Dart Programming Language

- **Dart** is an **open-source general-purpose** programming language. It is originally developed by **Google** and later approved as a standard by **ECMA**.
- **Dart** is a new **programming language** meant for the **server** as well as the **browser**.
- Introduced by **Google**, the **Dart SDK** ships with its **compiler**- the **Dart VM**. The **SDK** also includes a utility **–dart2js**, a **transpiler** that generate **JavaScript** equivalent of a **Dart Script**.
- **Dart** is **Object-Oriented Language** with **C-Style syntax** which can optionally **trans compile** into **JavaScript**.
- **Dart** supports a varied **range** of **programming aids** like **interfaces**, **classes**, **collections**, **generics**, and **optional typing**.
- **Dart** can be extensively used to **create single-page applications**.
- **Single page applications** apply only to **website** and we **applications**.
- **Single page applications** enable **navigations** between different **screens** of the **website** without loading a different **webpage** in the **browser**. A classic example is **Gmail**.

Comparison between Dart and JavaScript:

Serial Number	Feature	Dart	JavaScript
01.	Type System	Optional, Dynamic	Weak, Dynamic
02.	Classes	Yes, Single Inheritance	Prototypical
03.	Interfaces	Yes, Multiple Interfaces	No
04.	Concurrency	Yes, with Isolates	Yes, with HTML% web Workers

Dart Environment:

There are **two** methods of setting up the **Dart Environment**:

Executing Script Online with DartPad:

- You may **test** your **scripts** online by using the **online editor** “https://dartpad.dev/”.
- The **Dart Editor** executes the **script** and **displays** both **HTML** as well as **console Output**.
- **DartPad** also enables to **code** in a more **restrictive fashion**. This can be achieved by checking the strong **mode** option on the **bottom right** of the editor.

Setting up the Local Environment:

- Using IDE/Text Editor:

1. **Install** the **Dart SDK**.
2. **Verify** the **installation**.
3. **Download** the **editor** you like, for example: **WebStorm**, **Intellij IDEA** etc.
4. After **Installation**, **open** your **editor**.
5. **Click** on **New**, **name** your **project** and **create** it.
6. **Right click** on your **project**, new then **Dart file** and **name** it.
7. **Write** a **program** and **run** to **test** it.

Variables:

Built-in **Variables** in **Dart**:

1. **Number**
   a. **Int**
   b. **double**
2. **Strings**
3. **Booleans**
4. **Lists (Also known as Arrays)**
5. **Maps**
6. **Runes (For expressing Unicode characters in a String)**
7. **Symbols**

Note: All **Data Types** in **Dart** are **Objects** which means their **Default Value** will be **NULL** until we **Initialize** it.

SYNTAX OF DECLARING A VARIABLE:

`var variable_name = value;`

OR

You can write them with **Specific Data Types**.

For Example: `"int variable_name = value".`

CODE OF VARIABLES:

```
void main(){                          // main part of the program
//  TODO: Numbers Variables declaration
  int number = 10;                     // integer variable declaration
  var value = 20;                     // integer variable declaration
  print(number);                      // printing the value
  print(value);                      // printing the value

//  TODO: Double Variables Declaration
  double doubleValue = 5.08;           // double variable declaration
  double doubleValue_1 = 0x12345678;    // double variable declaration
  print(doubleValue_1);                // printing the value
  print(doubleValue);                 // printing the value

//  TODO: String Variables Declaration
  String first_name = "Ahtesham";       // string variable declaration
  var last_name = "Awan";               // string variable declaration
  print(first_name);                  // printing the value
  print(last_name);                   // printing the value

//  TODO: Boolean Variables Declaration
```

```
  bool isValid = true;                // Boolean variable declaration
  var isNotValid = false;              // Boolean variable declaration
  print(isValid);                   // printing the value
  print(isNotValid);                  // printing the value
}
```

Question: Why to use **Final** and **Constant Keyword**?

Answer: If you **Never** want to **Change Value** of a **Variable** in your program, then you have to use **Constant** or **Final Keyword**.

To Declare Final Variable:

Note: You can use **"const"** and **"Final"** Keyword only, but you can also use **Data Types.**

For Example:

```
final name = 'Ahtesham';
```

```
final String = 'Ahtesham';
```

To Declare Constant Variable:

For Example:

```
const PI = 3.14;
```

```
const double PI = 3.14;
```

Difference Between Final and Constant Keyword:

Final Variable can only be **Set Once** and it is **Initialized** when **Accessed**.

For Example: When you will use the **Variable Name** then only it's **Value** will be **Initialized** and **Memory** will be **Allocated** otherwise not.

Constant Variable is **Implicitly Final** but it is a **Compile Time Constant**. i-e **Constant Variable Initialized** during the **Compilation** and it used **Memory**.

Instance Variable (Variables with in a Class) can be **Final** but cannot be **Constant** so in that case

If you want **Constant Variable** in a **Class**, then you have to use **'static'** keyword along with **'const'**.

Literals:

If you **Define** some values as `10, 2.5, 'name'`, then these **Values** are known as **Literals**. You can assign the **Literals** to the **Variables**.

String Literal:

You can define a **String Literal** in **Single Quotes** as well as **Double Quotes**.

For Example: `'Single Quoted String'`, `"Double Quoted String"`.

String Interpolation:

String Interpolation includes **Different Operation** on **String** Like;

- **Combining Two Strings.**
- **Length Calculation of String and so on**.

Note: You can apply **Interpolations** on **Int**, **Double**, **Boolean** as well.

