Q.1 (a) What is Flutter? Explain Files and Folder structures of Flutter Project.

ANS: Flutter is an open-source UI software development kit (SDK) created by Google. It is used to develop applications for Android, iOS, Windows, Mac, Linux, Google Fuchsia, and the web from a single codebase. Flutter allows developers to create visually attractive, natively compiled applications with a single programming language (Dart) and codebase.

Key Features of Flutter:

- **Fast Development:** Hot reload feature allows quick iterations and development.
- **Expressive and Flexible UI:** Flutter provides a rich set of customizable widgets for building beautiful UIs.
- **Native Performance:** Flutter compiles to native ARM code for both iOS and Android, enabling high performance.
- **Single Codebase:** Write once, run anywhere approach reduces development time and effort.

File and Folder Structure of a Flutter Project:

When you create a new Flutter project, a set of files and directories are generated. Here's a typical structure and explanation of each:

```
my_flutter_app/
├── android/
├── ios/
├── lib/
|   ├── main.dart
├── test/
├── .gitignore
├── .metadata
├── pubspec.yaml
├── pubspec.lock
├── README.md
└── build/
```

1. android/ and ios/

These directories contain platform-specific code and configuration files.

- **android/**: Contains the Android-specific code, configurations, and Gradle build files. This folder is used when building the app for Android.
- **ios/**: Contains the iOS-specific code, configurations, and Xcode project files. This folder is used when building the app for iOS.

2. `lib/`

This is the main directory where your Dart code resides.

- **main.dart**: The entry point of the Flutter application. This file contains the main function that starts the app.

3. `test/`

Contains the test files for your Flutter app. You can write unit and widget tests to ensure your app works as expected.

4. `.gitignore`

Specifies files and directories that should be ignored by Git. This helps keep unnecessary files out of version control.

5. `.metadata`

Contains metadata about your Flutter project. It is used internally by Flutter tools.

6. `pubspec.yaml`

The most important configuration file for a Flutter project. It manages the project's dependencies, assets, and other configurations.

- **dependencies**: Lists the packages that your app depends on.
- **dev_dependencies**: Lists the packages needed only for development, such as testing packages.
- **flutter**: Configures assets and other Flutter-specific settings.

7. `pubspec.lock`

This file is generated automatically by the Dart package manager, `pub`, and it lists the exact versions of dependencies used in your project.

8. `README.md`

A markdown file that contains the description and instructions for your project. You can customize it to provide an overview of your app.

9. `build/`

This directory is generated automatically when you build your Flutter app. It contains the compiled output of your project. You generally do not need to interact with this directory directly.

(b) Create a Flutter program to display a message “All the Best for Your Exam” as a text in the centre of the screen also display demo of output.

ANS: A simple Flutter program to display the message "All the Best for Your Exam" centered on the screen.

