main() {
  List<String> names = ["ahmed", "omar", "ali"];
  Iterable lengthElement = names.map((e) {
    if (e == "ahmed") {
      return "Yes \"ahmed\" exists";
    }
    return "No";
  });
  print(lengthElement); // (Yes "ahmed" exists, No, No)
}
```

Example:

```
void main() {
  Map info = {'name': "ahmed", 'age': 20};
  Iterable infoKey = info.entries.map((e) => e.key);
  print(infoKey); // (name, age)
}
```

Note:

- If you try to assign an integer variable to a division operation, this throws an error because the result of the division operation is double, so you need to make the datatype of the variable double.
- You can solve this problem and let the datatype int as it's, but you need to use (~) before the (/).

Example:

```
void main() {
  int res1 = 10 / 2; // Error
  int res2 = 10 ~/ 2;
  double res3 = 10 / 2;}
```

try & catch

- Exceptions are errors that may occur in the program. If you don't catch the exceptions and handle them properly, the program will crash.
- Dart try-catch is used to execute a block of code that could throw an exception and handle the exception without letting the program terminate. If an exception, thrown by any of the code, is not handled via catch block, then the program could terminate because of the exception.
- For example, the following program defines a string variable message and attempts to access the character at the index 5:

```
void main() {
  String message = "Hello";
  print("The character at the position 5 is ${message[5]}.");
  print('Bye!');
}
```

- The program crashed and issued the following error:

```
/*

Unhandled exception:
RangeError (index): Invalid value: Not in inclusive range 0..4: 5

*/
```

- The error message shows that the RangeError exception was unhandled.
- To prevent the program from crashing, you need to handle the exception. To do that, you use the try catch statement.
- In the try-catch statement, you place the code that may cause an exception in the try block. If an exception occurs, the program jumps to the catch block immediately and skips the remaining code in the try block.

- In the catch block, you place the code that handles the exception. In practice, you may want to recover from the exception as much as possible such as showing a user-friendly message.
- Also, you can access the exception object (e) in the catch block. All exception classes are the subclasses of the Exception class.
- For example, the following program uses the try-catch statement to catch the RangeError exception:

```
void main() {
  String message = "Hello";

  try {
    print("The character at the position 5 is ${message[5]}.");
  } catch (e) {
    print("Error: $e");
  }

  print('Bye!');
}

/*
Output:

Error: RangeError (index): Invalid value: Not in inclusive range 0..4: 5
Bye!
*/
```

- In this example, the message[5] causes the RangeError exception. However, the program doesn't crash. and runs to the end.

Note:

- If you don't want to pass a variable to **catch() {}**, then you can pass underscore (_) as in the next example:

```dart
void main() {
  List<int> list1 = [11, 2, 3, 4, 6, 8];
  try {
    int x = list1.singleWhere((element) => element % 2 == 0);
  } catch (_) {
    throw Exception("Error: There's more than a Single Element");
  }
}
```

- We can use **try** with **on** not **catch**, but this when we know the type of exception.

Example:

```dart
import 'dart:io';

void main() {
  myLoop:
  for (int i = 0; i < 10; i--) {
    // Try Body
    try {
      print("Enter your birth year:");
      var birthYear = int.parse(stdin.readLineSync()!);

      var age = DateTime.now().year - birthYear;

      print("You are $age years old");
      // Break The Loop
      break myLoop;
    }
    // In Case There's An Error
    on FormatException {
      print("\t\twrong input".toUpperCase());
      print("================ Try Again ================\n");
    }
  }
}
```

Code explanation

- In the previous example, the exception type is **FormatException**, so if the type of the error wasn't as that type, it wouldn't be handled.
- We can use **on** to print something to alert the user that there's an error, for example, we can print **Wrong input**, but we can also print that and try our program again as we did in the previous example using **on** and for loop.
- I used an infinite loop to allow me to print the error message for infinite number of times, and if **try** body is executed, then the program is done.
- Note that we can name our loop as you can see.

shorthand if

Example:

```
void main() {
  int x = 3;
  x > 1 ? print("X > 1") : print("X < 1"); // X > 1
}
```

Example:

```
void main() {
  int x = 3;
  ((x > 1) && (x == 3)) ? print("X > 1 & X == 3") : print("X < 1 && X != 3");
}
```

User Input

- Instead of writing hard-coded values, you can give input to the computer. It will make your program more dynamic. You must import package import 'dart:io'; for user input.

Note:

- You won't be able to take input from users using dartpad. You need to run a program from your computer.

String User Input

- They are used for storing textual user input. If you want to keep values like somebody's name, address, description, etc., you can take string input from the user.

Example:

```
import 'dart:io';

void main() {
  print("Enter name:");
  String? name = stdin.readLineSync();
  print("The entered name is ${name}");
}
```

Integer User Input

- You can take integer input to get a numeric value from the user without the decimal point. E.g. 10, 100, -800 etc.

Example:

```
import 'dart:io';

void main() {
  print("Enter number:");
  int? number = int.parse(stdin.readLineSync()!);
  print("The entered number is ${number}");
}
```

To Get Your Age

We use **DateTime()** library.

Example:

```
import 'dart:io';

void main() {
  print("Enter Your Birth Year: ");
  var birthYear = stdin.readLineSync();
  var age = DateTime.now().year - int.parse(birthYear!);
  print("You Are $age Years Old");
}
```

Floating Point User Input

- You can use float input if you want to get a numeric value from the user with the decimal point. E.g. 10.5, 100.5, -800.9 etc.