CODE OF STRING VARIABLE AND INTERPOLATION:

```
void main(){
  String s1 = 'Single';     // string declaration and initialization
  String s2 = "Double";     // string declaration and initialization
  print(s1);       // printing string s1
  print(s2);       // printing string s2

  String s3 = 'It\'s easy';   // string declaration and initialization
  print(s3);       // printing string s3
  String s4 = "It's easy";    // string declaration and initialization
  print(s4);       // printing string s4

//  TODO: How to combine two strings in dart
//  First way of writing a string
  String s5 = 'This is going to be a very long String' +
      'This is Just a demo to write a long String in Dart Programming Language';

  //  Best Way of writing a long String in Dart is
  String s6 = 'This is going to be a very long String'
  'This is just a demo to write a long String in Dart Programming Language';
  print(s5);      // printing string s5
  print(s6);      // printing string s6

//  TODO: String Interpolation
//  This is not a good way of combining two or more strings in Dart
  String s7 = 'Ahtesham';     // string declaration and initialization
  String s8 = 'My Name is';   // string declaration and initialization
  print(s8 + s7);         // printing string s7 and s8

//  Best Way of Combining Two Strings in Dart is
  String s9 = 'Ahtesham';     // string declaration and initialization
  print('My Name is $s9');    // printing string s9

//  TODO: To calculate length of string
  print('Length of String Ahtesham is: ' + s9.length.toString());
//  Another way of calculating length of string is
  print('Length of String Ahtesham is: ${s9.length}');
//  TODO: Integer Interpolation
  int length = 10;      // integer variable declaration and initialization
  int breadth = 20;     // integer variable declaration and initialization
  print("The Sum of Length and Breadth is: ${length + breadth}");
  print("The Sum of $length and $breadth is:  ${length + breadth}");
 }
```

Conditional Statements/Control Flow Statements/Decision Statements/If-else Conditions:

In these statements, we take some **Decisions** on some certain **Types** of **Conditions**.

Conditional Expressions:

There are **Two** types of **Conditional Expressions**:

```
condition? exp1: exp2;
```

This expression is read as **"If Condition is True, then Return exp1, and if Condition is not True, then Return exp2"**.

```
exp1 ?? exp2;
```

This expression is read as **"If exp1 is not NULL then Return exp1, and if it is NULL then Return exp2"**.

CODE OF IF-STATEMENT, IF-ELSE STATEMENT, MULTIPLE IF-ELSE STATEMENT, AND CONDITIONAL EXPRESSIONS:

```
void main(){
//  TODO:if statement
  var salary = 25000;         // integer variable declaration
  if(salary > 20000){         // if condition
    print('Congratulations! You have got Promoted');    // printing a string
  }

//  TODO:if-else statement
  var number = 10;          // integer variable declaration
  if(number < 0){           // if condition
    print('$number is a Negative Number');       // printing a string
  }else{                    // else statement
    print('$number is a Positive Number');       // printing a string
  }



//  TODO: Multiple if-else statement
  var marks = 70;         // integer variable declaration
  if(marks > 90 && marks <= 100){     // if condition
    print('A+ Grade');            // displaying result
  }else if(marks >= 80 && marks <= 90){   // else if condition
    print('A Grade');             // displaying result
  }else if(marks >= 70 && marks <= 80){   // else if condition
   print('B Grade');            // displaying result
  }else if(marks >= 60 && marks <= 70){   // else if condition
   print('C Grade');            // displaying result
  }else if(marks >= 50 && marks <= 60){   // else if condition
   print('D Grade');            // displaying result
  }else{    // else statement
   print('F Grade');            // displaying result
  }

//  TODO: Conditional Expressions
  int a = 2;       // integer variable declaration
  int b = 3;       // integer variable declaration
  a < b ? print('$a is smaller') : print('$b is smaller');

  String name = "Ahtesham";     // string variable declaration
//  String name = null;        // string variable declaration
  String nameToPrint = name ?? "Guest User";      // conditional statement
  print(nameToPrint);          // displaying result
}
```

Switch Case Statement:

- **Switch Case Statement** is same as the **if-else conditions statements**.
- Only for **"int"** & **"String" Data Types**.

CODE OF SWITCH STATEMENT:

```
void main(){
  String grade = 'A';      // string variable declaration
  switch(grade){           // switch statement
   case 'A':               // case definition
    print("Excellent");   // printing result
    break;                 // break statement
   case 'B':               // case definition
    print("Very Good");   // printing result
    break;                 // break statement
   case 'C':               // case definition
    print("Good. Keep it up");  // printing result
    break;                 // break statement
   case "D":                 // case definition
    print("Need to Work Hard");   // printing result
    break;                 // break statement
   case "F":                 // case definition
    print("Failed");      // printing result
    break;                 // break statement
   default:               // case definition
    print("Invalid Input");   // printing result
  }
}
```

Loops Iterators/Loop Control Statement:

For Loop:

Syntax is:

```
for (initializer; condition; increment/decrement counter variable) {

// body of loop

}
```

CODE OF FOR LOOP:

```
void main() {
  print("For Loop Execution: "); // printing a string
  for (int i = 0; i <= 5; i++) { // for loop
    print(i); // printing output
  }
//  TODO: Nested For Loops
  for (int value1 = 1; value1 <= 3; value1++) { // outer for loop
    for (int value2 = 1; value2 <= 3; value2++) { // inner for loop
      print("$value1 $value2"); // printing the value of value1 & value2
    }
  }
}
```

While Loop:

Syntax is:

```
while (condition on counter variable) {

// body of loop

// increment/decrement the counter variable

}
```

CODE OF WHILE LOOP:

```
void main(){
  print("While Loop Execution: ");    // printing a string
  int i = 0;      // integer variable declaration and initialization
  while(i <= 5){      // while loop
    print(i);     // printing output
    i++;       // incrementing the value of i
  }
}
```

Do-While Loop:

Syntax is:

```
do {
```

```
// body of loop

// increment/decrement the counter variable

} while (condition on counter variable);
```

CODE OF DO-WHILE LOOP:

```
void main(){
  print("Execution of Do-While Loop: ");    // printing a string
  int i = 0;      // integer variable declaration and initialization
  do{     // do while loop
   print(i);    // printing output
   i++;     // incrementing the value
  }while(i <= 5);   // while condition
}
```

Difference between these 3 loops:

If you know **Exact Numbers** of **Iterations,** then use **for-loop (For Definite Numbers)**.

If you don't know the **Exact Numbers** of **Iterations,** then use **while/do-while loop (For Indefinite Numbers)**.

