Chapter-1

Dart Programming Languages Basics

Dart is a modern, object-oriented language developed by Google, often used for building web, server, and mobile applications. Understanding the basics of variable declaration, constants, data types, and control flow is essential for anyone starting with Dart.

Variable Declaration and Initialization

In Dart, variables can be declared using `var`, `final`, or `const`. Variables can also be explicitly typed.

Using `var`

The `var` keyword is used when you want the compiler to infer the type of the variable based on the assigned value.

```
void main() {
  var name = 'John Doe';    / Type inferred as String
  var age = 30;             / Type inferred as int
  var height = 5.9;         / Type inferred as double
  print(name);
  print(age);
  print(height);
}
```

Explicit Typing

You can explicitly declare the type of the variable.

```
void main() {
  String name = 'John Doe';
  int age = 30;
  double height = 5.9;
  print(name);
  print(age);
  print(height);
}
```

Constants and Final Values

Dart provides `final` and `const` for declaring constants. These values cannot be changed once set.

A `final` variable can be set only once and is initialized when accessed.

```
void main() {
  final name = 'John
  Doe'; final int
  age = 30;
  print(name);
  print(age);
}
```

A `const` variable is a compile-time constant. It is initialized at compile-time.

```
void main() {
  const name = 'John
  Doe'; const int
  age = 30;
  print(name);
  print(age);
}
```

Difference between `final` and `const`:
`final`: The value is set at runtime and can be different each time the program is run.
`const`: The value is set at compile-time and remains constant.

Data Types

Dart supports various data types including numeric values, strings, and boolean types.

Numeric Values
Dart has two numeric types: `int` and

`double`. int: Represents integer

values.

```
void
  main()
  { int
```

```
  age =
  30;
  int year =
  2023;
  print(age);
  print(year);
}
```

double: Represents floating-point values.

```
void main() {
  double height = 5.9;
  double weight = 70.5;
  print(height);
  print(weight);
}
```

Strings

Strings are a sequence of characters enclosed in single or double quotes.

```
void main() {
  String singleQuotes = 'Hello, Dart!';
  String doubleQuotes = "Hello, Dart!";
  print(singleQuotes);
  print(doubleQuotes);
  }
```

String interpolation allows you to include variables in a string.

```
void main() {
  String name = 'John';
  int age = 30;
  String greeting = 'Hello, my name is $name and I am $age years
old.';
  print(greeting);
}
```

Boolean Types

The `bool` type represents Boolean values, either `true` or `false`.

```
void main() {
  bool isVisible = true;
  bool isEnabled = false;
  print(isVisible);
  print(isEnabled);
}
```

Chapter-2

Operators in Dart

In Dart programming, operators are specialized symbols that perform operations on values (operands). They provide a concise and readable way to manipulate data and control program flow, forming the backbone of any Dart program.

Arithmetic Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	+	Addition	`a + b`	`5 + 3`	Adds two numbers
2	-	Subtraction	`a - b`	`5 - 3`	Subtracts one number from another
3	*	Multiplication	`a * b`	`5 * 3`	Multiplies two numbers
4	/	Division	`a / b`	`5 / 3`	Divides one number by another, returns a double
5	~/	Integer Division	`a ~/ b`	`5 ~/ 3`	Divides one number by another, returns an integer
6	%	Modulus	`a % b`	`5 % 3`	Returns the remainder of division

Relational Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	==	Equality	`a == b`	`5 == 3`	Checks if two values are equal
2	!=	Inequality	`a != b`	`5 != 3`	Checks if two values are not equal
3	>	Greater than	`a > b`	`5 > 3`	Checks if one value is greater than another

4	<	Less than	`a < b`	`5 < 3`	Checks if one value is less than another
5	>=	Greater than or equal to	`a >= b`	`5 >= 3`	Checks if one value is greater than or equal to another
6	<=	Less than or equal to	`a <= b`	`5 <= 3`	Checks if one value is less than or equal to another