```
import 'dart:io';

void main() {
  print("Enter a floating number:");
  double number = double.parse(stdin.readLineSync()!);
  print("The entered num is $number");
}
```

Platform class

- It's a class in 'dart:io' library.
- Information about the environment in which the current program is running.
- Platform provides information such as the operating system, the hostname of the computer, the value of environment variables, the path to the running program, and other global properties of the program being run.

Some uses:

```dart
import 'dart:io';

void main() {
  print(Platform.executable);
  print(Platform.numberOfProcessors);
  print(Platform.operatingSystem);
  print(Platform.operatingSystemVersion);
  print(Platform.script);
  print(Platform.version);
  print("================================================");
  // THE MOST IMPORTANT ONES
  print(Platform.isAndroid);
  print(Platform.isIOS);
  print(Platform.isMacOS);
  print(Platform.isWindows);
  print(Platform.isFuchsia);
  print(Platform.isLinux);
  print("================================================");
  Platform.environment.forEach((key, value) {
    print("$key: $value");
  });
  /*
  Another way to get the environment:

  Platform.envornemnt.forEach((key, _){
    print("$key: ${Platform.environment[key]}");
  });
   */
}
```

Extension Function

- Extension method is a way to add functionality to existing libraries.

Example:

- In this example we are going to add a new functionality to **String** library.

```dart
void main() {
  print(int.parse('5') + 5); // 10
```

```dart
  print('3'.parseInt()); // 3
  print('3'.parseInt().runtimeType); // int
  /*
  We want to make a function has the same functionlaity
  of int.parse()
  */
}

extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
    /*
    this ==> is the object that calls the extension.
    */
  }
}
```

Note:

- We can use the name of the extension **(NumberParsing)** if the extension is in another dart file.

Example:

```dart
void main() {
  print(NumberParsing('3').parseInt()); // 3
}

extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
  }
}
```

Separated Operator

Example:

```dart
void main() {
  List list1 = [10, 20, 30];
```

```
  List list2 = [
    0,
    ...list1,
    ...[40, 50]
  ];

  print(list2); // [0, 10, 20, 30, 40, 50]
}
```

In this example, the separated operator is used to get the elements of the list that it came before.

typedef

- A typedef can be used to specify a function signature that we want specific functions to match.
- A function signature is defined by a function's parameters (including their types). The return type is not a part of the function signature.
- Its syntax is as follows. typedef function_name(parameters).

Example:

```
void main() {
  calc(3, 4, add); // 7
  calc(20, 10, info); // Error
}

typedef operation(a, b);

calc(x, y, operation z) {
  z(x, y);
}

info() {
  print("hello");
}

add(n1, n2) {
```

```
  print(n1 + n2);
}

sub(n1, n2) {
  print(n1 - n2);
}

mult(n1, n2) {
  print(n1 * n2);
}

div(n1, n2) {
  print(n1 / n2);
}
```

In this example, the **operation** can be any function that takes two parameters, so it can be **add()**, **sub()** or any other function but not the **info()** function because it doesn't take two parameters.

Example:

```
void main() {
  operation op;

  op = add;

  op(3, 7); // 10
}

typedef operation(a, b);

add(n1, n2) {
  print(n1 + n2);
}
```

OOP

- OOP In Dart Object-oriented programming (OOP) is a programming method that uses objects and their interactions to design and program applications. It is one of the most popular programming paradigms and is used in many programming languages, such as Dart, Java, C++, Python, etc.
- In OOP, an object can be anything, such as a person, a bank account, a car, or a house. Each object has its attributes (or properties) and behavior (or methods). For example, a person object may have the attributes name, age and height, and the behavior walk and talk.

Advantages

- It is easy to understand and use.
- It increases reusability and decreases complexity.
- The productivity of programmers increases.
- It makes the code easier to maintain, modify and debug.
- It promotes teamwork and collaboration.
- It reduces the repetition of code.

Features of OOP

- Class
- Object
- Encapsulation

- Inheritance
- Polymorphism
- Abstraction

Note:

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

Class

- In object-oriented programming, a class is a blueprint for creating objects. A class defines the properties and methods that an object will have. For example, a class called **Dog** might have properties like **breed**, **color** and methods like **bark, run**.
- You can declare a class in dart using the **class** keyword followed by class name and braces {}. It's a good habit to write class name in **PascalCase**. For example, **Employee, Student, QuizBrain**, etc.

Syntax

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

In the above syntax:

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

- **Functions** are used to perform the operations. It is also known as **methods**.

Declaring a class

- In this example below, there is class **Animal** with three properties: **name**, **numberOfLegs**, and **lifeSpan**. The class also has a method called **display**, which prints out the values of the three properties.