Difference between while & do-while loop:

In **While Loop**, we first **Check** the **Condition** and then we **Check** the **Condition.**

Break Statement:

Break Statement is used when you want a **Certain Value** to be **Print Out.**

CODE OF BREAK STATEMENT:

```
void main(){
  for(int a = 1; a <= 10; a++){     // for loop
    print(a);              // printing value of a
    if(a == 5){            // if condition to check the value of a
      break;              // break statement
    }
  }
}
```

Continue Statement:

CODE OF CONTINUE STATEMENT:

```
void main(){
//  TODO: Continue Statement
```

```
  for(int number_1 = 1; number_1 <= 10; number_1++){   // for loop definition
    if(number_1 == 5){        // if condition
      continue;       // value 5 will be skipped
    }
    print(number_1);        // printing value of number_1
  }

//    TODO: Continue statement in Nested and Labelled For Loops
  outerLoop: for(int i = 1; i <= 3; i++){      // outer loop definition
    innerLoop: for(int j = 1; j <= 3; j++){   // inner loop definition
      if(i == 2 && j == 2){         // if condition
//          continue outerLoop;       // this will skip the value 2 2
        continue innerLoop;        // this will skip the value 2 2
      }
      print('$i $j');             // printing values of i and j
    }
  }
}
```

Functions/Methods in Dart:

Functions are defined as **"Collection of Statements Grouped Together to Perform an Operation"**.

Syntax:

```
return_type function_name (list_of_parameters) {

// body of function}
```

CODE OF FUNCTION:

```
void main(){
  function();                // function call
  int result = add(2, 3); // function call with 2 parameters and storing value
  print(result);             // printing result
}

//  TODO: Function with void type
  void function(){                // void function definition
    print("I am in Void Function");   // printing a line
  }

//  TODO: function definition with integer return type
  int add(int number1, int number2){
    return number1 + number2; // summation of 2 numbers and returning value
  }
```

Properties of Functions:

1. **Functions** in **Dart Programming Language** Are **Objects** which means **Functions** can be **Assigned** to a **Variable** or **Passed** as **Parameter** to other **Functions**.
2. All the **Functions** in **Dart Return** a **Value**.
3. If **No Return Value** is **Specified**, then **Function** returns **NULL**.
4. **Specifying Return Type** is **Optional** but it is recommended as **Per Code Conventions**.

Functions as Expressions/Expressions in Functions:

We can **Optimize** our **Functions** by using the concept of **FAT Arrow**.

Note: You can't use **Curly Braces** with **FAT Arrow**.

Syntax is:

```
return_type function_name(list_of_parameter) => expression;
```

CODE OF FUNCTION AS EXPRESSION/EXPRESSION IN FUNCTION:

```
void main(){
  function();        // function call
  print(add(1, 2));   // function call
}

//  TODO: Above functions with the concept of FAT ARROW
  void function() => print("I am written by using FAT Arrow Concept");
  int add(int number1, int number2) => number1 + number2;
```

Dart Optional Positional Parameters:

Parameters are of **Two** types:

1. **Required**
2. **Optional**

   Optional Parameters are further **Divided** into **Three Parameters**

   a. **Position**
   b. **Named**
   c. **Default**

Required Parameters:

You can't **Skip** all the **Parameters** You **Passed** to a **Function**.

2.a Position Parameters:

Syntax is:

Put your **Parameters** in **Square Brackets [ ]**. It will **Return Null Value** till you don't it.

2.b Optional Named Parameter:

Basically used to **Prevent Errors** if there are **Larger Number** of **Arguments**.

Syntax is:

Put **Arguments** in **Curly Brackets { }**.

2.c Optional Default Parameters:

You can **Assign Default Values** to **Parameters**.

Syntax is:

Put **Default Valued** argument in **Curly Brackets {}.**

CODE OF PARAMETERS IN FUNCTIONS:

```
void main(){
//  TODO:For Required Parameters
  printCities("Islamabad", "Karachi", "Rawalpindi");  // function call
  print("\n");      // printing a new line

//  TODO: For Optional Positional Parameter
  printCountries("Pakistan", "Iran", "Iraq");   // function call
  printCountries("Pakistan", "Iran");          // function call
  print("\n");       // printing a new line
// TODO: For Optional Named Parameters
  var result = findVolume(10, breadth: 3, height: 10);   // function call
  print(result);      // printing result
  print("\n");       // printing a new line
// TODO: For Optional Default Parameters
  var result_1 = find_Volume(10, 3);      // function call
  print(result_1);      // printing result
}

//  TODO: Function with Required Parameters Definition
  void printCities(String name1, String name2, String name3){
  print("Name 1 is: $name1");     // printing value
  print("Name 2 is: $name2");     // printing value
  print("Name 3 is: $name3");     // printing value
  }

//  TODO: Function with Optional Positional Parameters Definition
  void printCountries(String name1, String name2, [String name3]){
  print("Name 1 is: $name1");     // printing value
  print("Name 2 is: $name2");     // printing value
  print("Name 3 is: $name3");     // printing value
  }

//  TODO: Optional Named Parameters
  int findVolume(int length, {int breadth, int height}){
    return length * breadth * height;   // returning result
  }

//  TODO: Optional Default Parameters
  int find_Volume(int length, int breadth, {int height = 10}){
    return length * breadth * height;   // returning result
  }
```

Exception Handling in Dart:

- When **Normal Flow** of **Program** is **Disrupted** and the **Application Crashes**.
- Due to some **Exception** or **Bugs** in our **Code**.
- Some of the **Common Exceptions** are:
    - **Format Exception.**
    - **IO Exception.**
    - **IntegerDivisionByZero etc.**

How exception arises in our code:

For Example: If we **Divide** a **Number** by **Zero** then it will **Crash** our **Application**.

To handle Exception Handling we have many cases:

Case I (ON Clause):

When you **Don't Know** the **Exception Name**.

Syntax is:

```
try {
// body of try clause
} on exception_name {
// body of ON Clause
}
```

Case II (CATCH Clause):

When you **Don't Know** the **Exception**.

```
try {
// body of try clause
} catch (identifier) {
// body of catch clause
}
```

Case III (CATCH Clause with Exception Object & Stack Trace Object):

By using **STACK TRACE,** we can **Know What** are the **Events** occur **Before Exception** was **Thrown**.

Syntax is:

```
try {
// body of try clause
} catch (list of identifiers) {
// body of catch clause
}
```

Case IV (Custom Class Exception Handling):

In this case we **Define** a **Custom Class** for **Handling Exception** in our **Application**.

For Example:

Let take example of a **Bank** in which a **User** Can't **Deposit Negative Money**, so there should be an **Error Shown** to **User** that you can't **Enter Negative Number**.

