1. **Quick start — run this first**

void main() {

print('Hello, Dart!');

}

Save as main.dart and run dart run main.dart (or copy to DartPad).

1. **Variables & basic datatypes**

int, double, num, bool, String

var (type inferred), dynamic (runtime flexible)

final (runtime constant), const (compile-time constant)

null-safety: Type? can be null

late for deferred initialization

void main() {

int a = 10;

double pi = 3.1416;

num n = 7; // can be int or double later

bool ok = true;

String name = 'Hakim';

var city = 'Dhaka'; // inferred as String

dynamic anything = 5; anything = 'five';

final version = '1.0.0';

const int maxUsers = 100;

int? maybeNull;

late String token;

token = 'abc123';

print('Hi $name from $city!'); // string interpolation

print(maybeNull ?? 'no value'); // null-coalescing

}

1. **Operators (quick)**

Arithmetic: + - \* / / % (/ integer division)

Comparison: == != < <= > >=

Logical: && || !

Type test: is, is!, cast with as

Null-aware: ??, ?., ??=

Cascade: .. (chain calls)

void main() {

print(7 ~/ 2); // 3

print(null ?? 'fallback');

String? s;

print(s?.length); // null safe

var sb = StringBuffer()..write('hi')..write('!');

print(sb);

}

1. **Conditionals**

void main() {

int marks = 76;

if (marks >= 80) {

print('A+');

} else if (marks >= 70) {

print('A');

} else {

print('Keep trying');

}var day = 'fri';

switch (day) {

case 'fri':

print('Jummah');

break;

case 'sat':

case 'sun':

print('Weekend');

break;

default:

print('Workday');

}

}

1. **Loops**

void main() {

for (int i = 0; i < 3; i++) print('for $i');

var list = [10, 20, 30];

for (var v in list) print('for-in $v');

int w = 0;

while (w < 2) { print('while $w'); w++; }

int d = 0;

do { print('do $d'); d++; } while (d < 2);

list.forEach((x) => print('forEach $x'));

}

1. **Functions**

positional parameters, optional positional [], named {} (often better)

int add(int a, int b) => a + b; // arrow for single expr

String hello([String name = 'Guest']) => 'Hi $name'; // optional positional

String greet({required String name, String prefix = 'Hello'}) =>

'$prefix, $name'; // named parameter

void main() {

print(add(2,3));

print(hello());

print(greet(name: 'Hakim'));

}

Higher-order functions: pass a function as argument:

List transform(List l, int Function(int) f) => l.map(f).toList();

1. **Collections**

**List, Set, Map**

void main() {

var list = [3,1,2];

list.add(4);

list.sort();

var set = {'a','b','a'}; // unique

var map = {'Ali':30, 'Babu':25};

print(list); print(set); print(map['Ali']);

print(list.where((x) => x.isEven).toList());

var expanded = [0, ...list]; // spread operator

print(expanded);}

1. **Errors & exceptions**

void main() {

try {

int x = int.parse('abc');

print(x);

} on FormatException catch (e) {

print('Format error: $e');

} catch (e) {

print('Other error: $e');

} finally {

print('always run');

}

}

Throw your own:

void check(int n) {

if (n < 0) throw ArgumentError('n must be >= 0');

}

1. **Async basics (Future / async / await / Stream)**

Future fetchUser() async {

await Future.delayed(Duration(milliseconds: 300));

return 'Hakim';

}

Stream counter(int n) async\* {

for (int i = 1; i <= n; i++) {

await Future.delayed(Duration(milliseconds: 100));

yield i;

}

}

void main() async {

var user = await fetchUser();

print('User: $user');

await for (var v in counter(3)) {

print('tick $v');

}

}

1. **Short practice tasks (try now)**

Function to return even numbers from List.

Map Map → return name with max value.

Small Future returning "OK" after 100ms and await it.

If you want, I can show the solutions.

**===== Separate: OOP in Dart — explained simply & clearly =====**

I’ll explain each OOP concept with simple analogy, then code.

Analogy:

Think of classes as blueprints (like a car blueprint). An object is a made car from that blueprint.

OOP helps organize code as objects that hold data (state) and behavior (methods).

Key concepts (short)

Class — blueprint/type.

Object (instance) — concrete item created from class.

Encapsulation — keep internal data private, expose methods.

Constructor — how to create an object.Inheritance — child class reuses/extends parent.

Polymorphism — different classes respond to same method call differently.

Abstraction — hide complex details; expose essential interface (abstract classes).

Mixin / Composition — reuse code (mixins) or include objects (composition).

