Module 2 : Dart Programming Essentials

Theory Assignments:

Name : Chandvaniya Abhay

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

1. int (Integer)

Represents whole numbers (positive, negative, or zero).

Examples:

```
int age = 25;
int temperature = -5;
```

Use cases:

- Counting items
- Loop counters
- Values that never require decimals

2. double (Floating-point number)

Represents numbers with decimal points.

Examples:

```
double price = 29.99;
double pi = 3.14159;
```

Use cases:

- Calculations requiring precision
- Measurements, mathematical formulas

3. num

A parent type for both int and double.

num x = 10;
x = 10.5; // valid

Use case:
When a variable might hold either an integer or a decimal.

4. String

Used to store text.

Examples:

String name = "Alice";
String message = 'Hello World';

Use cases:

- User input
- Display messages
- Data formatting

5. bool (Boolean)

Represents truth values: true or false.

Examples:

bool isLoggedIn = true;
bool isActive = false;

Use cases:

- Conditions (if, while)
- Flags and states

6. List (Ordered Collection)

A list is similar to an array; it stores an ordered sequence of items.

Examples:

```
List<int> numbers = [1, 2, 3, 4];
var names = ["Alice", "Bob"];
```

Use cases:

- Ordered data
- Iteration and collection operations

Dart lists are 0-indexed (first element at index 0).

7. Map (Key-Value Pair Collection)

A map stores data as key–value pairs.

Examples:

```
Map<String, int> scores = {
  "Alice": 90,
  "Bob": 85
};
```

Use cases:

- Fast lookup of values by keys
- Representing structured data (e.g., JSON-like objects)

8. Set (Unique Collection)

A collection of **unique** items (no duplicates).

```
Set<String> fruits = {"apple", "banana", "orange"};
fruits.add("apple"); // still only one "apple"
```

Use cases:

- Removing duplicates
- Membership testing

9. var, final, const (Not data types, but important keywords)

var

Type is inferred at compile time; value can change.

```
var x = 10;  // int
x = 20;      // OK
```

final

Value cannot change after assignment (runtime constant).

```
final name = "Alice";
```

const

Compile-time constant.

```
const pi = 3.14;
```

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

Control Structures in Dart

Control structures allow you to control the flow of execution in a Dart program. Dart supports typical decision-making and looping constructs found in most modern languages.

1. if Statement

Used to execute code only when a condition is true.

Example

```
void main()
{
  int age = 20;

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

2. if–else Statement

Provides an alternate action when the condition is false.

Example

```
void main()
{
  int age = 16;

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

3. for Loop

Used to repeat code a specific number of times.

Example

```
void main()
{
  for (int i = 1; i <= 5; i++)
  {
    print("Count: $i");
  }
}
```

4. while Loop

Repeats code *while* a condition remains true.

Example

```
void main()
{
  int i = 1;

  while (i <= 5)
  {
    print("Number: $i");
    i++;
  }
}
```

5. switch Statement

Used to execute different actions based on a variable's value. Works best with integers, strings, and enums.

Example

```
void main()
{
 String grade = "B";

 switch (grade)
 {
  case "A":
   print("Excellent");
   break;

  case "B":
   print("Good");
   break;

  case "C":
   print("Fair");
   break;

  default:
   print("Invalid grade");
 }
}
```

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

OOP Concepts in Dart

Dart is a fully object-oriented language where **everything is an object**, and OOP principles are central to the language.

1. Classes

A **class** is a blueprint for creating objects. It defines properties (variables) and methods (functions).

Example

```
class Car
{
  String brand;
  int year;

  Car(this.brand, this.year);

  void displayInfo()
  {
    print('Brand: $brand, Year: $year');
  }
}

void main()
{
  Car car1 = Car('Toyota', 2020);
  car1.displayInfo();
}
```

2. Inheritance

Inheritance allows one class to **extend** another and reuse its properties and methods.

Example

```
class Animal
{
  void eat()
  {
    print('Animal is eating');
  }
}

class Dog extends Animal
{
  void bark()
  {
    print('Dog is barking');
  }
}

void main()
{
  Dog dog = Dog();
  dog.eat();   // inherited method
  dog.bark();  // subclass method
}
```

3. Polymorphism

Polymorphism means a subclass can **override** methods of its parent class and be treated as the parent type.

Example: Method Overriding

```
class Animal
{
  void sound()
  {
    print('Animal makes a sound');
  }
}

class Cat extends Animal
{
  @override
  void sound()
  {
    print('Cat says: Meow');
  }
}

void main()
{
  Animal animal = Cat();  // polymorphism
  animal.sound();        // Cat overridden method
}
```

4. Interfaces

Dart does not have a keyword interface.
Instead, **every class can act as an interface**, and you implement it using implements.

Classes that implement an interface **must override all its methods**.

Example

```
class Printable
{
  void printData() {}
}

class Document implements Printable
{
  @override
  void printData()
  {
    print('Printing document...');
  }
}

void main()
{
  Printable doc = Document();
  doc.printData();
}
```

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

1. Future

A **Future** represents a value that will be available **later**.
It's similar to a promise of a value that may arrive in the future.

A Future can be:

- **completed with data** (success)
- **completed with an error** (failure)

Example: Creating and using a Future

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

void main()
{
  fetchData().then((value)
  {
    print(value);
  });
}
```

2. async and await

async and await make asynchronous code look like synchronous code.

- Add **async** to a function to mark it as asynchronous.
- Use **await** to pause the function until the Future completes.

Example: Using async and await

```
Future<String> fetchData() async
{
  await Future.delayed(Duration(seconds: 2));
  return "Data loaded";
}

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

3. Stream

A Stream is used for receiving **multiple asynchronous values over time**.

You use a Stream when:

- You expect **a sequence of values**
- The data arrives **continuously** (like events, messages, or sensor data)

Example: Simple Stream

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

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