CODE OF EXCEPTION HANDLING:

```
void main(){
//  TODO: How exception arises in our code
  int result = 12 ~/ 4;     // tilt operator to convert double value to integer

//  TODO: Case I: ON Clause(when you don't know the exception name)
  try{      // try clause
    int result = 12 ~/ 0;   // integer variable declaration & division
    print(result);    // displaying result
  }on IntegerDivisionByZeroException {     // on class with exception name
    print("Can't be divided by Zero");  // printing a string
  }

//  TODO: Case II: CATCH Clause(When you don't know the exception)
  try{    // try clause
    int result = 12 ~/ 0; // integer variable declaration & division
    print(result);    // displaying result
  }catch(e){      // catch clause with identifier
    print("The Exception thrown is $e");    // this will show the exception
  }
//  TODO: CATCH clause with Exception Object & Stack Trace Object.
    try{      // try clause
      int result =  12 ~/ 0;    // integer variable declaration & division
      print(result);    // displaying result
    }catch(e, s){     // catch class with identifiers
      print("STACK TRACE is: $s");  // printing a string
    }

//  TODO: Custom Exception Handling
  try{      // try clause
    depositMoney(-200);   // function call
  }catch(e){      // catch clause with identifier
    print(e.errorMessage());    // printing a message by function call
  }
}
```

```
// TODO: Custom Class Exception Handling
class depositException implements Exception{    // class declaration
  String errorMessage(){      // string type function definition
    return "You can't enter ammount less than 0";   // returning a string
  }
}

// TODO: Method Defined
void depositMoney(int money){   // function definition
  if(money < 0){      // if condition
    throw new depositException();   // trowing exception
  }
}
```

Class and Objects:

- A **Class** is an **Extensible Program-Code-Template** for **Creating Objects**, providing **Initial Values** for **State (Member Variables)** and **Implementation** of **Behavior (Member Functions or Methods)**.
- To **Define** a **Class**, the **Syntax** is:

```
class class_name {

// Body of Class

}
```

How to create objects of a class?

```
var object_name = new class_name ();
```

OR

```
var object_name = class_name ();
```

- Use **"."** **Operator** to **Access** the **Properties** of a **Class**.
- **Declared Properties** of a **Class** are known as **Instance Variables** or **Field Variable**.
- By **Default**, the **Value** of **Instance Variables** or **Field Variables** is **Null**.
- You can **Create** as many as **Objects** you want.

CODE OF CLASS AND OBJECTS:

```
void main() {
//  creating object First Method
  var student_1 = new Student(211, "Ahtesham");   // object declaration
  student_1.name = "Ahtesham";                  // assigning value
  student_1.id = 211;                     // assigning value
  print("ID = ${student_1.id} and Name = ${student_1.name}"); // print result
  student_1.study();                        // function call
  student_1.sleep();                        // function call
  print("");                                // printing a new line

//  Creating object Second Method
  var student_2 = Student(212, "Aamir");         // object declaration
  student_1.name = "Aamir";                      // assigning value
  student_1.id = 212;                            // assigning value
  print("ID = ${student_2.id} and Name = ${student_2.name}"); // print result
  student_1.study();                        // function call
  student_1.sleep();                        // function call

}

//  TODO: Declaration & Definition of a Class
class Student{
  int id;       // Instance/Field Variables
  String name;  // Instance/Field Variables

  Student(int id, String name){
    this.id = id;
    this.name = name;
  }
//  TODO: Functions definition
  void study(){     // function definition
    print("${this.name} is Studying");    // print statement
  }

  void sleep(){   // function definition
    print("${this.name} is Sleeping");    // print statement
  }
}
```

Constructors:

- **Constructors** always **Execute** before our other **Code** gets **Executed**.
- **Constructors** doesn't have any **Return Type**.
- **Constructors** are used to **Create Objects**.
- You can **Initialize** the **Instance/Fields variables** within the **Constructor**.
- You can't have **Default Constructor** & **Parameterized Constructors** at the same time.
- You can have as many as **Named Constructors** you want.

Types of Constructors:

Default Constructor:

- We can't see the **Default Constructors** in our **Code** but it is already **Declared** with our **Class**.
- You can also **Declare** your **Default Constructor**.
- **Syntax** is:

```
class_name(){
// Body of the default constructor
}
```

Parameterized Constructor:

- You can **Pass** a **List of Parameters** into a **Constructor**.
- Syntax is:

```
class_name(list_of_parameters){
        this.instance_name = instance_name;
        this.instance_name = instance_name;
        // and so on
}
```

- **Another** way of **Defining** a **Parameterized Constructor** is:

```
        class_name(this.instance_name, this.instance_name, ...);
```

Named Constructor:

- You can **Define** your **Own Constructor**.
- Syntax is:

```
class_name.constructor_name(){
// body of named constructor
}
```

- **Another** way of **Defining** a **Named Constructor** is:

```
class_name.constructor_name(this.instance_name, this.instance_name, ...);
```

CODE OF CONSTRUCTORS AND CONSTRUCTOR TYPES:

```
void main(){
//  TODO: For Parameterized Constructors
  var student_3 = Student(45, "Obaid"); // object declaration
  print("ID = ${student_3.id} and Name = ${student_3.name}");   // printing
//  TODO: For Named Constructor
  var student_4 = Student.myCustomConstructor();    // constructor call
  student_4.id = 54;      // assigning value to instance variable
  student_4.name = "Arshad";    // assigning value to instance variable
  print("ID = ${student_4.id} and Name = ${student_4.name}"); // printing
  var student_5 = Student.myAnotherCustomConstructor(56, "Asad");
  print("ID = ${student_5.id} and Name = ${student_5.name}");
}


//  TODO: Declaration & Definition of a Class
class Student{
  int id;       // Instance/Field Variables
  String name;  // Instance/Field Variables
//  TODO: Constructors Concepts
//  TODO: Default  Constructor
//  Student(){
//    print("This is our Defualt Constructor");
//  }
//  TODO: Parameterized Constructor
  Student(int id, String name){
    this.id = id;       // assigning value to instance variable
    this.name = name;   // assigning value to instance variable
  }
//  Another way of defining Parameterized Constructor is
//  Student (this.id, this.name);
//  TODO: Named Constructor
  Student.myCustomConstructor(){
    print("This is my Custom Named Constructor");
  }
//  Another way of defining Custom Constructor is
  Student.myAnotherCustomConstructor(this.id, this.name);
}
```

Summary:

- A **Class** is a **Blueprint** to create **Object**.
- You can **Define Constructors** in **Class**.
- You have **Default Constructor**, **Parameterized Constructors**, **Named Constructors**.
- In **Class**, you have **Special Type** of **Constructor** called **"Named Constructor"**.
- You can **Pass** as many as **Arguments** into a **Constructors** you need.
- **Variables** declare in a **Class** are known as **Instance/Field Variables**.
- **Variables** declare in a **Function** within a **Class** are known as **"Local Variables"**.
- **Defined Objects** are known as **"Reference Variable"**.
- **Reference variables** are used to create **Objects**.
- You can **Create** as many as **Classes** of anything.