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


void main() {
  runApp(MyApp());
}


class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Exam Wishes'),
        ),
        body: Center(
          child: Text(
            'All the Best for Your Exam',
            style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
          ),
        ),
      ),
    );
  }
}
```

Explanation

1. **main. dart**:
    - This is the entry point of the application.

    - `runApp` is called with the `MyApp` widget which starts the app.
2. **MyApp class**:
    - A stateless widget that returns a `MaterialApp`.
    - The `MaterialApp` has a `Scaffold` with an `AppBar` and a `Center` widget.
3. **Scaffold**:
    - Provides the basic visual layout structure.
    - Contains an `AppBar` with the title "Exam Wishes".
4. **Center**:
    - Canters its child widget.
    - The child is a `Text` widget displaying "All the Best for Your Exam" with styling.

Q.2 (a) List out the Operators in Dart. Explain any two types of Operator in Dart with example.

Ans.: Dart provides various operators that you can use to perform operations on values and variables. Here's a list of some common operators in Dart:

1. Arithmetic Operators:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)

2. Relational Operators:

- == (Equality)
- != (Inequality)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

3. Logical Operators:

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

4. Assignment Operators:

- = (Assignment)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

5. Increment/Decrement Operators:

- ++ (Increment)
- -- (Decrement)

6. Type Test Operators:

- is (Type test)
- as (Type cast)

7. Bitwise Operators (for integer types):

- & (Bitwise AND)
- | (Bitwise OR)
- ^ (Bitwise XOR)
- ~ (Bitwise NOT)
- << (Shift left)
- >> (Shift right)

8. Conditional Operator (Ternary Operator):

- condition ? expr1 : expr2 (Returns expr1 if condition is true, otherwise expr2)

Explanation of Two Types of Operators in Dart

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations. Dart supports the following arithmetic operators:

- **Addition (+)**: Adds two operands.
- **Subtraction (-)**: Subtracts the right operand from the left operand.
- **Multiplication (*)**: Multiplies two operands.
- **Division (/)**: Divides the left operand by the right operand and returns a double.
- **Integer Division (~/)**: Divides the left operand by the right operand and returns an integer.
- **Modulus (%)**: Returns the remainder when the left operand is divided by the right operand.

```
void main() {
  int a = 10;
  int b = 3;

  print('a + b = ${a + b}'); // Addition: 10 + 3 = 13
  print('a - b = ${a - b}'); // Subtraction: 10 - 3 = 7
  print('a * b = ${a * b}'); // Multiplication: 10 * 3 = 30
  print('a / b = ${a / b}'); // Division: 10 / 3 = 3.3333333333333335
  print('a ~/ b = ${a ~/ b}'); // Integer Division: 10 ~/ 3 = 3
  print('a % b = ${a % b}'); // Modulus: 10 % 3 = 1
}
```

2. Logical Operators

Logical operators are used to combine conditional statements. Dart supports the following logical operators:

- **Logical NOT (!)**: Inverts the value of a boolean.
- **Logical AND (&&)**: Returns true if both operands are true.
- **Logical OR (||)**: Returns true if at least one of the operands is true.

Example

```
void main() {
  bool x = true;
  bool y = false;

  print('!x = ${!x}'); // Logical NOT: !true = false
```

```
  print('x && y = ${x && y}'); // Logical AND: true && false = false
  print('x || y = ${x || y}'); // Logical OR: true || false = true
}
```

(b) Explain flow control statements in a Flutter with suitable example.

ANS: Flow control statements in Flutter, which uses the Dart programming language, are essential for managing the execution flow of your code. Here are the main types of flow control statements in Dart:

- If and else
- for loops
- while and do-while loops
- break and continue
- switch and case
- assert

1. if and else: Dart like any other programming language supports the **if** statement with optional **else**. This statement helps to redirect flow based on the truthfulness of a condition.

```
if(isOpen())
{
       enter();
}
Else
{
       ringTheBello();
}
```

2. For Loops: - In Dart, you can iterate with a standard **for loop** as well as a **for-in loop** (if the object that you are trying to iterate over is an Iterable and you don't need to know the current iteration counter).

```
// For loop Example
for(var i = 0; i<3; i++)
{
       print(i);
}// For-in loop Example List
list1 = [1,2,3,4,5,6];
for (var item in list1)
{
```

```
        print(item);
}
```

3. while and do-while loops: These loops are based on the condition but with some differences. The **while loop** evaluates the condition before entering the loop.

```
while(doorIsOpen())
{
        doSomething();
}
```

Do-while: The **do-while loop** executes the body at least once before checking the truthiness of the condition.

```
Do
{
        doSomething ();
}
while(doorIsOpen());
```

4. Break and continue :- You can use the **break** to stop the loop and **continue** to skip to the next iteration.

