Question Answer

1.Explain the concepts of positional parameters ,named parameters and required parameters in Dart with examples

2.Explain the concept of async and await in Dart. How do they help with asynchronous programming? Provide an example demonstrating their usage.

3.Explain the concept of widget lifecycle in Flutter and describe the different lifecycle methods available. Provide an example demonstrating the usage of lifecycle methods.

4.What is the difference between StatefulWidget and StatelessWidget in Flutter? Provide an example demonstrating the usage of both.

5.What is null safety in Dart? Explain the benefits of using null safety.

6.Explain the difference between a nullable and non-nullable type in Dart. Provide an example demonstrating the usage of each type.

7.What is the difference between a GET request and a POST request? Provide an example use case for each in Flutter.

8.List different types of APIs and explain any two of them.

9.Why would you choose Flutter for mobile app development? Provide at least three advantages of using Flutter.

10.Describe the main features and benefits of Dart as a programming language
11. Explain Flutter architectural layers.

Short notes:
1.Dio package
2.Http request
3.Exception handling
4.Status code

5.Types of widgets in Flutter

## 1.Explain the concepts of positional parameters ,named parameters and required parameters in Dart with examples

Answer

**Positional parameter:**

Positional parameters are the default type of parameters in Dart. They are defined by their position or order in the function declaration. **When calling a function with positional parameters, you must provide the arguments in the same order as the parameters in the function declaration.**

```
void greet(String name, [String? prefix = 'Hello']) {
  print('$prefix $name');
}

void main() {
  greet('John'); // Output: Hello John
  greet('Jane', 'Hi'); // Output: Hi Jane
}
```

In this example, the **greet** function has two positional parameters: **name** and **greeting**. When calling the **greet** function, the arguments **'John'** and **'Hello'** are provided in the same order as the parameters.

**Named Parameters:**

Named parameters in Dart allow you to specify arguments using their names, rather than relying on their position. They are defined by enclosing them in curly braces ({}). When calling a function with named parameters, you explicitly provide the parameter names along with their corresponding values.

```
void greet({String? name, String? greeting}) {
  print('$greeting, $name!');
}
void main() {
  greet(name:'John',greeting: 'Hello'); // Output: Hello, John!

}
```

In this example, the greet function has two named parameters: name and greeting. When calling the greet function, the arguments are provided using their corresponding parameter names.

**Required Parameters:**

In Dart, required parameters are used when you want to enforce that certain arguments must be provided when calling a function. Required parameters do not have default values and are denoted by not using any square brackets or curly braces.

```dart
void greet({required String name, String? prefix}) {
  print('$prefix $name');
}

void main() {
  greet(name: 'John', prefix: 'Hi');
}
```

In this example, the name parameter is marked as required. It must be provided when calling the greet function. If a required parameter is not provided, a compilation error will occur.

## 2.Explain the concept of async and await in Dart. How do they help with asynchronous programming? Provide an example demonstrating their usage. (8 marks)

Answer:

Asynchronous programming in Dart allows for non-blocking execution of tasks, such as network requests or file operations, without blocking the execution of the main thread. Dart provides the async and await keywords to handle asynchronous operations.

Here's an example demonstrating the usage of async and await:

```dart
Future<void> fetchData() async {
  print('Fetching data...');
  await Future.delayed(Duration(seconds: 2));
  print('Data fetched!');
}

void main() async {
  print('Start');
  await fetchData();
  print('End');
}
```

In this example, the fetchData function is marked as asynchronous using the async keyword. The await keyword is used to pause the execution of the function until the Future.delayed completes after a 2-second delay. The await keyword ensures that the code following it will not execute until the awaited operation is complete. The output will be:

```
Start
Fetching data...
Data fetched!
End
```

3.Explain the concept of widget lifecycle in Flutter and describe the different lifecycle methods available. Provide an example demonstrating the usage of lifecycle methods. (8 marks)

4.What is the difference between StatefulWidget and StatelessWidget in Flutter? Provide an example demonstrating the usage of both. (8 marks)

Answer:

The main difference between StatefulWidget (STF) and StatelessWidget (STL) in Flutter lies in their ability to manage and update state.

StatelessWidget :

A StatelessWidget is immutable and does not hold any mutable state. It represents part of the user interface that does not change over time. It is typically used for displaying static content. Here's an example:

```dart
import 'package:flutter/material.dart';

class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.blue,
      child: Text('Hello, Stateless Widget!'),
    );
  }
}
```