Logical Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	&&	Logical AND	`condition1 && condition2`	`(5 > 3) && (5 < 10)`	**Returns true if both conditions are true**
2	||	Logical OR	`condition1 || condition2`	`(5 > 3) || (5 > 10)`	**Returns true if at least one condition is true**
3	!	Logical NOT	`!condition`	`!(5 > 3)`	**Inverts a boolean value**

Assignment Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	=	Assignment	`a = b`	`int a = 5`	Assigns a value to a variable
2	+=	Add and assign	`a += b`	`a += 3`	Adds and assigns a value
3	-=	Subtract and assign	`a -= b`	`a -= 2`	Subtracts and assigns a value
4	*=	Multiply and assign	`a *= b`	`a *= 2`	Multiplies and assigns a value

5	/=	Divide and	a /= b	a /= 3	Divides and assigns a
		assign			value
6	%=	Modulus and assign	a %= b	a %= 5	Takes modulus and assigns a value

Increment and Decrement Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	++	Increment	a++ or ++a	int a = 5; a++	Increases a value by one
2	--	Decrement	a-- or --a	int a = 5; a--	Decreases a value by one

Conditional Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	?:	Conditional	condition ? expr1 : expr2	(5 > 3) ? 5 : 3	Returns one of two values based on a condition
2	??	Null-aware	a ?? b	int? a; int b = a ?? 10	Returns the left operand if not null, otherwise returns the right operand

Bitwise and Shift Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	&	Bitwise AND	a & b	5 & 3	Performs a bitwise AND
2	|	Bitwise OR	a | b	5 | 3	Performs a bitwise OR

3	^	Bitwise XOR	a ^ b	5 ^ 3	Performs a bitwise XOR
4	~	Bitwise NOT	~a	~5	Inverts the bits
5	<<	Left Shift	a << n	5 << 2	Shifts bits to the left
6	>>	Right Shift	a >> n	5 >> 2	Shifts bits to the right

Type Test Operators

Sl.No	Symbol	Operator Name	Syntax	Example	Explanation
1	is	Type Test	a is Type	5 is int	Checks if an object has a certain type
2	as	Type Cast	a as Type	a as int	Casts the object to a specified type
3	is!	Not Type Test	a is! Type	5 is! String	Checks if an object is not of a certain type

Null Safety in Dart: Keeping it Safe

- Enforces types by default, preventing null-related errors.
- Use ? to declare nullable variables.

Nullish Coalescing Operator (??): Defaults Made Easy

- Provides a default value if a variable is null or undefined (unlike ||).
- Cleaner way to handle missing values.

Example:

```
String? Name;
String greeting = name  ? "There";
  / greeting will be "There" if name is null
```

Chapter-3

Flow Control Constructs

Flow control constructs are fundamental elements in programming languages that manage the order in which statements, instructions, or function calls are executed or evaluated. They enable the programmer to dictate the sequence and conditions under which different parts of the code are executed, making the program dynamic and responsive to various inputs and conditions.

Flow control constructs are essential for:

1. Decision Making: Allowing the program to choose different paths of execution based on conditions.
2. Looping: Repeating a set of instructions until a certain condition is met.
3. Branching: Redirecting the flow of execution to different parts of the code based on certain criteria.

Common Flow Control Constructs

1. **Conditional Statements:** These include `if`, `if-else`, `if-else if ladder`, and `switch` statements. They allow the execution of specific code blocks based on whether a condition is true or false.
2. **Loops:** Constructs like `for`, `while`, and `do-while` loops that repeat a block of code multiple times.
3. **Branching Statements:** Such as `break`, `continue`, and `return`, which control the flow within loops and functions.

1. `if` Statement

The `if` statement allows you to execute a block of code if a specified condition is true.

Syntax

```
if (condition) {
   / code to execute if the condition is true
}
```

Example: Checking if a user is an adult.

```
void main() {
  int age = 20;
  if (age  = 18) {
```

```
    print('You are an adult.');
  }
}
```

Explanation:

- An integer variable `age` is initialized with a value of 20.
- The `if` statement checks if `age` is greater than or equal to 18.
- Since the condition is true (20 >= 18), it prints "You are an adult."