Getter and Setter in Dart:

- In **Dart**, whenever we defined **Instance Variable**, then this **Variable** acts as **Default Getter** and **Setter**.
- The **Instance/Field Variables** acts as **Default Getter** and **Setter**.
- You can **Define** your own **Custom Getter** and **Setter**.

Declaration of Private Variable:

- You **Can't** declare **Private Variable** in **Dart**.
- You **Can't** have **Private** or **Public Keywords** in **Dart**.
- But you **Can** put **"_"** before any **Variable** to **make** it **Private** within its own **Library** but you **cannot** make it **Private** to its own **Class**.

CODE OF GETTER, SETTER, AND PRIVATE VARIABLE:

```
void main(){
//   TODO: Default Getter & Setters
  var student_1 = Student();      // declaration of object
//  For default getter and setter
  student_1.name = "Ahtesahm";      // assigning value to instance variable
  print(student_1.name);      // printing result

//    For Custom getter and setter
  student_1.percentage = 438.0;     // assigning value to instance variable
  print(student_1.percentage);
}

class Student{
//  Instance variables come with default getter & setter
  String name;      // instance variable
//  Defining Custom Getter and Setter
  double _percent;    // declaration of Private variable

  void set percentage(double marksSecured) => _percent = (marksSecured/500)*100;

  double get percentage => _percent;

}
```

Inheritance:

- **Inheritance** is a **Mechanism** in which **One Object** acquires the **Properties** of its **Parent Class Object**.
- **Super Class** of any **Class** is an **Object**.
- **Inheritance** provides **Default Implementation** of:
  - **toString()**, **Return** the **String Representation** of the **Object**.
  - **hashCode Getter**, **Returns** the **Hash Code** of an **Object**.
  - **Operator ==**, use to **Compare Two Objects**.

Advantages of Inheritance:

- **Code Reusability**.
- **Method Overriding.**
- **Cleaner Code, no repetition.**

CODE OF INHERITANCE:

```
void main(){
//  TODO: Inheritance
  var dog = Dog();       // object declaration of child class
  dog.breed = "Labredor"; // assigning value
  dog.color = "Black";    // assigning value
  dog.eat();       // function call of parent class
  dog.bark();      // function call
  print(dog.color);

  var cat = Cat();    // object declaration of child class
  cat.color = "White";    // assigning value
  cat.age = 2;        // assigning value
  cat.eat();       // function call of parent class
  cat.meow();      // function call

  var animal = Animal();  // object declaration of parent class
  animal.color = "Brown"; // assigning value
  animal.eat();     // function call
}

//  Parent Class
class Animal {      // parent class definition
  String color; // string declaration
  void eat() {    // method definition
    // function definition
    print('Eat'); // printing a string
  }
}

// Child Class
class Dog extends Animal {      // child class definition
//   inheriting properties
  String breed; // string declaration
  String color = "Black";     // declaration and assigning value

  void bark(){      // function definition
    print('Bark');  // printing a string
  }
}
//  Child Class/Sub Class
class Cat extends Animal{   // inheriting properties
  int age;       // integer variable declaration

  void meow(){    // function definition
    print('Meow');  // printing a string
 }
}
```

Method Overriding:

Method Overriding is a **Mechanism** by which a **Child Class Redefines** a **Method** in **Parent Class**.

CODE OF METHOD OVERRIDING:

```
void main(){
  var dog = Dog();    // object declaration
  dog.eat();        // method call
}

//  Parent Class
class Animal{       // class definition
  String color = "Brown";     // string declaration

//  TODO: For MethodOverriding
  void eat(){       // function definition
    print("Animal is eating");       // printing a string
  }
}

//  Child Class/Sub Class
class Dog extends Animal {      // child class definition
//   inheriting properties
  String breed; // string declaration
  String color = "Black";     // assigning value and declaration of variable

//  TODO: For method overriding
  void eat(){     // method definition
    super.eat();      // super keyword to call super class function
    print("Dog is eating");   // printing a string
  }
  void bark(){      // function definition
    print('Bark');  // printing a string
  }
}
```

Constructors in Inheritance:

In **Dart**, whenever we **Define** a **Constructor** in **Child Class**, it **Implicitly** called **Constructor** of **Super Class**. Remember that whenever we have **Child Class** and **Super Class** it is **Mandatory** that your **Super Class** should have **Zero Argumented Constructor**.

If your **Child Class** have an **Argumented Constructor,** then you have to **Manually** called **Super Class Constructor**.

When you **Write Named Constructor** in **Child Class**, then **Super Class** should have **Zero Argumented Constructor**. If it **Doesn't Have,** then you have to **Manually Define** it.

