GUJARAT TECHNOLOGICAL UNIVERSITY

MCA INTEGRATED– SEMESTER IX- EXAMINATION –WINTER-2022

Subject Code: 2698606 **Date: 31/12/2022**

Subject Name: Flutter Application Development

Q.1 (a) Do as Directed.

1. Everything in Flutter is a Widget**.**

2. Stateful Widgets are static widgets. (False)

3. A collection that can contain only documents is a Cloud Firestore collection. (True)

4. Which of the following is the Stateful widgets:

a. Checkbox b. Radio Button c. Slider d. All

Ans.: The correct answer is d.

5. All the methods declared with static keyword are termed as Static Method.

6. AnimatedCrossFade constructor takes duration, firstChild, secondChild, crossFadeState, sizeCurve, and many other arguments. (True)

7. The GestureDetector gestures that you can listen for and take appropriate action:

a. Tap b. Double tap c. Long press d. All

Ans.: The Correct Option is d.

(b) i) What is Await expressions in dart.

Ans.: In Dart, the await keyword is used in conjunction with the async and await pattern to work with asynchronous operations. When you mark a function as async, you can use await within that function to pause the execution until an asynchronous operation, typically a Future or a Stream, is completed. Here's how it works:

1. Mark a Function as async: To use await, you first need to mark the enclosing function with the async keyword. This signals that the function contains asynchronous code.

```
Future<void> fetchData() async
{
        // Asynchronous code with 'await' goes here
}
```

2. Use await to Wait for an Asynchronous Operation: Inside the async function, you can use the await keyword before an asynchronous operation, like a Future, to pause the execution of the function until the operation is complete. The value returned by the Future is assigned to the variable.

```
Future<void> fetchData() async
{
    String data = await fetchDataFromServer();
    // 'await' pauses until data is fetched print(data);
   // Use the fetched data here
}
```

3. Error Handling: You can handle errors using try and catch as you normally would in synchronous code. If an error occurs in the awaited operation, it will throw an exception that you can catch.

```
Future<void> fetchData() async
{
        try
        {
                String data = await fetchDataFromServer();
                 // 'await' pauses until data is fetched print(data);
        }
        catch (error)
        {
                 print("Error: $error");
        }
 }
```

The async/await pattern in Dart makes it easier to work with asynchronous code by making it look more like synchronous code. This helps improve readability and maintainability, especially when dealing with complex asynchronous operations, such as network requests or file I/O.

ii) Describe data types in Dart with literal.

Ans.: Dart, like many programming languages, supports various data types. Here are some common data types in Dart with examples of their literals:

1. Numbers:
    - int: Integer literals (e.g., 42, -10, 0).
    - double: Double-precision floating-point literals (e.g., 3.14, -0.5, 1.0).
2. Strings:
    - String: String literals enclosed in single or double quotes (e.g., "Hello, Dart!", 'Single quotes work too').
3. Booleans:
    - bool: Boolean literals (e.g., true, false).
4. Lists:
    - List: Lists are collections of objects.

      ```
      List<int> numbers = [1, 2, 3];
      List<String> names = ["Alice", "Bob", "Charlie"];
      ```
5. Maps:
    - Map: Maps are key-value pairs.

      ```
      Map<String, int> ages = {
              "Alice": 25,
              "Bob": 30,
              "Charlie": 22,
      };
      ```
6. Runes (for representing Unicode characters):

- Runes: Represents a sequence of Unicode characters.
  Runes runes = Runes('\u{1F604}');
  // Represents a smiling emoji

7. Symbols:
   - Symbol: Represents an operator or identifier.
     Symbol sym = #example;
8. Null:
   - null: Represents the absence of a value.
     var missingData = null;
9. Dynamic (for when the type is not known at compile time):
   - dynamic: A variable that can hold values of any type.
     dynamic x = 42;
     x = "Hello, Dart!";

Q.2 (a) Describe flutter framework.

