Flutter Assignment
Module 2 – Dart Programming Essentials
Name :- Dhruv M Patel

1. Explain the fundamental data types in Dart (int, double, String, List, Map, etc.) and their uses.
Ans:-
1. `int`

- **Description:** Represents integer values, which are whole numbers without a decimal point.
- **Usage:** Used for counting, indexing, and performing arithmetic operations.

Example:

```
int age = 25;

int year = 2025;
```

2. `double`

- **Description:** Represents double-precision floating-point numbers, which can contain decimal points.
- **Usage:** Used for calculations that require fractional values, such as measurements and financial calculations.

Example:

```
double price = 99.99;

double pi = 3.14159;
```

3. `String`

- **Description:** Represents a sequence of characters, used for text.

- **Usage:** Used for storing and manipulating text data, such as names, messages, and any other textual information.

Example:

String name = 'Dhruv';

String message = "Welcome to Dart!";

4. `bool`

- **Description:** Represents a boolean value, which can be either `true` or `false`.
- **Usage:** Used for conditional statements and logical operations.

Example:

bool isLoggedIn = true;

bool hasPermission = false;

5. `List`

- **Description:** Represents an ordered collection of items, which can be of any data type.
- **Usage:** Used for storing multiple values in a single variable, allowing for easy iteration and manipulation of collections.

Example:

List<String> fruits = ['apple', 'banana', 'mango'];

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

6. `Map`

- **Description:** Represents a collection of key-value pairs, where each key is unique.
- **Usage:** Used for storing related data, allowing for efficient retrieval based on keys.

Example:

```
Map<String, String> user = {

  'name': 'Dhruv',

  'email': 'dhruv@example.com'

};

Map<String, int> marks = {

  'Math': 90,

  'Science': 85

};
```

7. `Set`

- **Description:** Represents an unordered collection of unique items.
- **Usage:** Used when you need to store a collection of items without duplicates.

Example:

```
Set<int> numbers = {1, 2, 3, 4, 2};  // duplicates are ignored
```

8. `dynamic`

- **Description:** A special type that can hold values of any type.
- **Usage:** Used when the type of a variable is not known at compile time or can change.

Example:

```
dynamic value = 10;

value = 'Now a string';
```

9. `var`

- **Description:** A keyword that allows Dart to infer the type of a variable based on the assigned value.
- **Usage:** Used for declaring variables without explicitly specifying their type.
- **Usage:** Makes a variable immutable after assignment.

Example:

```
final String name = 'Dhruv';

print(name);  // Output: Dhruv

// Trying to reassign will cause an error:

name = 'Patel'; // ❌ Error: Final variable can't be reassigned.
```

2. Describe control structures in Dart with examples of if, else, for, while, and switch.

Ans:- Control structures in Dart allow you to dictate the flow of your program based on certain conditions or to repeat actions. Here's a brief overview of some common control structures in Dart, along with examples for each.

1. If-Else Statement

The `if` statement executes a block of code if a specified condition is true. The `else` statement can be used to execute a block of code if the condition is false.

Example:-

```
void main() {
  int number = 10;
  if (number > 0) {
    print('The number is positive.');
  } else if (number < 0) {
    print('The number is negative.');
  } else {
    print('The number is zero.');
  }
}
```

2. For Loop:-

The `for` loop is used to execute a block of code a specific number of times. It consists of an initialization, a condition, and an increment/decrement operation.

Example:-

```
void main() {
  for (int i = 0; i < 5; i++) {
    print('Iteration: $i');
  }
```

```
}
```

3. While Loop

The `while` loop repeatedly executes a block of code as long as a specified condition is true.

Example:

```
void main() {
  int count = 0;

  while (count < 5) {
    print('Count: $count');
    count++;
  }
}
```

4. Switch Statement

The `switch` statement allows you to execute different blocks of code based on the value of a variable. It is often used as an alternative to multiple `if-else` statements.

