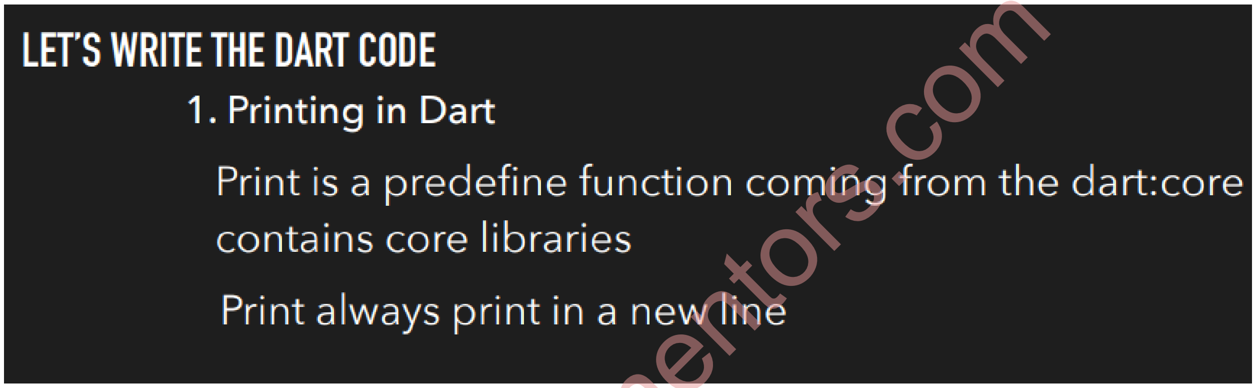
Dart Basics


LET'S WRITE THE DART CODE

1. Printing in Dart

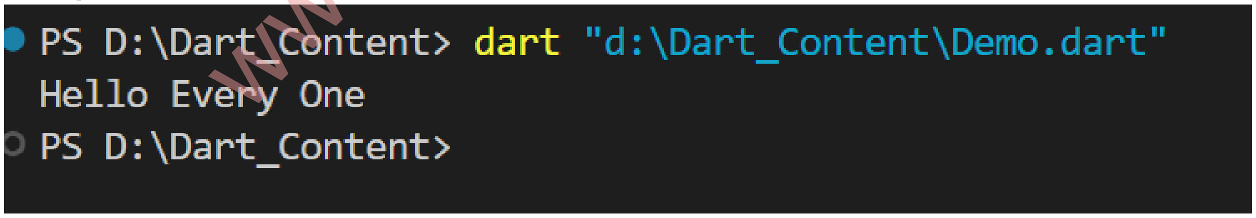
Print is a predefine function coming from the dart:core contains core libraries

Print always print in a new line

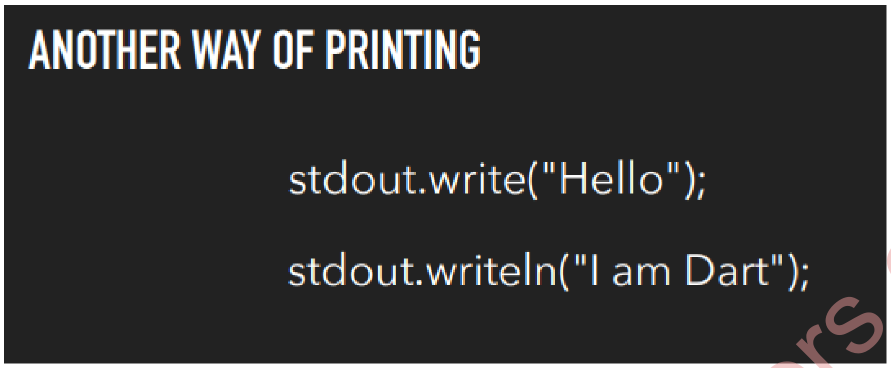
Source code:

Void main(){
print("Hello Every One..");
}

Output:


```
PS D:\Dart_Content> dart "d:\Dart_Content\Demo.dart"
Hello Every One
PS D:\Dart_Content>
```

Another method:



ANOTHER WAY OF PRINTING

```
stdout.write("Hello");

stdout.writeln("I am Dart");
```

Dart operators

Dart provides a wide range of operators that you can use in expressions to manipulate data.

Here’s a list of Dart operators by name only:

1. Arithmetic Operators
2. Equality Operators
3. Relational Operators
4. Logical Operators
5. Assignment Operators
6. Type Test Operators
7. Bitwise Operators
8. Conditional Operators
9. Cascade Operator
10. Indexing Operator
11. Function Call Operator

1. Arithmetic Operators

These operators are used to perform mathematical operations.

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`~/`	Integer division (returns int)	`a ~/ b`
`%`	Modulus (remainder)	`a % b`
`-expr`	Unary minus (negation)	`-a`

2. Equality and Relational Operators

These operators are used to compare two operands.

Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`

3. Logical Operators

These operators are used to combine or negate boolean expressions.

Operator	Description	Example
`&&`	Logical AND	`a && b`
`		`
`!`	Logical NOT	`!a`

4. Assignment Operators

These operators are used to assign values to variables.

Operator	Description	Example
`=`	Assign value	`a = b`
`??=`	Assign value if null	`a ??= b`
`+=`	Add and assign	`a += b`
`-=`	Subtract and assign	`a -= b`
`*=`	Multiply and assign	`a *= b`
`/=`	Divide and assign	`a /= b`
`~/=`	Integer divide and assign	`a ~/= b`
`%=`	Modulus and assign	`a %= b`

5. Type Test Operators

These operators are used to test the type of an object.

Operator	Description	Example
`is`	True if the object is of the specified type	`a is int`
`is!`	True if the object is **not** of the specified type	`a is! int`
`as`	Cast object to a specific type	`a as String`

6. Bitwise Operators

These operators are used to perform bitwise operations.

Operator	Description	Example
`&`	Bitwise AND	`a & b`
`	`	Bitwise OR
`^`	Bitwise XOR	`a ^ b`
`~`	Bitwise NOT (complement)	`~a`
`<<`	Left shift	`a << b`
`>>`	Right shift	`a >> b`

7. Conditional Operators

These operators are used to evaluate conditions.

Operator	Description	Example
`condition ? expr1 : expr2`	Ternary operator, if condition is true, expr1 is evaluated, otherwise expr2	`a > b ? a : b`
`??`	If the left-hand side is `null`, return the right-hand side	`a ?? b`

8. Cascade Notation (..)

Allows performing multiple operations on the same object.

Operator	Description	Example
`..`	Cascade operator	`object..method()`

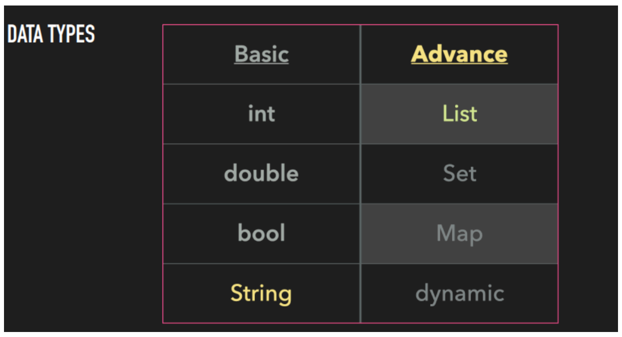
9. Other Operators

Some additional useful operators.

Operator	Description	Example
`()`	Function call	`myFunction()`
`[]`	Indexing operator	`myList[index]`
`?.`	Conditional member access	`object?.property`