```
class Animal {
  String? name;
  int? numberOfLegs;
  int? lifeSpan;

  void display() {
    print("Animal name: $name.");
    print("Number of Legs: $numberOfLegs.");
    print("Life Span: $lifeSpan.");}}
```

Object

- In object-oriented programming, an object is a self-contained unit of code and data. Objects are created from templates called classes. An object is made up of properties(variables) and methods(functions). An object is an instance of a class.
- For example, a bicycle object might have attributes like color, size, and current speed. It might have methods like changeGear, pedalFaster, and brake.
- It's a good practice to declare the object name in lower case.
- Once you create an object, you can access the properties and methods of the object using the dot(.) operator.

Instantiation

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

Declaring an object

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

Syntax

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

Example:

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

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

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

void main() {
  // Here bicycle is object of class Bicycle.
  Bicycle bicycle = Bicycle();
  bicycle.color = "Red";
```

```
  bicycle.size = 26;
  bicycle.currentSpeed = 0;
  bicycle.changeGear(5);
  bicycle.display();
}
```

Example:

```
class Car {
  String? color;
  String? name;
  int? model;

  void display() {
    print("Color: $color " "\t" " Name: $name " "\t" " Model: $model");
  }
}

void main() {
  Car car = new Car();
  car.color = "Red";
  car.name = "BMW";
  car.model = 2020;
  car.display(); // Color: Red       Name: BMW       Model: 2020
}
```

Note:

- If we don't use the question mark (?) after the datatype of the properties, the complier throws an error because in dart you must initialize any variable or property in a class.
- We use it, to stop this error.

Constructor

- A constructor is a special method used to initialize an object. It is called automatically when an object is created, and it can be used to set the initial

values for the object's properties. For example, the following code creates a Person class object and sets the initial values for the name and age properties:

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

Without Constructor

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

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

Things to Remember

- The constructor's name should be the same as the class name.
- Constructor doesn't have any return type.
- When you create an object of a class, the constructor is called automatically. It is used to initialize the values when an object is created.

Syntax

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

Default Constructor

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

Declaring a Default Constructor

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

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

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

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

Example:

- In this example below, there is a class **Student** with four properties: **name**, **age**, **schoolname** and **grade**. The default constructor is used to initialize the values of the school name. The reason for this is that the school name is the

same for all the students. We also have an object of the class **Student** called **student**. The default constructor is called automatically when you create an object of the class.

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

  // Default Constructor
  Student() {
    print(
        "Constructor called"); // this is for checking the constructor is called
or not.
    schoolname = "ABC School";
  }
}

void main() {
  // Here student is object of class Student.
  Student student = Student();
  student.name = "John";
  student.age = 10;
  student.grade = "A";
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("School Name: ${student.schoolname}");
  print("Grade: ${student.grade}");
}
```

Parameterized Constructor

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

Syntax

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

Declaring a Parameterized Constructor

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

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

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

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

Note:

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

Write Constructor in a Single Line

- You can also write the constructor in short form. You can directly assign the values to the properties. For example, the following code is the short form of the constructor in one line:

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

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

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

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

Constructor with Optional Parameters

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

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

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

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

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

Parameterized Constructor with Named Parameters

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

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

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

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

Parameterized Constructor with Default Values

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

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

```
  int? age;

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

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

Note:

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

Named Constructor

- In most programming languages like java, c++, c#, etc., we can create multiple constructors with the same name. But in Dart, this is not possible. Well, there is a way. We can create multiple constructors with the same name using **named constructors**.
- Named constructors improves code readability. It is useful when you want to create multiple constructors with the same name.

Declaring a Named Constructor

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

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

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

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

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

Real Life Example of Named Constructor

- In this example below, there is a class **Person** with two properties **name** and **age**. The class has three constructors. The first is a parameterized constructor

which takes two parameters **name** and **age**. The second and third constructors are named constructors. Second constructor fromJson is used to create an object of the class **Person** from a JSON. The third fromJsonString is used to create an object of the class **Person** from a JSON string. We also have an object of the class **Person** called **person**.

```
import 'dart:convert';

class Person {
  String? name;
  int? age;

  Person(this.name, this.age);

  Person.fromJson(Map<String, dynamic> json) {
    name = json['name'];
    age = json['age'];
  }

  Person.fromJsonString(String jsonString) {
    Map<String, dynamic> json = jsonDecode(jsonString);
    name = json['name'];
    age = json['age'];
  }
}

void main() {
// Here person is object of class Person.
  String jsonString1 = '{"name": "Bishworaj", "age": 25}';
  String jsonString2 = '{"name": "John", "age": 30}';

  Person p1 = Person.fromJsonString(jsonString1);
  print("Person 1 name: ${p1.name}");
  print("Person 1 age: ${p1.age}");

  Person p2 = Person.fromJsonString(jsonString2);
  print("Person 2 name: ${p2.name}");
  print("Person 2 age: ${p2.age}");
}
```

required

Example:

- In this example we replace (?) by **required**.

```
class Person {
  String name;
  int? age = null;
  Person({required this.name, this.age}); // NAMED CONSTRUCTOR
}

void main() {
  Person per = new Person(name: "Ahmed");
  print(per.name); // Ahmed
  print(per.age); // null
}
```

late

- It's used to initialize it later. In short, this variable will never be null.

Constant Constructor

- Constant constructor is a constructor that creates a constant object. A constant object is an object whose value cannot be changed. A constant constructor is declared using the keyword const.
- Constant Constructor is used to create an object whose value cannot be changed. It Improves the performance of the program.

Rule for Declaring a Constant Constructor

- All properties of the class must be final.
- It does not have any body.

- Only class containing **const** constructor is initialized using the **const** keyword.

Declaring a Constant Constructor

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

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

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

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

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

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

Note:

- Here p1 and p2 has the same hash code. This is because p1 and p2 are constant objects. The hash code of a constant object is the same. This is because the hash code of a constant object is computed at compile time. The

hash code of a non-constant object is computed at run time. This is why p3 and p4 have different hash code.

