



Fundamentals of Dart

What is dart?

→ Dart is a client-optimized, object-oriented, modern programming language to build apps fast for many platforms like android, iOS, web, desktop, etc. Client optimized means optimized for crafting a beautiful user interface and high-quality experiences. Google developed Dart as a programming language.

- Free and open-source.
- Object-oriented programming language.
- Used to develop android, iOS, web, and desktop apps fast.
- Can compile to either native code or javascript.
- Offers modern programming features like null safety and asynchronous programming.
- You can even use Dart for servers and backend.

→ Dart has many similarities to other languages such as Java, C#, Swift and Javascript. Some of its features include:

- Statically typed
- Type inference
- String expressions
- Multi-paradigm including object-oriented and functional programming
- Null safe

Difference Between Dart & Flutter

→ **Dart** is a client optimized, object-oriented programming language. It is popular nowadays because of flutter. It is difficult to build complete apps only using Dart because you have to manage many things yourself.

→ **Flutter** is a framework that uses dart programming language. With the help of flutter, you can build apps for android, iOS, web, desktop, etc. The

framework contains ready-made tools to make apps faster.

→ `main` function is the starting point where the execution of your program begins.

```
void main (){  
  //void is return type, which means method  
  //wont return anything back.  
  //main is the name of the method  
}
```

Comment

→ Comments may be used for noting something on the code, creating a reminder or explaining the code base.

```
// This is a single-line comment.  
  
/*  
  This is a multi-line comment block.  
  This is useful for long  
  comments that span several lines.  
*/  
  
/// Documentation comments for your code.
```

Identifiers

→ Identifiers are names given to elements in a program like variables, functions etc. Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.

- Identifiers cannot include special symbols except for underscore (`_`) or a dollar sign (`$`).
- Identifiers cannot be keywords.
- They must be unique.

- Identifiers are case-sensitive.
- Identifiers cannot contain spaces.

Variables and Data Types

→ A variable is “a named space in the memory” that stores values. In other words, it acts a container for values in a program.

→ Dart is **statically typed**, meaning that each variable in Dart has a type that must be known when you compile the code. The variable type cannot change when you run the program

→ Variable Types

- **String**: For storing text value. E.g. “John” [Must be in quotes]
- **int**: For storing integer value. E.g. 10, -10, 8555 [Decimal is not included]
- **double**: For storing floating point values. E.g. 10.0, -10.2, 85.698 [Decimal is included]
- **num**: For storing any type of number. E.g. 10, 20.2, -20 [both int and double]
- **bool**: For storing true or false. E.g. true, false [Only stores true or false values]
- **var**: For storing any value. E.g. ‘String’, 12, ‘z’, true

```
//Basic Types in Dart
void main (){
  var number = 15; //assign 15 to variable
  num age = 20; // used to store any types of numbers
  num height = 5.9;
  int integerNumber = 15; //int: Integers
  //double: Floating-point numbers
  double doubleNumber = 14.56;
  bool isRight = true; //bool: True or false
  //String: Sequences of characters
```

```
String stringValue = "Character set";  
}
```



`int` and `double` both derive from a type named `num`.

→ Round Double Value To 2 Decimal Places : The `.toStringAsFixed(2)` is used to round the double value up to 2 decimal places in dart. You can round to any decimal places by entering numbers like 2, 3, 4, etc.

```
// Declaring Variables  
double price = 1130.2232323233233; // valid.  
print(price.toStringAsFixed(2));
```

→ Dart also uses the dynamic variable definitions such as `var` and `dynamic`.

- Variables with dynamic and var keyword, can take any value such as int, string, bool and other types.
- There is a one key difference between var and dynamic. When variable with var keyword is assigned to a type, it can not be changed. However, type of variable with dynamic keyword can be changed afterwards.

```
var variableOne = "Name"; //String  
variable = 256; // Creates error  
  
dynamic variableTwo = "Name"; //String  
variable = 256; // Does not create any error
```

Numbers

→ The **num** type is inherited by the **int** and **double** types. The **int** data type is used to represent whole numbers. **double** – 64-bit (double-precision) floating-point numbers.

→ The **parse()** static function allows parsing a string containing numeric literal into a number.