Dart DataTypes

Dart programming language has several built-in data types



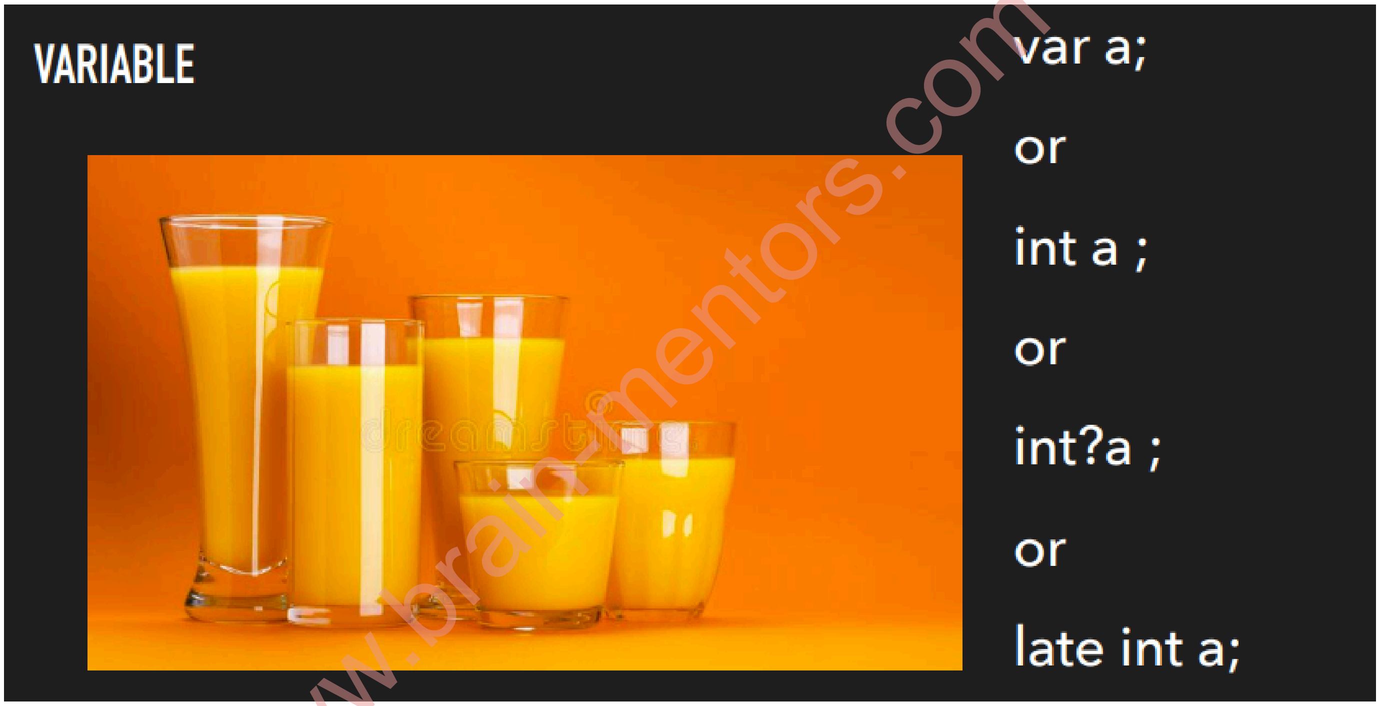
DATA TYPES

Basic	Advance
int	List
double	Set
bool	Map
String	dynamic

www.brain-mentors.com

Brain Mentors
We Develop IT Brains

Dart Variables



```
var a;
or
int a ;
or
int?a ;
or
late int a;
```

In Dart, variables are used to store data that can be referenced and manipulated in your programs.

Variable Type	Description	Approximate Size	Example
int	Represents integer values (whole numbers).	4 bytes	`int age = 30;`
double	Represents floating-point numbers (decimal values).	8 bytes	`double height = 5.9;`
String	Represents a sequence of characters.	2 bytes per character	`String name = "Alice";`
bool	Represents a boolean value (`true` or `false`).	1 byte	`bool isActive = true;`
List	An ordered collection of items (same/different types).	Varies (size of elements)	`List<int> numbers = [1, 2, 3];`
Map	A collection of key-value pairs.	Varies (size of keys & values)	`Map<String, int> scores = {'Alice': 90};`
Set	An unordered collection of unique items.	Varies (size of elements)	`Set<String> fruits = {'apple', 'banana'};`
dynamic	A variable that can hold values of any type.	Varies (depends on assigned value)	`dynamic value = 10;`
final	A variable that can only be set once.	Varies (depends on assigned value)	`final int maxScore = 100;`
const	A compile-time constant.	Varies (depends on value)	`const double pi = 3.14;`
late	A variable that will be initialized later.	Varies (depends on assigned value)	`late String description;`

1. **Using `var`**:
- The `var` keyword allows you to declare a variable without specifying its type. Dart infers the type based on the assigned value.

2. **Using Explicit Types**:
- You can declare a variable with an explicit type if you want to make your code clearer.

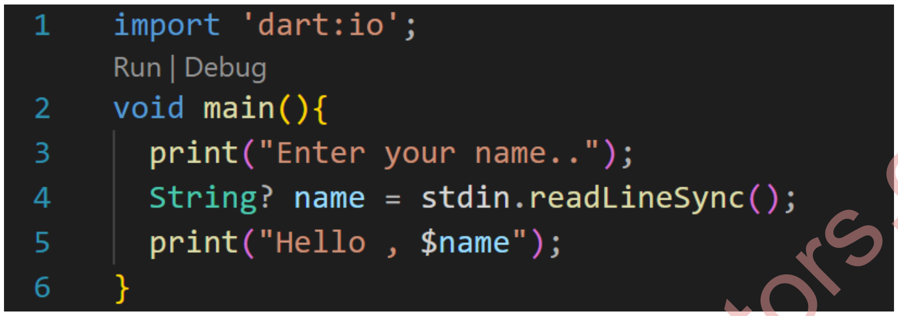
3. **Using `dynamic`**:
- The `dynamic` keyword allows you to declare a variable that can hold values of any type. The type can change during runtime.

4. **Using `late`**:
- The `late` keyword is used to declare a variable that will be initialized later. This is useful for non-nullable types when the value cannot be immediately assigned.

5. **Using `final`**:
- The `final` keyword declares a variable that can only be set once. You can assign a value at runtime, but it cannot be changed afterward.

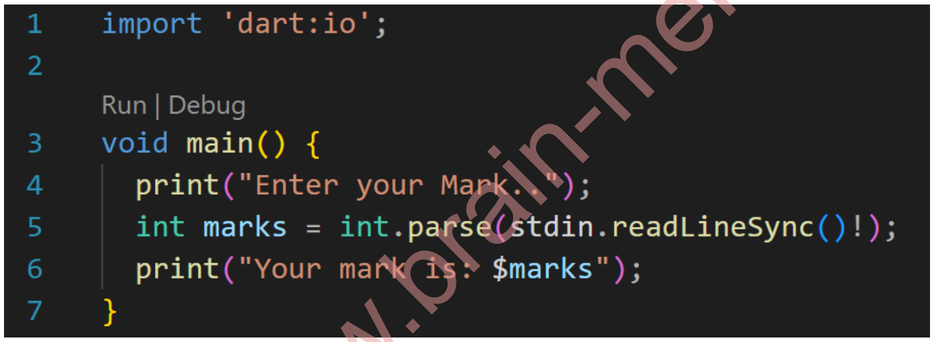
6. **Using `const`**:
- The `const` keyword is used to declare compile-time constants. The value must be known at compile time and cannot change.

How to take input in dart:



```dart
import 'dart:io';
void main(){
  print("Enter your name..");
  String? name = stdin.readLineSync();
  print("Hello , $name");
}
```

Type conversion:



```dart
import 'dart:io';

void main() {
  print("Enter your Mark..");
  int marks = int.parse(stdin.readLineSync()!);
  print("Your mark is: $marks");
}
```

Single-line comments

A single-line comment begins with `//`. Everything between `//` and the end of line is ignored by the Dart compiler.

Multi-line comments

A multi-line comment begins with `/*` and ends with `*/`. Everything between `/*` and `*/` is ignored by the Dart compiler . Multi-line comments can nest.

Dart Conditional Statements

Conditionals in Dart are responsible for controlling the flow of a program. They are the basic building blocks for decision-making. Conditionals allow a block of code to be executed after specific predefined conditions are met based on whether they are true or false.

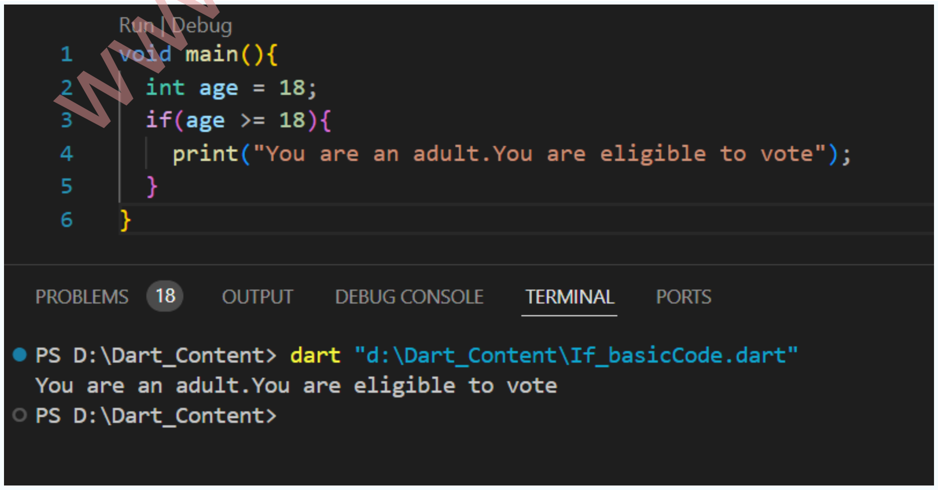
There are four ways to achieve this:

- if Statement
- if-else Statement
- else-if Ladder
- Nested if Statement

if Statement

Syntax

```
if(condition){
  // Code to be executed
}
```



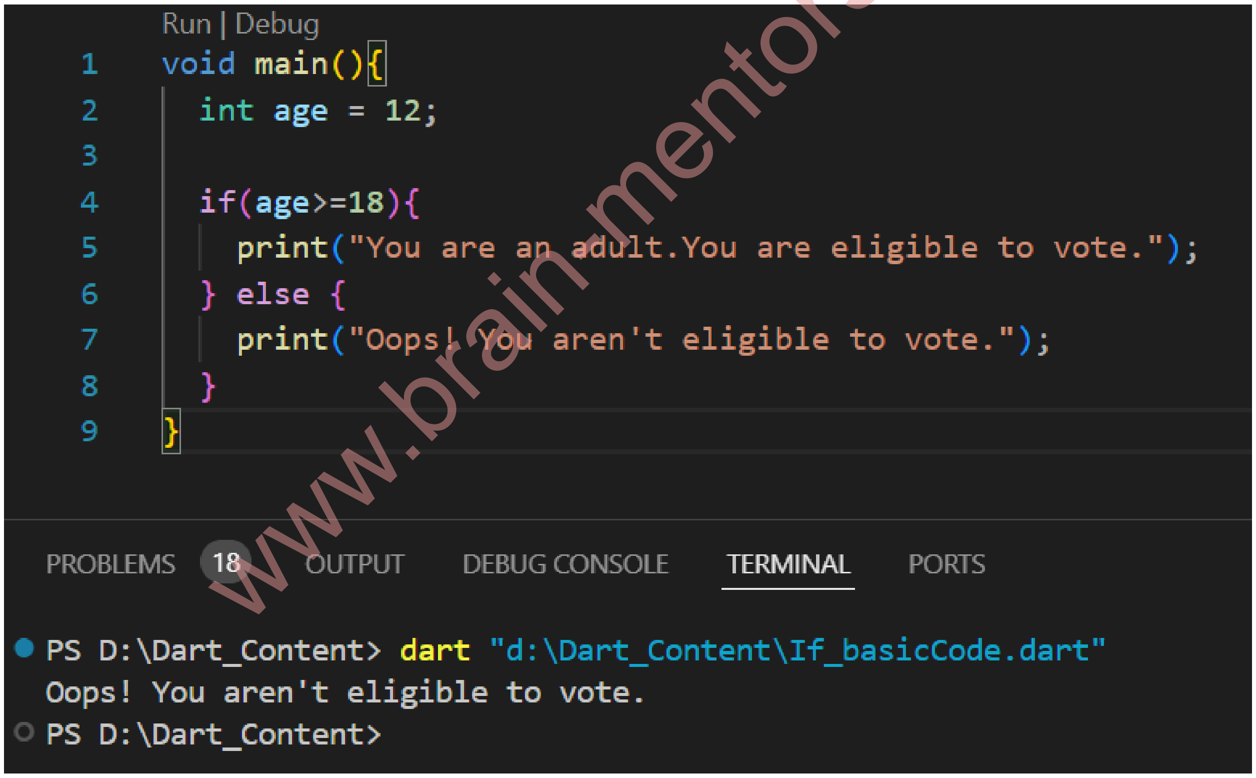
```
Run | Debug
void main(){
  int age = 18;
  if(age >= 18){
    print("You are an adult.You are eligible to vote");
  }
}
```

```
PS D:\Dart_Content> dart "d:\Dart_Content\If_basicCode.dart"
You are an adult.You are eligible to vote
PS D:\Dart_Content>
```

if-else Statement

Syntax

```
if(condition){
  // Code to be executed if and only if the condition is true
}else{
  // Code to be executed if the above condition is false
}
```



Run | Debug

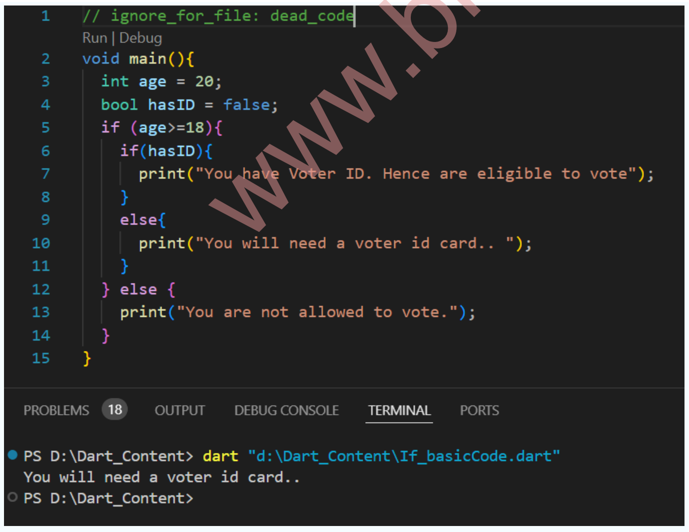
```dart
void main(){
  int age = 12;

  if(age>=18){
    print("You are an adult.You are eligible to vote.");
  } else {
    print("Oops! You aren't eligible to vote.");
  }
}
```

```
PS D:\Dart_Content> dart "d:\Dart_Content\If_basicCode.dart"
Oops! You aren't eligible to vote.
PS D:\Dart_Content>
```

Nested if-else Statement

Syntax

```
if(condition1) {
  // Code to be executed if condition1 is true
  if(condition2) {
    // Code to be executed if condition1 and condition2 are both
true
  } else {
    // Code to be executed if condition1 is true but condition2
is false
  }
} else {
  // Code to be executed if condition1 is false
  if(condition3){
    // Code to be executed if condition1 is false and condition3
is true
  } else {
    // Code to be executed if both condition1 and condition3 are
false
  }
}
```



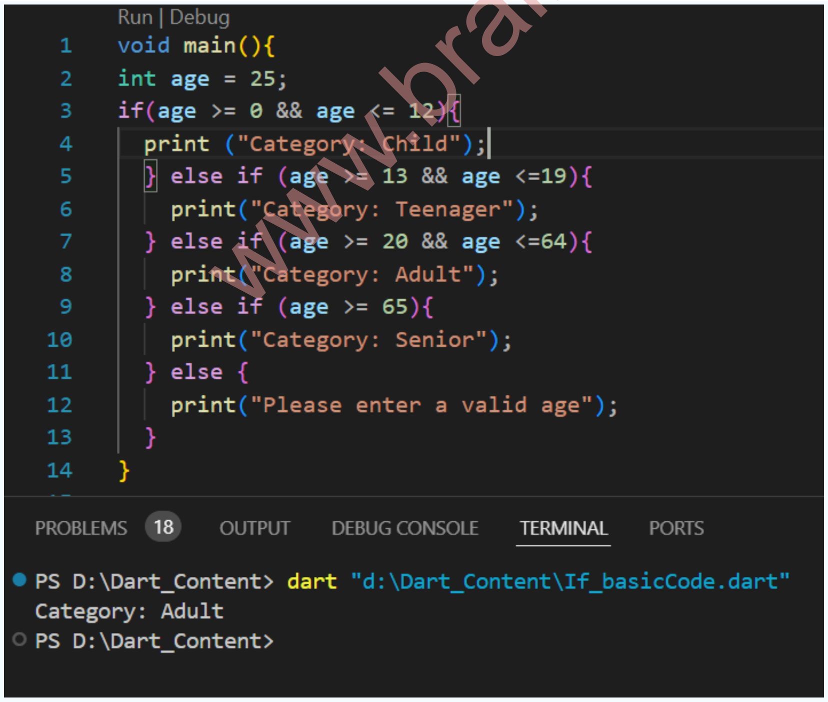
```
// ignore_for_file: dead_code
Run | Debug
void main(){
  int age = 20;
  bool hasID = false;
  if (age>=18){
    if(hasID){
      print("You have Voter ID. Hence are eligible to vote");
    }
    else{
      print("You will need a voter id card.. ");
    }
  } else {
    print("You are not allowed to vote.");
  }
}
```

```
PS D:\Dart_Content> dart "d:\Dart_Content\If_basicCode.dart"
You will need a voter id card..
PS D:\Dart_Content>
```

if-else-if Ladder

Syntax

```
if (condition1) {
    // Code to execute if condition1 is true
  } else if (condition2) {
    // Code to execute if condition1 is false but condition2 is
true
  } else if (condition3) {
    // Code to execute if condition1 and condition2 are false
but condition3 is true
  } else {
    // Code to execute if all the above conditions are false
}
```



```
void main(){
int age = 25;
if(age >= 0 && age <= 12){
  print ("Category: Child");
  } else if (age >= 13 && age <=19){
    print("Category: Teenager");
  } else if (age >= 20 && age <=64){
    print("Category: Adult");
  } else if (age >= 65){
    print("Category: Senior");
  } else {
    print("Please enter a valid age");
  }
}
```

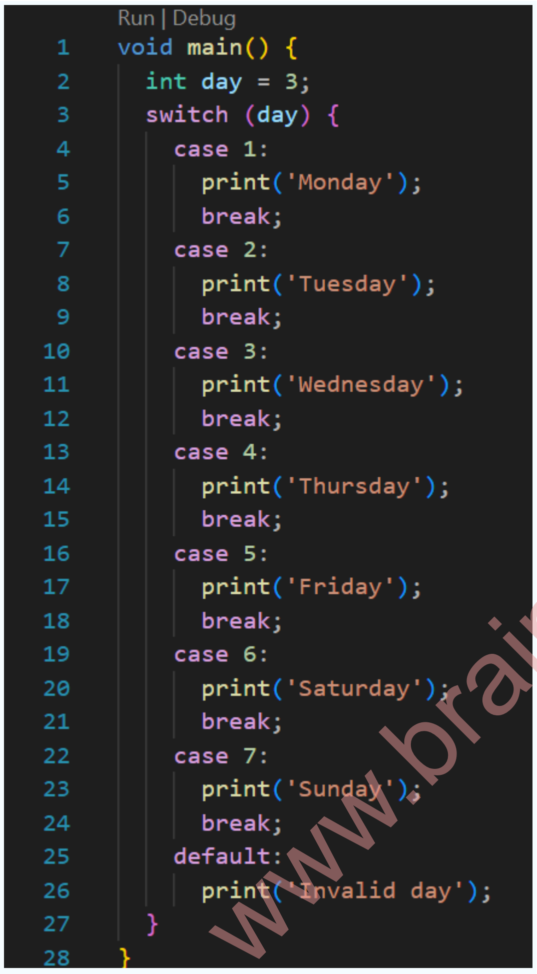
```
PS D:\Dart_Content> dart "d:\Dart_Content\If_basicCode.dart"
Category: Adult
PS D:\Dart_Content>
```

Switch Statement

The switch statement is used to execute a code block based on a specific value of a expression.