```
////// break example
while(true)
{
        if(doorIsOpen()) break;
        doSomething();
}/////// continue example
for(var i = 0; i<4; i++)
{
        if(item < 2) continue;
        print(item);
}
```

5. Switch and case : A switch statement is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via search and map. Switch statements in dart compare integer, string, or compile-time constants using == . The compared objects must all be instances of the same class(and not of any subtype of its subtypes), and the class must not override ==.

```
var result = 'LOADED';
switch(result)
{
        case 'LOADING':
                executeLoading();
                break;
        case 'LOADED':
                executeLoaded();
```

```
            break;
      default:
            executeUnknown();
}
```

6. Assert: You can use the assert statement during development (does not work in production) to disrupt the normal execution if a Boolean condition is false.

```
assert (name! = null);
assert (number < 100);
assert (name == 'Richard', "Name error. name does not
correspond");
```

Q.3 (a) How to collect Text inputs and set text limits in flutter?

ANS: In Flutter, collecting text input and setting text limits can be done using the `TextField` or `TextFormField` widgets. These widgets allow users to input and edit text in a form or other UI elements. You can also use `TextEditingController` to manage the text input and perform validation or set constraints like text limits.

Example: -

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

import 'package:flutter/services.dart';


void main() {

 runApp(MyApp());

}


class MyApp extends StatelessWidget {

 @override

 Widget build(BuildContext context) {

  return MaterialApp(

   home: MyForm(),

  );

 }

}


class MyForm extends StatefulWidget {
```

```
  @override
  _MyFormState createState() => _MyFormState();
}

class _MyFormState extends State<MyForm> {
  final TextEditingController _controller = TextEditingController();

  void _submitData() {
    print('Submitted Text: ${_controller.text}');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Text Input Example')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: _controller,
              decoration: InputDecoration(
                labelText: 'Enter Text',
                counterText: '', // Optional: to hide the counter
              ),
              maxLength: 10, // Limit to 10 characters
              inputFormatters: [
                LengthLimitingTextInputFormatter(10), // Limit to 10 characters
              ],
            ),
            SizedBox(height: 20),
```

```
          ElevatedButton(
            onPressed: _submitData,
            child: Text('Submit'),
          ),
        ],
      ),
    ),
  );
 }
}
```

(b) Describe following widgets in flutter: Scaffold, AppBar, Container, Buttons.

ANS: Flutter provides a rich set of widgets to create beautiful and responsive user interfaces. Here, we'll describe some commonly used widgets: Scaffold, AppBar, Container, and various types of buttons.

1. Scaffold

Scaffold is a basic layout structure for a Flutter app. It provides APIs for showing drawers, snack bars, and bottom sheets. It helps implement the visual structure of a Material Design app.

Key Properties:

- appBar: A top app bar providing a title, actions, and navigation.
- body: The primary content of the screen.
- floatingActionButton: A button that floats above the content, typically used for primary actions.
- drawer: A panel that slides in from the side of the screen, often used for navigation.

Example:

dart
Copy code

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Scaffold Example'),
        ),
        body: Center(
          child: Text('Hello, World!'),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: () {},
          child: Icon(Icons.add),
        ),
      ),
    );
  }
}
```

2. **AppBar**

AppBar is a Material Design app bar. It's typically placed at the top of the Scaffold. It can contain a title, actions, and navigation controls.

Key Properties:

- title: A widget to display in the center of the app bar, usually a Text.
- actions: A list of widgets to display in a row after the title widget, usually IconButtons.
- leading: A widget to display before the title, typically an IconButton for navigation.

Example:

dart
Copy code