2. `if-else` Statement

The `if-else` statement allows you to execute one block of code if the condition is true and another block if the condition is false.

```
Syntax:
if (condition) {
   / code to execute if the condition is true
} else {
   / code to execute if the condition is false
}
```

Example: Checking if a user is logged in or not.

```
void main() {
  bool isLoggedIn = true;

  if (isLoggedIn) {
    print('Welcome back!');
  } else {
    print('Please log in.');
  }
}
```

Explanation:

- A boolean variable `isLoggedIn` is initialized with a value of `true`.
- The `if` statement checks if `isLoggedIn` is `true`.
- Since the condition is true, it prints "Welcome back!"
- If `isLoggedIn` were `false`, it would print "Please log in."

3. `if-else if` Ladder

The `if-else if` ladder allows you to check multiple conditions in sequence. It executes the block of code associated with the first true condition.

```
Syntax:
if (condition1) {
   / code to execute if condition1 is true
} else if (condition2) {
   / code to execute if condition2 is true
} else {
   / code to execute if none of the conditions are true
}
Example: Determining the level of discount based on the purchase
amount.
void main() {
  double purchaseAmount = 150.0;
  if (purchaseAmount > 200) {
    print('You get a 20% discount.');
  } else if (purchaseAmount > 100) {
    print('You get a 10% discount.');
  } else if (purchaseAmount > 50) {
    print('You get a 5% discount.');
  } else {
    print('No discount available.');
  }
}
```

Explanation:

- A double variable `purchaseAmount` is initialized with a value of 150.0.
- The first `if` statement checks if `purchaseAmount` is greater than 200. It's false.
- The first `else if` statement checks if `purchaseAmount` is greater than 100. It's true.
- Since the second condition is true, it prints "You get a 10% discount."
- If the amount were less than or equal to 100 but greater than 50, the next condition would be checked, and so on.

4. `switch` Statement

The `switch` statement provides a way to dispatch execution to different parts of code based on the value of an expression. It's often used as an alternative to a series of `if-else` statements.

Syntax:

```
switch (expression) {
  case value1:
     / code to execute if expression  = value1
    break;
  case value2:
     / code to execute if expression  = value2
    break;
   / more cases
  default:
     / code to execute if none of the cases match
}
```

Example: Determining the day of the week based on a number input.

```
void main() {
  int dayNumber = 3;
  switch (dayNumber) {
    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 C:
      print('Saturday');
      break;
    case 7:
```

```
        print('Sunday');
        break;
      default:
        print('Invalid day number');
    }
}
```

Explanation:

- An integer variable `dayNumber` is initialized with the value 3.
- The `switch` statement compares `dayNumber` to each case.
- When `dayNumber` matches 3, it prints "Wednesday" and breaks out of the `switch` statement.
- If `dayNumber` were not between 1 and 7, the `default` block would execute, printing "Invalid day number."

Chapter-4

Definition of Looping Statements

Looping statements are constructs in programming that allow a set of instructions to be executed repeatedly based on a condition or a set number of times. They are essential for tasks that require repetitive operations, such as iterating over elements in a collection, performing operations on data sets, and automating repetitive tasks.

Types of Looping Statements

1. **for Loop:** Executes a block of code a specified number of times.
2. **while Loop:** Repeatedly executes a block of code as long as a specified condition is true.
3. **do-while Loop:** Similar to the `while` loop, but the condition is checked after the code block has executed, ensuring the block runs at least once.