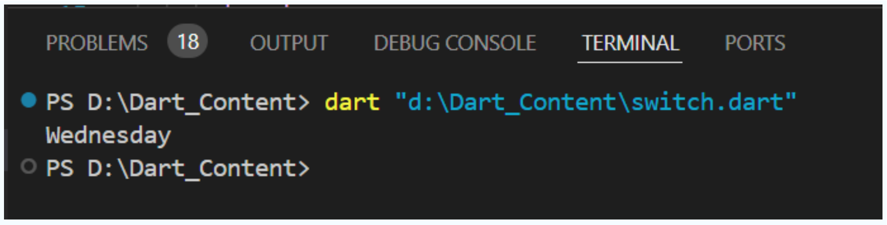
Syntax

```
switch (expression) {
  case value1:
    // Code to execute if expression == value1
    break;
  case value2:
    // Code to execute if expression == value2
    break;
  case value3:
    // Code to execute if expression == value3
    break;
  default:
    // Code to execute if the value of the expression does not match
any case
}
```



```
Run | Debug
void main() {
  int day = 3;
  switch (day) {
    case 1:
      print('Monday');
      break;
    case 2:
      print('Tuesday');
      break;
    case 3:
      print('Wednesday');
      break;
    case 4:
      print('Thursday');
      break;
    case 5:
      print('Friday');
      break;
    case 6:
      print('Saturday');
      break;
    case 7:
      print('Sunday');
      break;
    default:
      print('Invalid day');
  }
}
```

Output:



```
PS D:\Dart_Content> dart "d:\Dart_Content\switch.dart"
Wednesday
PS D:\Dart_Content>
```

Brain Mentors
We Develop IT Brains

Dart Loops

Loops are essential constructs in programming that allow you to execute a block of code repeatedly based on a specified condition. Dart provides several types of loops, including `for`, `while`, and `do-while` loops. Each type has its own syntax and use cases.

1. For Loop

The `for` loop is used when the number of iterations is known. It consists of an initialization, a condition, and an increment/decrement operation.

```
for (initialization; condition; increment/decrement) {
  // Code to be executed
}
```

Examples:

void main() {

for (int i = 1; i <= 5; i++) {

print('Iteration $i');

} }

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

2. While Loop

The `while` loop continues to execute as long as the specified condition is true. It checks the condition before executing the loop body.

```
while (condition) {
  // Code to be executed
}
```

Examples:

```
void main() {
 int count = 1;
 while (count <= 5) {
 print('Iteration $count');
 count++;
} }
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

3. Do-While Loop

The `do-while` loop is similar to the `while` loop, but it checks the condition after executing the loop body. This means that the loop will execute at least once.

Syntax:

```
do {
  // Code to be executed
} while (condition);
```

Examples:

```
void main() {
 int count = 1;
 do {
 print('Iteration $count');
count++;
} while (count <= 5);
}
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

4. For-In Loop

The `for-in` loop is used to iterate over elements in a collection, such as a list or a set. It simplifies the syntax for accessing elements.

```
for (var item in collection) {
  // Code to be executed
}
```

Examples:

void main() {

List<String> fruits = ['Apple', 'Banana', 'Cherry'];

for (var fruit in fruits) {

print(fruit);

} }

Output:

```
Apple
Banana
Cherry
```

5. ForEach Loop

The `forEach` method is a function available for collections that allows you to apply a function to each element in the collection.

```
collection.forEach((item) {
  // Code to be executed
});
```

Examples:

```
void main() {
 List<String> colors = ['Red', 'Green', 'Blue'];
 colors.forEach((color) {
 print(color);
 });
 }
```

Output:

```
Red
Green
Blue
```

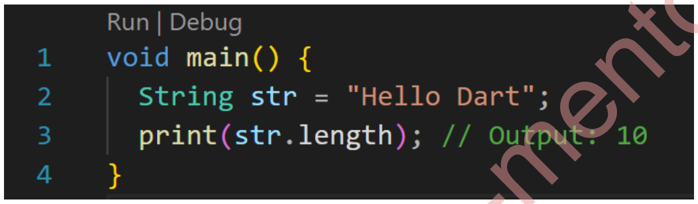
String & List Evaluation..

String:

`String` is nothing but a plain text that can contain one or more characters including a combination of letters and numbers (alpha-numeric) or special characters. This kind of data representation is considered to be `String` as long as it is included in *single* or *double-quotes*.

1. Length

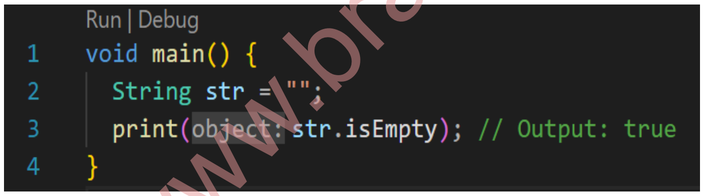
Returns the number of characters in the string.



```
Run | Debug
void main() {
  String str = "Hello Dart";
  print(str.length); // Output: 10
}
```

2. isEmpty

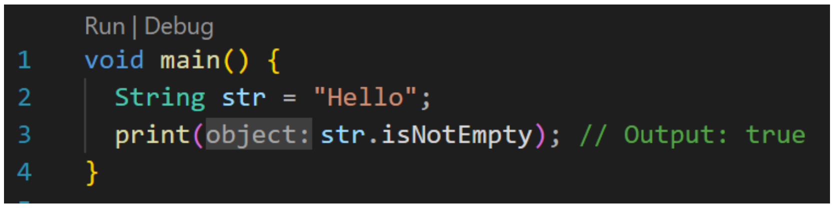
Checks if the string is empty.



```
Run | Debug
void main() {
  String str = "";
  print(object: str.isEmpty); // Output: true
}
```

3. isNotEmpty

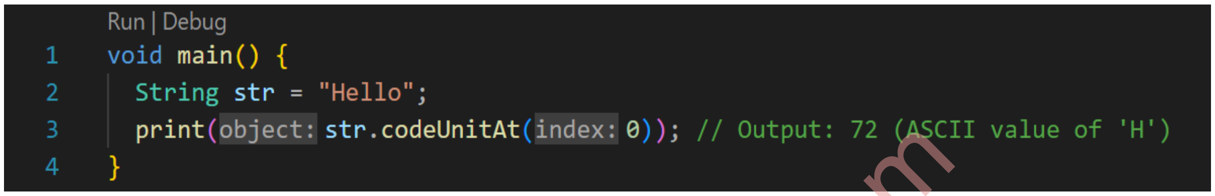
Checks if the string is not empty.



```
Run | Debug
void main() {
  String str = "Hello";
  print(object: str.isNotEmpty); // Output: true
}
```

4. codeUnitAt()

Returns the Unicode code unit of the character at the given index.



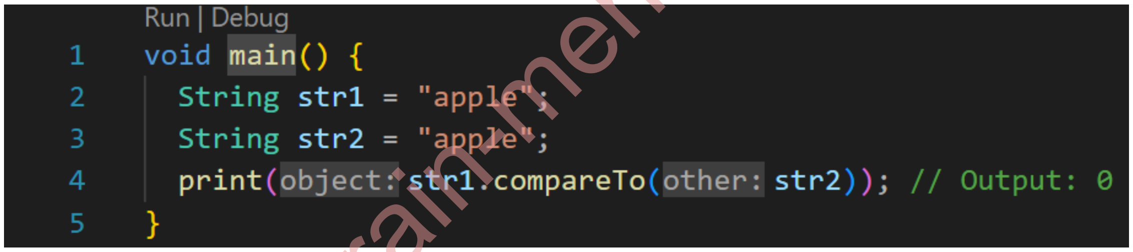
```
void main() {
  String str = "Hello";
  print(object: str.codeUnitAt(index: 0)); // Output: 72 (ASCII value of 'H')
}
```

5. compareTo() -

Compares two strings lexicographically.

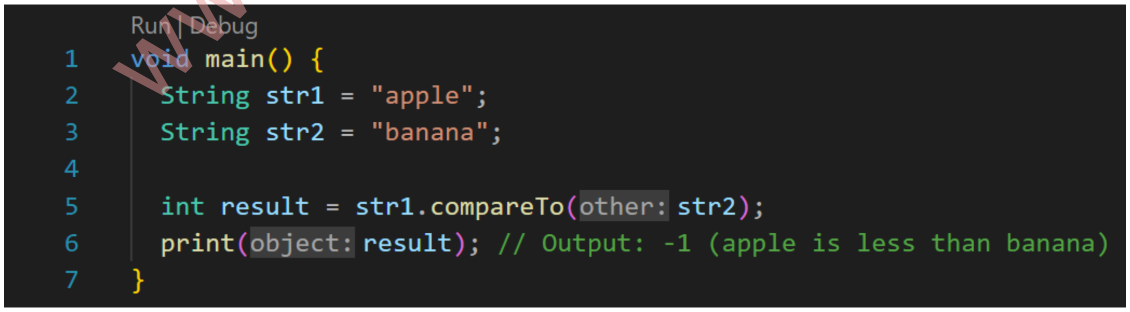
Return Values:

- **Returns 0: if both strings are equal.**



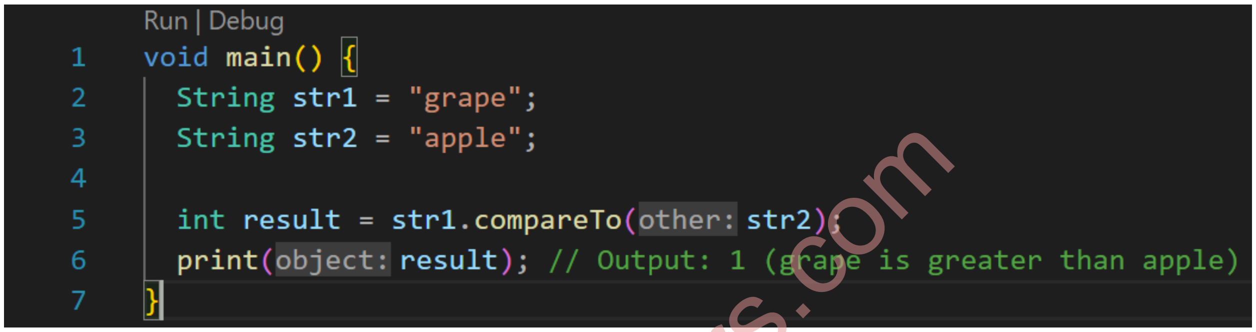
```
void main() {
  String str1 = "apple";
  String str2 = "apple";
  print(object: str1.compareTo(other: str2)); // Output: 0
}
```

- **Returns a negative integer: if the calling string is lexicographically less than the `other` string.**



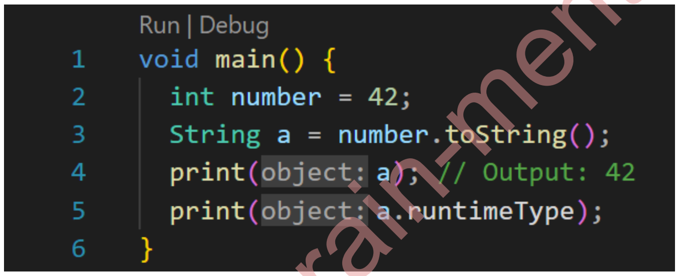
```
void main() {
  String str1 = "apple";
  String str2 = "banana";

  int result = str1.compareTo(other: str2);
  print(object: result); // Output: -1 (apple is less than banana)
}
```

- **Returns a positive integer: if the calling string is lexicographically greater than the `other` string.**


```
Run | Debug
void main() {
  String str1 = "grape";
  String str2 = "apple";

  int result = str1.compareTo(other: str2);
  print(object: result); // Output: 1 (grape is greater than apple)
}
```

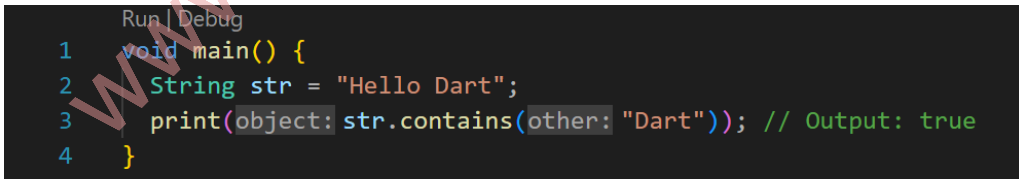
6. toString()

Returns the number of characters in the string.


```
Run | Debug
void main() {
  int number = 42;
  String a = number.toString();
  print(object: a); // Output: 42
  print(object: a.runtimeType);
}
```

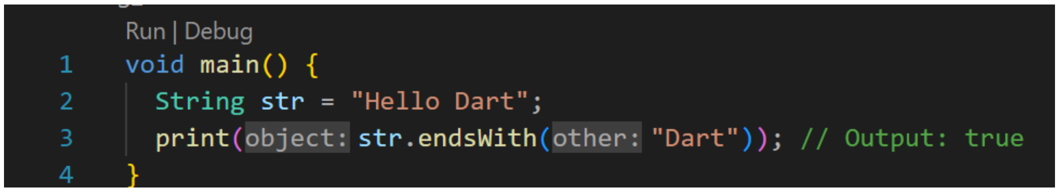
7. contains()

Checks if the string contains a specific substring.


```
Run | Debug
void main() {
  String str = "Hello Dart";
  print(object: str.contains(other: "Dart")); // Output: true
}
```

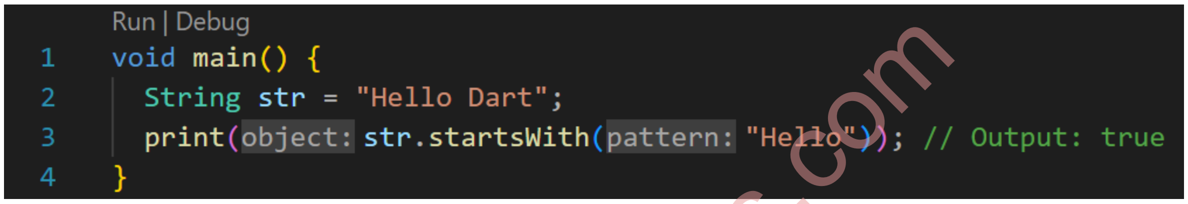
8. endsWith()

Checks if the string ends with a specific substring.


```
Run | Debug
void main() {
  String str = "Hello Dart";
  print(object: str.endsWith(other: "Dart")); // Output: true
}
```

9. startsWith()

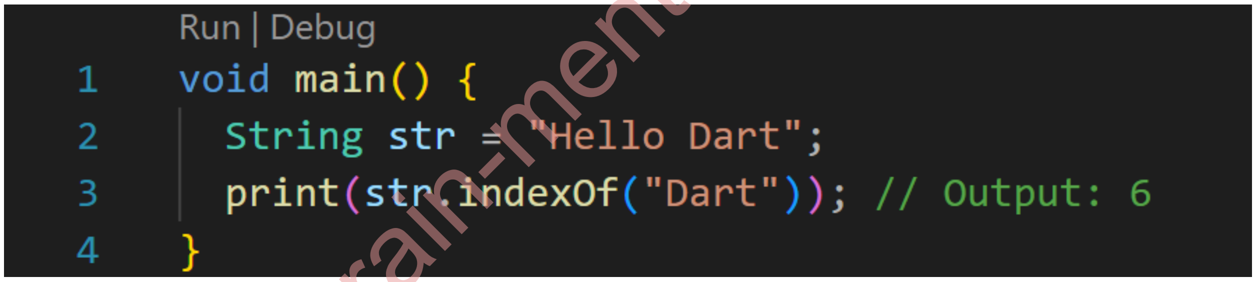
Checks if the string starts with a specific substring.



```
void main() {
  String str = "Hello Dart";
  print(object: str.startsWith(pattern: "Hello")); // Output: true
}
```

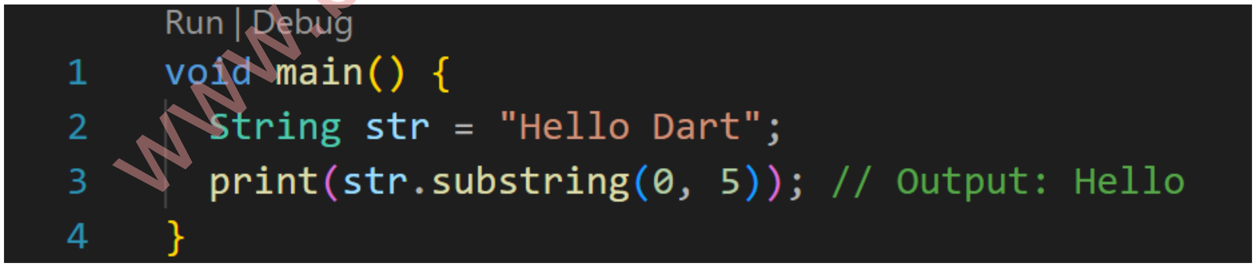
10. indexOf()

Finds the index of the first occurrence of a substring.



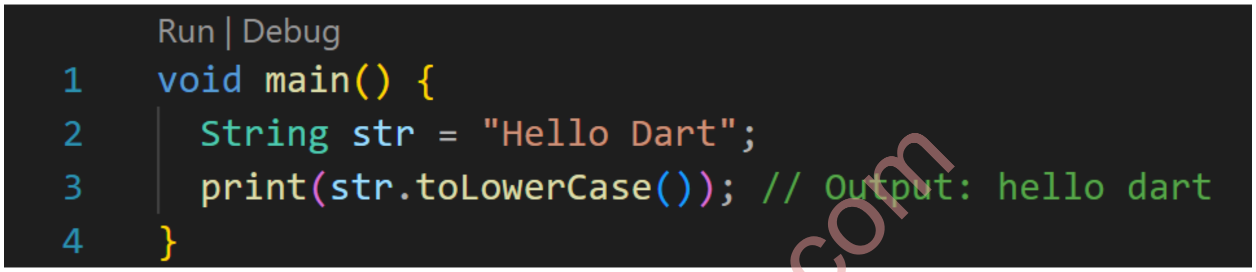
```
void main() {
  String str = "Hello Dart";
  print(str.indexOf("Dart")); // Output: 6
}
```

11. substring()



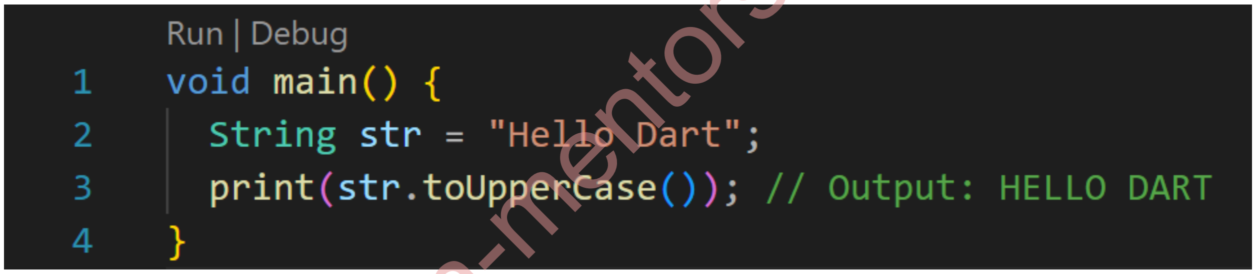
```
void main() {
  String str = "Hello Dart";
  print(str.substring(0, 5)); // Output: Hello
}
```

