Dart&Flutter concepts

1️⃣ Dart Basics

1.1 🔹 Lambda (Arrow) Functions

- Represent short functions using `=>`.
- Often used with **callbacks**.
- Can be **named or anonymous**.
- Must contain **only one expression** after `=>`.

```
int square(int x) => x * x;

var greet = (String name) => "Hello $name";
```

If multiple statements are needed, use a block function:

```
int sum(int a, int b) {
  return a + b;
}
```

1.2 🔹 Higher-Order Functions (HOF)

A function that:

- Takes another function as a parameter, OR
- Returns a function.

```
void execute(Function action) {
  action();
}
```

Example with a List:

```
var numbers = [1, 2, 3];
numbers.forEach((n) => print(n));
```

1.3 🔹 `where()` Method

- A **higher-order function** on `Iterable`.
- Works like `filter()` in other languages.
- Returns an **Iterable**, NOT a tuple.

```
var numbers = [1, 2, 3, 4];
var even = numbers.where((n) => n % 2 == 0);
print(even); // (2, 4)
```

1.4 🔹 Iterable

An `Iterable` is a collection that can be looped over.

Examples:

- `List`
- `Set`
- `Map` (iterates over keys by default)
- `String` (iterable of characters)

```
Iterable<int> nums = [1, 2, 3];
```

1.5 🔹 Lexical Closure

A function defined inside another function that:

- Accesses outer variables
- Remembers them even after outer function ends

```
Function counter() {
  int count = 0;
  return () {
    count++;
    return count;
  };
}
```

1.6 🔹 Extension Methods

Add methods to existing classes **without modifying them**.

```
extension StringExtension on String {
  String shout() => toUpperCase() + "!";
}
```

1.7 🔹 Platform Class (`dart:io`)

Detects OS or system info:

```
import 'dart:io';

if (Platform.isWindows) {
```

```
  print("Running on Windows");
}
```

2️⃣ Object-Oriented Programming (OOP)

2.1 🔹 Everything is an Object

- `int`, `double`, `String`, `List` — all are objects.
- Every class implicitly extends `Object`.

Important `Object` members:

- `runtimeType`
- `hashCode`
- `toString()`
- `noSuchMethod()`

2.2 🔹 Abstract Class

- Cannot create objects directly.
- May contain abstract methods (without body).

```
abstract class Animal {
  void makeSound();
}
```

2.3 🔹 Inheritance (`extends`)

- Dart supports **single inheritance only**.

```
class Dog extends Animal {
  @override
```

```
  void makeSound() => print("Bark");
}
```

2.4 🔹 Interfaces (`implements`)

- Any class can act as an interface using `implements`.
- Must override **all methods and properties**.
- Supports multiple interfaces.

```
class Robot implements Animal {
  @override
  void makeSound() {
    print("Beep");
  }
}

class MyClass implements A, B {}
```

2.5 🔹 Mixins (`with`)

Used to reuse behavior across multiple classes.

```
mixin Fly {
  void fly() => print("Flying");
}

class Bird with Fly {}
```

- Enables **multiple behavior reuse**
- Not true multiple inheritance

2.6 🔹 Polymorphism

- Greek meaning: *Many forms*
- A subclass can be treated as its superclass

```
Animal a = Dog();
a.makeSound();
```

- Achieved through **method overriding** (not overloading)

2.7 🔹 Method Overriding

```
@Override
void makeSound() {
  print("Different sound");
}
```

2.8 🔹 Named Constructors

- Since Dart does not support function overloading:

```
class Person {
  String name;
  Person(this.name);
  Person.guest() : name = "Guest";
}
```

2.9 🔹 Optional Parameters

Positional

```
void greet([String name = "Guest"]) {}
```

Named

```
void greet({String name = "Guest"}) {}
```

2.10 🔹 Enums

Used for a fixed set of constants:

```
enum Status { loading, success, error }
```

- Prevents using raw strings.