Constant Constructor with Named Parameters

- In this example below, there is a class **Car** with three properties: **name**, **model**, and **prize**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Car** called **car**.

```
class Car {
  final String? name;
  final String? model;
  final int? prize;

  // Constant Constructor
  const Car({this.name, this.model, this.prize});
}

void main() {
  // Here car is object of class Car.
  const Car car = Car(name: "BMW", model: "X5", prize: 50000);
  print("Name: ${car.name}");
  print("Model: ${car.model}");
  print("Prize: ${car.prize}");
}
```

Benefits of Constant Constructor

- Improves the performance of the program.

Factory Constructor

- In object-oriented programming, a factory constructor is a special type of constructor that returns an instance of a class from a factory method. A factory constructor is a static method that returns an object of that class.

Why We Use Factory Constructor?

- Use factory constructor when creating new instances of a class is too expensive.
- If you want to create only one instance of a class, then you can use factory constructor.
- A factory constructor is a static method that returns an object of that class.
- A factory constructor is used to return an instance of a class from a factory method.

Note:

- The factory constructor is called when you create an object of a class. The factory constructor is used to return an instance of a class from a factory method.

Properties of Factory Constructor

Here is the list of properties of a factory constructor in dart.

- A factory constructor is a static method.
- A factory constructor is used to create an object of a class.

- A factory constructor is used to return an instance of a class from a factory method.

Example:

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

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

  Student._internal(this.name, this.age, this.rollNumber);
}

void main() {
  Student student = Student("John", 30, 1);
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("Roll Number: ${student.rollNumber}");
}
```

Reset the Value of a Variable in a Class

First Way

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

  Info() {
    print("a default constructor is called.");
    name = "\0";
    age = 0;
  }
```

```
  Info.parameterized(String? n, int? g) {
    print("A parameterized Constructor is called");
    this.name = n;
    this.age = g;
  }

  void display() {
    print("Name: $name " "\t" " Age: $age");
  }
}

void main() {
  Info person1 = Info.parameterized("Ahmed", 20);
  person1.display(); // Name: Ahmed      Age: 20
  person1.name = "Mohamed";
  person1.display(); // Name: Mohamed    Age: 20
}
```

Second Way – Cascade Operator (..)

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

  Info() {
    print("a default constructor is called.");
    name = "\0";
    age = 0;
  }

  Info.parameterized(String? n, int? g) {
    print("A parameterized Constructor is called");
    this.name = n;
    this.age = g;
  }

  void display() {
    print("Name: $name " "\t" " Age: $age");
  }
}

void main() {
  Info person1 = Info.parameterized("Ahmed", 20)..name = "Mohamed"..age = 35;
```

```
  person1.display(); // Name: Mohamed    Age: 35
}
```

Third Way – Using a Parameterized Constructor

You know it and we have used it many times so far.

Setter & Getter

Getter

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

Syntax

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

Why is Getter Important in Dart?

- To access the value of private property.
- To restrict the access of data members of a class.

Note:

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

Example:

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

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

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

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

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

Example:

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

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

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

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

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

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

Note:

- In the above example, a getter **name** and **prize** are used to access the value of the properties **_name** and **_prize**.

Example – Getter with Data Validation

- In this example below, there is a class named **NoteBook**. The class has two private properties **_name** and **_prize**. There are two getters **name** and **prize** to access the value of the properties. If you provide a blank name, then it will return **No Name**.

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

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

  // Getter to access private property _name
  String get name {
    if (_name == "") {
      return "No Name";
    }
    return this._name;
  }
```

```
  // Getter to access private property _prize
  double get prize {
    return this._prize;
  }
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook("Apple", 1000);
  print("First Notebook name: ${nb.name}");
  print("First Notebook prize: ${nb.prize}");
  NoteBook nb2 = new NoteBook("", 500);
  print("Second Notebook name: ${nb2.name}");
  print("Second Notebook prize: ${nb2.prize}");
}
```

Example:

- In this example below, there is a class named **Doctor**. The class has three private properties **_name**, **_age** and **_gender**. There are three getters **name**, **age**, and **gender** to access the value of the properties. It has **map** getter to get Map of the object.

```
class Doctor {
// Private properties
  String _name;
  int _age;
  String _gender;

// Constructor
  Doctor(this._name, this._age, this._gender);

// Getters
  String get name => _name;
  int get age => _age;
  String get gender => _gender;

// Map Getter
  Map<String, dynamic> get map {
    return {"name": _name, "age": _age, "gender": _gender};
```

```
  }
}

void main() {
// Create an object of Doctor class
  Doctor d = Doctor("John", 41, "Male");
  print(d.map);
}
```

Setter

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

Syntax

```
set property_name (value) {
  // Setter body
}
```

Why is Setter Important in Dart?

- It is used to set the value of a private property.
- It is also used for data validation.
- It gives you better control over the data.

Note:

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

Example:

```
class Person {
  String? name;
```

```
  int? age;

  set setName(String n) => this.name = n;
  set setAge(int g) => this.age = g;

  String get getName {
    if (name == "") {
      return "No Name";
    }
    return this.name!;
  }

  int get getAge {
    return this.age!;
  }
}

void main() {
  Person per1 = new Person();
  Person per2 = new Person();
  per1.setName = "Ahmed";
  per1.setAge = 20;
  per2.setName = "";
  per2.setAge = 30;
  print("First Person's Name: ${per1.getName}");
  print("First Person's Age: ${per1.getAge}");
  print("=========================================");
  print("Second Person's name: ${per2.getName}");
  print("Second Person's Age: ${per2.getAge}");
}
```

Note:

- In the above example, a setter **name** and **prize** are used to update the value of the properties **_name** and **_prize**.