Ans.: Flutter is an open-source UI (User Interface) software development framework developed by Google. It is designed for building natively compiled applications for mobile, web, and desktop from a single codebase. Here are some key characteristics and components of the Flutter framework:

Certainly, here are a few key points about Flutter in simple terms:

1. **Cross-Platform Development:** Flutter lets you write code once and use it to create apps for multiple platforms like Android, iOS, web, and desktop.
2. **Fast Development:** Flutter's "Hot Reload" feature allows developers to see instant changes, making the development process faster.
3. **Attractive User Interfaces:** Flutter provides pre-designed components and tools to create visually appealing and consistent user interfaces.
4. **High Performance:** Apps built with Flutter are fast and responsive, thanks to natively compiled code.
5. **Good for Beginners:** Flutter's easy-to-learn Dart programming language and comprehensive documentation make it suitable for developers of all levels.
6. **Growing Community:** A supportive developer community contributes to the framework's growth and offers resources for assistance.
7. **Native Features:** Flutter lets you access native device features and APIs easily.
8. **Single Codebase:** You can maintain a single codebase for your app, reducing development effort and ensuring consistency across platforms.

(b) Create a Flutter program to display handing title “My First Flutter app” and text “Hello Flutter” in the centre of the screen also display demo of output.

Ans.:

```
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('My First Flutter App'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text(
                'Hello Flutter',
                style: TextStyle(
                  fontSize: 24,
                  fontWeight: FontWeight.bold,
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

(b) List out the Operators in Dart. Explain Type Test 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)

Type Test Operator (is):

The is operator in Dart is used to check if an object is an instance of a specific class or implements a particular interface. It returns true if the object matches the specified type and false otherwise.

Example:

```
class Animal {
  void makeSound() {
    print("Some generic animal sound");
  }
}