CODE OF CONSTRUCTORS IN INHERITANCE:

```
void main(){
//  TODO: For Constructors in Inheritance
//  TODO: For Default Constructor
//  var dog = Dog();         //object declaration

//  TODO: For Parameterized Constructor
//  var dog = Dog("Labredor");      // object declaration
//  var dog = Dog("Lebrador", "White"); // object declaration

// TODO: For Named Constructors
//  var dog = Dog.myNamedConstructor();   // object declaration
}

// TODO: For Constructors in Inheritance
class Animal{         // parent class definition
  String color;       // string varaible declaration
//  For Default Constructor
//  Animal(){         // constructor definition
//    print("This is Animal Class Default Constructor");   // printing a string
//  }

//  For Parameterized Constructor
  Animal(String color){         // constructor definition
    print("This is Animal Class Parameterized Constructor");
  }

//  For Named Constructor
  Animal.myNamedConstructor(){  // constructor definition
    print("This is Animal Class Named Constructor");
  }
}

class Dog extends Animal{   // child class definition
  String breed;      // string variable declaration
// Default Constructor
//  Dog(){        // constructor definition
//    print("This is Dog Class Default Constructor");
//  }

//  Dog():super(){      // constructor definition
//    print("This is Dog Class Default Constructor");
//  }

//  TODO: For Parameterized Constructor
  Dog(String breed, String color):super(color){   // constructor definition
    print("This is Dog class Parameterized Constructor");
  }

//  TODO: For Named Constructor
  Dog.myNamedConstructor(): super.myNamedConstructor(){
    print("This is Dog class Named Constructor");
  }
 }
```

Abstract Class:

To **Define Abstract Class**, use keyword **"abstract"** before keyword **"class". Abstract Class** can't be **Instantiated**; you **Can't Create Objects** of **Abstract class**. **Abstract Class** can have **Abstract Methods**,

Normal Methods, and **Instance/Field Variables**. Whenever you will **Extend** the **Sub Class** by **Abstract Class** then you have to **Override** the **Function** of the **Abstract Class**.

Abstract Methods:

Abstract Methods Only **Exist** in **Abstract Class**. To define **Abstract Method**, you just need to put **Semicolon** after **Parenthesis**.

CODE OF ABSTRACT CLASS AND ABSTRACT METHOD:

```
void main(){
//  var draw = Draw();
var circle = Circle();      // declaration of object of circle class
circle.draw();            // method call
}

abstract class Draw{      // abstract class
  void draw();          // abstract method
}

class Circle extends Draw{          // class definition
  void draw(){          // overriding abstract method
    print("Drawing");
  }
}
```

Interface:

- **Dart** does not have any **Special Syntax** to **Declare INTERFACE**.
- An **INTERFACE** is a **Normal Class**.
- An **INTERFACE** is **Used** when you **Need Concrete Implementation** of **All** of its **Functions within** its **Sub Class**.
- It is **mandatory** to **Override** all **Methods** in the **Implementing Class**.
- You can **Implement Multiple Classes** but you **Cannot Extend Multiple Classes** during **Inheritance**.

CODE OF INTERFACE:

```
void main(){
  var remote = Remote();    // object declaration
  remote.volumeUp();       // method call
  remote.volumeDown();      // method call
  var tv = Television();    // object declaration
  tv.volumeUp();           // method call
  tv.volumeDown();         // method call
}

class Remote{             // class definition
  void volumeUp(){         // method declaration & definition
    print("_____Volume Up from Remote_____");   // printing a string
  }

  void volumeDown(){       // method declaration & definition
```

```
    print("_____Volume Down from Remote_____"); // printing a string
  }
}

class AnotherClass{         // class definition
  void anotherMethod(){     // method declaration & definition
//    Code
  }
}

class Television implements Remote, AnotherClass{   // implementing interface
  void volumeUp(){        // overriding method of remote class
    print("_____Volume Up in Television_____");
  }

  void volumeDown() {     // overriding method of remote class
    print("_____Volume Down in Television_____");
  }

  void anotherMethod(){     // overriding method of AnotherClass
//    Code
  }
}
```

Functional Programming in Dart:

Lambda:

- A **Function** without a **Name**.
- **Lambda** is also known as **Anonymous Function**.
- Remember that **Function** in **Dart** is an **Object**. For Example:
  - `int sum = 4;     // object`
  - `String message = "Hello";        // object`
  - `Function addNumber = //some value; // object`

To Declare a Lambda Function:

```
Function VariableName = (List of Parameters){

// Code of Lambda Function

}
```

Note: **Function** is a **Class**.

CODE OF LAMBDA FUNCTION:

```
void main(){
  addSum(10, 20);     // function call
/*
*   To define a Lambda Function/Expression, there are two ways
* */

//  First way of defining Lambda Function/Expression
Function addTwoNumbers = (int num1, int num2){
  var sum = num1 + num2;  // calculating sum
  print(sum);         // printing result
};

//  To call a Lambda Function/Expression
  addTwoNumbers(20, 50);

//  Second Way of defining Lambda Function/Expression
  var multiplyByTwo = (int number) => print(2*number);

//  To call a Lambda Function/Expression
  multiplyByTwo(2);
}

//  Normal Function Definition
void addSum(int value1, int value2){
  var sum = value1 + value2;    // declaration and assigning result
  print(sum);       // printing result
}
```

Higher Order Functions:

- A **Function** that can **Accept** another **Function** as a **Parameter**.
- A **Function** that can **Return** a **Function**.
- A **Function** that can do **both**.

CODE OF HIGHER-ORDER-FUNCTIONS:

```
void main(){
//  Defining Lambda Function/Expression
  Function addNumbers =
      (int number_1, int number_2) => print((number_1 + number_2));
//  Calling Higher-Order-Function
  /*
  * 1. Calling Higher-Order-Function.
  * 2. Passing a String "Hello".
  * 3. Calling Lambda Function/Expression
  * */
  someOtherFunction("Hello", addNumbers);   // calling function



//  Calling Higher-Order-Function
  var myFunc = taskToPerform(); // declaration of Variable & Function call
  print(myFunc(10));             // passing value to Function
  /*
  * At Run-Time:
  * 1. myFunc will become multiplyByTwo
  * 2. number * 2
  * 3. 10 * 2
  * 4. Output: 20
  * */
}

void someOtherFunction(String message, Function myFunction){
  print(message);                // printing message
//  Calling Function Parameter
  myFunction(2,4);               // function call
  /*
  * At Run-time:
  * 1. addNumbers(2,4)
  * 2. print(number_1 + number_2)
  * 3. print(2 + 4)
  * 4. Output: 6
  * */
}

Function taskToPerform(){
//  Defining Lambda Function/Expression
  Function multiplyByTwo = (int number) => number*2;
//  Returning Lambda Function
  return multiplyByTwo;   // returning function
}
```

Closure Function:

- It is a **Special Function**.
- Within in a **Closure Function** you can **mutate** (**modify**) the **value** of **variables** present in the **Parent Scope**.
- A **Closure** is a **Function** that has **access** to the **Parental Scope**, even after the **Scope** has been **closed**.
- A **Closure** is a **Function** that has **access** to **Variables** in its **Lexical Scope**, even when the **Function** is **used Outside** of its **Original Scope**.
- By word **Parent Scope**, we mean **main () method** of the **Program**.