```
AppBar(
  title: Text('AppBar Example'),
  actions: [
    IconButton(
      icon: Icon(Icons.search),
      onPressed: () {},
    ),
    IconButton(
      icon: Icon(Icons.more_vert),
      onPressed: () {},
    ),
  ],
)
```

3. Container

Container is a versatile widget that can be used to create rectangular visual elements. It can contain a single child and allows you to apply padding, margins, borders, and other styling properties.

***Key Properties**:*

- child: The widget that is contained by the Container.
- padding: The padding inside the Container.
- margin: The margin outside the Container.
- color: The background color of the Container.
- decoration: An object to paint behind the child (e.g., BoxDecoration).

***Example**:*
dart
Copy code
```
Container(
  padding: EdgeInsets.all(16.0),
  margin: EdgeInsets.all(8.0),
  color: Colors.blue,
  child: Text('This is a Container'),
)
```
4. Buttons

Flutter provides several types of buttons to perform actions when pressed. Some common buttons include ElevatedButton, TextButton, OutlinedButton, and IconButton.

Elevated Button

A Material Design raised button that provides a slightly elevated effect.

Key Properties:

- onPressed: A call-back function that is called when the button is tapped.
- child: The widget inside the button, typically a Text.

Example:
dart
Copy code
```
ElevatedButton (
  onPressed: () {
    print('ElevatedButton pressed');
  },
  child: Text('ElevatedButton'),
)
```
TextButton

A flat button without elevation, typically used for less emphasized actions.

Key Properties:

- onPressed: A callback function that is called when the button is tapped.
- child: The widget inside the button, typically a Text.

Example:
dart
Copy code
```
TextButton(
  onPressed: () {
    print('TextButton pressed');
  },
  child: Text('TextButton'),
)
```
OutlinedButton

A button with an outlined border, often used for secondary actions.

Key Properties:

- onPressed: A callback function that is called when the button is tapped.
- child: The widget inside the button, typically a Text.

Example:
dart
Copy code
```
OutlinedButton(
  onPressed: () {
    print('OutlinedButton pressed');
  },
  child: Text('OutlinedButton'),
)
```

IconButton : *A button with an icon, typically used for actions without text.*

Key Properties:

- onPressed: A callback function that is called when the button is tapped.
- icon: The icon to display inside the button.

Example:
dart
Copy code
```
IconButton(
  onPressed: () {
    print('IconButton pressed');
  },
  icon: Icon(Icons.thumb_up),
)
```

Q.4 (a) Explain the purpose of the Bottom Navigation Bar and how it enhances navigation in a Flutter app.

ANS: The Bottom Navigation Bar is a material design widget that provides quick navigation between top-level views in an app. It typically appears at the bottom of the app screen and consists of multiple

items, each representing a different section or feature of the app. When a user taps on a navigation item, the app transitions to the corresponding view.

Benefits of Using a Bottom Navigation Bar:-

1. **Improved Navigation**: It allows users to quickly switch between different sections of an app, enhancing the overall user experience.
2. **Accessibility**: Being located at the bottom of the screen, it is easily reachable with the thumb, making it convenient for one-handed use.
3. **Consistency**: Provides a consistent way to navigate, adhering to material design principles, which users are familiar with.
4. **Space Management**: Helps to declutter the interface by consolidating navigation options in a single area, freeing up space for content.
5. **Visibility**: Keeps navigation options always visible, which can improve usability by reducing the number of steps required to access different parts of the app.

Implementing a Bottom Navigation Bar in Flutter

To implement a Bottom Navigation Bar in a Flutter app, you typically use the `BottomNavigationBar` widget along with a `Scaffold` to manage the layout. Here's how you can set it up:

Example Implementation

1. **Setup**: Create a new Flutter project and set up the basic structure with `Scaffold`.
2. **BottomNavigationBar**: Add a `BottomNavigationBar` to the `Scaffold` and define the navigation items.
3. **State Management**: Use a `StatefulWidget` to manage the state of the selected navigation item and update the content accordingly.

Example :

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


void main() {
 runApp(MyApp());
}


class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: HomeScreen(),
  );
```

```
  }
}


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Bottom Navigation Bar Example'),
      ),
      body: Center(
        child: _widgetOptions.elementAt(_selectedIndex),
      ),
      bottomNavigationBar: BottomNavigationBar(
        items: const <BottomNavigationBarItem>[
          BottomNavigationBarItem(
            icon: Icon(Icons.home),
            label: 'Home',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.search),
            label: 'Search',
          ),
          BottomNavigationBarItem(
            icon: Icon(Icons.person),
            label: 'Profile',
          ),
        ],
        currentIndex: _selectedIndex,
        selectedItemColor: Colors.blue,
        onTap: _onItemTapped,
```

```
    ),
  );
 }
}
```

(b) Explain Animation Controller with example.

