Mobile Programming

Dart Programming

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

Dart | Mobile Systems | Mobile Engineering | Production



Table of Contents

1. Why Dart for Mobile?
2. Language Essentials (Production-Focused)
3. Null Safety & Type System
4. OOP & Architectural Thinking
5. Generics & Extensions
6. Structured Error Handling
7. Asynchronous Programming Model
8. Futures & Streams
9. Isolates & Concurrency
10. Event Loop & Microtasks
11. Serialization & Data Modeling
12. Performance Thinking

Why Dart for Mobile?

Flutter Runs on Dart

- Dart runtime
- Event loop
- Async execution model
- Memory management

Weak Dart → unstable apps.

Mobile Engineering Constraints

- Limited CPU & memory
- Battery sensitivity
- Network unreliability
- Strict UI responsiveness

Blocking UI = unacceptable.

Language Essentials (Production-Focused)

Our focus

- Clean Architecture
- Async correctness
- Testable systems
- Scalable applications
- Performance awareness

```
No unnecessary syntax theory (iykyk)
```



Null Safety & Type System

```
String? username;
String title = "Mobile Programming";

void greet(String? user) {
  print(user?.toUpperCase() ?? "Guest");
}
```

Why It Matters

- Eliminates null crashes
- Explicit data contracts
- Required for safe API handling

OOP & Architectural Thinking

```dart
abstract class Repository<T> {
  Future<List<T>> getAll();
}

class UserRepository implements Repository<User> {
  @override
  Future<List<User>> getAll() async {
    return [];
  }
}
```

Principles

- Depend on abstractions
- Separate domain from data
- Enable mocking & testing

Generics & Reusability

```
class Result<T> {
  final T? data;
  final String? error;

  Result.success(this.data) : error = null;
  Result.failure(this.error) : data = null;
}
```

Use generics for:

- API responses
- Repository layers
- Strong type guarantees

Extensions for Cleaner APIs

The "Old" Way vs. The "Extension" Way

Utility Class

```
if (ListUtils.isEmpty(users)) {
  return null;
} else {
  return users.first;
}
```

Verbose, hard to read.

Extensions

```
// Define once
extension ListX<T> on List<T> {
  T? firstOrNull() =>
      isEmpty ? null : first;
}

// Use naturally
final active = users.firstOrNull();
```

Intuitive and direct.

Extensions for Cleaner APIs

Key Benefits:

- **Expressive Logic:** Code reads like a sentence (`users.firstOrNull()`).
- **No Boilerplate:** Eliminates static `StringUtils` or `ListUtils` files.
- **Better Discovery:** IDE suggests these methods automatically when you type.

Structured Error Handling

```
try {
  final user = await fetchUser();
} on TimeoutException {
  print("Request timed out");
} catch (e, stack) {
  logError(e, stack);
}
```

Mobile rule:

- Never swallow errors
- Always log failures
- Recover when possible

Asynchronous Programming Model

Mobile apps are

- Event-driven
- Network-heavy
- I/O-bound

Understanding async: `mandatory`

Futures

```
Future<String> fetchUser() async {
  await Future.delayed(Duration(seconds: 1));
  return "Alo";
}
```

Used for:

- HTTP requests
- Database calls
- File operations

Streams: Data Over Time

Concept

Unlike a `Future` (which delivers **one** value asynchronously), a **Stream** delivers **a sequence** of values asynchronously. Think of it as a pipe with water flowing through it.

The Code

```
// "async*" marks a generator function
Stream<int> counter() async* {
  for (int i = 0; i < 5; i++) {
    await Future.delayed(
       Duration(seconds: 1)
    );
    yield i; // Sends value down the pipe
  }
}
```

The Visualization



Streams: Data Over Time

Real-World Use Cases:

- **Real-time Feeds:** Live stock prices or sports scores.
- **User Input:** Handling text changes in a search bar.
- **Chat Systems:** Receiving messages from a WebSocket.
- **I/O Operations:** Reading a file chunk-by-chunk.