Property & Description	
hashCode	Returns a hash code for a numerical value.
isFinite	True if the number is finite; otherwise, false.
isInfinite	True if the number is positive infinity or negative infinity; otherwise, false.
isNan	True if the number is the double Not-a-Number value; otherwise, false.
isNegative	True if the number is negative; otherwise, false.
sign	Returns minus one, zero or plus one depending on the sign and numerical value of the number.
isEven	Returns true if the number is an even number.
isOdd	Returns true if the number is an odd number.

Method & Description
abs Returns the absolute value of the number.
ceil Returns the least integer no smaller than the number.
compareTo Compares this to other number.
Floor Returns the greatest integer not greater than the current number.
remainder Returns the truncated remainder after dividing the two numbers.
Round Returns the integer closest to the current numbers.
toDouble Returns the double equivalent of the number.
toInt Returns the integer equivalent of the number.
toString Returns the string equivalent representation of the number.
truncate Returns an integer after discarding any fractional digits.

String

→ The String data type represents a sequence of characters.

```
String variable_name = 'value';
```

```
String variable_name = "value";
```

```
String variable_name = '''line1
line2''';
```

```
String variable_name = """line1
line2""";
```

→ In Dart, you can use the `+` operator or use **interpolation (\$)** to concatenate the String

```
print(firstName + " " + lastName+".");
print("Interpolation, $firstName $lastName.");
```

```
// Reversing the string
print("${input.split('').reversed.join()}");
```

Property & Description
codeUnits Returns an unmodifiable list of the UTF-16 code units of this string.
isEmpty Returns true if this string is empty.
Length Returns the length of the string including space, tab and newline characters.

Methods & Description
toLowerCase() Converts all characters in this string to lower case.
toUpperCase() Converts all characters in this string to upper case.
trim() Returns the string without any leading and trailing whitespace.
compareTo() Compares this object to another.
replaceAll() Replaces all substrings that match the specified pattern with a given value.
split() Splits the string at matches of the specified delimiter and returns a list of substrings.
substring() Returns the substring of this string that extends from startIndex, inclusive, to endIndex, exclusive.
toString() Returns a string representation of this object.
codeUnitAt() Returns the 16-bit UTF-16 code unit at the given index.

Bool

→ It is called boolean, which takes true or false.

```
bool isAdult = true;
bool isMarried = false;
```

Runes

→ With runes, you can find Unicode values of String.

```
String value = "a";  
print(value.runes);
```

Runtime Type

→ you can check runtime type in dart with `.runtimeType` after the variable name.

```
var a = 10;  
print(a.runtimeType);  
print(a is int); // true
```

Scope

→ The scope is a concept that refers to where values can be accessed or referenced.

→ Method scope : If you created variables inside the method, you can use them inside the method block but not outside the method block.

→ Global Scope : You can define a variable in the global scope to use the variable anywhere in your program.

→ Define your variable as much as close **Local** as you can. It makes your code clean and prevents you from using or changing them where you shouldn't.

Operators

→ Some examples of Dart's operators include:

- arithmetic
- equality
- increment and decrement
- comparison
- logical



Dart also allows for **operator overloading**.

```
void main(){

  //Arithmetic Operators
  40 + 2; // +, addition
  44 - 2; // -, subtraction
  21 * 2; // *, multiplication
  84 / 2; // /, dividing
  50 % 8; // %, modulo
  70 ~/5; // Divide, returning an integer result

  //Equality Operators
  42 == 43; // Equals?
  42 != 43; // Not equals?

  //Comparison Operators
  42 < 43; //Less than (<)
  42 > 43; //Greater than (>)
  42 => 43; //Equal and greater than (=>)
  42 =< 43; //Equal and less than (=<)

  // Type Test Operators
  variable is int // True if it has specified type
  variable is! int // False if it doesn't have specified type

  //Arithmetic/assignment operators
  var value = 42.0;
  value += 1; print(value); // 43.0
  value -= 1; print(value); // 42.0
  value *= 2; print(value); // 84.0
  value /= 2; print(value); // 42.0
```

```
//short form of +=1 -> ++ var++, ++var
//short form of -=1 -> -- var--, --var

//Logical Operators
(41 < 42) && (42 < 43); // && -> and logical operator
(41 < 42) || (42 > 43); // || -> or logical operator
!(41 < 42);           // ! -> not logical operator

//String Operations - It can be both defined
//with single and double quotes
var firstName = 'Albert';
String lastName = "Einstein";

//${expression}
var statement = "$firstName $lastName likes number ${84 / 2}"

//Escaping Strings
"\n" //for a new line
"\t" //For new tab
//If there are special characters, use \ to escape them
"you can\'t explain"

r"If you can't come\nyou don't need to pay."
//If you need to show escape sequences within the string,
//you can use raw strings, which are prefixed by r
}
```

