Introduction to Dart

1.1 What is Dart ?

Dart is an open-source, general-purpose programming language developed by Google, designed for building fast, scalable applications across multiple platforms, such as mobile (with Flutter), web, and server-side applications. Dart is optimized for client-side development and offers a modern, clean syntax with powerful features.

Key features and importance of Dart:

- **Object-Oriented Language**: Dart is an object-oriented language, meaning everything in Dart is an object, including primitive data types like numbers and booleans.
- **Hot Reload with Flutter**: Dart is the language behind Flutter, a popular open-source UI framework developed by Google for creating natively compiled applications for mobile, web, and desktop from a single codebase. With Flutter, Dart enables "hot reload," where changes to the code can be instantly reflected in the running application without restarting it.
- **Optimized for Performance**: Dart is designed for high-performance execution, making it ideal for building fast apps that require responsive user interfaces.
- **Cross-Platform Development**: Dart and Flutter allow developers to write once and deploy on multiple platforms like Android, iOS, web, desktop, etc., ensuring consistent performance and appearance.
- **Strong Typing & Safety**: Dart is a statically typed language with optional types, which makes the code more predictable and helps catch errors early in the development process.
- **Concurrency with Isolates**: Dart offers Isolates, a way to perform parallel operations without worrying about threads or locking mechanisms.

1.2. Setting up Dart:

The Dart SDK (Software Development Kit) is the package that contains the necessary tools to develop and run Dart applications, including a Dart VM, libraries, and command-line tools like the Dart analyzer and compiler.

Here’s how to install the Dart SDK:

For Windows:

1. Visit the official **Dart** website: https://dart.dev/get-dart.
2. Download the **Dart SDK** for Windows.
3. Extract the downloaded **.zip** file to a directory, for example, C:\dart.
4. Add Dart to the system environment variables:

- Go to Control Panel > System > Advanced system settings > Environment Variables.
- Under System Variables, find the Path variable and click Edit.
- Add the path to the Dart SDK's bin directory (e.g., C:\dart\dart-sdk\bin).

5. Verify the installation by opening a terminal (Command Prompt) and running:

```
dart --version
```

This should display the installed Dart version.

For macOS:

1. Install Dart via Homebrew (a package manager for macOS):

- First, install Homebrew (if not already installed) by running the following in Terminal:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Then, install Dart with: brew tap dart-lang/dart brew install dart

2. Verify the installation by running:

```
dart --version
```

For Linux:

1. Add the Dart repository and install the Dart SDK using APT:

- Run the following commands in the terminal:

```
sudo apt update -y
sudo apt install apt-transport-https
sudo sh -c 'wget -qO-
```

```
https://storage.googleapis.com/download.dartlang.org/linux/debian/dart_stable.list
> /etc/apt/sources.list.d/dart_stable.list'
sudo apt update -y
sudo apt install dart
```

2. Verify the installation by running:

```
dart --version
```

- Once the Dart SDK is installed, you can start developing Dart applications in any text editor or IDE. Popular IDEs for Dart include **Visual Studio Code**, **IntelliJ IDEA**, and **Android Studio**, all of which support Dart plugins for enhanced development.

3. Running Your First Program:

- To run a Dart program, you'll need to write some code and execute it using the Dart SDK.

Here's how you can run your first Dart program:

Step 1: Write Your First Dart Program

1. Open any text editor (e.g., Visual Studio Code, Sublime Text) and create a new file called hello.dart.

2. Write the following Dart code:

```
void main() {
print('Hello, Dart!');
}
```

Explanation:

- **void main()** is the entry point for every Dart application.
- The **print()** function outputs text to the console. In this case, it will print **"Hello, Dart!"**.

Step 2: Run the Dart Program

1. Open your terminal (Command Prompt on Windows, Terminal on macOS/Linux).

2. Navigate to the folder where your hello.dart file is saved. For example:

```
cd path/to/your/dart/file
```

3. Run the Dart program by executing the following command:

```
dart run hello.dart
```

Step 3: Observe the Output

- The terminal will display: Hello, Dart! Running Dart Without an IDE:
- You can run any Dart program from the command line using the Dart SDK. This is helpful for quick testing and running smaller programs. Using IDEs:
- If you're using an IDE like **Visual Studio Code**, you can install the Dart and Flutter extensions, which provide features like syntax highlighting, code completion, and an integrated terminal for running Dart code directly within the editor.