Isolates (True Parallelism)

Why Isolates?

Dart uses a single thread (Event Loop) by default. If you run a heavy calculation (like parsing a huge JSON or encrypting a file) on the main thread, **the UI freezes.**

Isolates are independent workers ("threads") with their own memory. They don't share state; they **message** each other.

Isolates (True Parallelism)

```
void heavyTask(SendPort sendPort) { // 1. The Isolate (The Worker)
  int sum = 0;
  // Runs on a separate thread
  for (int i = 0; i < 100000000; i++) {sum += i;}
  // "Yield": Send result back
  sendPort.send(sum);
}
void runTask() async { // 2. Main Thread (The Manager)
  final receivePort = ReceivePort();
  // Spawn the worker
  await Isolate.spawn(
    heavyTask,
    receivePort.sendPort
  );
  // Wait for result
  final result = await receivePort.first;
}
```

Isolates (True Parallelism)

Architecture

No shared memory = No race conditions.

Benefits:

- **Smooth UI:** 60 FPS maintained.
- **Stability:** A crash in the worker won't kill the UI.
- **Multi-Core:** Uses all CPU cores available.

Event Loop & Microtasks

The Concept

The Dart Event Loop manages the execution queue. It prioritizes **Microtasks** (internal quick ops) over **Events** (I/O, Timers, Gestures).

Rule: The Loop finishes the **entire** Microtask Queue *before* processing the first item in the Event Queue.

Event Loop & Microtasks

```dart
void main() {
  // 1. Add to Event Queue
  Future(() => print("Event"));

  // 2. Add to Microtask Queue
  scheduleMicrotask(() {
    print("Microtask");
  });

  print("Main");
}
```

Execution Flow:

1. `Main` (Synchronous code runs first)
2. `Microtask` (Microtask Queue empties)
3. `Event` (Event Queue processes)

Why? Because Microtasks have higher priority.

Why This Matters:
If you queue 1,000 microtasks, you will **block** the UI (clicks, scrolling) until all 1,000 are finished, even if timers are firing.

Serialization & Data Modeling

The Concept

Serialization is the bridge between your code (Dart Objects) and the outside world (JSON strings/Maps). It converts structured data into a format your app can process.

JSON (API) ⇄ **fromJson() / toJson()** ⇄ **Dart Object**

Serialization & Data Modeling

```
class User {
  final int id;
  final String name;

  User({required this.id, required this.name});

  // 1. Factory: Creates object from JSON
  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      id: json['id'],
      name: json['name'],
    );
  }

  // 2. Method: Creates JSON from object
  Map<String, dynamic> toJson() =>
      {'id': id, 'name': name};
}
```

Serialization & Data Modeling

Engineering Rules (Best Practices):

- **Immutable Models:** Use `final` fields to prevent data corruption after creation.
- **Explicit Types:** Never use `var` or `dynamic` for model properties (e.g., use `int`, not `var`).
- **Null Safety:** Define strictly which fields are optional (`String?`) vs required (`String`).

Performance Thinking

Avoid

- Blocking the main thread
- Excessive object creation
- Dynamic types

Prefer

- const constructors
- final fields
- Efficient collections
- Lazy initialization

Mini Lab

Build an Async Task Scheduler

Requirements

1. Define `Task` model
2. Create `TaskRepository`
3. Execute tasks asynchronously
4. Handle failures properly
5. Log execution

Real-Life Application

Ride-Sharing Logic Simulation

Implement

- Driver model
- Rider model
- Trip lifecycle
- Async booking flow
- Error recovery

Focus

- Clean domain modeling
- Async correctness
- Zero UI

Learning Outcomes

- Think asynchronously
- Avoid UI blocking
- Write crash-resistant code
- Design repository-based systems
- Build scalable Dart foundations

> Strong Dart engineers build scalable Flutter apps.

Thank You! 🙌

Questions?

"Q: Why did the Java developer refuse to cross the street?

*A: Because he was afraid of getting caught by the Garbage Collector while he was still **referenced**!"*