Code for Closure Function:

```
void main(){
  String message = "Dart is Good";    // string variable declaration & initialization
  Function showMessage = (){          // lambda function definition
    message = "Dart is Awesome";      // assigning value to string
    print(message);                   // printing value
  };
  showMessage();                      // function call

  Function talk = (){                 // function definition
    String msg = "Hi";                  // string variable declaration & initialization
    Function say = (){                // lambda function definition
      msg = "Hello";                  // assigning value to variable
      print(msg);                   // displaying value
    };
    return say;                     // returning function
  };
  var speak = talk();                // variable declaration & initialization
  speak();                        // function call
}
```

List:

- In **Dart**, **Array** is known as **LIST**.
- **List** is **Ordered Collection**.
- **Elements** are **Ordered** in a **Sequence**.
- **Default Value** of **Elements** is **NULL**.
- Each **Element** in **List** contains **Address** called **Index** of **Element**.
- **Index** of **Element Start** with **0**.

Types of List:

There are **Two** types of **List**.

1. **Fixed Length List.**
2. **Grow-able List.**

Fixed Length List:

- **Once** the **Length** is **Defined**, you **Can't Change** it.
- You can't apply **List Operations** on the **Fixed Length List** such as **add ()**, **remove ()**, **clear ()** etc.

Syntax of defining Fixed Length List is:

```
List<type_parameter> list_name = List(length);
```

Type_Parameter may be **Integer**, **Double**, **String**, **Boolean Data Types**.

Code of Fixed Length List:

```
void main(){
//  TODO: Fixed Length List
  List<int> numberList = List(5);   // declaration of Fixed Length list
  numberList[0] = 23;     // assigning value
  numberList[3] = 85;     // assigning value
  numberList[4] = 100;    // assigning value

//  TODO: Method 1: To access List Elements(MANUALLY)
  print(numberList[0]);   // output will be 23
  print(numberList[1]);   // output will be null
  print(numberList[2]);   // output will be null
  print(numberList[3]);   // output will be 85
  print(numberList[4]);   // output will be 100

  print("\n");      // printing new line

//  TODO: Method 2: To access List Elements (FOR LOOP)
  for(int element in numberList){     // body of for-in loop
```

```
    print(element);     // printing out elements of list
  }

  print("\n");    // printing new line

//  TODO: Method 3: To access List Elements(FOR-EACH LOOP)
  numberList.forEach((element)=> print(element));// printing out elements of list
  print("\n");  // printing new line

//  TODO: Method 4: To access List Elements(PRIMITIVE FOR LOOP)
  for(int index = 0; index < numberList.length; index++){
    print(numberList[index]);      // printing out elements of list
  }
  print("\n");
}
```

Grow-able List:

- **Grow-able List Increase** or **Decrease** as your **Per-Requirements**.
- It is **Extendable List**.
- By **Default**, **List** is **Empty**.

Syntax of defining Grow-able list is:

```
List list_name = List();
```

Another Way is if you **Know** some **Values**:

```
List list_name = [values];
```

Operations on Grow-able List:

To add elements in the list, use add () method. Syntax is:

```
list_name.add(value);
```

To remove elements from the list, use remove(value) method with value. Syntax is:

```
list_name.remove(value);
```

You can remove element from the list by using index number, use removeAt () method. Syntax is:

```
list_name.removeAt(index_value);
```

To update element of the list, the syntax is:

```
list_name[index_value] = "value";    // for string

list_name[index_value] = value;     // for integer & double
```

Code of Grow-able List:

```
void main(){
//  TODO: Grow-able List
  List numberList = List();     // declaration of grow-able list
  List countryList = ["Pakistan", "USA", "UK"];   // Another declaration
  numberList.add(73);     // adding element to the list
  numberList.add(67);     // adding element to the list
  numberList.add(90);     // adding element to the list
  numberList.add(99);     // adding element to the list

  numberList[0] = 200;      // updating element in the list
  numberList.remove(99);    // removing element 99
  numberList.removeAt(1);   // removing element at index 1
//  numberList.clear();       // removing all the elements in the list
//  TODO: Method 1: To access List Elements(MANUALLY)
//  print(numberList[0]);   // output will be 23
//  print(numberList[1]);   // output will be null
//  print(numberList[2]);   // output will be null
//  print(numberList[3]);   // output will be 85
//  print(numberList[4]);   // output will be 100

  print("\n");      // printing new line

//  TODO: Method 2: To access List Elements (FOR LOOP)
  for(int element in numberList){     // body of for-in loop
    print(element);     // printing out elements of list
  }

  print("\n");    // printing new line

//  TODO: Method 3: To access List Elements(FOR-EACH LOOP)
  numberList.forEach((element)=> print(element));// printing out elements of list

  print("\n");  // printing new line

//  TODO: Method 4: To access List Elements(PRIMITIVE FOR LOOP)
  for(int index = 0; index < numberList.length; index++){
    print(numberList[index]);      // printing out elements of list
  }
  print("\n");
}
```

Set:

- **Unordered Collection** of **Unique Items**.
- It **Doesn't** contain **Duplicate Elements**.
- You **Can't** get **Elements** by **INDEX**, since the **Items** are **Unordered**.

Declaration of Set:

To **Define** a **Set**, **Syntax** is:

Method 1:

`Set<Data Type> Set_Name = Set.from([Item1, Item2, Item3, ...]);`

Method 2:

`Set<Data Type> Set_Name = Set();`

To access the elements of the set:

Set use the **Same Methods** of **Accessing Elements** as of **List** except of **Index Method** because **Set** contains **Unordered Elements**.