Immutability

→ Dart uses the keywords `const` and `final` for values that don't change. Use `const` for values that are known at compile-time. Use `final` for values that don't have to be known at compile-time but cannot be reassigned after being initialized.

→ Dart throws an exception if an attempt is made to modify variables declared with the **final** or `const` keyword.

→ You can explicitly state the type with either `final` or `const`

```
const speedOfLight = 299792458;

final planet = 'Jupiter';

planet = 'Mars'; // error: planet can only be set once

final String moon = 'Europa';

print('$planet has a moon, $moon');
// Jupiter has a moon, Europa
```

Nullability

→ Dart *guarantees* that a non-nullable type will never contain a null value. This is known as null safety.

→ To tell Dart that you want to allow the value `null`, add `?` after the type.

```
String variable = null;
print(middleName); // gives null-point error

String? variable = null;
print(variable); // null

//The default for a nullable type is null.
String? variable;
print(variable) // null
```

Null Aware Operators

→ The double-question-mark operator, `??`, returns the left-hand operand if the object isn't null. Otherwise, it returns the right-hand value:

```
var variable;  
var name = variable ?? 'none';  
print(name); // none
```

→ The `?.` operator protects you from accessing properties of null objects. It returns `null` if the object itself is null. Otherwise, it returns the value of the property on the right-hand side.

```
print(variable?.length); // null
```

Conditionals

If Else Statement

→ If-else is used as decision mechanism

```
if (variable?.length == 11) {  
    print("Length of identification number matches");  
}  
else{  
    print("Length of identification number does not match");  
}  
  
// Multiple if else conditions can be used  
if (variable?.length. == 11) {  
    print('This is a identification number.');} else if (variable?.length. == 10) {  
    print('This is a phone number.');} else {  
    print('This number is unknown.');}
```

→ Ternary operator

```
condition ? exprIfTrue : exprIfFalse
```

```
var variable = number > 10 ? "More than 10", "Less than 10";
```

Switch Case

→ The switch statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case.

- There can be any number of case statements within a switch.
- The case statements can include only constants. It cannot be a variable or an expression.
- The data type of the variable_expression and the constant expression must match.
- Unless you put a break after each block of code, the execution flows into the next block.
- The case expression must be unique.
- The default block is optional.

```
var grade = "A";
switch(grade) {
    case "A": { print("Excellent"); }
    break;

    case "B": { print("Good"); }
    break;

    case "C": { print("Fair"); }
    break;

    case "D": { print("Poor"); }
    break;

    default: { print("Invalid choice"); }
```

```
    break;
}
```

Assert

→ It takes conditions as an argument. If the condition is true, nothing happens. If a condition is false, it will raise an error.

```
assert(condition);
// or
assert(condition, "Your custom message");
```

Loops

While Loop

```
while(condition){
    //statement(s);
    // Increment (++) or Decrement (--) Operation;
}

while (i < 10) {
    print(i);
    i++;
}
```

Do-while Loop

→ In a do-while loop, the statements will be executed at least once time, even if the condition is false. It is because the statement is executed before checking the condition.

```
do{
    statements;
}while(condition);
```

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

For Loop

```
for(initialization; condition; increment or decrement){  
    statements;  
}  
  
for (var i = 1; i <= 10; i++) {  
    sum += i;  
}
```

For Each Loop

```
collection.forEach(void f(value));  
  
List<String> footballplayers=['Ronaldo', 'Messi', 'Neymar'];  
footballplayers.forEach( (names)=>print(names));
```

For in Loop

```
for(String variable in collection){  
    statements;  
}  
  
for(var codePoint in name.runes){  
    print("Unicode of");  
    print("${String.fromCharCode(codePoint)} is $codePoint.");  
}
```

Continue - Break

→ **continue**: Skips remaining code inside a loop and immediately goes to the next iteration.

```
continue;

for(num = 0; num <= 20; num++) {
    if (num % 2 == 0) {
        continue;
    }
    count++;
}
```

→ **break**: Stops the loop and continues execution after the body of the loop.

```
break;

i = 1;
do {
    print(i);
    if (i == 5) {
        break;
    }
    ++i;
} while (i < 10);
```

Collections

→ Collections are useful for grouping related data.

