Day 2:
Asynchronous Programming in Dart/Flutter

Why Asynchronous Programming?

User Expectations:

- Fast, responsive apps
- No freezing or lag
- Immediate feedback

Common Blocking Operations:

- Database queries
- Network requests (HTTP, APIs)
- File I/O operations
- OS-dependent delays

The Problem with Synchronous Code

Example:

```
int processData(int param1, double param2) {
  // Long computation...
  return httpGetRequest(value);
}
```

Issue:

- Blocks `main()` for seconds
- App appears frozen
- Poor user experience

Introducing Futures

Future<T>:

- Represents a value (or error) available *later*
- Used for time-consuming operations
- Returns immediately, processes in background

Example:

```
Future<int> processData(...) {
  return Future<int>.value(res);
}
```

Working with Futures – `then()`

Non-blocking Execution:

```
void main() {
  Future<int> val = processData(1, 2.5);
  val.then((result) => print(result));
  print("Welcome to Dart!");
}
```

Output:

```
Welcome to Dart!
result = 10
```

Chaining and Error Handling

Method Chaining:

```
val.then((r) => anotherFunction1(r))
   .then((a) => anotherFunction2(a))
   .catchError((e) => print(e.message));
```

Key Points:

- Chain multiple `.then()` callbacks
- Use `.catchError()` for exceptions
- Avoids "callback hell" but can be verbose

Useful Future Constructors

- `Future.delayed()` – runs after a delay
- `Future.error()` – completes with an error
- `Future.value()` – wraps a value in a Future
- `Future.sync()` – executes callback immediately
- `Future.wait()` – waits for multiple Futures

Async/Await – The Better Way

Syntactic Sugar:

- Makes async code look synchronous
- Reduces verbosity
- Improves readability

Before (`then`):

```
processData(...).then((data) => print(data));
```

After (`async/await`):

```
void main() async {
  final data = await processData(...);
  print(data);
}
```

Async/Await Rules

1. Use `await` only in `async` functions
2. Mark function with `async` before body
3. `await` works only on `Future<T>`
4. Exceptions handled with `try/catch`

Example with Error Handling:

```
try {
  final result = await processData(...);
} on Exception catch (e) {
  print(e.message);
}
```

Good Practices

Official Guideline:

> "Prefer async/await over raw Futures"

Why:

- Cleaner, more readable code
- Easier debugging
- Avoids nested callbacks
- Looks like synchronous code

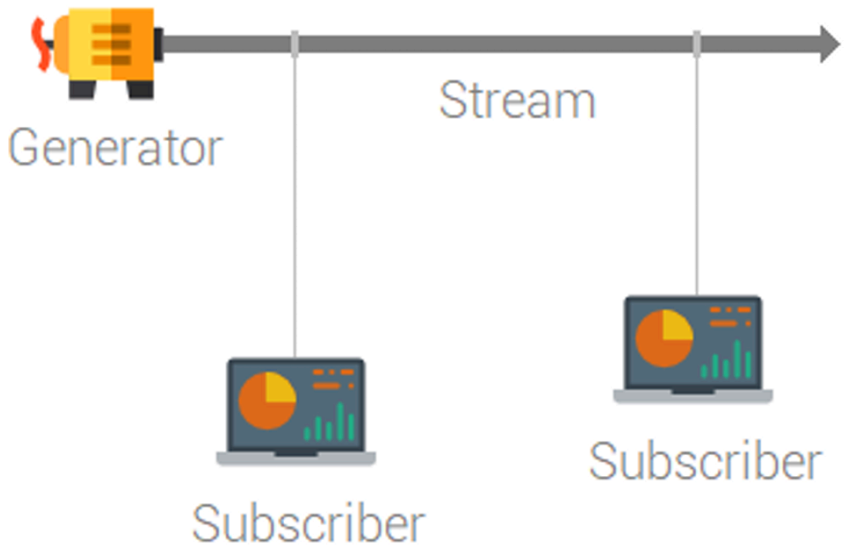
Introducing Streams

Stream<T>:

- Sequence of asynchronous events
- Listener is notified automatically
- Used for continuous data flows

Analogy:

- Generator → Stream → Subscriber
- Like a pipe with flowing data


Generator
Stream
Subscriber
Subscriber

Asynchronous Generators (`async*`)

Example:

```
Stream<int> randomNumbers() async* {
  for (var i = 0; i < 100; ++i) {
    await Future.delayed(Duration(seconds: 1));
    yield random.nextInt(50) + 1;
  }
}
```

Key Points:

- Use `async*` and `yield`
- No `return` statement
- Data emitted lazily

Synchronous Generators (`sync*`)

Example:

```
Iterable<int> randomNumbers() sync* {
  for (var i = 0; i < 100; ++i) {
    sleep(Duration(seconds: 1));
    yield random.nextInt(50) + 1;
  }
}
```