class Dog extends Animal {
  void makeSound() {
    print("Bark!");
  }
}
class Cat extends Animal {
  void makeSound() {
    print("Meow!");
  }
}
void main() {
  Animal myPet = Dog();
  if (myPet is Dog) {
    myPet.makeSound(); // Calls the Dog's makeSound method
  } else if (myPet is Cat) {
    myPet.makeSound(); // Calls the Cat's makeSound method
  } else {
    myPet.makeSound(); // Calls the generic animal sound
  }
}
```

Q.3 (a) i) What are the advantages and Disadvantages of Flutter?

Ans.: Advantages of Flutter

1. Fast Development

Flutter's fast development cycle allows developers to see changes to the app in real time as they make modifications to the code. This can greatly increase the speed and efficiency of the development process of the applications.

2. Beautiful User Interfaces

Flutter provides a rich set of customizable widgets that can be used to create beautiful and user-friendly interfaces. The framework also offers a strong emphasis on design and visual appeal, making it an attractive choice for app development projects that require a high degree of visual appeal.

3. High Performance

Flutter offers fast and smooth animations and transitions, and is designed to run smoothly on older devices. The framework is optimized for performance, making it an attractive choice for demanding mobile applications. As a result the number of targeted users increases.

4. Cross-Platform Development

Flutter supports not only mobile app development but also web and desktop app development. This makes it a versatile tool for developing applications that need to run on multiple platforms without any issues.

5. Open-Source

Flutter is a free and open-source framework, making it accessible to a wide range of developers and companies. The large community of developers and users working with the framework helps to ensure that it continues to evolve and expand its capabilities.

Disadvantages of Flutter

1. Limited Third-Party Libraries

While Flutter has a growing number of packages and plugins available, the framework is still relatively new, and the number of third-party libraries available for it is limited compared to more established frameworks such as React Native.

2. Steep Learning Curve

The Dart programming language used by Flutter can be challenging for some developers, and there may be a steep learning curve for those who are not already familiar with it.

3. Limited Corporate Adoption

While Flutter has gained significant traction in the development community, it is still relatively new, and its adoption by large corporations is limited compared to more established frameworks.

ii) Define Dart.

Ans.: Dart is a programming language developed by Google. It's often used to build web, mobile, and desktop applications. Dart is known for its simplicity and efficiency, and it's the primary language for building apps with the Flutter framework. Flutter is particularly popular for creating cross-platform mobile applications.

Dart is object-oriented and has a C-style syntax. It supports features like strong typing, just-in-time (JIT) compilation for development speed, and ahead-of-time (AOT) compilation for producing efficient machine code. Overall, Dart is a versatile language that can be used for a variety of application development purposes.

(b) Explain all the basic/common widgets used in flutter.

Ans.: Flutter provides a wide range of widgets that you can use to build user interfaces for your mobile, web, and desktop applications. Here, I'll explain some of the most commonly used and fundamental widgets in Flutter:

1. Container:
   - A versatile widget for creating boxes with padding, margins, and decoration.
   - Can be used to contain other widgets and style them with a background color, borders, or shadows.
2. Text:
   - Used for displaying text in your app.
   - Supports styling with fonts, colors, and alignment.
3. Row and Column:
   - Layout widgets for arranging child widgets in a horizontal (Row) or vertical (Column) direction.
   - Useful for creating structured layouts.
4. Image:
   - Widget for displaying images.
   - Supports loading images from various sources, such as assets or URLs.
5. Icon:
   - Represents an icon from the Material Icons or Cupertino Icons library.
   - Can be used to display icons in your app.
6. ListView:
   - A scrollable list of widgets.
   - Useful for displaying lists of items, such as contacts or messages.
7. Card:
   - A Material Design card, which is a rectangular piece of material with a shadow.
   - Often used to group related information together.
8. AppBar:
   - A Material Design app bar that typically appears at the top of the screen.
   - Contains options like the app's title, navigation, and actions.
9. TextField:
   - Allows users to input text.
   - You can control various properties, such as keyboard type, validation, and decoration.
10. FlatButton, RaisedButton, TextButton:
    - Widgets for creating different types of buttons.
    - Can be customized with text, icons, and various visual styles.
11. Scaffold:
    - A basic structure for your app's pages.
    - Includes app bars, body content, and a floating action button.

12. ListView.builder:
    - An efficient way to create a scrollable list of items, especially when working with large or dynamic lists.
13. Expanded:
    - A flexible widget that can be used within a Row or Column to distribute available space among its children.
14. Container:
    - A versatile widget for creating boxes with padding, margins, and decoration.
    - Can be used to contain other widgets and style them with a background color, borders, or shadows.
15. Stack:
    - Allows you to stack multiple widgets on top of each other.
    - Useful for creating complex layouts and overlapping widgets.
16. GridView:
    - A grid-based layout for arranging widgets in rows and columns.
    - Great for creating grids of items or images.
17. PageRoute:
    - Represents a screen or page in your app and provides navigation between different screens using routes.
18. Dialog:
    - A widget for displaying alert dialogs or pop-up dialogs to the user.
19. BottomNavigationBar:
    - A navigation bar at the bottom of the screen, commonly used for switching between different sections or pages in the app.
20. TabBar and TabView:
    - Widgets for creating tabbed interfaces, allowing users to switch between different content sections.

Q.3 (a) Explain form widgets in flutter.

Ans.: A Form widget is a fundamental building block for creating and managing forms within your user interfaces. A form typically consists of various input fields, such as text fields, checkboxes, radio buttons, and buttons, which allow users to input data or make selections. The Form widget provides a way to group and manage these form fields, validate user input, and handle form submissions efficiently.

Here are the key aspects and functionalities of the Form widget in Flutter:

1. Grouping Form Fields: The Form widget allows you to group form fields together within a single parent widget. This organization makes it easier to manage and validate user input.
2. Form Validation: The Form widget provides built-in support for form validation. You can specify validation rules for each form field, such as required fields, minimum and maximum lengths, or custom validation logic. Flutter can automatically validate and handle errors as users interact with the form.

3. Form Key: To manage and interact with a Form, you typically use a GlobalKey<FormState>. This key provides access to the state of the form, including methods to validate, reset, and save the form data.
4. Text Input Fields: Within a Form, you can use various text input widgets like TextFormField, TextField, and CupertinoTextField to create input fields. These widgets can be used for text input, password entry, and more.
5. Buttons: You can include buttons (e.g., ElevatedButton, TextButton, or IconButton) within a Form to trigger form submissions or other actions. For example, a "Submit" button can be used to send the form data to a server.
6. Submit and Reset Actions: A Form allows you to define actions to be performed when the form is submitted (e.g., data validation and submission to a server) and when it's reset (e.g., clearing the form fields).
7. Form Auto-Validation: You can configure the autovalidate property of the Form to automatically validate fields as users interact with them. This provides real-time feedback to users.
8. Handling Form Submissions: When a user submits the form, you can use the onPressed event of a button to call a function that handles the form data. The FormState object can be used to access the form's data.

Here's a simple example of a Flutter Form:

```
GlobalKey<FormState> _formKey = GlobalKey<FormState>();
String _name = '';

Widget build(BuildContext context) {
  return Form(
    key: _formKey,
    child: Column(
      children: [
        TextFormField(
          validator: (value) {
            if (value.isEmpty) {
              return 'Please enter your name';
            }
            return null;
          },
          onSaved: (value) {
            _name = value;
          },
        ),
        ElevatedButton(
          onPressed: () {
            if (_formKey.currentState.validate()) {
```

```
            _formKey.currentState.save();
            // Handle the form submission, e.g., send _name to a server.
          }
        },
        child: Text('Submit'),
      ),
    ],
  ),
 );
}
```

(b) Differentiate:

i) Hot reload and Hot restart.

Ans.: <u>Hot Reload</u>

- It performs very fast as compared to hot restart or default restart of flutter.
- If we are using the state in our app then hot reload will not change the state of the app.
- We perform hot reload by using key ctrl+\.

<u>Hot Restart</u>

- It is slower than hot reload but faster than the default restart.
- It doesn't preserve the state of our it starts from the initial state of our app.
- We perform hot restart using ctrl+shift+\

ii) Stateless Widget and Stateful Widget.

Ans.: <u>Stateless Widget:</u>

- Stateless Widgets are static widgets.
- They do not depend on any data change or any behavior change.
- Stateless Widgets do not have a state, they will be rendered once and will not update themselves, but will only be updated when external data changes.
- For Example Text, Icon, and *Raised Button* are Stateless Widgets.

<u>Stateful Widget:</u>

- Stateful Widgets are dynamic widgets.
- They can be updated during runtime based on user action or data change.
- Stateful Widgets have an internal state and can re-render if the input data changes or if Widget's state changes.
- For Example: Checkbox, Radio Button, Slider are Stateful Widgets

Q.4 (a) List functions in Dart. Explain any one in detail.

Ans.: Dart, as a versatile programming language, provides a wide range of built-in functions for performing various operations. Here's a list of some common functions in Dart:

1. print(): Used for printing text and values to the console.
2. main(): The entry point for Dart applications.
3. String methods (e.g., toUpperCase(), toLowerCase(), substring(), indexOf()).

4. List methods (e.g., add(), remove(), forEach()).
5. Set methods (e.g., add(), remove(), contains()).
6. Map methods (e.g., putIfAbsent(), remove(), containsKey()).
7. Control flow functions (e.g., if, for, while, switch).
8. assert(): Used for debugging and testing.
9. is: Used for type checking and casting.
10. throw: Throws an exception.
11. try, catch, finally: Used for exception handling.
12. async and await: Used for asynchronous programming.
13. Future and Stream methods: Used for asynchronous operations.
14. RegExp methods: Used for working with regular expressions.
15. Function and closure creation.

Now, let's explain one of these functions in detail:

Function: async and await

The async and await keywords are used for asynchronous programming in Dart. They allow you to work with asynchronous operations, such as network requests or file I/O, in a more synchronous and readable manner. Here's an explanation and example of how they work:

- async Function:
  - A function declared with the async keyword is marked as asynchronous. It can perform non-blocking operations and use the await keyword to pause its execution until the awaited operation is complete.
- await Expression:
  - The await keyword is used inside an async function to wait for the result of an asynchronous operation (typically a Future) to complete.
  - When an await expression is encountered, the function is paused until the awaited operation finishes. This allows you to write code that looks synchronous but doesn't block the event loop.

Example:

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

Future<void> fetchData() async {
 try {
  var response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));
  if (response.statusCode == 200) {
   print('Data: ${response.body}');
  } else {
   print('Failed to fetch data: ${response.reasonPhrase}');
  }
 } catch (e) {
```

```
    print('Error: $e');
  }
}

void main() {
  fetchData();
  print('Fetching data...');
}
```

(b) Explain AnimationController 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 AnimationController are:

Step 1: First, instantiate an AnimationController 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 callback 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.4 (a) Describe steps to adding a cloud firestore database and implementing security.