1. **Class & object (basic)**

class Person {

String name;

int \_age; // private to library (leading underscore)

Person(this.name, this.\_age); // constructor

void sayHi() => print('Hi, I am $name, age $\_age');

int get age => \_age; // getter

set age(int value) { // setter with check

if (value < 0) throw ArgumentError('Age cannot be negative');

\_age = value;

}

@override

String toString() => 'Person(name: $name, age: $\_age)';

}

void main() {

var p = Person('Rahim', 25);

p.sayHi();

p.age = 26;

print(p);

}

\_age is private to the file/library. Use getter/setter for control (encapsulation).

1. **Constructors: default, named, factory, initializer list**

class Point {

double x, y;

// default constructor

Point(this.x, this.y);

// named constructor

Point.origin() : x = 0, y = 0;

// factory constructor (useful for singletons or caching)

static final Map \_cache = {};

factory Point.fromKey(String key) {

return \_cache.putIfAbsent(key, () => Point(0,0));

}

}

1. **Inheritance & override**

class Student extends Person {

String id;

Student(String name, int age, this.id) : super(name, age);

@override

void sayHi() => print('Student $name [$id], age $age');

}

extends reuses parent. super(...) calls parent constructor.

@override indicates overriding a method.

1. **Abstract classes & polymorphismabstract**

class Animal {

void speak(); // abstract method (no body)

}

class Dog extends Animal {

@override

void speak() => print('Woof');

}

class Cat extends Animal {

@override

void speak() => print('Meow');

}

void main() {

List zoo = [Dog(), Cat()];

for (var a in zoo) a.speak(); // polymorphism: same call, different behavior

}

1. **Interfaces (implements) vs inheritance**

In Dart, any class can be used like an interface.

class Flyable {

void fly() => print('flying');

void land() => print('landing');

}

// implement forces full implementation of members

class Bird implements Flyable {

@override

void fly() => print('Bird flaps wings');

@override

void land() => print('Bird lands softly');

}

implements means you must provide all methods yourself (no inherited behavior).

1. **Mixins (code reuse)**

mixin Coder {

void code() => print('Coding...');

}

mixin Tester {

void test() => print('Testing...');

}

class Dev with Coder, Tester {}

void main() {

var d = Dev();

d.code();

d.test();

}

with adds behavior. Use on in advanced cases to restrict mixin to certain base types.

7) Static members & utility methods

class Counter {

static int total = 0;

Counter() { total++; }

}

void main() {Counter(); Counter();

print(Counter.total); // 2

}

8) Composition vs Inheritance (preferred when possible)

Composition: class A has-a B (use B as a member).

Inheritance: class A is-a B.

Example: A Car has an Engine (composition):

class Engine {

void start() => print('Engine started');

}

class Car {

final Engine engine;

Car(this.engine);

void start() => engine.start();

}

Composition is often safer and more flexible.

9) Practical OOP example: Small bank account (with code)

class BankAccount {

String owner;

double \_balance;

BankAccount(this.owner, [this.\_balance = 0]);

double get balance => \_balance;

void deposit(double amount) {

if (amount <= 0) throw ArgumentError('deposit > 0');

\_balance += amount;

}

void withdraw(double amount) {

if (amount <= 0) throw ArgumentError('withdraw > 0');

if (amount > \_balance) throw StateError('Insufficient funds');

\_balance -= amount;

}

@override

String toString() => 'BankAccount(owner: $owner, balance: $\_balance)';

}

void main() {

var acc = BankAccount('Asha', 100.0);

acc.deposit(50);

try {

acc.withdraw(200);

} catch (e) {

print('Error: $e');

}

print(acc);

}

This demonstrates encapsulation (\_balance), methods for behavior, and error handling.

OOP practice tasks (with easy solutions)

Task: Create abstract Shape with double area(); implement Circle and Rectangle.

Solution (quick):

import 'dart:math';abstract class Shape {

double area();

}

class Circle implements Shape {

double r;

Circle(this.r);

@override

double area() => pi \* r \* r;

}

class Rectangle implements Shape {

double w, h;

Rectangle(this.w, this.h);

@override

double area() => w \* h;

}

void main() {

List shapes = [Circle(2), Rectangle(3,4)];

for (var s in shapes) print('area: ${s.area()}');

}

Task: Make a mixin Logger that adds log(String) to any class; use in a Service class.

Solution (quick):

mixin Logger {

void log(String msg) => print('[LOG] $msg');

}

class Service with Logger {

void run() => log('Service started');

}

void main() {

Service().run();

}

Tips / Best Practices (short)

Prefer named parameters for readability in functions/constructors.

Keep fields private (underscore) and expose controlled getters/setters.

Prefer composition over inheritance when possible.

Use async/await for readable asynchronous code (avoid nested callbacks).

Use immutable const/final where possible.

If you want, I can:

Convert this to a single ready-to-run file with mini-examples and comments.

Translate the OOP section into Bengali step-by-step.

Give more practice problems and full solutions (banking, shapes, small console app).