1. `for` Loop

The `for` loop is used when the number of iterations is known before entering the loop. It consists of three parts: initialization, condition, and increment/decrement.

```
Syntax:
for (initialization; condition; increment/decrement) {
   / code to be executed
}
```

```
Example: Printing numbers from 1 to 5.
void main() {
  for (int i = 1; i  = 5; i +) {
    print(i);
  }
}
```

Explanation:

1. **Initialization: `int i = 1` sets the starting point.**
2. **Condition: `i <= 5` checks if `i` is less than or equal to 5.**
3. **Increment: `i++` increases `i` by 1 after each iteration.**
4. **The loop runs 5 times, printing numbers 1 to 5.**

2. `while` Loop

The `while` loop is used when the number of iterations is not known beforehand. It continues to execute as long as the condition is true.

```
Syntax:
while (condition) {
   / code to be executed
}

Example: Printing numbers from 1 to 5.
void main() {
  int i = 1;
  while (i  = 5) {
    print(i);
    i +;
  }
}
```

Explanation:

1. **An integer variable `i` is initialized with a value of 1.**
2. **The `while` loop checks if `i` is less than or equal to 5.**
3. **If the condition is true, it prints `i` and increments `i` by 1.**
4. **The loop runs 5 times, printing numbers 1 to 5.**

3. `do-while` Loop

The `do-while` loop is similar to the `while` loop, but it ensures that the code block runs at least once because the condition is checked after the block has executed.

```
Syntax:
do {
   / code to be executed
} while (condition);

Example: Printing numbers from 1 to 5.
void main() {
  int i = 1;

  do {
```

```
    print(i);
    i +;
  } while (i  = 5);
}
```

Explanation:

1. An integer variable `i` is initialized with a value of 1.
2. The `do` block prints `i` and increments `i` by 1.
3. After the block executes, the `while` loop checks if `i` is less than or equal to 5.
4. The loop runs 5 times, printing numbers 1 to 5.

Break and Continue Statements
These statements are used to control the flow within loops.

Break Statement
The `break` statement terminates the loop immediately, exiting it before the condition is false.

```
Syntax:
for (initialization; condition; increment/decrement) {
  if (condition) {
    break;
  }
   / code to be executed
}
Example: Exiting a loop when a number is found.
void main() {
  for (int i = 1; i  = 10; i +) {
    if (i  = 5) {
      print('Found 5, exiting loop.');
      break;
    }
    print(i);
  }
}
```

Explanation:

1. **The loop initializes `i` to 1 and increments it up to 10.**
2. **If `i` equals 5, it prints a message and breaks out of the loop.**
3. **The loop terminates when 5 is found, only printing numbers 1 to 4.**

Continue Statement

The `continue` statement skips the current iteration and moves to the next iteration of the loop.

```
Syntax:
for (initialization; condition; increment/decrement) {
  if (condition) {
    continue;
  }
   / code to be executed
}

Example: Skipping even numbers.
void main() {
  for (int i = 1; i  = 10; i +) {
    if (i % 2  = 0) {
      continue;
    }
    print(i);
  }
}
```

Explanation:

1. **The loop initializes `i` to 1 and increments it up to 10.**
2. **If `i` is an even number (`i % 2 == 0`), it skips the current iteration using `continue`.**
3. **It only prints odd numbers (1, 3, 5, 7, 9).**

Chapter-5

Functions in Dart

1. What are Functions and Their Benefits?

Functions are self-contained blocks of code designed to perform a specific task. They allow for better organization, modularity, and reusability of code. By breaking down complex tasks into smaller, manageable pieces, functions make code easier to read, maintain, and debug.

Benefits of Functions:

- **Code Reusability:** Functions allow you to write code once and reuse it multiple times.
- **Modularity:** Functions break down complex problems into smaller, manageable sub-problems.
- **Maintainability:** Functions make code easier to read, understand, and maintain.
- **Abstraction:** Functions hide the implementation details and expose only the necessary interface.

2. Components of a Function Definition

A function in Dart consists of the following components:
Return Type: Specifies the type of value the function returns.
Function Name: Identifies the function.
Parameters: Inputs to the function.
Function Body: The block of code that defines what the function does.

3. Basic Syntax of Function Definition

```
Syntax:
ReturnType functionName(ParameterType parameter1, ParameterType
parameter2) {
   / function body
  return value;
}

Example: Adding two numbers
int add(int a, int b) {
  return a + b;
}
```

Explanation:
1. The function `add` takes two integer parameters, `a` and `b`.
2. It returns the sum of `a` and `b` as an integer.