List

→ Lists are zero-based, so the first item in the list is at index 0.

```
// Integer List
List<int> ages = [10, 30, 23];
// String List
List<String> names = ["Raj", "John", "Rocky"];
```



```

// Mixed List
var mixed = [10, "John", 18.8];

// Mutable List
List<String> names = ["Raj", "John", "Rocky"];
// Immutable List
const List<String> names = ["Raj", "John", "Rocky"];

// Growable list, size can increase
List variables = [42, -1, 299792458, 100];
var variable = variables[1]; // Accessing a list
variables.add(5); // Adding a element to a list
variables.remove(-1); // Removing a element from a list
print(variables); // Prints a list

//Growable list, size can increase
var variables = new List();
variables.add(12);
variables.add(13);

// Not growable list
var variables = new List(3);
variables[0] = 12;
variables[1] = 13;
variables[2] = 11;

// Going thorough every element of a list
for (var dessert in desserts) {
  print('I love to eat $dessert. ');
}

// Conditions in List
bool sad = false;
var cart = ['milk', 'ghee', if (sad) 'Beer'];

// Where in list
List<int> numbers = [2,4,6,8,10,11,12,13,14];

```

```
List<int> even =
    numbers.where((number) => number.isEven).toList();
```

Method	Description
remove()	Removes one element at a time from the given List.
removeAt()	Removes an element from the specified index position and returns it.
removeLast()	Remove the last element from the given List.
removeRange()	Removes the item within the specified range.

add()	Add one element at a time and returns the modified List object.
addAll()	Insert the multiple values to the given List, and each value is separated by the commas and enclosed with a square bracket ([]).
insert()	Provides the facility to insert an element at a specified index position.
insertAll()	Insert the multiple value at the specified index position.

Methods & Description
first Returns the first element in the list.
isEmpty Returns true if the collection has no elements.
isNotEmpty Returns true if the collection has at least one element.
length Returns the size of the list.
last Returns the last element in the list.
reversed Returns an iterable object containing the lists values in the reverse order.
Single Checks if the list has only one element and returns it.

Map

→ When you want a list of paired values, `Map` is a good choice.

→ Elements of a map are called **key-value pairs**, where the key is on the left of a colon and the value is on the right.

→ If a map doesn't contain the key that you're looking up, it'll return a `null` value.

```
// Creating a Map
Map<String, String> people = {
  'name': 'Chase',
  'middleName': 'John',
  'surname': 'Brook',
};

// Creating a Map
Map<String, int> calories = {
  'cake': 500,
  'donuts': 150,
  'cookies': 100,
};

// Accessing a element from map
var variable = people['name'];
// Adding a element to a map
calories['brownieFudgeSundae'] = 1346;

// Creating a Map
var details = new Map();
details[key] = value;
details[key] = value;
```

Properties	Work
keys.toList()	Convert all Maps keys to List.
values.toList()	Convert all Maps values to List.
containsKey('key')	Return true or false.
containsValue('value')	Return true or false.
clear()	Removes all elements from the Map.
removeWhere()	Removes all elements from the Map if condition is valid.

Property & Description
Keys Returns an iterable object representing keys
Values Returns an iterable object representing values
Length Returns the size of the Map
isEmpty Returns true if the Map is an empty Map
isNotEmpty Returns true if the Map is an empty Map

Function Name & Description
addAll() Adds all key-value pairs of other to this map.
clear() Removes all pairs from the map.
remove() Removes key and its associated value, if present, from the map.
forEach() Applies f to each key-value pair of the map.

Sets

→ An unordered collection of unique items is called set in dart. You can store unique data in sets.

→ The list allows you to add **duplicate items**, but the Set doesn't allow it.

```
Set <variable_type> variable_name = {};
```

```
Set<String> weekdays = {"Mon", "Tue", "Wed", "Thu", "Fri"};
```

Method	Description
clear()	Removes all elements from the Set.
difference()	Creates a new Set with the elements of this that are not in other.
elementAt()	Returns the index value of element.
intersection()	Find common elements in two sets.