Example – Setter with Data Validation

- In this example, there is a class named **NoteBook**. The class has two private properties **_name** and **_prize**. If the value of **_prize** is less than 0, we will

throw an exception. There are also two setters name and prize to update the value of the properties. The class also has a method **display()** to display the values of the properties.

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

  // Setter to update the value of name property
  set name(String name) => _name = name;

  // Setter to update the value of prize property
  set prize(double prize) {
    if (prize < 0) {
      throw Exception("Price cannot be less than 0");
    }
    this._prize = prize;
  }

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

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook();
  // setting values to the object using setter
  nb.name = "Dell";
  nb.prize = 250;

  // Display the values of the object
  nb.display();
}
```

Note:

- It is generally best to not allow the user to set the value of a field directly. Instead, you should provide a setter method that can validate the value before setting it. This is very important when working on large and complex programs.

Example:

- In this example, there is a class named **Student**. The class has two private properties **_name** and **_classnumber**. We will also create two setters **name** and **classnumber** to update the value of the properties. The **classnumber** setter will only accept a value between 1 and 12. The class also has a method **display()** to display the values of the properties.

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

  set setGender(String gen) {
    if (gen == "male" || gen == "Male") {
      this.gender = gen;
    } else {
      throw Exception("Gender Must Be \"Male\" or \"male\"");
    }
  }

  set setAge(int g) {
    if (g > 12 && g <= 16) {
      this.age = g;
    } else {
      throw Exception("Age must between 13 & 16");
    }
  }

  String get getGender {
    if (gender == "") {
```

```
      return "No Gender";
    }
    return this.gender!;
  }

  int get getAge => this.age!;
}

void main() {
  Student std1 = new Student();
  std1.setGender = "male";
  std1.setAge = 13;
  print("The Student Gender: ${std1.getGender}");
  print("The Student Age: ${std1.getAge}");
}
```

Static

- It generally manages the memory for the global data variable. Static variables are the same for every instance of the class. Using the class name, you can also call the static method and variables from other classes using the class name.

Note:

- The static keyword is used to declare the class variable and method as static.

Static Variable

- All the variables declared using the static keyword inside the class are static variables. These are the member of the class instead of a specific instance. The static variables are treated the same for all instances of the class; it means a single copy of the static variable is shared among all instances of classes.

Syntax

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

Example:

```
class Student {
  static String studentName = "Suraj Subedi";
  static int rollNo = 213;
  static String address = "Pokhara";
}

void main() {
  print("Name of the student: ${Student.studentName}");
  print("Name of the student: ${Student.rollNo}");
  print("Name of the student: ${Student.address}");
}
```

Static Method

- The static methods are the member of the class instead of the class instance. The static methods can use only static variables and can invoke the static method of the class. you don't need to create an instance of the class to access it.

Syntax

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

Example:

```
class Student {
  static String studentName = "Suraj Subedi";
  static int rollNo = 213;
  static String address = "Pokhara";

  static void printStudentDetails() {
    print("Name of the Student: ${Student.studentName}");
    print("Roll No: ${Student.rollNo}");
    print("Address: ${Student.address}");
  }
}

void main() {
  Student.printStudentDetails();
}
```

Cascade Operator

- We can call a method from a class without creating an object, using dot and this is called **Cascade operator.**

Example:

```
class MyClass {
  void printHello() {
    print("Hello");
  }

  void printHi() {
    print("Hi");
  }
}

void main() {
  new MyClass().printHello();
}
```

- Using the way in the previous example to call a method, doesn't allow us to call more than a method at the same time.
- We can call more than a method at the same time using two dots (..)

Example:

```
class MyClass {
  void printHello() {
    print("Hello");
  }

  void printHi() {
    print("Hi");
  }
}

void main() {
  new MyClass()..printHello()..printHi();
}
```

Inheritance

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

Note:

- Whenever you use inheritance, it always creates a **is-a** relation between the parent and child class like **Student is a Person**, **Truck is a Vehicle**, **Cow is an Animal** etc.
- Dart supports single inheritance, which means that a class can only inherit from a single class. Dart does not support multiple inheritance which means that a class cannot inherit from multiple classes.

Syntax

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

class ChildClass extends ParentClass {
  // Child class code
}
```

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

Terminology

- **Parent Class**: The class whose properties and methods are inherited by another class is called parent class. It is also known as base class or super class.
- **Child Class**: The class that inherits the properties and methods of another class is called child class. It is also known as derived class or sub class.

Advantages of Inheritance

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

Types of Inheritance

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

Example:

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

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

  set setName(String n) {
    this.name = n;
  }

  set setAge(int g) {
    this.age = g;
  }
```

```
  set setGender(String gen) {
    if (gen == "male" || gen == "female") {
      this.gender = gen;
    } else {
      throw Exception("Error: Invalid Gender");
    }
  }

  String get getName => this.name!;
  int get getAge => this.age!;
  String get getGender => this.gender!;

  void display() {
    print("Name: $name " "\t" " Age: $age " "\t" " Gender: $gender");
  }
}

class Student extends Person {
  String? schoolName;
  String? schoolAddress;

  set setSchoolName(String schName) {
    this.schoolName = schName;
  }

  set setSchoolAddress(String schAdd) {
    this.schoolAddress = schAdd;
  }

  String get getSchoolName => this.schoolName!;
  String get getSchoolAddress => this.schoolAddress!;

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

void main() {
  Student std = new Student();
  std.setName = "Ahmed";
  std.setAge = 20;
  std.setGender = "male";
  std.setSchoolName = "ABC School";
  std.setSchoolAddress = "Baltim";
```

```
  std.display();
  std.displaySchoolInfo();
}
```