4. Function Call Syntax

To use a function, you call it by its name and pass the required arguments.

```
Syntax:
functionName(argument1, argument2);
Example:
void main() {
  int result = add(5, 3);
  print(result);  / Output: 8
}
```

Explanation:
1. The `add` function is called with arguments `5` and `3`.
2. The result (8) is stored in the variable `result` and printed to the console.

5. Positional Parameters and Optional Positional Parameters

Positional Parameters are parameters that are mandatory and must be provided in the correct order when the function is called.

```
Syntax:
ReturnType functionName(ParameterType parameter1, ParameterType
parameter2) {
   / function body
  return value;
}

Example: Calculating the area of a rectangle
int calculateArea(int length, int width) {
  return length * width;
}

void main() {
  print(calculateArea(5, 3));  / Output: 15
}
```

Explanation:
1. The `calculateArea` function takes two integer parameters, `length` and `width`.
2. It returns the product of `length` and `width` as the area.

Optional Positional Parameters: are parameters that are optional and can be omitted when calling the function. If omitted, they take on a default value if provided.

Syntax:

```
ReturnType functionName(ParameterType parameter1, [ParameterType
parameter2 = defaultValue]) {
   / function body
  return value;
}
```

Example: Greeting a user with an optional title

```
String greet(String name, [String title = "Mr."]) {
  return 'Hello, $title $name';
}

void main() {
  print(greet("John"));  / Output: Hello, Mr. John
  print(greet("John", "Dr."));  / Output: Hello, Dr. John
}
```

Explanation:

1. The function `greet` takes a mandatory parameter `name` and an optional parameter `title`.
2. If `title` is not provided, it defaults to "Mr."

Null Safety with Optional Positional Parameters:

```
String greet(String name, [String? title])
{
 title  ?= "Mr.";
 return 'Hello, $title $name';
}

void main() {
  print(greet("John"));  / Output: Hello, Mr. John
  print(greet("John", "Dr."));  / Output: Hello, Dr. John
}
```

Explanation:

1. The `title` parameter can be null.
2. If `title` is null, it defaults to "Mr." using the null-aware operator (`??=`).

6. Named Parameters

Named Parameters: are parameters that are identified by name rather than position. They make function calls more readable and allow for specifying only the necessary parameters.

Syntax:

```
ReturnType functionName({ParameterType parameter1, ParameterType
parameter2}) {
   / function body
  return value;
}
```

Example: Greeting a user with an optional title

With Optional Named Parameter:

```
String greet({required String name, String title = "Mr."}) {
  return 'Hello, $title $name';
}

void main() {
  print(greet(name: "John"));  / Output: Hello, Mr. John
  print(greet(name: "John", title: "Dr."));  / Output: Hello, Dr. John
}
```

Without Optional Named Parameter:

```
String greet({required String name, required String title}) {
  return 'Hello, $title $name';
}

void main() {
  print(greet(name: "John", title: "Mr."));  / Output: Hello, Mr. John
  print(greet(name: "John", title: "Dr."));  / Output: Hello, Dr. John
}
```

Explanation:

1. The function `greet` takes named parameters `name` and `title`.
2. `name` is required, while `title` is optional with a default value of "Mr."

Null Safety with Named Parameters:

```
String greet({required String name, String? title}) {
  title  ?= "Mr.";
  return 'Hello, $title $name';
}

void main() {
  print(greet(name: "John"));  / Output: Hello, Mr. John
  print(greet(name: "John", title: "Dr."));  / Output: Hello, Dr. John
}
```

Explanation:
1. The `title` parameter can be null.
2. If `title` is null, it defaults to "Mr." using the null-aware operator (`??=`).

7. Arrow Functions

Arrow Functions provide a concise syntax for functions with a single expression.

```
Syntax
ReturnType functionName(ParameterType parameter) => expression;
Example:
int square(int x) => x * x;
void main() {
  print(square(4));  / Output: 1C
}
```

Explanation:
1. The function `square` takes an integer `x` and returns its square.
2. The arrow (`=>`) replaces the need for curly braces and the `return` keyword for single-expression functions.