Ans.: The animation controller is a class that allows us to control the animation. It always generates new values whenever the application is ready for a new frame. For example, it gives the controlling of start, stop, forward, or repeat of the animation. Once the animation controller is created, we can start building other animation based on it, such as reverse animation and curved animation. Here, the duration option controls the duration of the animation process, and vsync option is used to optimize the resource used in the animation. The basic steps necessary for using an Animation Controller are:

Step 1: First, instantiate an Animation Controller with parameters, such as duration and vsync.

Step 2: Add the required listeners like addListener () or addStatusListener ().

Step 3: Start the animation.

Step 4: Perform the action in the listener call-back methods (for example, setState).

Step 5: Last, dispose of the animation.

For Example:

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget
{
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context)
 {
        return MaterialApp
        (
                title: 'Flutter Animation',
```

```
            theme: ThemeData
            (
                  // This is the theme of your application.
                  primarySwatch: Colors.blue,
            ),
            home: MyHomePage(),
      );
 }
}

class MyHomePage extends StatefulWidget
{
      _HomePageState createState() => _HomePageState();
}
class _HomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin
{
      Animation<double> animation;
      AnimationController animationController;
      @override
      void initState()
      {
       super.initState();
       animationController = AnimationController(vsync: this, duration:
       Duration(milliseconds: 2500));
       animation = Tween<double>(begin: 0.0, end: 1.0).animate(animationController);
       animation.addListener(()
       {
            setState(()
            {
                  print (animation.value.toString());
            });
```

```
        });
        animation.addStatusListener((status)
        {
                if(status == AnimationStatus.completed)
                {
                        animationController.reverse();
                }
                else if(status == AnimationStatus.dismissed)
                {
                        animationController.forward();
                }
        });
        animationController.forward();
       }
        @override
        Widget build(BuildContext context)
        {
                return Center
                (
                        child: AnimatedLogo
                        (
                                animation: animation,
                        )
                );
        }
}
class AnimatedLogo extends AnimatedWidget
{
        final Tween<double> _sizeAnimation = Tween<double> (begin: 0.0, end: 500.0);
        AnimatedLogo({Key key, Animation animation}):super(key: key, listenable: animation);
        @override
```

```
    Widget build(BuildContext context)
    {
        final Animation<double> animation = listenable;
        return Transform.scale
        (
            scale: _sizeAnimation.evaluate(animation),
            child: FlutterLogo(),
        );
    }