12. toLowerCase()



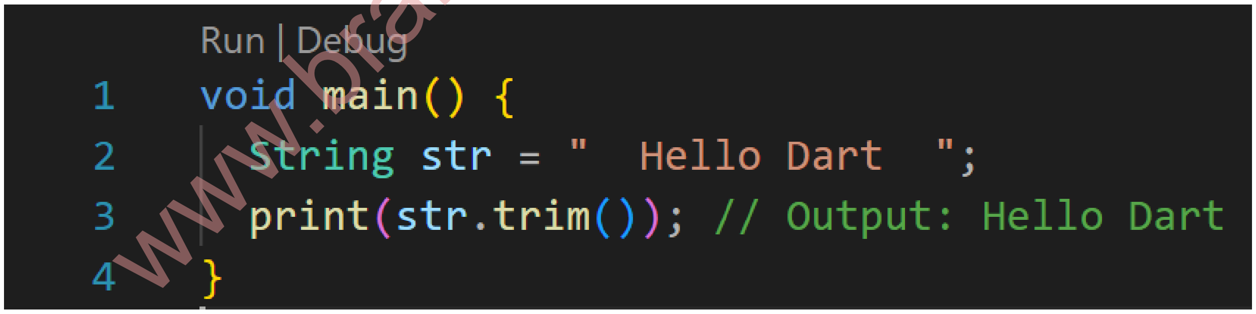
```dart
void main() {
  String str = "Hello Dart";
  print(str.toLowerCase()); // Output: hello dart
}
```

13. toUpperCase()



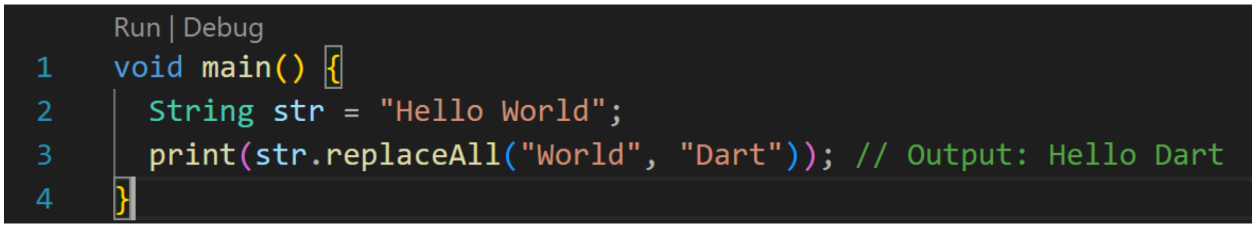
```dart
void main() {
  String str = "Hello Dart";
  print(str.toUpperCase()); // Output: HELLO DART
}
```

14. trim()



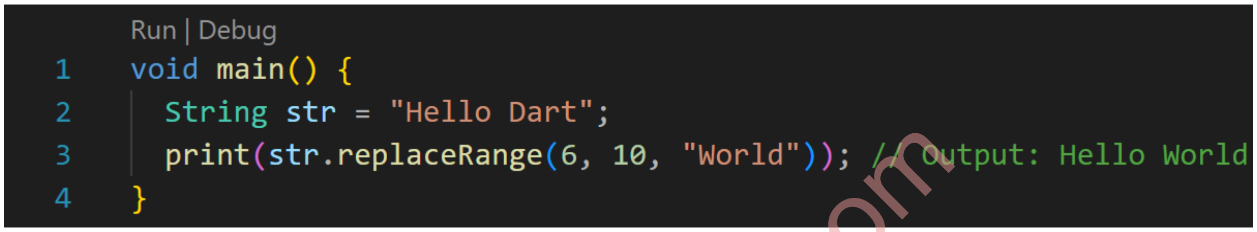
```dart
void main() {
  String str = "   Hello Dart   ";
  print(str.trim()); // Output: Hello Dart
}
```

15. replaceAll()



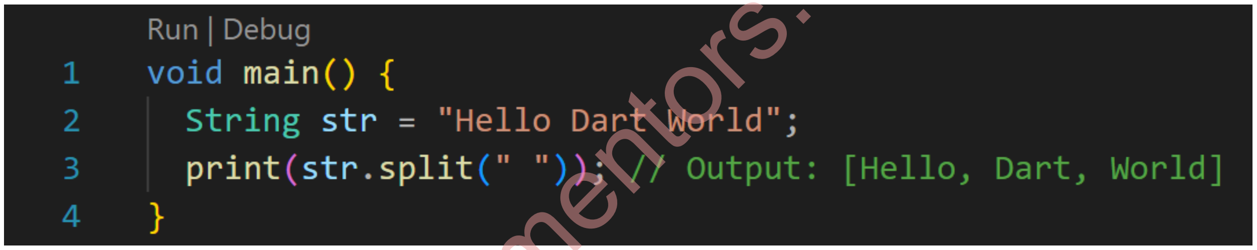
```dart
void main() {
  String str = "Hello World";
  print(str.replaceAll("World", "Dart")); // Output: Hello Dart
}
```

16. replaceRange()



```dart
void main() {
  String str = "Hello Dart";
  print(str.replaceRange(6, 10, "World")); // Output: Hello World
}
```

17. split()



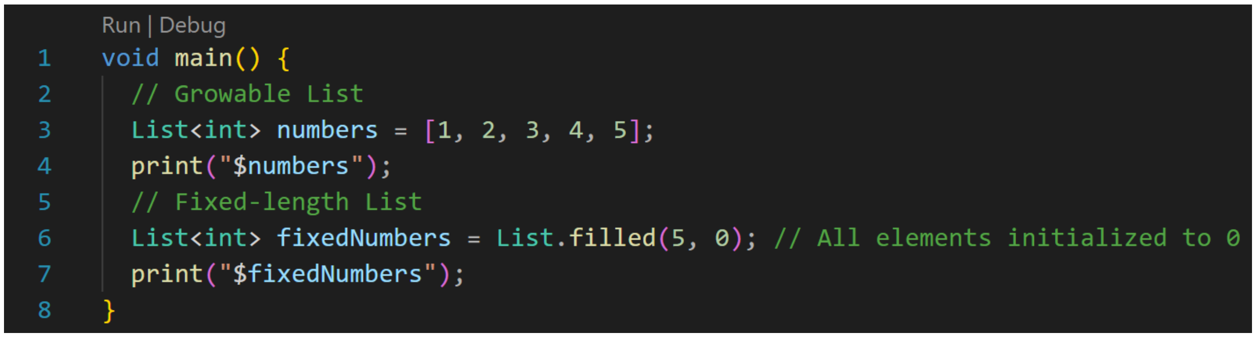
```dart
void main() {
  String str = "Hello Dart World";
  print(str.split(" ")); // Output: [Hello, Dart, World]
}
```

List:

In Dart, a `List` is an ordered collection of objects, and it is similar to arrays in other programming languages. Lists can be either fixed-length or growable.

Declaring a List in Dart:

You can create a list in Dart using square brackets `[]` or the `List` constructor.

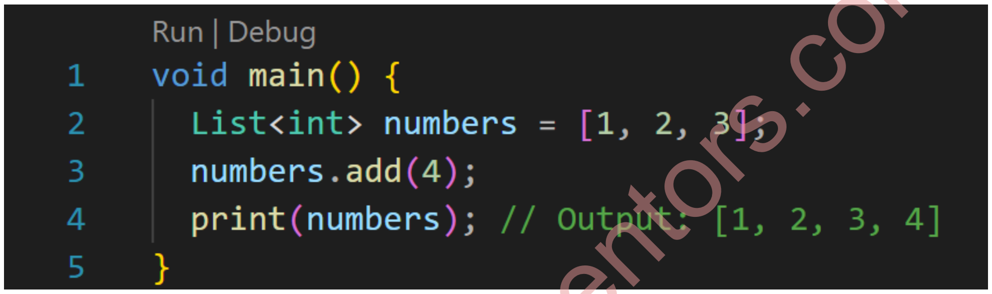


```dart
void main() {
  // Growable List
  List<int> numbers = [1, 2, 3, 4, 5];
  print("$numbers");
  // Fixed-length List
  List<int> fixedNumbers = List.filled(5, 0); // All elements initialized to 0
  print("$fixedNumbers");
}
```

Important Built-in Functions of Dart List:

1. add()

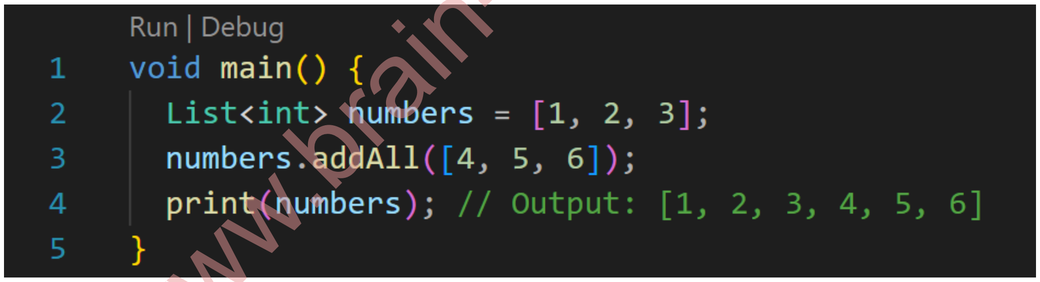
Adds a single element to the end of the list (only for growable lists).



```dart
void main() {
  List<int> numbers = [1, 2, 3];
  numbers.add(4);
  print(numbers); // Output: [1, 2, 3, 4]
}
```