Example:

```
void main() {
  String day = 'Monday';
  switch (day) {
    case 'Monday':
      print('Start of the work week.');
      break;
    case 'Friday':
      print('End of the work week.');
      break;
    case 'Saturday':
    case 'Sunday':
      print('Weekend!');
      break;
    default:
      print('Midweek days.');
  }
}
```

3. Explain object-oriented programming concepts in Dart, such as classes, inheritance, polymorphism, and interfaces.

Ans:- Object-oriented programming (OOP) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. Dart, being an object-oriented language, supports several key OOP concepts, including classes, inheritance, polymorphism, and interfaces. Here's a detailed explanation of each concept with examples.

1. Classes

A class is a blueprint for creating objects. It defines properties (attributes) and methods (functions) that the objects created from the class can use.

Example:

```
class Animal {
  String name;

  Animal(this.name);

  void speak() {
    print('$name makes a sound.');
  }
}

void main() {
  Animal dog = Animal('Dog');
  dog.speak(); // Output: Dog makes a sound.
}
```

2. Inheritance

Inheritance allows a class (subclass) to inherit properties and methods from another class (superclass). This promotes code reusability.

Example:

```
class Dog extends Animal {
  Dog(String name) : super(name);

  @override
  void speak() {
    print('$name barks.');
```

```
  }
}

void main() {
  Dog dog = Dog('Buddy');
  dog.speak(); // Output: Buddy barks.
}
```

3. Polymorphism

Polymorphism allows methods to do different things based on the object that it is acting upon. In Dart, this is often achieved through method overriding and interfaces.

Example:

```
class Cat extends Animal {
  Cat(String name) : super(name);

  @override
  void speak() {
    print('$name meows.');
  }
}

void main() {
  List<Animal> animals = [Dog('Buddy'), Cat('Whiskers')];

  for (var animal in animals) {
    animal.speak(); // Output: Buddy barks. Whiskers meows.
  }
}
```

4. Interfaces

In Dart, any class can act as an interface. A class can implement multiple interfaces, allowing for a form of multiple inheritance. When a class implements an interface, it must provide implementations for all the methods defined in the interface.

Example:

```
abstract class Flyer {
```

```
  void fly();
}

class Bird extends Animal implements Flyer {
  Bird(String name) : super(name);
  @override
  void speak() {
    print('$name chirps.');
  }
  @override
  void fly() {
    print('$name is flying.');
  }
}

void main() {
  Bird bird = Bird('Sparrow');
  bird.speak(); // Output: Sparrow chirps.
  bird.fly();   // Output: Sparrow is flying.
}
```

4. Describe asynchronous programming in Dart, including Future, async, await, and Stream.

Ans:- Asynchronous programming in Dart allows you to perform tasks without blocking the main thread, which is particularly useful for operations that take time, such as network requests or file I/O. Dart provides several constructs for handling asynchronous programming, including `Future`, `async`, `await`, and `Stream`. Here's a detailed explanation of each concept:

1. Future

A `Future` represents a potential value or error that will be available at some time in the future. It is used to handle asynchronous operations. A `Future` can be in one of three states: uncompleted, completed with a value, or completed with an error.

Example:

```
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () {
```

```
    return 'Data fetched';
  });
}

void main() {
  fetchData().then((data) {
    print(data); // Output: Data fetched (after 2 seconds)
  });
}
```

2. Async and Await

The `async` keyword is used to mark a function as asynchronous, allowing you to use the `await` keyword inside it. The `await` keyword pauses the execution of the function until the `Future` completes, making the code easier to read and write.

Example:

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

Future<void> main() async {
  print('Fetching data...');
  String data = await fetchData();
  print(data); // Output: Data fetched (after 2 seconds)
}
```

3. Stream

A `Stream` is a sequence of asynchronous events. It allows you to listen for multiple values over time, making it suitable for handling data that arrives in chunks, such as user input or data from a web socket.

Example:

```
Stream<int> countStream() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i; // Yielding values to the stream
  }
```

```
}

void main() async {
  await for (var value in countStream()) {
    print(value); // Output: 1, 2, 3, 4, 5 (each after 1 second)
  }
}
```