Properties	Work
first	To get first value of Set.
last	To get last value of Set.
isEmpty	Return true or false.
isNotEmpty	Return true or false.
length	It returns the length of the Set.

Method	Description
add()	Add one element to Set.
remove()	Removes one element from Set.
addAll()	Insert the multiple values to the given Set.

Where in Dart

→ You can use where in list, set, map to **filter specific items**. It returns a new list containing all the elements that satisfy the condition. This is also called **Where Filter** in dart.

```
Iterable<E> where(
  bool test(
    E element
  ))

List<int> numbers = [2, 4, 6, 8, 10, 11, 12, 13, 14];
List<int> oddNumbers =
    numbers.where((number) => number.isOdd).toList();
```

Enumeration

→ An enumeration is used for defining named constant values. An enumerated type is declared using the **enum** keyword.

```
// How to define enumeration
enum enum_name {
  enumeration list
}

// Example of enum
```

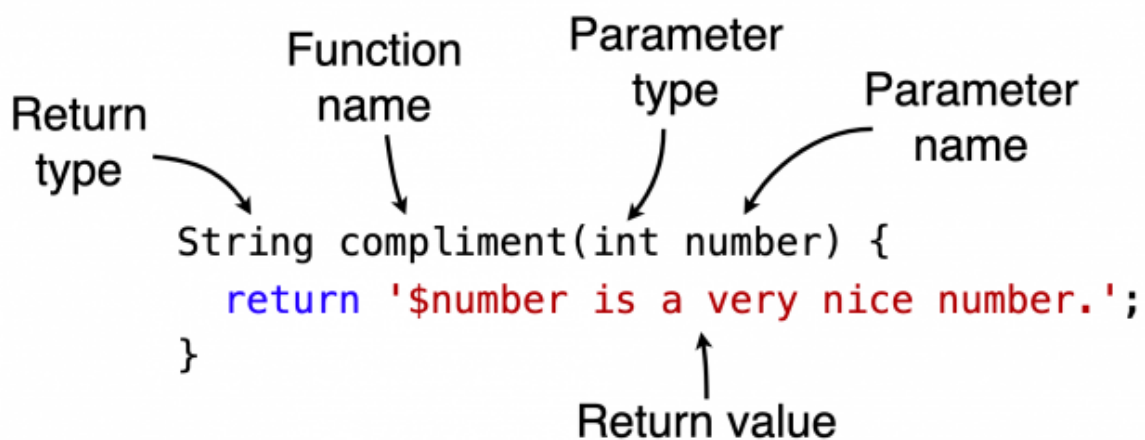
```
enum Status {
  none,
  running,
  stopped,
  paused
}

// Printing every value of the enum
Status.values.forEach((v) =>
    print('value: $v, index: ${v.index}'))
```

Functions

→ Functions let you package multiple related lines of code into a single body. You then summon the function to avoid repeating those lines of code throughout your app.

→ If a function doesn't need to return a value, you can set the return type to `void`



```
bool isValidIdNumber(String number) {
  return number.length == 11;}

```

Optional Parameters

→ If a parameter to a function is optional, you can surround it with square brackets and make the type nullable

```
String fullName(String first, String last, [String? title]) {
  if (title == null) {
    return '$first $last';
  } else {
    return '$title $first $last';
  }
}

fullName('Joe', 'Howard'); // Function calling -> Joe Howard
fullName('Albert', 'Einstein', 'Professor'); // Function call

// Optional parameter with default value
String fullName(String first, String last, [String title = 'Dr']) {
  if (title == null) {
    return '$first $last';
  } else {
    return '$title $first $last';
  }
}
fullName('Joe', 'Howard'); // Dr Joe Howard
```

Named Parameters

→ **named parameters**, surrounding the parameter list with curly brackets: `{ }`.