Ans.: Adding Cloud Firestore to your Flutter application and implementing security rules involves several steps. Here's a high-level overview of the process:

1. Set Up a Firebase Project:
    - Go to the Firebase Console (https://console.firebase.google.com/).
    - Click "Add Project" to create a new project.
    - Follow the setup wizard, providing a project name and enabling Google Analytics if desired.
2. Add Your App to the Project:
    - Once the project is created, click on "Add app" and select the appropriate platform (iOS or Android).
    - Follow the setup instructions to register your app with Firebase. This involves downloading configuration files and adding them to your Flutter project.
3. Add the FlutterFire Libraries:
    - In your Flutter project, open your pubspec.yaml file and add the dependencies for Firestore and Firebase authentication.
    - Run flutter pub get to fetch the added dependencies.
4. Initialize Firebase:
    - In your Flutter code, typically in the main.dart or the entry point of your app, initialize Firebase with Firebase.initializeApp(). This should be done before your app starts.
5. Create Firestore Security Rules:
    - In the Firebase Console, go to "Firestore" and select your project.
    - Navigate to the "Rules" tab to define security rules for your Firestore database.
    - Firestore security rules are written in a domain-specific language (rules language) that you can use to define who can read and write data. For example, you can control access based on user authentication, roles, or custom conditions.
6. Test and Deploy Rules:

- Before deploying your security rules, it's a good practice to test them. You can use the Firebase Emulator Suite for local testing.
- Once you're satisfied with your rules, deploy them to your production Firestore database.

7. Implement Authentication:
    - Use Firebase Authentication to identify and authenticate users in your app. Firestore security rules can be set to restrict access based on user authentication.
    - Set up authentication providers (e.g., email/password, Google, Facebook) in the Firebase Console and configure your app to use them.
8. Read and Write Data:
    - In your Flutter code, you can use the cloud_firestore package to read and write data to Firestore.
    - Use Firebase Authentication to sign in users, and then implement Firestore operations while adhering to your security rules.
9. Testing and Debugging:
    - Regularly test your app to ensure data is accessed and modified according to your security rules.
    - Use Firebase Console logging and debugging tools to help troubleshoot any security-related issues.
10. Monitor and Refine Security:
    - Continuously monitor your app for security issues and refine your Firestore security rules as necessary to adapt to changing requirements.

(b) Explain InkWell gestures with example.

Ans.: The InkWell widget is used to provide a visual response to user touch or tap gestures. It's commonly used to create interactive widgets that respond to user input by showing an ink splash animation when the user taps or presses on the widget. The ink splash effect is a visual indication that the widget is interactive and acknowledges the user's input.

Here's an explanation of InkWell with an example:

Example: Creating an InkWell with an Ink Splash Effect.

```
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('InkWell Example'),
      ),
      body: Center(
        child: MyButton(),
      ),
    ),
  );
 }
}

class MyButton extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return InkWell(
    onTap: () {
     // Handle the tap event here
     print('Button tapped');
    },
    child: Container(
     padding: EdgeInsets.all(16.0),
     decoration: BoxDecoration(
      borderRadius: BorderRadius.circular(8.0),
      color: Colors.blue,
     ),
     child: Text(
      'Tap Me',
      style: TextStyle(
       color: Colors.white,
       fontSize: 20.0,
      ),
     ),
    ),
  );
 }
}
```

In this example:

1. We create a Flutter app with a single **InkWell** button.
2. The **InkWell** widget is wrapped around a **Container** that represents the button. This container is given a blue background color.

3. The **onTap** property of the **InkWell** is set to a function that is called when the user taps the button. In this case, it simply prints a message to the console.
4. When the user taps the button, the **InkWell** widget displays the ink splash effect, indicating the touch event.
5. The button text is styled with white color and a font size of 20.

Q.5 (a) Explain Getters and Setters in Dart.

Ans.: In Dart, getters and setters are methods that allow you to control access to the properties (fields) of a class. They are used to provide read and write access to class properties while encapsulating the underlying implementation details. Getters are used to retrieve the values of properties, and setters are used to modify them. They enable you to define custom behavior when reading or writing property values.

Getters and setters provide a level of control and encapsulation over class properties, allowing you to hide the underlying implementation details and apply validation or custom logic when getting or setting property values.