Example:

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

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

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

class Toyota extends Car {
  String? model;
  int? prize;

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

void main() {
  var toyota = Toyota();
  toyota.color = "Red";
  toyota.year = 2020;
  toyota.model = "Camry";
  toyota.prize = 20000;
  toyota.start();
  toyota.showDetails();
}
```

Example – Single Inheritance

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

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

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

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

Example – Multilevel Inheritance

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

the properties of the sub class **Tesla**. The sub class has a property **color** and a method display to **display** the values of the properties.

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

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

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

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

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

Note:

- Here **super** keyword is used to call the method of the parent class.

Example – Multilevel Inheritance

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

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

  set setName(String n) {
    this.name = n;
  }

  set setAge(int g) {
    this.age = g;
  }

  set setGender(String gen) {
    if (gen == "male" || gen == "Male" || gen == "female" || gen == "Female") {
      this.gender = gen;
    } else {
      throw Exception("Error: Invalid Gender");
    }
  }
}

class Doctor extends Person {
  String? hospitalName;
  String? hospitalAddress;
  void display() {
    print("Name: $name " "\t" " Age: $age " "\t" " Gender: $gender");
    print("Hospital Name: $hospitalName "
        "\t"
        " Hospital Address: $hospitalAddress");
  }
}
```

```dart
class Specialist extends Doctor {
  String? specialization;
  void display() {
    super.display();
    print("Specialization: $specialization");
  }
}

void main() {
  Specialist sp = new Specialist();
  sp.setName = "Ahmed Ghaly";
  sp.setAge = 20;
  sp.setGender = "Male";
  sp.hospitalName = "Health Hospital";
  sp.hospitalAddress = "Cairo";
  sp.specialization = "Cardiologist";
  sp.display();
}
```

Example – Heretical Inheritance

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

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

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

class Triangle extends Shape {
  // Method to calculate the area of the triangle
```

```
  double area() {
    return 0.5 * diameter1! * diameter2!;
  }
}

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

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

Key Points

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

Why Dart Does Not Support Multiple Inheritance?

- Dart does not support multiple inheritance because it can lead to ambiguity. For example, if class **Apple** inherits class **Fruit** and class **Vegetable**, then

there may be two methods with the same name **eat**. If the method is called, then which method should be called? This is the reason why Dart does not support multiple inheritance.

What's Problem of Copy Paste Instead of Inheritance?

- If you copy the code from one class to another class, then you will have to maintain the code in both the classes. If you make any changes in one class, then you will have to make the same changes in the other class. This can lead to errors and bugs in the code.

We can solve the problem of multiple inheritance using **Interface**.

In this case we don't use the keyword **extends**, we use **implements** instead, and what's coming after it is an **interface**.

Interface is an **abstract** class.

Note that to declare a class that **implements** a class or more than a class, you must **override** this/these classes' methods or fields (variables).

Example:

```
abstract class A {
  infoA();
}

abstract class B {
  infoB();
}

class C implements A, B {
  @override
  infoA() {
    print("This is from Interface A");
  }
```

```
  @override
  infoB() {
    print("This is from Interface B");
  }

  infoC() {
    print("This is from Class C");
  }
}

void main() {
  C object = C();
  object.infoA();
  object.infoB();
  object.infoC();
}
```

Inheritance of Constructor

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

Example:

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

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

void main() {
```

```
  var macbook = MacBook();
}
```

Note:

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

Example - Inheritance of Constructor with Parameters

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

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

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

Example:

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

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

class Student extends Person {
```

```
  int rollNumber;

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

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

Example - Inheritance of Constructor with Named Parameters

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

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

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

Example - Calling Named Constructor of Parent Class

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

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

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

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

Override

- Method overriding occurs in dart when a child class tries to override the parent class's method. When a child class extends a parent class, it gets full access to the methods of the parent class and thus it overrides the methods of the parent class. It is achieved by re-defining the same method present in the parent class.
- This method is helpful when you have to perform different functions for a different child class, so we can simply re-define the content by overriding it.

Important Notes

- A method can be overridden only in the child class, not in the parent class itself.

- Both the methods defined in the child and the parent class should be the exact copy, from name to argument list except the content present inside the method i.e., it can and can't be the same.
- A method declared final or static inside the parent class can't be overridden by the child class.
- Constructors of the parent class can't be inherited, so they can't be overridden by the child class.

Example:

```
class SuperGeek {
  // Creating a method
  void show() {
    print("This is class SuperGeek.");
  }
}

class SubGeek extends SuperGeek {
  // Overriding show method
  void show() {
    print("This is class SubGeek child of SuperGeek.");
  }
}

void main() {
  // Creating objects
  //of both the classes
  SuperGeek geek1 = new SuperGeek();
  SubGeek geek2 = new SubGeek();

  // Calling same function
  // from both the classes
  // object to show method overriding
  geek1.show();
  geek2.show();
}
```

Example:

```
class SuperGeek {
  // Creating a method
  void show() {
    print("This is class SuperGeek.");
  }
}

class SubGeek1 extends SuperGeek {
  // Overriding show method
  void show() {
    print("This is class SubGeek1 child of SuperGeek.");
  }
}

class SubGeek2 extends SuperGeek {
  // Overriding show method

  void show() {
    print("This is class SubGeek2 child of SuperGeek.");
  }
}

void main() {
  // Creating objects of both the classes
  SuperGeek geek1 = new SuperGeek();
  SubGeek1 geek2 = new SubGeek1();
  SubGeek2 geek3 = new SubGeek2();