Summary of Key Concepts:

- **What is Dart**: Dart is a client-optimized language for building mobile, web, and server-side applications, especially popular due to its use in Flutter.
- **Installing Dart SDK**: The SDK provides all necessary tools for developing, compiling, and running Dart programs. The installation process differs based on the operating system.
- **Running Your First Program**: A simple "Hello, Dart!" program shows how to write and run Dart code using the dart command in a terminal. These steps form the foundation for learning Dart and building more complex applications, particularly when integrated with Flutter for mobile and web development.

1.3 Dart Basics

Dart Basics cover the foundational concepts that you need to understand before diving deeper into the language. Let's explore each subsection in detail:

1.3.1 Variables and Data Types

Variables are used to store data, while data types define the type of data a variable can hold. Dart is a strongly typed language, meaning every variable has a type. However, it supports type inference, so you don’t always need to specify the type explicitly. Variables store data, and data types define the type of data stored.

Dart supports:

```
int, double, String, bool, List, Map, dynamic, final, const
```

Key Points:

Declaring Variables:

```
int age = 25; // Explicit type declaration
var name = "John"; // Type inference (String)
dynamic anyType = "Can change"; // Can hold any type of value
```

Final and Const:

- **final**: Immutable variables; their value is set only once at runtime.
- **const**: Compile-time constants.

  ```
  final String city = "New York";
  const double pi = 3.14159;
  ```

Data Types:

- **Numbers**: int, double

  ```
  int count = 10;
  double price = 99.99;
  ```

- **Strings**: A sequence of characters enclosed in quotes.

  ```
  String message = 'Hello, Dart!';
  ```

- **Booleans**: Represent true or false.

  ```
  bool isActive = true;
  ```

- **Lists** (Arrays): Ordered collections.

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

- **Maps**: Key-value pairs.

```
Map<String, String> user = {'name': 'John', 'age': '30'};
```

Challenges:

- Declare a **final** variable to store the name of your favorite book and try changing its value.
- Create a variable using **dynamic** and assign a value. Then, change its type and print it.
- Write a Dart program to calculate the area of a rectangle using double variables for length and breadth.
- Declare a List of integers and calculate their sum.
- Create a Map to store three cities as keys and their population as values. Retrieve and print the population of a specific city.

1.3.2 Functions

Functions are reusable blocks of code designed to perform specific tasks. They allow you to organize and modularize your code.

Key Points:

- Defining a Function:

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

- Calling a Function:

```
greet(); // Outputs: Hello!
```

- Parameters: Positional:

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

- Parameters:Named (with optional defaults):

```
void greet({String name = 'Guest'}) {
    print('Hello, $name!');
}
```

- Return Values:

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

- Arrow Syntax (Short-hand):

```
int square(int n) => n * n;
```

Challenges:

- Create a function multiply that takes two numbers as parameters and returns their product.
- Write a function that takes a list of numbers and returns their average.
- Define a function that prints the first N Fibonacci numbers.
- Write an arrow function to calculate the cube of a number.
- Create a function that takes a Map of students' names and marks, and prints the name of the student with the highest marks.

1.3.3 Control Flow

Control flow structures like loops and conditionals help you control how code executes based on conditions. Key Points:

- Conditional Statements:

```
if (age > 18) {
   print('Adult');
} else {
  print('Minor');
}
```

- Switch-Case:

```
switch (day) {
  case 'Monday':
    print('Start of the week');
    break;
  default:
    print('Not Monday');
}
```

- Loops:

  for Loop:

```
for (int i = 0; i < 5; i++) {
 print(i);
}
```

  while Loop:

```
int i = 0;
while (i < 5) {
print(i);
i++;
}
```

  do-while Loop:

```
int i = 0;
do {
print(i);
i++;
} while (i < 5);
```

Challenges:

- Write a program that prints all even numbers from 1 to 20 using a for loop.