Chapter-6

Collections in Dart

1. Define Collections

Collections in Dart are data structures that store multiple values. They are used to hold a group of objects and provide various methods to manage these objects efficiently.

2. Collection Types:

The main collection types in Dart are:

- List
- Map

Lists in Dart

1. Define List:

A List in Dart is an ordered collection of items, where each item can be accessed by its index. Lists are similar to arrays in other programming languages.

2. How to Create a List:

Using Square Brackets:

```
List<int> numbers = [1, 2, 3, 4, 5];
```

Example Explanation:

This creates a list of integers with elements 1, 2, 3, 4, and 5.

Using List Constructor:

```
List<String> fruits = List<String>();
fruits.addAll(['Apple', 'Banana', 'Mango']);
```

Example Explanation

This creates an empty list of strings and then adds 'Apple', 'Banana', and 'Mango' to it.

3. Accessing List Elements:

```
print(fruits[0]);  / Output: Apple
```

Example Explanation:

Accesses the first element of the list `fruits`.

4. Iterating Over List Elements:

```
for (var fruit in fruits) {
  print(fruit);
}
```

Example Explanation:
Iterates over each element in the list and prints it.

5. Appending Element to List:

```
fruits.add('Orange');
```

Example Explanation:
Adds 'Orange' to the end of the list `fruits`.

6. Modifying List Elements:

```
fruits[0] = 'Strawberry';
```

Example Explanation:
Changes the first element of the list to 'Strawberry'.

7. Removing List Elements:

```
fruits.remove('Banana');
```

Example Explanation:
Removes 'Banana' from the list `fruits`.

8. Filtering List Elements Using `where` Method:

```
var filteredFruits = fruits.where((fruit) =>
fruit.startsWith('S')).toList();
```

Example Explanation
Filters the list to include only fruits that start with 'S' and converts the result back to a list.

9. List Containing Elements Maps Example:

```
List<Map<String, String > users = [
  {'name': 'Alice', 'email': 'alice@example.com'},
  {'name': 'Bob', 'email': 'bob@example.com'}
];
```

Example Explanation
Creates a list of maps, where each map contains a user's name and email.

10. Converting Maps to List of Objects Using `map()` and `toList()` Method:

```
List<String> userEmails = users.map((user) => user['email']).toList();
```

Example Explanation:

Extracts the email addresses from each user map and converts the result to a list.

Maps in Dart

1. Define Map:
A Map in Dart is an unordered collection of key-value pairs, where each key is unique and is used to access its corresponding value.

2. Creation of Map:

Using Curly Braces:

```
Map<String, int> scores = {'Alice': 50, 'Bob': 75};
```

Example Explanation:
This creates a map with keys 'Alice' and 'Bob', and their corresponding values 50 and 75.

Using `Map()` Constructor:

```
Map<String, String> capitals = Map();
 capitals['USA'] = 'Washington D.C.';
 capitals['Canada'] = 'Ottawa';
```

Example Explanation:
This creates an empty map and then adds key-value pairs for 'USA' and 'Canada'.

Using Map Types:

```
 Map<int, String> items = {1: 'Item1', 2: 'Item2'};
```

Example Explanation:
Creates a map where the keys are integers and the values are strings.

3. Accessing Map Elements:

```
print(scores['Alice']);  / Output: 50
```

Example Explanation:
Accesses the value associated with the key 'Alice'.

4. Iterating Over Maps Using for Loop:

```
for (var entry in capitals.entries) {
  print('The capital of ${entry.key} is ${entry.value}');
}
```

Example Explanation:
Iterates over each entry in the map and prints the key and value.

5. Checking Whether an Element is Contained in Map:

```
print(scores.containsKey('Bob'));  / Output: true
```

Example Explanation:
Checks if the key 'Bob' exists in the map.

6. Modifying Elements in Map:

```
scores['Alice'] = 85;
```

Example Explanation:
Updates the value associated with the key 'Alice' to 85.