Here's how to define and use getters and setters in Dart:

Getters:

- Getters are used to retrieve the value of a property.
- They are defined using the get keyword followed by the property name.
- Getters are called like properties, without parentheses.

Setters:

- Setters are used to modify the value of a property.
- They are defined using the set keyword followed by the property name.
- Setters are called like properties, but you can use the assignment operator (=) to assign a new value to the property.

Here's an example of how to define and use getters and setters:

```
class Person {
  String _name = ''; // Private field
  // Getter for the 'name' property
  String get name {
    return _name;
  }
  // Setter for the 'name' property
  set name(String newName) {
    if (newName.isNotEmpty) {
      _name = newName;
    }
  }
}
void main() {
```

```
  var person = Person();
  // Use the getter to retrieve the 'name' property
  print('Name: ${person.name}'); // Output: Name:
  // Use the setter to update the 'name' property
  person.name = 'Alice';
  // Use the getter again to retrieve the updated 'name' property
  print('Name: ${person.name}'); // Output: Name: Alice
}
```

In this example:

1. We define a Person class with a private field _name. By convention, an underscore prefix (_) indicates that a field is private and should not be accessed directly from outside the class.
2. We define a getter and a setter for the name property. The getter allows us to retrieve the value of _name, and the setter allows us to update _name. The setter also performs a simple validation to ensure that the name is not empty.
3. In the main function, we create an instance of the Person class and demonstrate the use of the getter and setter to access and modify the name property.

(b) Explain InkResponse gestures with example.

Ans.: The InkResponse widget is used to create interactive components that respond to touch gestures by displaying visual feedback in the form of ink splashes or highlights. It's similar to InkWell, but with a few differences. InkResponse provides more control over the ink effect and is particularly useful when you want to create custom interactions that go beyond simple button taps.

Here's an explanation of InkResponse with an example:

Example: Creating an InkResponse with Custom Touch Interaction

```
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('InkResponse Example'),
        ),
```

```
      body: Center(
        child: InteractiveBox(),
      ),
    ),
   );
  }
}

class InteractiveBox extends StatefulWidget {
  @override
  _InteractiveBoxState createState() => _InteractiveBoxState();
}

class _InteractiveBoxState extends State<InteractiveBox> {
  bool isTapped = false;

  void handleTap() {
    setState(() {
      isTapped = !isTapped;
    });
  }

  @override
  Widget build(BuildContext context) {
    return InkResponse(
      onTap: handleTap,
      onLongPress: handleTap,
      splashColor: Colors.blue, // Custom splash color
      child: Container(
        width: 200.0,
        height: 200.0,
        color: isTapped ? Colors.green : Colors.red, // Change color on tap
        alignment: Alignment.center,
        child: Text(
          isTapped ? 'Tapped!' : 'Tap me',
          style: TextStyle(
            color: Colors.white,
            fontSize: 20.0,
          ),
        ),
      ),
```

```
    );
  }
}
```

Q.5 (a) Explain GridView.count with example.

Ans.: GridView.count is a widget that allows you to create a scrollable grid of widgets with a fixed number of rows or columns. You specify the number of rows or columns you want, and the GridView will automatically handle the layout of the items within those rows or columns. It's a useful widget for creating grids of items like images, cards, or other widgets.

Here's an explanation of GridView.count with an example:

Example: Creating a Grid of Items with GridView.count

```
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('GridView.count Example'),
        ),
        body: GridDemo(),
      ),
    );
  }
}

class GridDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GridView.count(
      crossAxisCount: 2, // Number of columns in the grid
      children: List.generate(6, (index) {
        return Container(
          color: Colors.primaries[index % Colors.primaries.length],
          child: Center(
            child: Text('Item $index'),
```

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

In this example:

1. We create a Flutter app with a **GridView.count** as the main content of the app.
2. The **crossAxisCount** property is set to **2**, which means we want two columns in the grid. You can adjust this value to control the number of columns in the grid.
3. We use the **List.generate** function to create a list of colorful **Container** widgets. Each container represents an item in the grid. The background color of each container is determined by the **Colors.primaries** list.
4. Inside each container, we place a **Text** widget to display the item number.

(b) Define firebase and cloud firestore. State difference between 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.

****