2. addAll()

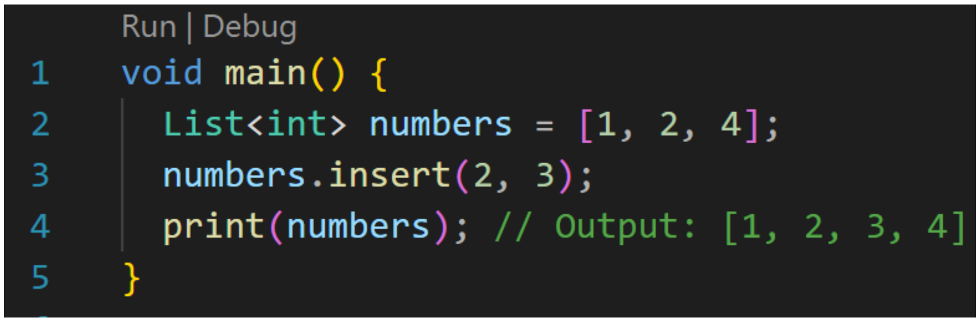
Adds all elements of another list to the end of the list.



```dart
void main() {
  List<int> numbers = [1, 2, 3];
  numbers.addAll([4, 5, 6]);
  print(numbers); // Output: [1, 2, 3, 4, 5, 6]
}
```

3. insert()

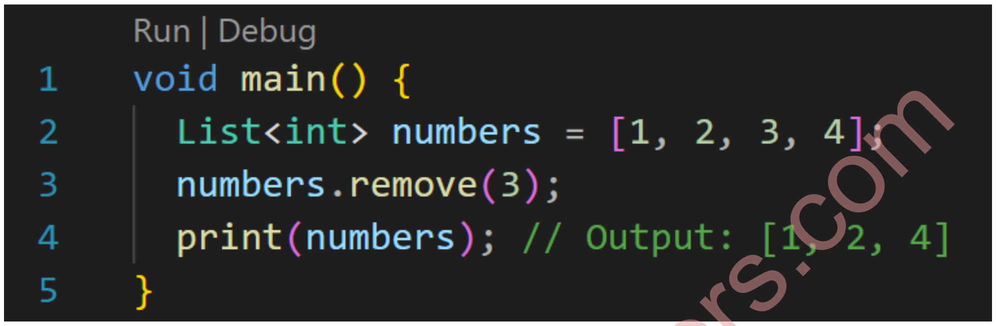
Inserts an element at the specified index.



```dart
void main() {
  List<int> numbers = [1, 2, 4];
  numbers.insert(2, 3);
  print(numbers); // Output: [1, 2, 3, 4]
}
```

4. remove()

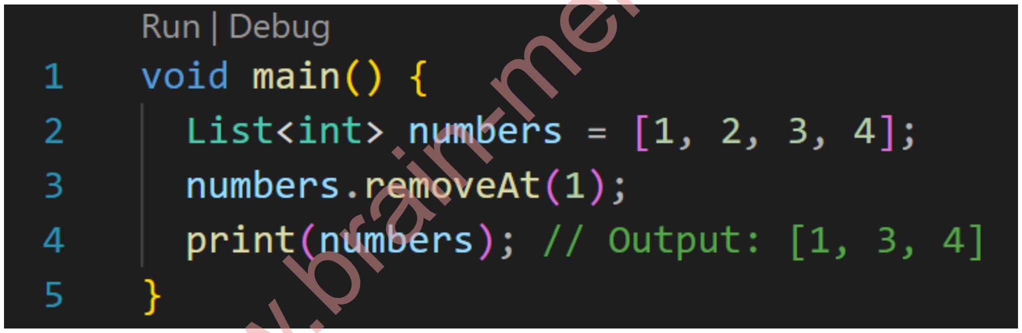
Removes the first occurrence of the specified element from the list.



```dart
void main() {
  List<int> numbers = [1, 2, 3, 4];
  numbers.remove(3);
  print(numbers); // Output: [1, 2, 4]
}
```

5. removeAt()

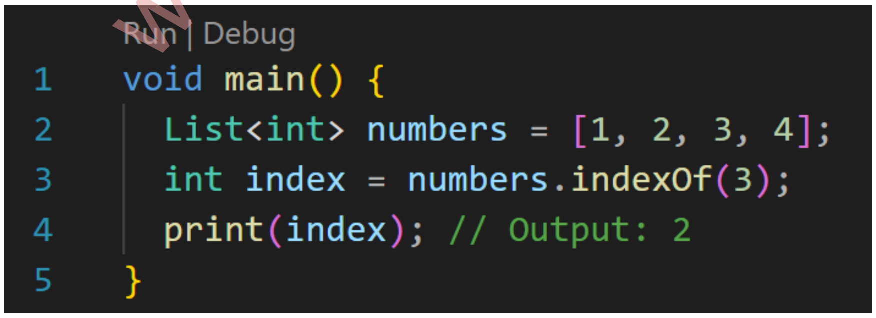
Removes the element at the specified index.



```dart
void main() {
  List<int> numbers = [1, 2, 3, 4];
  numbers.removeAt(1);
  print(numbers); // Output: [1, 3, 4]
}
```

6. indexOf()

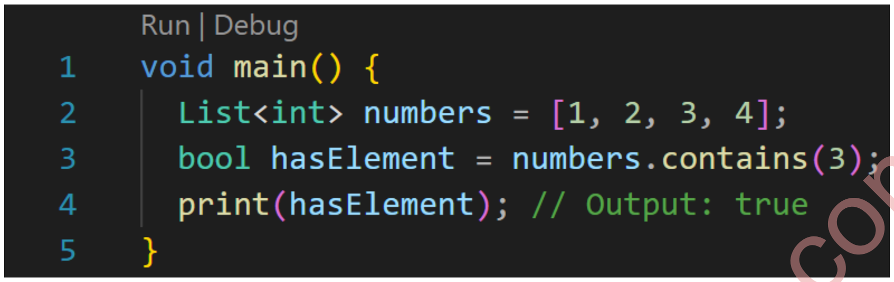
Returns the index of the first occurrence of the specified element.



```dart
void main() {
  List<int> numbers = [1, 2, 3, 4];
  int index = numbers.indexOf(3);
  print(index); // Output: 2
}
```

7. contains()

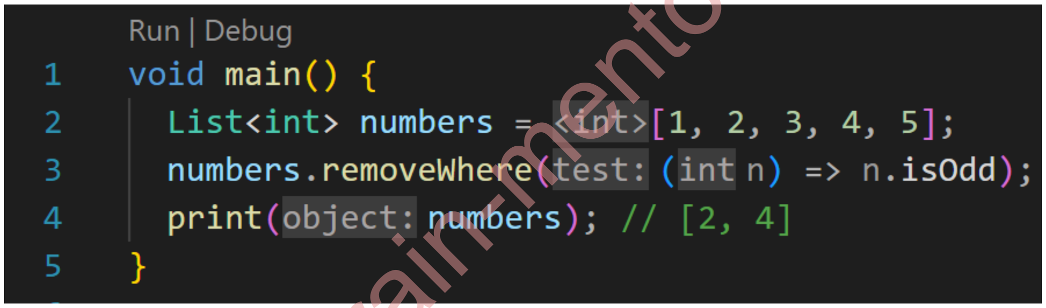
Checks if the list contains the specified element.



```dart
void main() {
  List<int> numbers = [1, 2, 3, 4];
  bool hasElement = numbers.contains(3);
  print(hasElement); // Output: true
}
```

8. removeWhere()

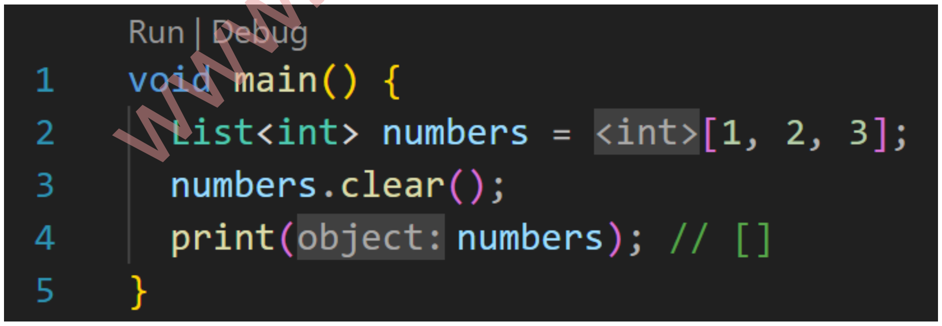
Removes all elements that satisfy a condition.



```dart
void main() {
  List<int> numbers = <int>[1, 2, 3, 4, 5];
  numbers.removeWhere(test: (int n) => n.isOdd);
  print(object: numbers); // [2, 4]
}
```

9. clear()