Operations on Set:

You can perform many **Operations** on **Set** as like **Grow-able List** like;

- **remove(value).**
- **contain(value).**
- **add(value).**
- **isEmpty.**
- **length.**
- **clear() and much more.**

Code of Set:

```
void main(){
// Method 1: To define a Set
//Set<String> countriesSet = Set.from(["Pakistan", "Australia", "Saudia"]);

// Method 2: To define a Set
  Set<int> numberSet = Set();
  numberSet.add(73);        // adding 73 to set
  numberSet.add(61);        // adding 61 to set
  numberSet.add(79);        // adding 79 to set
  numberSet.add(99);        // adding 99 to set
  print("\n");       // printing new line



//  Operations on Set
  print("Length of Set: ${numberSet.length}");    // output will be 4
  print(numberSet.contains(61));    // output will be true
  print(numberSet.contains(101));   // output will be false
  numberSet.remove(73);          // remove 73 from set
  print(numberSet.isEmpty);     //output will be false
  print(numberSet.isNotEmpty);  // output will be true
//  numberSet.clear();            // it will clear all the elements

//  TODO: Method 2: To access List Elements (FOR LOOP)
  for(int element in numberSet){     // body of for-in loop
    print(element);     // printing out elements of list
  }

  print("\n");    // printing new line

//  TODO: Method 3: To access List Elements(FOR-EACH LOOP)
  numberSet.forEach((element)=> print(element));// printing out elements of list

}
```

Hash-Set:

- **Implementation** of **Unordered Set**.
- It is **Based** on **Hash Table** based on **Set Implementation**.

Map:

- It is **Unordered Collection** of **Key-Value Pairs**.
- **Key Value** can be of any **Object Type**.
- The **Value** can be **Repeated**.
- Each **Key** in the **Map** should be **Unique**.
- Commonly called as **Hash** or **Dictionary**.
- **Size of Map** is not **Fixed;** it can be **Increased** or **Decreased** as **Per the Number of Elements**.

Hash-Map:

- **Implementation** of **Map Class**.
- **Based** on **Hash-Table** that is **Each Element** is **Identified** by its **Hash Value**.

Syntax to define a Map:

```
Map<key(Data-type), value> mapName = Map();
```

Another Way to implement the Map is in terms of Literals.

```
Map<key(Data-Type), value> mapName = {

key:value,

key:value,

key:value

// and so on };
```

To add values in Map:

```
mapName[key] = value;
```

To read values from Map:

```
mapName[key];
```

To read all keys from Map:

```
for(var/Data-Type key in mapName.keys){

    print(key);

}
```

To read all values from Map:

```
for(var/Data-Type value in mapName.value){

print(value);

}
```

To print Keys and Values at the same time:

- **Done** in **Terms** of **Lambda Expression**.

```
mapName.forEach(key, value) => print($key $value);
```

Operations on Map:

Different **Operations** can be performed on **Map** like;

- **containKey(key);**
- **update(key, value here will be updated in terms of Lambda Expression);**
- **remove(key);**
- **isEmpty;**
- **clear();**
- **length and much more.**

Code of Map/Hash Map:

```
void main(){
//  Declaration of Map
  Map<String, String> fruits = Map();
  fruits["apple"] = "Green";     // adding key and value
  fruits["orange"] = "Orange";   // adding key and value
  fruits["grapes"] = "Purple";   // adding key and value
  fruits["guava"] = "Yellow";    // adding key and value

//  To Access element of Map
```

```
  print(fruits["apple"]);         // output will be Green
  print(fruits["orange"]);        // output will be Orange
  print(fruits["grapes"]);        // output will be Purple
  print(fruits["guava"]);         // output will be Yellow

//  To Access keys in Map
  print('\nKeys in Map are:');      // printing a string
  for(String key in fruits.keys){   // for-in loop to print keys
    print(key);                   // printing keys
  }

//  To Access values in Map
  print("\nValues in Map are:");    // printing a string
  for(String value in fruits.values){   // for-in loop to print values
    print(value);         // printing values
  }

//  To access keys and value at the same time
  print("\nTo Access Keys and Values from Map at the Same Time:");    // printing a string
  fruits.forEach((key, value) =>
      print("Key is: $key and Value is: $value"));  // printing keys and values

//  Another way to define Map
  Map<String, int> countryDialCodes = {
    "Pakistan" : 92,    // assigning key and value
    "India" : 91,       // assigning key and value
    "USA" : 1           // assigning key and value
  };

//  To Access element of Map
  print(countryDialCodes["Pakistan"]);  // output will be 92
  print(countryDialCodes["India"]);    // output will be 91
  print(countryDialCodes["USA"]);      // output will be 1

//  To Access keys in Map
  print('\nKeys in Map are:');    // printing a string
  for(String key in countryDialCodes.keys){   // for-in loop to print keys
    print(key);     // printing keys
  }
//  To Access values in Map
  print("\nValues in Map are:");    // printing a string
  for(int value in countryDialCodes.values){  // for-in loop to print values
    print(value);   // printing values
  }

//  To access keys and value at the same time
  print("\nTo Access Keys and Values from Map at the Same Time:");    // printing a string
  countryDialCodes.forEach((key, value) =>
      print("Key is: $key and Value is: $value"));    // printing keys and values

//  Operations on Maps
  print(fruits.containsKey("apple")); // output will be true
  print(countryDialCodes.isNotEmpty);   // output will be true
  print(fruits.isEmpty);              // output will be false
  countryDialCodes.remove("India");   // this will remove India key and value
  print(fruits.length);               // output will be 4
  fruits.clear();                     //  this will clear all values
}
```

Callable Class:

- When **Dart Class** is called like a **Function**.
- It **Implements** the **call() Function**.
- You can **Pass** as many as **Arguments** to **call() Function** you want.
- While **Returning** some **Value** from **call() Function**, you must need to **Mention Return Type**.

Code of Callable Class:

```
void main(){
//  Object declaration
  var personOne = Person();     // object declaration
//  personOne();  // this will call the whole class

//  call method with passing arguments
//  personOne(23, "Ahtesham");    // this will call the whole class

// call method with return type
  var msg = personOne(23, "Ahtesham");  // variable declaration and initialization
  print(msg);     // printing result
}

//  Class Definition
class Person{
//  Call method declaration and definition
//  call(){     // call method definition
//  }

//  Call method with passing arguments
//  call(int age, String name){ // call method definition
//    print("Name: $name Age: $age");   // printing result
//  }

//  Call method with return type
  String call(int age, String name){      // call method definition
    return "Name: $name Age: $age";       // returning string
  }
}
```