StatefulWidget :
A StatefulWidget can hold mutable state and is used for parts of the user interface that can change dynamically. It allows for stateful interactions and updates based on user input or other triggers. Here's an example:

```dart
import 'package:flutter/material.dart';

class MyStatefulWidget extends StatefulWidget {
  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  int _counter = 0;

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Counter: $_counter'),
        ElevatedButton(
          onPressed: () {
            setState(() {
              _counter++;
            });
          },
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

In this example, the MyStatefulWidget extends StatefulWidget and has an associated state class _MyStatefulWidgetState that extends State. The state class holds the mutable state (_counter in

this case) and the setState() method is used to update the state and trigger a rebuild of the widget whenever _counter changes.

## 5.What is null safety in Dart? Explain the benefits of using null safety.

Answer:
Null safety is a feature in Dart that ensures variables are non-null by default, reducing the risk of null reference errors. It introduces the concepts of nullable and non-nullable types. Non-nullable types (denoted as Type) cannot be assigned a value of null unless explicitly marked as nullable (Type?).

Benefits of using null safety in Dart:

1. Elimination of Null Reference Errors: Null safety helps prevent null reference errors, which occur when a variable is unexpectedly null at runtime. By defaulting to non-nullability, Dart encourages developers to handle null values explicitly, reducing the likelihood of null-related errors.

2. Enhanced Code Reliability: With null safety, code becomes more reliable as developers can express nullability intentions explicitly. This makes it easier to reason about the code's behavior and reduces the risk of encountering unexpected null values.

3. Improved Tooling Support: Dart's null safety is supported by powerful IDEs, providing enhanced static analysis, warnings, and code suggestions. These tools can detect potential null-related issues during development, helping developers catch errors early and write more robust code.

4. Sound Type System: Null safety enhances Dart's type system, allowing it to differentiate between nullable and non-nullable types. This provides more accurate type information, catches type-related errors, and improves overall code correctness.

5. Better Documentation and Readability: By explicitly annotating variables with nullable or non-nullable types, null safety enhances code readability and documentation. It helps other developers understand the expected nullability of variables and reduces the need for additional null checks or comments.

## 6.Explain the difference between a nullable and non-nullable type in Dart. Provide an example demonstrating the usage of each type. (8 marks)

Answer:

Nullable Types:

A nullable type in Dart is denoted by appending a question mark (?) to the type declaration. It indicates that a variable can hold either a non-null value of the specified type or a null value.


Non-Nullable Types:

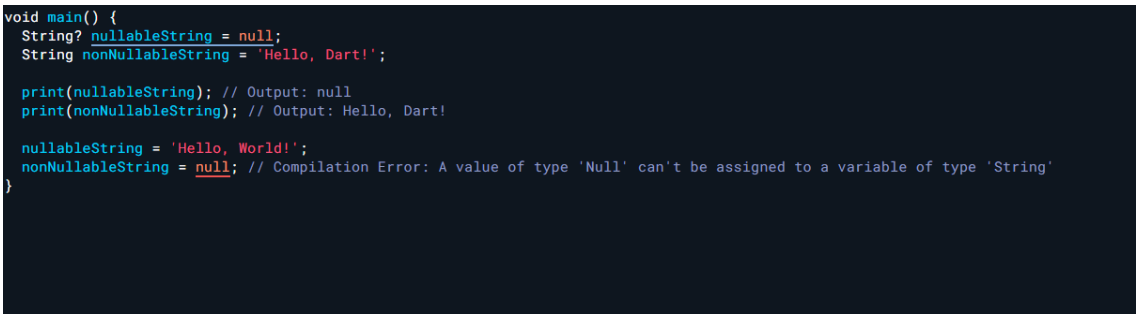
A non-nullable type in Dart is the default behavior when the question mark (?) is not used. It indicates that a variable can only hold non-null values of the specified type.


In this example, the variable nonNullableString is declared as a non-nullable String type. It can only hold non-null string values and cannot be assigned a value of null.

Usage differences between nullable and non-nullable types:

1. Assignment:
    - Nullable type: Can be assigned either a non-null value or null.
    - Non-nullable type: Can only be assigned non-null values; assigning null will result in a compilation error.
2. Nullability Enforcement:
    - Nullable type: Allows you to explicitly handle the possibility of null values and perform null checks before accessing the value.
    - Non-nullable type: Provides a guarantee that the variable will never hold a null value, reducing the need for explicit null checks.

Example showcasing the usage of nullable and non-nullable types:



```dart
void main() {
  String? nullableString = null;
  String nonNullableString = 'Hello, Dart!';

  print(nullableString); // Output: null
  print(nonNullableString); // Output: Hello, Dart!

  nullableString = 'Hello, World!';
  nonNullableString = null; // Compilation Error: A value of type 'Null' can't be assigned to a variable of type 'String'
}
```

In this example, nullableString is initially assigned the value null and later assigned the value 'Hello, World!'. The print statement correctly outputs null for the nullable variable. On the other hand, assigning null to nonNullableString results in a compilation error since it's a non-nullable type.

The usage of nullable and non-nullable types allows you to express nullability intentions clearly in your code and helps prevent null reference errors.

## 7.What is the difference between a GET request and a POST request? Provide an example use case for each in Flutter.

The main difference between a GET request and a POST request is the way data is sent to the server. In a GET request, data is sent via query parameters in the URL, while in a POST request, data is sent in the request body. Example use cases:

- GET: Fetching a list of products from an e-commerce API.
- POST: Sending user registration data to a server for creating a new account.

Post request:
Data can be passed to an API request using Dio by providing it in the data parameter of the request method. For example:

```
Dio().post(
  'https://api.example.com/endpoint',
  data: {
    'name': 'John Doe',
    'email': 'johndoe@example.com',
  },
```

```
);
```

Get request:
To parse JSON data received from an API response using Dio, you can access the data property of the Response object and use JSON parsing methods. For example:

```
Response response = await Dio().get('https://api.example.com/data');
Map<String, dynamic> data = response.data;
// Parse the JSON data and use it in your app
```

## 8.List different types of APIs and explain any two of them.

There are several types of APIs commonly used in software development. Here are five different types of APIs, along with explanations for two of them:

1. RESTful APIs (Representational State Transfer):
   RESTful APIs are based on the principles of the REST architectural style. They use HTTP methods to perform operations on resources identified by unique URLs. RESTful APIs are stateless and return responses in formats like JSON or XML. They are widely used, flexible, and scalable.

2. SOAP APIs (Simple Object Access Protocol):
   SOAP APIs use the SOAP protocol for communication. They define a set of XML-based standards for exchanging structured information between systems. SOAP APIs typically involve sending XML-formatted requests over HTTP or other protocols. They provide a strict interface definition through Web Services Description Language (WSDL) and offer features such as security and reliability.

3. GraphQL APIs:
   GraphQL APIs allow clients to query and retrieve precisely the data they need. Clients specify their data requirements using a query language, and the server responds with the requested data in a JSON format. GraphQL APIs provide efficient data fetching and enable real-time updates through subscriptions.

4. WebSocket APIs:

WebSocket APIs enable bidirectional communication between clients and servers over a single, long-lived connection. Unlike traditional HTTP requests, WebSocket APIs allow real-time, full-duplex communication, making them suitable for applications requiring instant data updates, such as chat applications or real-time collaboration tools.

5. JSON-RPC APIs:
JSON-RPC (Remote Procedure Call) APIs use the JSON format to define remote method invocations. They allow clients to make requests to a server by specifying a method and passing parameters in a JSON payload. The server processes the request and returns the response, also in JSON format. JSON-RPC APIs are lightweight and suitable for client-server communication.

Let's dive deeper into RESTful APIs and WebSocket APIs as examples:

RESTful APIs:
RESTful APIs follow the principles of REST architecture, which include using HTTP methods, stateless communication, and resource-based URLs. They allow clients to perform operations like retrieving, creating, updating, and deleting resources using standard HTTP methods.

WebSocket APIs:
WebSocket APIs provide a persistent connection between clients and servers, allowing real-time, bidirectional communication. Once established, the connection remains open, enabling instantaneous data transfer between the client and server. WebSocket APIs are well-suited for applications that require real-time updates or instant messaging capabilities.

WebSocket APIs differ from traditional HTTP APIs because they don't require the client to repeatedly initiate new connections for each interaction. Instead, the connection is kept open, enabling efficient and instantaneous data exchange between the client and server.

WebSocket APIs use the WebSocket protocol, which is designed to provide a full-duplex communication channel over a single TCP connection. This allows both the client and server to send and receive messages at any time, making it

ideal for applications that rely on real-time data synchronisation or live streaming.

WebSocket APIs are commonly used in applications such as real-time chat, multiplayer games, collaborative tools, and financial applications that require immediate updates on stock prices or market data. They offer improved performance and reduced overhead compared to other APIs that require repeated requests and responses.

Overall, RESTful APIs and WebSocket APIs represent different approaches to building APIs. RESTful APIs follow a stateless, resource-based approach using standard HTTP methods, while WebSocket APIs provide persistent, bidirectional communication for real-time updates. The choice between these types of APIs depends on the specific requirements of the application and the desired functionality.

## 9.Why would you choose Flutter for mobile app development? Provide at least three advantages of using Flutter.

- Flutter offers cross-platform development, allowing you to write code once and deploy it on multiple platforms like iOS, Android, and even the web. This saves development time and resources.
- Flutter provides a rich set of customizable UI components called widgets. These widgets allow for fast and flexible UI development, resulting in visually appealing and highly interactive user interfaces.
- Flutter uses the Dart programming language, which offers features like a just-in-time (JIT) compiler for faster development cycles, a reactive programming model, and modern language constructs that simplify app development.

## 10.Describe the main features and benefits of Dart as a programming language

- Dart supports both just-in-time (JIT) and ahead-of-time (AOT) compilation, providing flexibility in development and efficient execution.
- Dart has a strong type system, enabling static type checking to catch errors during development, leading to more robust and reliable code.
- Dart supports asynchronous programming using async and await keywords, making it easier to handle asynchronous operations and avoid blocking the user interface.
- Dart offers features like garbage collection, isolates for concurrency, and a comprehensive standard library, making it suitable for building complex and scalable applications.