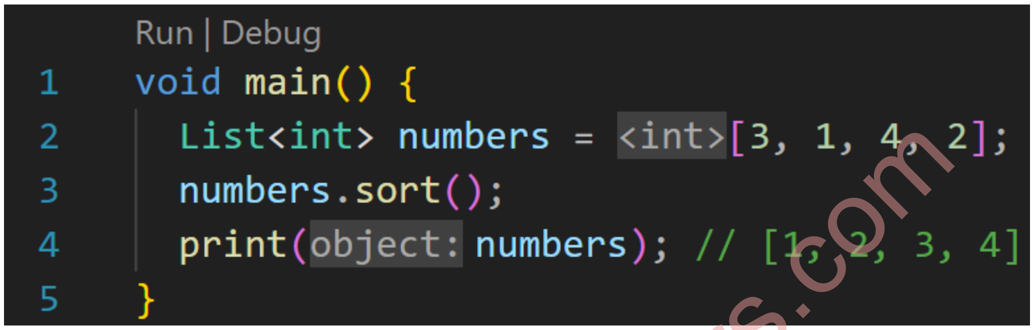
Removes all elements from the list.



```dart
void main() {
  List<int> numbers = <int>[1, 2, 3];
  numbers.clear();
  print(object: numbers); // []
}
```

10.sort()

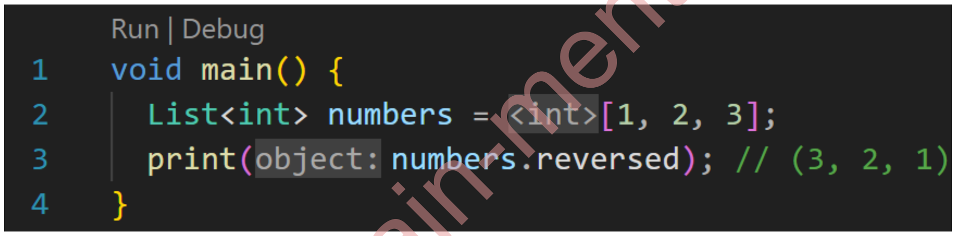
Sorts the list in ascending order.



```
void main() {
  List<int> numbers = <int>[3, 1, 4, 2];
  numbers.sort();
  print(object: numbers); // [1, 2, 3, 4]
}
```

11.reversed

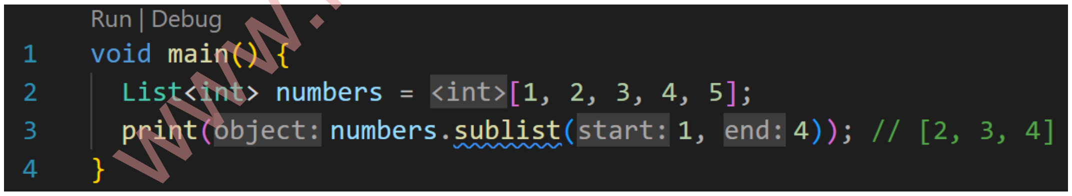
Returns an iterable of the list in reverse order.



```
void main() {
  List<int> numbers = <int>[1, 2, 3];
  print(object: numbers.reversed); // (3, 2, 1)
}
```

12.sublist()

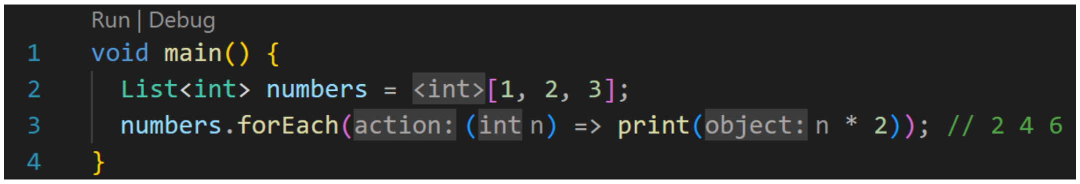
Returns a part of the list as a new list.



```
void main() {
  List<int> numbers = <int>[1, 2, 3, 4, 5];
  print(object: numbers.sublist(start: 1, end: 4)); // [2, 3, 4]
}
```

13.forEach()

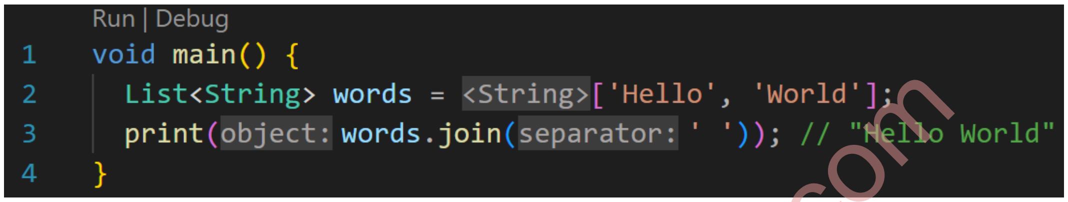
Applies a function to each element of the list.



```
void main() {
  List<int> numbers = <int>[1, 2, 3];
  numbers.forEach(action: (int n) => print(object: n * 2)); // 2 4 6
}
```

14.join()

Joins elements into a string with an optional separator.



```
Run | Debug
void main() {
  List<String> words = <String>['Hello', 'World'];
  print(object: words.join(separator: ' ')); // "Hello World"
}
```

Dart Functions

In Dart, functions are a key building block for writing modular, reusable code. Functions allow you to encapsulate logic and return results.

Code Organization: Functions help break down complex programs into smaller, manageable pieces.

Reusability: Once defined, functions can be called multiple times from different parts of your code, reducing repetition.

Abstraction: Functions allow you to hide complex implementations behind simple interfaces.

Modularity: By encapsulating logic in functions, you can easily maintain and update your code.

Testability: Functions make it easier to test individual parts of your program in isolation.

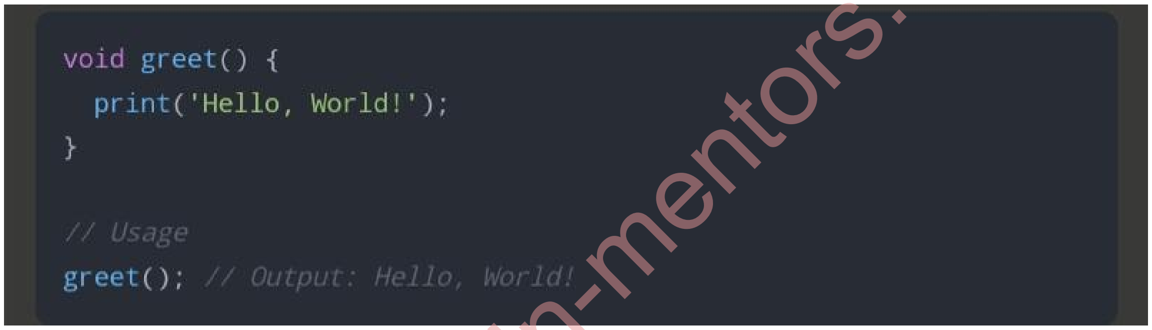
Readability: Well-named functions make code more self-documenting and easier to understand.

a. Basic Function Syntax

Definition: A block of code that performs a specific task.

- Use: For organizing code and promoting reusability.

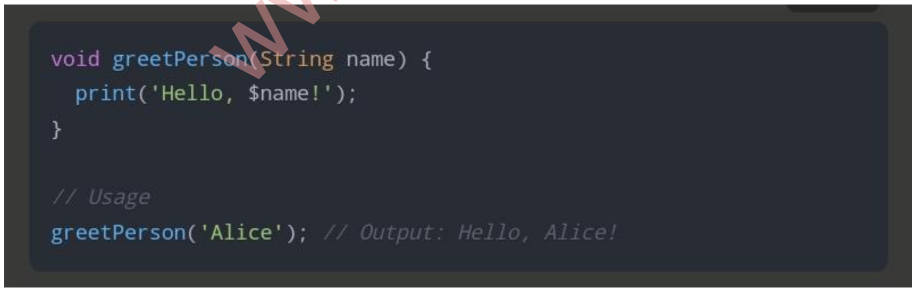
A basic Dart function looks like this:



```dart
void greet() {
  print('Hello, World!');
}

// Usage
greet(); // Output: Hello, World!
```

b. Function with Parameters

- Definition: Functions that accept input values.
- Use: To make functions more flexible and reusable.

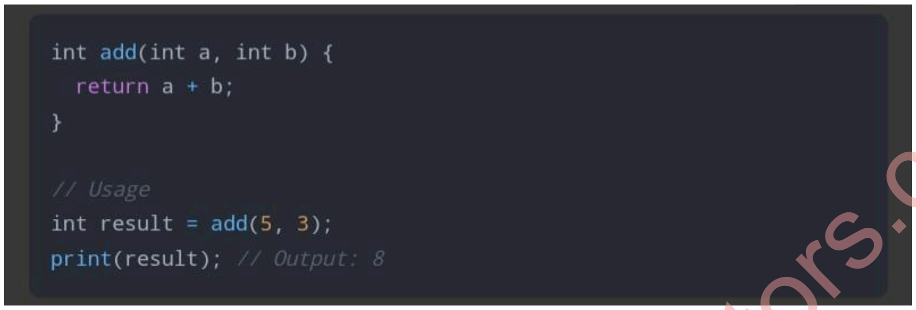


```dart
void greetPerson(String name) {
  print('Hello, $name!');
}

// Usage
greetPerson('Alice'); // Output: Hello, Alice!
```

c. Function with Return Value

- Definition: Functions that return a value after execution.

- Use: When you need to compute and return a result.



```
int add(int a, int b) {
  return a + b;
}

// Usage
int result = add(5, 3);
print(result); // Output: 8
```