Use When:

- No asynchronous operations needed
- Data produced synchronously

Stream Constructors

- `Stream.periodic()` – emits at intervals
- `Stream.value()` – single value stream
- `Stream.error()` – error event stream
- `Stream.fromIterable()` – from a list
- `Stream.fromFuture()` – from a Future
- `Stream.empty()` – empty stream

Subscribing to Streams

Using `await for`:

```
void main() async {
  final stream = randomNumbers();
  await for (var value in stream) {
    print(value);
  }
}
```

Key Points:

- `await for` works like a regular loop
- Listens until stream completes
- Use `try/catch` for error handling

StreamController – Advanced Stream Management

Why Use It?:

- More control over stream lifecycle
- Handles multiple subscribers
- Better for larger apps

Example Setup:

```
final _controller = StreamController<int>(
  onListen: _startStream,
  onCancel: _stopTimer,
);
Stream<int> get stream => _controller.stream;
```

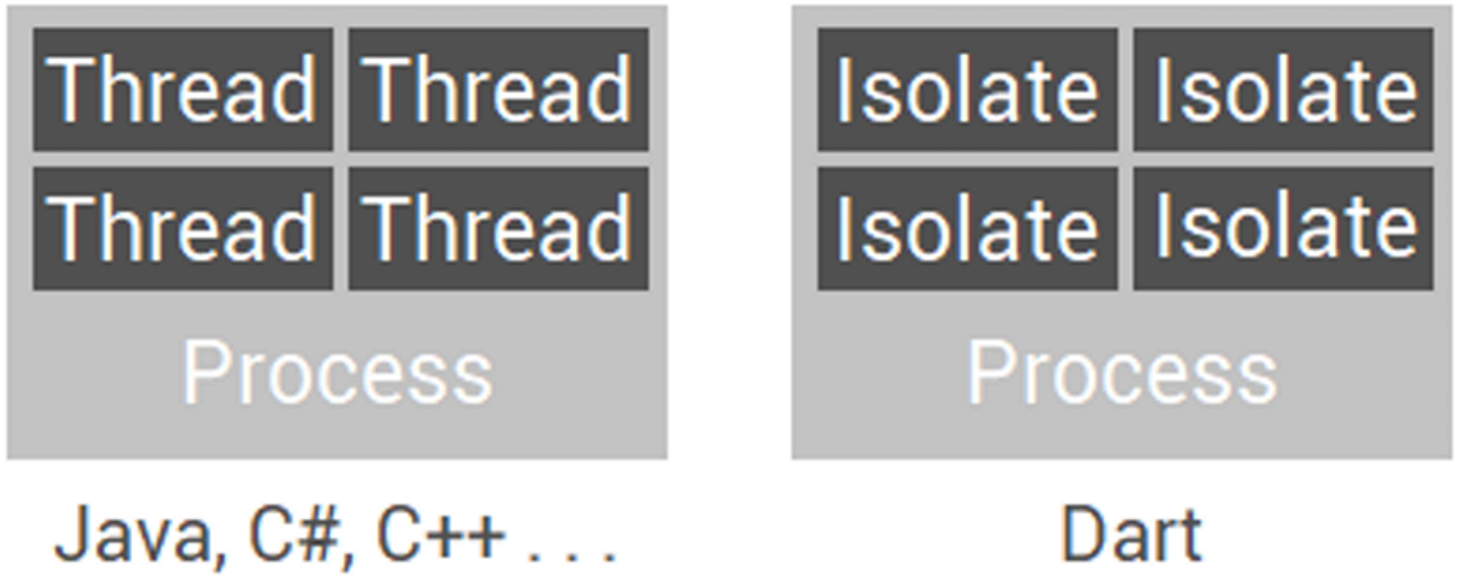
Isolates – Dart's Concurrency Model

No Traditional Threads:

- Dart uses *isolates*
- Each isolate has its own memory and event loop
- No shared memory → no data races

Comparison:

- Java/C#: Threads share memory
- Dart: Isolates do not share memory


Thread
Thread
Thread
Thread
Process
Java, C#, C++ . . .
Isolate
Isolate
Isolate
Isolate
Process
Dart