2.11 🔹 Generics

Allow classes to work with **any data type**.

```
class Box<T> {
  T value;
  Box(this.value);
}
```

- Type is decided at compile time (not runtime)

2.12 🔹 Built-in Methods Example

```
print(12.gcd(8)); // 4
```

2.13 🔹 Creating External Packages

- Create reusable Dart files
- Publish packages
- Import using:

```
import 'package:my_package/my_file.dart';
```

3️⃣ Additional Dart Concepts

3.1 🔹 Math Library Highlights

Common functions from `dart:math`:

```
sin(x)
cos(x)
tan(x)
sqrt(x)
pow(x, y)
max(a, b)
min(a, b)
log(x)
exp(x)
```

Important constants

```
pi
e
ln2
sqrt2
```

3.2 🔹 DateTime Basics

Current time

```
DateTime now = DateTime.now();
```

Add duration

```
now.add(Duration(days: 60));
```

Difference

```
now2.difference(now).inDays;
```

3.3 🔹 Platform (`dart:io`)

Provides system information:

```
Platform.operatingSystem
Platform.numberOfProcessors
Platform.version
Platform.environment
```

3.4 🔹 Operator Overloading

Custom behavior for operators:

```
Num operator +(Num other) {
  return Num(this.num + other.num);
}
```

3.5 🔹 Polymorphism (Overriding)

```
class Shape {
  type() => print("shape");
}

class Circle extends Shape {
  @override
```

```
  type() => print("circle");
}
```

3.6 🔹 Lexical Closure

```
Function counter() {
  int count = 0;
  return () {
    count++;
    print(count);
  };
}
```

3.7 🔹 Callable Class

```
class Human {
  call() {
    print("called");
  }
}

Human h = Human();
h(); // works like a function
```

3.8 🔹 Extension Method

```
extension parsingStrNum on String {
  int parseInt() => int.parse(this);
}

"26".parseInt();
```

4️⃣ Asynchronous Programming

4.1 🔹 Synchronous vs Asynchronous

Synchronous

- Executes **one task at a time**
- Blocking behavior

Asynchronous

- Tasks can **start without waiting** for previous ones
- Non-blocking
- Used for network, file, timers, etc.

4.2 🔹 Future in Dart

Represents a value that will be available **later**.

```
Future f1 = Future(() => 5);
```

Handling Future

```
f1.then((value) => print(value))
  .catchError((error) => print(error));
```

4.3 🔹 async and await

```
Future<int> getNumber() async {
  return 4;
}

int num = await getNumber();
```

- `await` pauses the current async function
- Does **not block** the whole program

4.4 🔹 Stream

- Emits **multiple asynchronous values** over time

Periodic Stream

```
Stream<int> s1 = Stream.periodic(Duration(milliseconds: 500),
(a) => a).take(15);
```

- `.take(15)` closes stream after 15 events
- Triggers `onDone`

Listening

```
final sub = s1.listen(
  (data) => print(data),
  onDone: () => print("Done"),
  onError: (e) => print(e),
);
```

- `subscription.cancel()` does **not trigger** `onDone`

4.5 🔹 async* and yield

```
Stream<int> getNumbers() async* {
  yield 1;
  await Future.delayed(Duration(seconds: 2));
  yield 3;
}
```

- `async*` returns a Stream

- `yield` emits values

4.6 🔹 StreamController

```
StreamController<int> sc = StreamController<int>();
sc.add(10);
await sc.close();
```

- `broadcast` allows multiple listeners

```
StreamController<int> sc = StreamController<int>.broadcast();
```

4.7 🔹 Key Stream Concepts

Concept	Trigger
onData	new data
onError	error occurs
onDone	stream finishes
cancel()	stops subscription (does NOT call onDone)

4.8 🔹 await for

```
await for (int val in stream) {
  print(val);
}
```

- Acts like a subscription
- Cannot listen to the same single-subscription stream twice

5️⃣ Flutter Setup

5.1 🔹 Installation Steps

1. Download Flutter SDK (zip)
2. Extract to `C:\flutter` (avoid `Program Files` & `Downloads`)
3. Add `flutter/bin` to **PATH**
4. Run:

```
flutter--version
```

5.2 🔹 Install Tools

- Android Studio / VS Code
- Dart & Flutter extensions
- AVD (Android Virtual Device)
- SDK platforms & tools

5.3 🔹 Useful Commands

```
flutter doctor
flutter create app_name
dart run file.dart
```

5.4 🔹 VS Code Extension

- **Error Lens** → Shows errors inline while typing