- Create a program that asks the user for a number and prints whether it's positive, negative, or zero.
- Write a Dart program using a switch statement to output the name of the day based on a number (1 for Monday, 2 for Tuesday, etc.).
- Implement a program that calculates the factorial of a number using a while loop.
- Write a program using a do-while loop to repeatedly ask the user for a password until they enter the correct one.

1.3.4 Classes

Dart is an object-oriented programming language. Classes are blueprints for creating objects, encapsulating data, and functionality.

Key Points:

- Defining a Class:

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

  Person(this.name, this.age);

  void displayInfo() {
    print('Name: $name, Age: $age');
  }
}
```

- Creating Objects:

```
var person = Person('Alice', 30);
person.displayInfo(); // Outputs: Name: Alice, Age: 30
```

- Inheritance:

```
class Employee extends Person {
  double salary;

  Employee(String name, int age, this.salary) : super(name, age);

  @override
  void displayInfo() {
        super.displayInfo();
        print('Salary: $salary');
    }
}
```

Challenges:

- Create a class Student with properties for name, age, and marks. Add a method to calculate the grade based on marks.
- Implement a class Animal with a method makeSound and create subclasses like Dog and Cat with their own sounds.
- Add a static property totalCars in the Car class that counts the number of objects created.
- Write a Dart program to demonstrate method overriding using inheritance.
- Create a class BankAccount with deposit and withdrawal methods, and ensure the balance doesn't go negative.

1.3.5 Exception Handling

Dart provides robust mechanisms to handle runtime errors using try-catch blocks. Key Points:

- Try-Catch:

```
try {
  int result = 10 ↙ 0; // Throws an exception
} catch (e) {
  print('Error: $e');
}
```

- Finally Block: Executes code regardless of whether an exception occurs.

```
try {
  int result = 10 ↙ 2;
} finally {
  print('Execution completed.');
}
```

- Custom Exceptions:

```
class CustomException implements Exception {
String cause;
CustomException(this.cause);
}

  void testException() {
  throw CustomException('This is a custom exception');
  }
 ```
```

**Challenges:**

- Write a program that catches a FormatException when parsing an invalid integer.
- Implement a function that takes two numbers, divides them, and handles division by zero.
- Create a custom exception class NegativeNumberException and throw it if a negative number is passed.
- Write a program that uses try-finally to ensure a resource (like a file) is always closed after use.
- Modify the provided code to log the exception details to a file instead of printing them.

## Solved Questions

**Q1**: What is Dart and why is it important in modern application development?

Answer:

Dart is an open-source programming language developed by Google that is designed for building fast and scalable applications. It is particularly important in the realm of cross-platform mobile development, as it serves as the primary language for Flutter. Dart offers key features like:

- Object-oriented structure, making it easy to manage complex codebases.
- High performance due to its just-in-time (JIT) compilation and ahead-of-time (AOT) compilation.
- Strong typing with optional static types, helping developers catch errors early.
- Concurrency support via isolates, making it easier to write performant multi-threaded code.
- Cross-platform development, allowing you to write code once and run it on multiple platforms (iOS, Android, Web, Desktop).
```

Q2: Explain the process of installing Dart SDK on a Windows machine. Answer: To install the Dart SDK on Windows:

1. Download the Dart SDK:

- Go to the official Dart website: https://dart.dev/get-dart.
- Download the .zip file for Windows.

2. Extract the SDK:

- Extract the .zip file to a folder on your computer (e.g., **C:\dart**).

3. Add Dart to the PATH:

- Right-click on This PC > Properties > Advanced system settings > Environment Variables.
- Under System Variables, find Path, click Edit, and add the path to Dart's bin directory (e.g., **C:\dart\dart-sdk\bin**).

4. Verify Installation:

- Open Command Prompt and run **dart --version**. You should see the installed Dart version printed in the terminal.

Q3: Write a simple Dart program that prints "Hello, Dart!" to the console.

Answer:

- Here's a simple Dart program:

```
    void main() {
    print('Hello, Dart!');
}
```

Explanation:

- **void main()** is the entry point for every Dart program.
- **print()** is a function that outputs the string to the console.

Points to Remember

1. Dart's Main Function: Every Dart program must have a main() function as the entry point. It's where the program execution starts. **:v**
2. Dart Syntax: Dart uses semicolons **;** to terminate statements. Always ensure each statement ends with a semicolon.

3. Hot Reload in Flutter: Dart powers Flutter, which provides the feature of hot reload, allowing developers to see changes instantly in their applications during development.
4. Variables in Dart: Dart supports both var, dynamic, and specific types like int, double, String, etc. The type of the variable can be inferred by Dart or explicitly defined.

Exercise

1. Exercise 1: Write a Dart program to add two numbers.

- Problem: Write a Dart program that takes two numbers as input and outputs their sum.
- Solution:

```
import 'dart:io';

void main() {
print('Enter first number:');
var num1 = int.parse(stdin.readLineSync()!);

print('Enter second number:');
var num2 = int.parse(stdin.readLineSync()!);

var sum = num1 + num2;

print('The sum of $num1 and $num2 is: $sum');
}
```

Explanation:

- The program uses stdin.readLineSync() to read input from the user.
- The **int.parse()** method is used to convert the string input to an integer.
- The result is printed using string interpolation.

2. Exercise 2: Write a Dart program to check if a number is even or odd. Solution:

```
import 'dart:io';

void main() {
print('Enter a number:');
var num = int.parse(stdin.readLineSync()!);

   if (num % 2 == 0) {
         print('$num is even.');
   } else {
         print('$num is odd.');
```

```
    }
  }
```

Explanation:

- The program checks if the number is divisible by 2. If true, it is even; otherwise, it is odd.

Activity

Activity 1: Install Dart SDK and Write Your First Program

1. Install the Dart SDK on your system by following the instructions for your operating system.
2. Write a simple Dart program that prints `"Hello, Dart!"` to the console.
3. Experiment with changing the printed message, such as `"Hello, World!"` or your name, and re-run the program. Activity 2: Experiment with Variables and Data Types
4. Create a Dart program that declares variables of different types (e.g., int, double, String) and prints them.
5. Try using var to let Dart infer the type of the variable.
6. Modify the program to perform simple arithmetic (e.g., addition, subtraction) with these variables and print the result.

Challenges

Challenge 1: Create a Simple Calculator

- Write a Dart program that simulates a basic calculator. It should allow the user to enter two numbers and select an operation (addition, subtraction, multiplication, division) to perform. The program should output the result of the operation.

Challenge 2: Fibonacci Series

- Write a Dart program that generates and prints the first n Fibonacci numbers, where n is provided by the user. Solution Outline:

- Implement a function that calculates the Fibonacci series.

- Prompt the user to enter n, and print the first n Fibonacci numbers.

Summary

- Solved Questions provide foundational knowledge about Dart, including its importance and installation process.

- Points to Remember highlight key aspects of Dart syntax and behavior.
- Exercises reinforce the learning through practical coding tasks.
- Activities encourage hands-on learning and experimentation with Dart.
- Challenges push the learner to apply what they've learned and solve more complex problems.

1.4 Advanced Dart Concepts

This section explains more advanced Dart concepts in-depth, covering Null Safety, Generics, Collections, Futures, Mixins, Extensions, and Packages. It includes code examples and challenges to help reinforce each concept.

1.4.1 Null Safety

What Is Null Safety?

Null safety in Dart is a feature that prevents null reference errors at compile-time"An informative illustration depicting working with APIs in Flutter. The image should show how Flutter communicates with a remote API using the http package for sending and receiving data. Visualize the process of making an HTTP request, handling JSON responses, and updating the UI with the fetched data. Include representations of API endpoints, network requests, and data parsing. The design should clearly display the flow from Flutter app to server and back, demonstrating efficient API interaction in a clean and modern style.". By default, variables cannot hold null unless explicitly declared as nullable. This feature helps avoid many runtime errors related to null dereferencing.

Key Points:

- Non-nullable variables: A variable must always have a value and cannot be null.