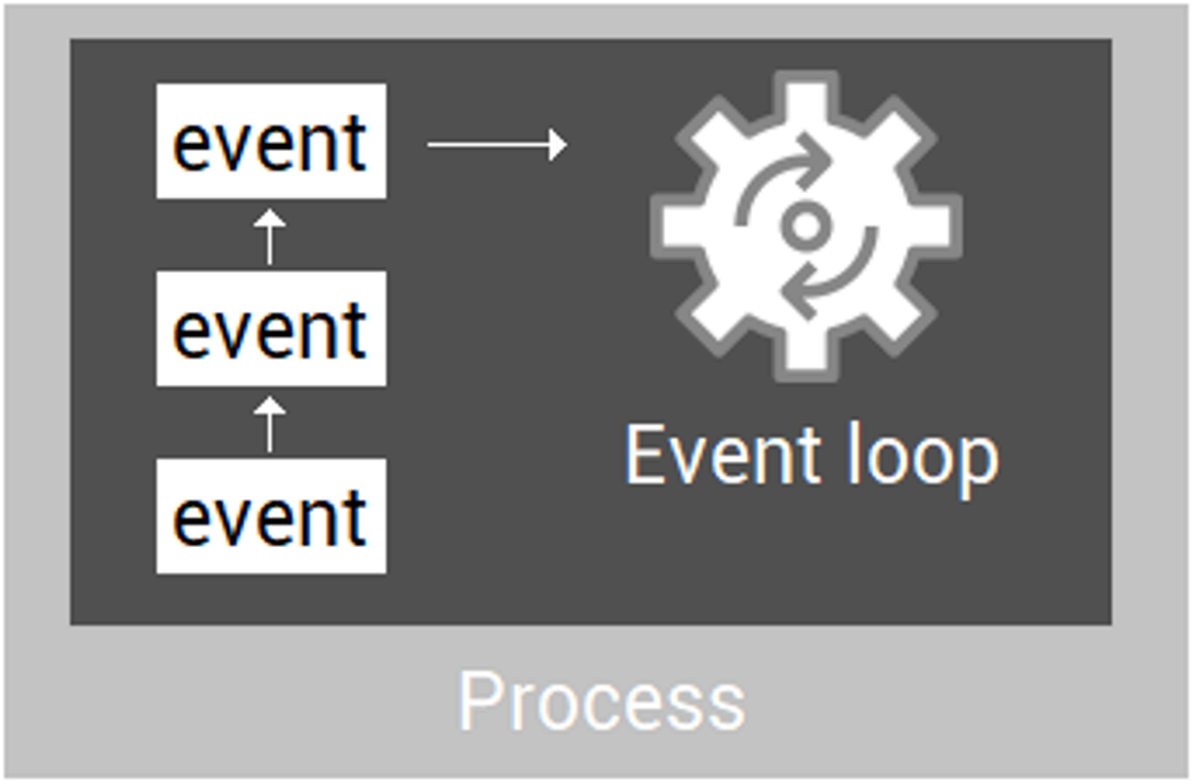
Inside an Isolate

Event Loop Architecture:

```
Event → Event Queue → Event Loop → Process
```

How Async Works:

- Long tasks split into events
- Callbacks scheduled later
- Event loop never blocks


event
event
event
Event loop
Process

Multiple Isolates

Communication via Messages:

- `SendPort` / `ReceivePort`
- No shared state
- Used for heavy computation

Spawning Isolates:

```
Isolate.spawn(entryFunction, message);
```

Isolates in Flutter (`compute`)

Flutter's Helper:

```
Future<int> heavyCalculations() {
  return compute(sumOfPrimes, 50000);
}
```

Rules:

- Function must be top-level or static
- Parameters must be serializable
- Use model class for multiple parameters

When to Use Isolates

Use Cases:

- Heavy mathematical computations
- Image/video processing
- Large data parsing
- Anything that drops FPS below 60

Otherwise:

- Stick with `async/await` and `Streams`

Comparing Async Tools

Tool	Use Case
`Future`	Single delayed value
`Stream`	Continuous data flow
`Isolate`	CPU-heavy, parallel tasks
`async/await`	Write async code synchronously

Best Practices Summary

1. **Prefer `async/await` over `then()`**
2. **Use `Streams` for real-time data**
3. **Use `Isolates` for CPU-intensive work**
4. **Never block the event loop**
5. **Keep UI thread responsive**

Common Pitfalls

- Blocking `main()` with sync code
- Nested callbacks (callback hell)
- Forgetting `await` in async functions
- Misusing `StreamController` lifecycle
- Overusing isolates for simple tasks

Coding Principles with Dart

Beyond Syntax: Good Practices

Topics Beyond This Book:

- Design Patterns (Gang of Four & modern patterns)
- Test-Driven Development (TDD)
- Domain-Driven Design (DDD)
- Clean Code & Architecture

Recommended Resources:

- ResoCoder: Flutter TDD & DDD courses
- Dart Effective Dart Guide
- General software design books

Why These Principles Matter

Even If You Skip This Chapter:

- SOLID & DI are language-agnostic
- Essential for professional code quality
- Critical for team collaboration & maintenance
- Flutter apps scale better with good architecture

Bottom Line:

Learn to write code that lasts.

SOLID Principles Overview

S – Single Responsibility Principle
O – Open/Closed Principle
L – Liskov Substitution Principle
I – Interface Segregation Principle
D – Dependency Inversion Principle

Goal: Maintainable, flexible, testable code

Single Responsibility Principle (SRP)

A class should have only one reason to change.