→ These parameters are optional by default, but you can give them default values or make them required by using the `required` keyword.

```
// Value is required, while min and max are
// optional with default values.
bool withinTolerance({required int value, int min = 0, int max = 100}) {
  return min <= value && value <= max;
}
```

```
// With named parameters, you can pass in
// arguments in a different order by supplying
// the parameter names with a colon.
withinTolerance(min: 1, max: 5, value: 11);

//Leave off the parameters with default
//values when calling the function.
withinTolerance(value: 5)
```

Arrow Functions

→ The arrow function is represented by \Rightarrow symbol. It is a shorthand syntax for any function that has only one expression.

```
returnType functionName(parameters...) => expression;

// function that calculate interest
double calculateInterest(double principal,
double rate, double time) {
    double interest = principal * rate * time / 100;
    return interest;
}

// arrow function that calculate interest
double calculateInterest(double principal,
double rate, double time) =>
    principal * rate * time / 100;
```

Anonymous Functions

→ Dart supports ***first-class functions***, meaning that it treats functions like any other data type. You can assign them to variables, pass them as arguments and return them from other functions.

```
final onPressed = () {
    print('button pressed');
};
```



```

onPressed(); // Function calling
// Simplify functions by using arrow syntax.
final onPressed = () => print('button pressed');

var cube = (int number) {
    return number * number * number;
};
print("The cube of 2 is ${cube(2)}");

// Usage of anonymous functions
List users = ['John', 'Chase', 'John'];
// .map takes all the list values and
// returns a new collection with them.
// .map is a convenient way to transform
// one collection into another.
final usernames = list.map(
    (name) => name.toLowerCase()
);

```



Don't confuse the `.map` method with the `Map` type.

Taking User Input

→ You must import the package `import 'dart:io';` for user input.

```

import 'dart:io';

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

```

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

// Double Input Taking
void main() {
    print("Enter a floating number:");
    int? number = double.parse(stdin.readLineSync());
    print("The entered number is ${number}");
}
```

Exception Handling

→ An exception is an error that occurs at runtime during program execution.

→ When the exception occurs, the flow of the program is interrupted, and the program terminates abnormally.

→ **Try** You can write the logical code that creates exceptions in the try block.

→ **Catch** When you are uncertain about what kind of exception a program produces, then a catch block is used. It is written with a try block to catch the general exception.

→ The **finally** block is always executed whether the exceptions occur or not. It is optional to include the final block, but if it is included, it should be after the try and catch block is over.

→ **On** block is used when you know what types of exceptions are produced by the program.

```
try {
    // Your Code Here
}
```

```

catch(ex){
// Exception here
}

// Try-catch with Finally block
try {
// Your Code Here
}
on Exception1 {
// Exception here
}
catch Exception2 {
// Exception here
}
finally {
// code that should always execute
//whether an exception or not.
}

```

Throwing an Exception

→ The throw keyword is used to raise an exception explicitly. A raised exception should be handled to prevent the program from exiting unexpectedly.

```

throw new Exception_name()

try {
    check_account(-10);
} catch (e) {
    print('The account cannot be negative');
}

void check_account(int amount) {
    if (amount < 0) {
        // Raising explanation externally
    }
}


```

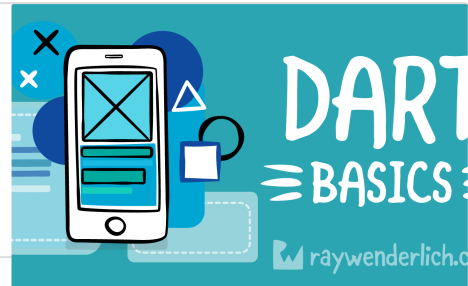
```
throw new FormatException();  
}
```

References

Dart Basics


Get an introduction to the basics of the Dart programming language, used for development with the Flutter SDK for mobile, web and beyond.

 <https://www.kodeco.com/22685966-dart-basics/page/5?page=1#toc-anchor-001>



Introduction and Basics

Learn the basics of the dart programming language. You will learn dart introduction, installation, basic program, variables, data types, comments, operators, input, and string.

 <https://dart-tutorial.com/introduction-and-basics/>



Dart Programming Tutorial

Dart Programming Tutorial - 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

 https://www.tutorialspoint.com/dart_programming/index.htm



Author → Serhat Kumas

<https://www.linkedin.com/in/serhatkumas/>

SerhatKumas - Overview

Computer engineering student who loves coding in different fields instead of focusing on a one spesific area. - SerhatKumas

 <https://github.com/SerhatKumas>