}
```

Q.5 (a) Define firebase and cloud Firestore. Differentiate SQL Server Database and Cloud Firestore.

Ans.: Firebase: Firebase is a backend platform for building Web, Android and IOS applications. It offers real time databases, different APIs, multiple authentication types, and hosting platform.

Cloud Firestore: Cloud Firestore is a flexible, scalable database for mobile, web, and server development from Firebase and Google Cloud. Cloud Firestore also offers seamless integration with other Firebase and Google Cloud products, including Cloud Functions.

The differences between SQL Server Database and Cloud Firebase are given below:

SQL Server and Cloud Firestore are two different types of database management systems, each with its own strengths and use cases. Here are some key differences between them:

1. Database Type:

- SQL Server is a traditional relational database management system (RDBMS). It uses structured tables with rows and columns to store data, and it enforces a schema.
- Cloud Firestore is a NoSQL database that stores data in flexible, semi-structured documents, such as JSON. It does not require a fixed schema, making it more flexible for handling unstructured or rapidly changing data.

2. Data Model:

- SQL Server uses a tabular data model where data is organized into tables, and relationships between tables are defined using foreign keys.

- Cloud Firestore uses a document-based data model. Data is stored in collections, and each document can have its own structure, allowing for nested data and more flexibility.

3. Scalability:

- SQL Server can be scaled vertically (by adding more resources to a single server) or horizontally (through clustering and sharding). Scaling is often complex and may require downtime for maintenance.
- Cloud Firestore is designed to be horizontally scalable "out of the box." It can easily handle large amounts of data and traffic without the need for complex scaling operations. Firestore is a serverless, managed service.

4. Complex Queries:

- SQL Server is optimized for complex queries and reporting. It supports SQL for querying data with powerful JOIN operations.
- Cloud Firestore is better suited for simple to moderately complex queries. It offers basic querying capabilities but lacks some of the advanced querying options found in SQL databases.

5. Real-Time Sync:

- Cloud Firestore is designed for real-time synchronization of data. It supports real-time listeners, making it suitable for applications that require live updates and collaboration.
- SQL Server does not provide built-in real-time synchronization. Achieving real-time capabilities typically requires additional libraries or frameworks.

6. Server Management:

- SQL Server requires manual server management, including tasks like updates, backups, and scaling.
- Cloud Firestore is a fully managed, serverless database, which means you don't need to manage infrastructure. Google handles server maintenance and scaling for you.

(b) **Explain Custom Scroll View using Slivers to customize scrolling behaviour in Flutter.**

ANS: Flutter provides powerful scrolling capabilities with its `CustomScrollView` widget, which allows for highly customizable scroll behavior using slivers. Slivers are special widgets that can create custom scrolling effects and layouts.

What are Slivers?

Slivers are portions of a scrollable area that you can define to create a custom scrolling effect. They are building blocks that let you create complex scrolling layouts, like lists, grids, and more.

CustomScrollView

`CustomScrollView` is a scroll view that creates custom scrolling effects using slivers. You can use various types of slivers to create different scrolling behaviors and layouts.

Commonly Used Slivers

1. **SliverAppBar**: A material design app bar that integrates with a `CustomScrollView`.
2. **SliverList**: A list of scrollable items.
3. **SliverGrid**: A scrollable grid of items.
4. **SliverToBoxAdapter**: A sliver that contains a single box widget.
5. **SliverPadding**: Adds padding around another sliver.

Example:

```
return Scaffold(
    body: CustomScrollView(
     slivers: <Widget>[
      SliverAppBar(
        pinned: true,
        expandedHeight: 200.0,
        flexibleSpace: FlexibleSpaceBar(
          title: Text('SliverAppBar'),
          background: Image.network(
            'https://flutter.dev/assets/homepage/carousel/slide_1-layer_0-2x-
72d3d5f18b51d13ec4de4b5941d0ecf2b3ef1200.png',
            fit: BoxFit.cover,
          ),
        ),
      ),
      SliverList(
        delegate: SliverChildBuilderDelegate(
          (BuildContext context, int index) {
            return Container(
              height: 100.0,
              color: index.isEven ? Colors.amber[200] : Colors.blue[200],
              child: Center(
```

```
            child: Text('List Item $index'),
          ),
        );
      },
      childCount: 10,
    ),
  ),
  SliverGrid(
    gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
      crossAxisCount: 2,
      mainAxisSpacing: 10.0,
      crossAxisSpacing: 10.0,
      childAspectRatio: 4.0,
    ),
    delegate: SliverChildBuilderDelegate(
      (BuildContext context, int index) {
        return Container(
          color: index.isEven ? Colors.green[200] : Colors.pink[200],
          child: Center(
            child: Text('Grid Item $index'),
          ),
        );
      },
      childCount: 10,
    ),
  ),
 ],
 ),
);
```