  // Calling same function
  // from both the classes
  // object to show method
  // overriding
  geek1.show();
  geek2.show();
  geek3.show();
}
```

Mixin

- In Dart, a mixin is a way of reusing a class's code in multiple class hierarchies. It allows a class to inherit methods and properties from multiple classes without having to define its own subclass for each one.
- Mixins are implemented as a special kind of abstract class that cannot be instantiated directly but can be added to a class using the `with` keyword.
- To create a mixin in Dart, you define a new class that extends the `Object` class and contains the methods or properties you want to reuse in other classes. Then you add the mixin to another class using the `with` keyword followed by the name of the mixin class.

Example:

```
mixin A {
  infoA() {
    print("Hi");
  }
}

mixin B {
  infoB() {
    print("Hi 2");
  }
}

class C {
  infoC() {
    print("Hi 3");
  }
}

class D extends C with A, B {}

void main() {
  D d = D();
```

```
  d.infoA();
  d.infoB();
  d.infoC();
}
```

Example:

```
mixin A {
  infoA() {
    print("Hi");
  }
}

mixin B {
  infoB() {
    print("Hi 2");
  }
}

class C {
  infoC() {
    print("Hi 3");
  }
}

class D extends C with A, B {
  @override
  infoA() {
    print("Hello");
    return super.infoA(); // this returns the mixin content too
  }

  @override
  infoB() {
    print("Hello 2");
  }

  @override
  infoC() {
    print("Hello 3");
  }
}

void main() {
```

```
  D d = D();
  d.infoA();
  d.infoB();
  d.infoC();
}
```

Example:

```
mixin A {
  infoA();
}

mixin B {
  infoB();
}

class C {
  infoC() {
    print("Hi 3");
  }
}

class D extends C with A, B {
  @override
  infoA() {
    print("Hello");
  }

  @override
  infoB() {
    print("Hello 2");
  }

  @override
  infoC() {
    print("Hello 3");
  }
}

void main() {
  D d = D();
  d.infoA();
  d.infoB();
  d.infoC();}
```

Note

- We can use an empty body function in **mixin** because it’s an abstract, but we can’t do so with a regular class.

Type map

HashMap

- A HashMap doesn't guarantee insertion order. If you insert an entry with key A and then another entry with key B, when you iterate through the map, there's a possibility that you'll get entry B first.

Example:

```
import 'dart:collection';

void main() {
  HashMap names = new HashMap();
  names.addAll({'first': 'ahmed'});
  names.addAll({'second': 'ali'});
  names.addAll({'third': 'omar'});
  print(names);
}
```

LinkedHashMap

- Data stored in a LinkedHashMap is sorted according to insertion order. If you insert an entry with key A and then another entry with key B, when you iterate through the map, you'll always get entry A first before entry B.

Example:

```
import 'dart:collection';
```

```
void main() {
  LinkedHashMap names = new LinkedHashMap();
  names.addAll({'first': 'ahmed'});
  names.addAll({'second': 'ali'});
  names.addAll({'third': 'omar'});
  print(names);
}
```

SplayTreeMap

- A SplayTreeMap is a self-balancing binary tree that allows recently accessed elements to be accessed quicker.
- Basic operations like insertion, look-up and removal can be done in O(log(n)).
- It performs tree rotations by bringing frequently accessed elements close to the root of the tree. Therefore, if some entries need to be accessed more often than the others, using SplayTreeMap is a good choice. However, if the data access frequency is almost the same on all entries, using SplayTreeMap is useless.

Example:

```
import 'dart:collection';

void main() {
  SplayTreeMap names = new SplayTreeMap();
  names.addAll({'first': 'ahmed'});
  names.addAll({'second': 'ali'});
  names.addAll({'third': 'omar'});
  print(names);
}
```

Enum

- An enum or enumeration is used for defining named constant values in a dart programming language. An enumerated type is declared using the keyword enum. It is used to hold the list of named constant values.

Characteristics of enum

- It must contain at least one constant value.
- Enums are declared outside the class.
- Used to store a large number of constant value.
- Makes code more readable and simple.
- It makes the code more reusable and makes it easier for developers.
- Values inside enumeration type cannot be changed dynamically.

Syntax

```
enum enum_name {
  constant_value1,
  constant_value2,
  constant_value3
  }
```

Example:

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

void main() {
  Enum today = days.Friday;

  switch (today) {
    case days.Sunday:
      print("Today is Sunday.");
      break;
```

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

Example – Printing All Values:

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

void main(List<String> args) {
  for (var day in days.values) {
    print(day);
  }
}
```

Enhance Enumerations in Dart

- The new version of the dart comes with the support for the enhanced enumerations. The main purpose of this is to increase the readability of the code and to make code a lot cleaner and more efficient to use.

Example:

```
enum Water {
```

```
  frezing(32),
  boiling(212);

  final int temperatureInF;
  const Water(this.temperatureInF);
}

void main() {
  print(
      "The ${Water.frezing.name} point for water is
${Water.frezing.temperatureInF} degree F.");
  print(
      "The ${Water.boiling.name} point for water is
${Water.boiling.temperatureInF} degree F. ");
}
```

Encapsulation

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

How to achieve Encapsulation in Dart?

Encapsulation can be achieved by:

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

Note:

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

Example:

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

// Getter method to access private property _id
  int getId() {
    return _id!;
  }

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

// Setter method to update private property _id
  void setId(int id) {
    this._id = id;
  }

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

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

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

Private Properties

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

Example - Private Properties

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

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

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

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

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

Why Aren't Private Properties Private?

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

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

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

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

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

Reason

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

Solution

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

Read-only Properties

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

Note:

- Properties declared with the final keyword must be initialized at the time of declaration. You can also initialize them in the constructor.

