Dart Theory Assignments

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

1. Numbers:

- int: Represents whole numbers (integers) without decimal points.
    - Use Cases:
        - Counting items (e.g., number of products in a cart)
        - Representing ages, years, etc.
        - Indices in lists and arrays

- double: Represents numbers with decimal points (floating-point numbers).
    - Use Cases:
        - Currency values (e.g., prices)
        - Measurements (e.g., height, weight)
        - Scientific calculations

2. String:

- Represents a sequence of characters enclosed in single or double quotes.
    - Use Cases:
        - Storing text (e.g., names, addresses, messages)
        - Displaying information to users
        - Working with textual data (e.g., parsing, formatting)

3. Boolean:

- Represents a logical value, either true or false.
    - Use Cases:
        - Conditional statements (if/else)
        - Controlling program flow
        - Checking for specific conditions

4. List:

- An ordered collection of objects (can be of the same or different types).
    - Use Cases:
        - Storing a series of items (e.g., a list of products, a list of users)
        - Iterating over collections
        - Accessing elements by index

5. Map:

- A collection of key-value pairs, where each key is unique.
    - Use Cases:
        - Storing data in a structured way (e.g., user information, configuration settings)
        - Efficiently retrieving values based on keys
        - Representing relationships between data

6. Set:

- A collection of unique objects (no duplicates).
    - Use Cases:
        - Removing duplicates from a list
        - Checking for membership (whether an item exists in the set)
        - Performing set operations (union, intersection)

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

1. Conditional Statements:

- if/else:
  - Executes a block of code only if a specific condition is true.
  - The else block is optional and executes if the condition is false.

```
int age = 25;

if (age >= 18) {
  print("You are an adult.");
} else {
  print("You are a minor.");
}
```

switch:

- Evaluates an expression and matches it against a series of cases.
- Executes the code block associated with the matching case.

```
int dayOfWeek = 3;


switch (dayOfWeek) {

  case 1:

    print("Monday");

    break;

  case 2:

    print("Tuesday");

    break;

  case 3:

    print("Wednesday");
    break;
  default:
    print("Other day of the week");
}
```

2. Loops:

for:

    - Executes a block of code a specified number of times.

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

while:

- Executes a block of code as long as a given condition is true.

```
int count = 0;
while (count < 3) {
  print("Count: $count");
  count++;
}
```

do-while:

- Similar to while, but the code block is executed at least once before the condition is checked.

```
int count = 0;

do {

  print("Count: $count");

  count++;

} while (count < 3);
```

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

1. Classes:

- Blueprint for Objects: A class is a blueprint or template that defines the properties (data) and behaviors (methods) of objects.

- Example:

```
class Car {

  String model;

  int year;


  void start() {

    print("Car started.");

  }

}
```

2. Objects:

- Instances of Classes: Objects are created from classes. They represent real-world entities with their own unique set of properties.

- Example:

  ```
  void main() {
    Car myCar = Car(); // Create an object of the Car class
    myCar.model = "Toyota Camry";
    myCar.year = 2023;
    myCar.start();
  }
  ```

3. Inheritance:

.Creating New Classes from Existing Ones: Allows you to create a new class (subclass or derived class) that inherits properties and methods from an existing class (superclass or base class).

```
class ElectricCar extends Car {
  double batteryCapacity;

  void charge() {
    print("Car is charging.");
  }
}
```

4.Polymorphism:

- "Many Forms": The ability of objects of different classes to be treated as objects of a common type.
- Method Overriding: Subclasses can override methods defined in the superclass to provide their own specific implementations.

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

class Dog extends Animal {
  @override
  void makeSound() {
    print("Woof!");
  }
}

class Cat extends Animal {
  @override
  void makeSound() {
    print("Meow!");
  }
}
```

5.Interfaces:

- Contracts: Define a set of methods that a class must implement.
- Example:

```
abstract class Flyable {
  void fly();
}
```

```
class Bird implements Flyable {
  @override
  void fly() {
    print("Bird is flying.");
  }
}
```

Key Concepts in OOP:

- Encapsulation: Bundling data (properties) and methods that operate on that data within a class.
- Abstraction: Hiding the internal implementation details of a class and only exposing necessary information.

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

1. Asynchronous Programming:

- Non-Blocking Operations: Enables you to perform operations that don't block the main thread of execution. This is crucial for tasks like network requests, file I/O, and database operations, which can be time-consuming.

2. Future:

- Represents a value that will be available in the future.
- Use Cases:
    - Representing the result of an asynchronous operation.
    - Chaining asynchronous operations.

```
Future<String> fetchData() async {
 // Simulate an asynchronous operation (e.g., network request)
 await Future.delayed(Duration(seconds: 2));
 return "Data fetched successfully!";
}

void main() async {
 String result = await fetchData();
 print(result);
}
```

3. async/await:

- Simplified Asynchronous Code:
  - async before a function makes it return a Future.
  - await pauses the execution of the current function until the Future completes and returns its value.

4. Stream:

- Sequence of Events: A stream represents a sequence of asynchronous events.
- Use Cases:
  - Handling real-time data (e.g., user input, sensor data).
  - Streaming data from various sources (e.g., network, files).

```
Stream<int> generateNumbers() async* {
  for (int i = 0; i < 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void main() async {
  await for (int number in generateNumbers()) {
    print(number);
  }
}
```