7. Deleting Specific Elements:

```
scores.remove('Bob');
```

Example Explanation:
Removes the key-value pair with the key 'Bob' from the map.

Chapter-7

Object-Oriented Programming (OOP) in Dart

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data and methods. Dart is an object-oriented language with classes and objects, supporting encapsulation, inheritance, and polymorphism.

1. Defining Classes
A class is a blueprint for creating objects. It defines properties (instance variables) and methods (functions).

```
class Person {
  String name;
  int age;
  void greet() {
    print('Hello, my name is $name.');
  }
}
```

Example Explanation:
This defines a `Person` class with two instance variables (`name` and `age`) and a method (`greet`).

2. Methods with Parameters
Methods can take parameters to perform actions based on input.

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

Example Explanation:
The `Calculator` class has an `add` method that takes two integers and returns their sum.

3. Methods with Required Parameters
In Dart, all parameters are required by default unless specified as optional. However, you can enforce required named parameters using the `required` keyword.

```
class Person {
  void greet({required String name, required int age}) {
```

```
    print('Hello, my name is $name and I am $age years old.');
  }
}
```

Example Explanation:
The `greet` method has two required named parameters (`name` and `age`). You must provide values for these parameters when calling the method.

4. Creating Objects
You create an object by calling the class constructor.

```
Person person = Person();
person.name = 'Alice';
person.age = 30;
person.greet();  / Output: Hello, my name is Alice.
```

Example Explanation:
Creates a `Person` object and assigns values to its properties.

5. Constructors

Constructors are special methods to initialize objects. Dart supports named constructors and default constructors.

```
class Person {
  String name;
  int age;
  Person(this.name,              this.age);
  Person.named(this.name, {this.age = 0});
}
```

Example Explanation:
The `Person` class has a default constructor and a named constructor (`named`) with an optional parameter `age`.

6. Subclasses
A subclass is a class that extends another class (superclass).

```
class Parent {
  void show() {
    print('Parent class');
  }
```

```
}
class Child extends Parent {
  void display() {
    print('Child class');
  }
}
```

Example Explanation:
The `Child` class extends the `Parent` class, inheriting the `show` method and adding its own `display` method.

7. Polymorphism
Polymorphism allows methods to do different things based on the object it is acting upon.

```
class Animal {
  void sound() {
    print('Some sound');
  }
}
class Dog extends Animal {
  @override
  void sound() {
    print('Bark');
  }
}
class Cat extends Animal {
  @override
  void sound() {
    print('Meow');
  }
}
void main() {
  Animal myDog = Dog();
  Animal myCat = Cat();
  myDog.sound();  / Output: Bark
  myCat.sound();  / Output: Meow
}
```

Example Explanation:
The `sound` method is overridden in the `Dog` and `Cat` classes, showing different behavior depending on the object.

Two Mark Questions

1. What is the difference between var, final, and const keywords used in variable declaration?
2. How do you declare an integer variable and initialize it with a value in Dart?
3. Write a simple expression to check if two strings are equal in Dart.
4. What is the output of the following code: int x = 10; print(x++);
5. What is the purpose of a for loop in Dart?
6. How do you access an element by its index in a list?
7. What is the difference between a Map and a List data structure in Dart?
8. Can you define a class named Person with attributes for name and age?
9. What is the syntax for calling a method on an object instance in Dart?
10. What is the benefit of using named parameters in a function definition?

Five Mark Questions

1. Explain the concept of null safety in Dart. How does it improve code reliability?
2. Compare and contrast if statements and switch statements in Dart. When would you use each one?
3. Explain the concept of inheritance in Dart using the concept of sub-classes. Provide a simple example with a base class and a sub-class .
4. Explain the concept of arrow functions in Dart and their benefits.
5. Describe the different types of loops available in Dart
6. Explain the concept of collections in Dart. Briefly discuss the differences between `List`, `and Map` data structures.
7. What are null-safety benefits in Dart compared to languages without it?