```
class Student {
  final _schoolname = "ABC School";

  String getSchoolName() {
    return _schoolname;
  }
}

void main() {
  var student = Student();
  print(student.getSchoolName());
  // This is not possible
  //student._schoolname = "XYZ School";
}
```

Note:

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

Why Encapsulation is Important?

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

Polymorphism

- Polymorphism is the concept of OOP, where one thing has various possible forms. The idea of the same thing having different shapes, sizes, and features distinguishes it from others. It reduces the time and effort to write repetitive code and provides you with the flexibility to override.

Syntax

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

Note:

- You use @override annotation to note/comment while overriding the method.

Example:

```
class Animal {
  void sound() {
    print("Animal produces sound.");
  }
}

class Cat extends Animal {}

class Dog extends Animal {
  @override
  void sound() {
    print("Dogs produces woof sound.");
  }
}

void main(List<String> args) {
  var dog = Dog();
  dog.sound();
  var cat = Cat();
  cat.sound();
}
```

- The above example shows the class Animal with the function sound. Later, you can see the class Cat and Dog extending Animal, meaning they also inherit the function sound. Then, a cat uses the same sound function as an Animal, whereas a Dog overrides it, making it possible for the same function to have a different form.

Example:

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

```
  }
}

class Honda extends Car {}

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

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

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

Note:

- Dart doesn't support method overloading, so each method name must be unique regardless of its parameters requirement due to the support of dynamic data type.

Abstract Class

- Abstract class are the classes that cannot be initialized, i.e. we can't use an object of it. It is used to define the behavior of a class that can be inherited by other classes. An abstract class is declared using the keyword abstract.

Note:

- Abstract class may or may not contain abstract methods, but a class having at least one abstract method must declare abstract.

Key Features

- Abstract class cannot be initialized.
- It can have both abstract and non-abstract methods.
- It is declared with abstract keyword.
- It can be inherited using the extend keyword then the abstract methods need to be implemented.

Syntax

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

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

Example:

```
abstract class Vehicle {
  printVehicleName();
  printSupplierName() {
    print('Supplier: TATA Motors');
  }
}

class Truck extends Vehicle {
  @override
  printVehicleName() {
    print("This is a truck.");
  }
}

void main(List<String> args) {
  var truck = Truck();
  truck.printVehicleName();
  truck.printSupplierName();
}
```

- In the above example, the Vehicle class is created and marked as an abstract class which contains the abstract method printVehicleName() and the non-abstract method printSupplierName(). Class Turck extends the class Vehicle and then implements the class Vehicle. In the main method, object of the truck is created which then calls the printVechicleName() and printSupplierName() from the parent class.

Generic

- They are the same as the dart collection. In dart, you used the different collections to store the group of data.

Example:

```
  void main()
{
    List list = ["Nabin", 19.0, 7500];
  print(list);
}
```

- Generics are the way of supporting the type-safety for all the Dart collections. The angular bracket is used to declare the generics in dart where the angular bracket is placed after the collection type which holds the data type which you want to enforce that collection to hold.

Syntax

```
CollectionName<dataType> variableName = CollectionName();
```

Note:

- Generics can be used for all the collection types, here is an example of generics for some of the common collections used in dart.

Example:

```
class College {
  List<String> studentsName = [
    "Suraj Subedi",
    "Rahul Jaishwal",
    "Sachin Khand"
  ];

  void printAllStudentName() {
    for (var i in studentsName) {
      print(i);
    }
  }
}

void main(List<String> args) {
  var college = College();
  college.printAllStudentName();
}
```

Example:

```
class College {
  //set only contains distict value, ingnores the duplicate data
  Set<String> studentsName = {
    "Suraj Subedi",
    "Rahul Gautam",
    "Mahan Gurung",
    "Suraj Subedi",
  };

  void printAllStudentName() {
    for (var i in studentsName) {
      print(i);
    }
  }
```

```
}

void main(List<String> args) {
  var college = College();
  college.printAllStudentName();
}
```

Example:

```
class College {
  //map holds data in key-value pair
  Map<int, String> studentsName = {
    20048: "Suraj Subedi",
    25412: "Rahul Gautam",
    56557: "Mahan Gurung",
    41245: "Sameen Kunwar",
  };

  void printAllStudentName() {
    studentsName.forEach((key, value) {
      print("Key:$key, value: $value");
    });
  }
}

void main(List<String> args) {
  var college = College();
  college.printAllStudentName();
}
```

Generic Class

Example:

```
class MyClass<T, E> {
  T? field1;
  E? field2;
}

void main() {
  MyClass object = MyClass<String, double>();
  object.field1 = "Hello From Field One";
```

```
  object.field2 = 25.3;
  print(object.field1);
  print(object.field2);

  MyClass objectTwo = MyClass<bool, int>();
  objectTwo.field1 = true;
  objectTwo.field2 = 20;

  print(objectTwo.field1);
  print(objectTwo.field2);
}
```