  ```
  int number = 10; // Non-nullable variable
  number = null; // Error: The value can't be null
  ```

- Nullable variables: Use the ? modifier to make a variable nullable.

  ```
  int? age = null; // Nullable variable
  ```

- Null-aware operators:

- ?: Used to check if a variable is null before performing an action.

```
String? name = null;
print(name?.length); // Safely returns null if name is null
```

- ??: Provides a default value if the variable is null.

```
String? name;
print(name ?? "Unknown"); // Output: Unknown
```

- !: Forces a nullable variable to be treated as non-null, but can cause errors if the value is actually null.

```
String? name = "Alice";
print(name!); // Output: Alice
```

Code Example:

```
void main() {
int? age = 30; // Nullable variable
int age2 = age ?? 25; // If age is null, use 25

String? name = null;
print(name?.length); // Output: null
print(name ?? "Anonymous"); // Output: Anonymous
}
```

Five Challenges:

1. Write a program that takes a nullable String and prints its length only if it's not null. Use null-aware operators.
2. Modify the above program to return a default value if the string is null.
3. Create a function that accepts a nullable int and returns a default value if it’s null.
4. Implement a function that safely handles nullable variables without using the ! operator.
5. Create a class with nullable and non-nullable properties, and demonstrate both null-aware operators and forced unwrapping.

1.4.2 Generics

What Are Generics?

Generics enable you to write reusable, type-safe code. Instead of using specific data types, you can define placeholders (type parameters) that allow functions, classes, and methods to work with any data type.

Key Points:

- Generic Classes: A class that can handle any type.

  ```
  class Box<T> {
  T value;
  Box(this.value);
  }


  void main() {
  var box = Box<int>(42);
  print(box.value); // Output: 42
  }
  ```

- Generic Functions: Functions can also accept generic types, allowing flexibility.

  ```
  T getFirst<T>(List<T> items) {
    return items.isEmpty ? null : items[0];
  }


  void main() {
  print(getFirst([1, 2, 3])); // Output: 1
  print(getFirst(["a", "b", "c"])); // Output: a
  }
  ```

  Code Example:

  ```
  // Generic class
  class Container<T> {
  T value;
  Container(this.value);

  void printValue() {
    print(value);
  }
  }
  ```

```
void main() {
var intContainer = Container<int>(10);
intContainer.printValue(); // Output: 10

var stringContainer = Container<String>("Hello");
stringContainer.printValue(); // Output: Hello
}
```

Five Challenges:

1. Create a generic Stack class to hold any type of elements and implement push and pop methods.
2. Write a generic function to find the middle element of a list.
3. Implement a generic class Pair<T, U> to store two values of different types and display them.
4. Create a generic function that swaps two values and returns them.
5. Modify the Container class to add methods for comparing equality between values of the same type.

1.4.3 Collections

What Are Collections?

Dart provides several built-in collections like Lists, Sets, and Maps to manage data. These collections allow you to store multiple elements and manipulate them efficiently. Key Points:

- **Lists**: Ordered collection of elements, accessible by index.

```
List<int> numbers = [1, 2, 3, 4, 5];
print(numbers[2]); // Output: 3
```

- **Sets**: Unordered collection of unique elements.

```
Set<int> uniqueNumbers = {1, 2, 3, 4};
uniqueNumbers.add(5);
print(uniqueNumbers); // Output: {1, 2, 3, 4, 5}
```

- **Maps**: Collection of key-value pairs.