Bad Example:

```
class Shapes {
  // Calculations + Painting + Networking + Caching
}
```

Good Example:

```
abstract class Shape {} // Calculations
class ShapePainter {}   // UI
class ShapesOnline {}   // Networking
```

Open/Closed Principle (OCP)

Open for extension, closed for modification.

Problem:

- Adding new shapes requires modifying `AreaCalculator`
- Chain of `if/else` or `switch` statements

Solution:

- Use abstractions (interfaces/abstract classes)
- Each shape implements its own `computeArea()`

OCP Example

Before (Violates OCP):

```
double calculate(Object shape) {
  if (shape is Rectangle) { ... }
  else if (shape is Circle) { ... }
  // Add more `if` for new shapes
}
```

After (Follows OCP):

```
abstract class Area { double computeArea(); }
class Rectangle implements Area { ... }
class Circle implements Area { ... }

double calculate(Area shape) {
  return shape.computeArea(); // No changes needed!
}
```

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

Classic Violation:

```
class Rectangle { double width, height; }
class Square extends Rectangle { ... } // Problem!
```

Issue:

- Square should have equal sides
- But can be mutated via Rectangle setters
- Logical inconsistency

Fixing LSP

Prefer composition over inheritance:

- Use interfaces/abstract classes
- Avoid extending concrete classes with different invariants

Better Design:

```
abstract class Shape { double area(); }
class Rectangle implements Shape { ... }
class Square implements Shape { ... }
```

Key Takeaway:

Inherit behavior, not structure, when logical constraints differ.

Interface Segregation Principle (ISP)

Clients shouldn’t depend on methods they don’t use.

Problem:

```
abstract class Worker {
  void work();
  void sleep(); // Robot doesn’t sleep!
}
```

Solution:

```
abstract class Worker { void work(); }
abstract class Sleeper { void sleep(); }
class Human implements Worker, Sleeper { ... }
class Robot implements Worker { ... }
```

Dependency Inversion Principle (DIP)

Depend on abstractions, not concretions. **Example**:

```
abstract class EncryptionAlgorithm {
  String encrypt();
}
class FileManager {
  void secureFile(EncryptionAlgorithm algo) {
    algo.encrypt(); // Knows only the abstraction
  }
}
```

Benefit:

- Easy to swap algorithms
- No changes to `FileManager` for new algorithms

Dependency Injection (DI) Introduction

What is Coupling?

- Class A depends on Class B if A cannot compile without B
- **Strong coupling**: Hard to change, fragile architecture

Example Problem:

- `PaymentProcessor` tightly coupled to `MasterCardValidator`
- Switching to PayPal requires massive refactoring

The Dependency Problem

Strong Coupling Diagram:

```
PaymentValidator
      ↑
Strong Dependency
      ↑
PaymentProcessor    Checker
```

Change Impact:

- Replace MasterCard with PayPal
- Requires changes in 3+ classes
- Cascade effect across hierarchy

Constructor Injection

Inject dependencies via constructor:

```
abstract class PaymentValidator {
  void validatePayment(int amount);
}

class PaymentProcessor {
  final PaymentValidator _validator;
  PaymentProcessor(this._validator); // Injected!
}
```

Usage:

```
final p1 = PaymentProcessor(MasterCard());
final p2 = PaymentProcessor(PayPal());
```

Benefits of Constructor Injection

1. **Weak coupling** – depends on abstraction
2. **Testability** – easy to inject mocks
3. **Flexibility** – swap implementations without changes
4. **Const constructors** possible
5. **Follows DIP & OCP**

Method Injection

For optional dependencies:

```
class PaymentProcessor {
  bool isProcessorActive(CheckProcessor check) {
    return check.isActive(); // Optional check
  }
}
```

When to Use:

- Constructor injection: **Essential** dependencies
- Method injection: **Optional** dependencies

Comparing Injection Types

Constructor Injection	Method Injection
Required dependencies	Optional dependencies
Class cannot work without it	Nice-to-have feature
Injected at creation	Injected at method call
Better for testing	More flexible usage

SOLID + DI in Practice

Complete Example:

```
// DIP + OCP
abstract class PaymentValidator { ... }

// SRP
class MasterCard implements PaymentValidator { ... }
class PayPal implements PaymentValidator { ... }

// Constructor Injection
class PaymentProcessor {
  final PaymentValidator _validator;
  PaymentProcessor(this._validator);
}

// LSP + ISP compliant
```

Practical Notes

1. **SOLID principles** guide better OOP design
2. **Dependency Injection** reduces coupling
3. **Prefer abstractions** over concrete implementations
4. **Constructor injection** for required dependencies
5. **Write testable, maintainable Flutter code**

END