```
Map<String, String> capitals = {
"USA": "Washington DC",
"India": "New Delhi"
};
print(capitals["India"]); // Output: New Delhi
```

Code Example:

```
void main() {
// List
List<String> fruits = ["Apple", "Banana", "Cherry"];
   fruits.add("Date");
   print(fruits); // Output: [Apple, Banana, Cherry, Date]

// Set
Set<String> colors = {"Red", "Green", "Blue"};
   colors.add("Yellow");
   print(colors); // Output: {Red, Green, Blue, Yellow}

// Map
Map<String, String> countries = {"USA": "Washington", "India": "New Delhi"};
   countries["Canada"] = "Ottawa";
   print(countries); // Output: {USA: Washington, India: New Delhi, Canada: Ottawa}
}
```

Five Challenges:

1. Create a program that stores a list of numbers, removes duplicates, and prints the unique numbers.
2. Write a Dart program that counts the frequency of each character in a string using a Map.
3. Implement a function that merges two lists and removes duplicate values.
4. Create a Map to store student names and their grades, and calculate the average grade.
5. Write a program that creates a Set of even numbers between 1 and 20.

1.4.4 Futures

What Are Futures?

A Future represents a value that might be available at some point in the future. Futures are commonly used for handling asynchronous operations, such as fetching data from a network or reading files.

Key Points:

- Creating a **Future**:

```
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () => "Data fetched");
}
```

- Using **then()** and **catchError()**` to handle results and errors:

```
fetchData().then((data) {
  print(data); // Output: Data fetched
}).catchError((e) {
  print(e);
});
```

Code Example:

```
void main() {
// Creating and handling a Future
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () => "Data fetched successfully");
}


fetchData().then((data) {
  print(data); // Output: Data fetched successfully
}).catchError((e) {
  print(e);
  });
}
```

Five Challenges:

1. Write a program that simulates an API call using Future.delayed and prints a message after 3 seconds.
2. Create a function that returns a Future and uses async/await to handle the result.
3. Modify the program to handle multiple asynchronous calls and output results in order.
4. Write a program that handles errors during asynchronous operations using try-catch.
5. Create a function that returns a Future, but throws an error if the input is less than 0.

1.4.5 Mixins and Extensions

What Are Mixins and Extensions?

Mixins allow you to add functionality to classes without using inheritance. Extensions enable you to add methods to existing classes.

Key Points:

- Mixins: Add functionality to classes using the mixin keyword.

```
mixin Swimming {
void swim() {
   print("Swimming");
   }
}

class Person with Swimming {
void talk() {
   print("Talking");
   }
}
```

- Extensions: Add methods to existing classes.

```
extension StringReversal on String {
String reverse() {
   return this.split('').reversed.join('');
   }
}

void main() {
String word = "Dart";
print(word.reverse()); // Output: traD
}
```

Code Example:

```
// Using a Mixin
mixin Flying {
void fly() {
print("Flying");
   }
}

class Bird with Flying {
void sing() {
   print("Singing");
```

```
    }
}

void main() {
Bird bird = Bird();
bird.fly(); // Output: Flying
bird.sing(); // Output: Singing
}

// Using an Extension
extension StringUppercase on String {
String toUpperCaseFirst() {
    return this.isNotEmpty ? this[0].toUpperCase() + this.substring(1) : this;
    }
}

void main() {
    print("hello".toUpperCaseFirst()); // Output: Hello
}
```

Five Challenges:

1. Write a program that uses a mixin to add the ability to jump to a Person class.
2. Create a Vehicle class with a move() method. Use a mixin to add FuelConsumption functionality.
3. Implement an extension on List that finds the median value of a list of numbers.
4. Write a mixin that logs method calls and apply it to a class.
5. Create an extension that adds a method to reverse the words in a sentence.

1.4.6 Packages

What Are Packages?

Packages are reusable libraries and tools that you can integrate into your projects. Dart uses the pub.dev repository to manage dependencies. Key Points:

- Adding Packages: Add a package to **pubspec.yaml**:

```
dependencies:
http: ^0.13.3
```

- Using the pub get command:

```
dart pub get
```

- Importing packages:

```
import 'package:http/http.dart' as http;
```

Code Example:

```
import 'package:http/http.dart' as http;

void main() async {
var response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));
print(response.body);
}
```

Challenges:

1. Add the http package to your project and fetch data from a public API.
2. Create a package that handles basic math operations and use it in your project.
3. Write a program that reads and writes to a file using the **path_provider** package.
4. Create a package that validates email addresses and use it in a Dart application.
5. Publish your own Dart package to pub.dev.