FLUTTER ISOLATES

WHAT IS

A THREAD ?

A thread is a unit of a process. It is a small set of instructions designed to be scheduled and executed independently of the parent process.

HOW ??

FLUTTER IS SINGLE THREADED

You might be wondering something: If Dart is a single-threaded language, how can asynchronous tasks in Flutter (like fetching data via an HTTP call) perform optimally without hindering other activities of the application? Well, this leads to two different concepts, concurrency and parallelism, the former standing for async tasks and the latter for tasks running on an isolate.

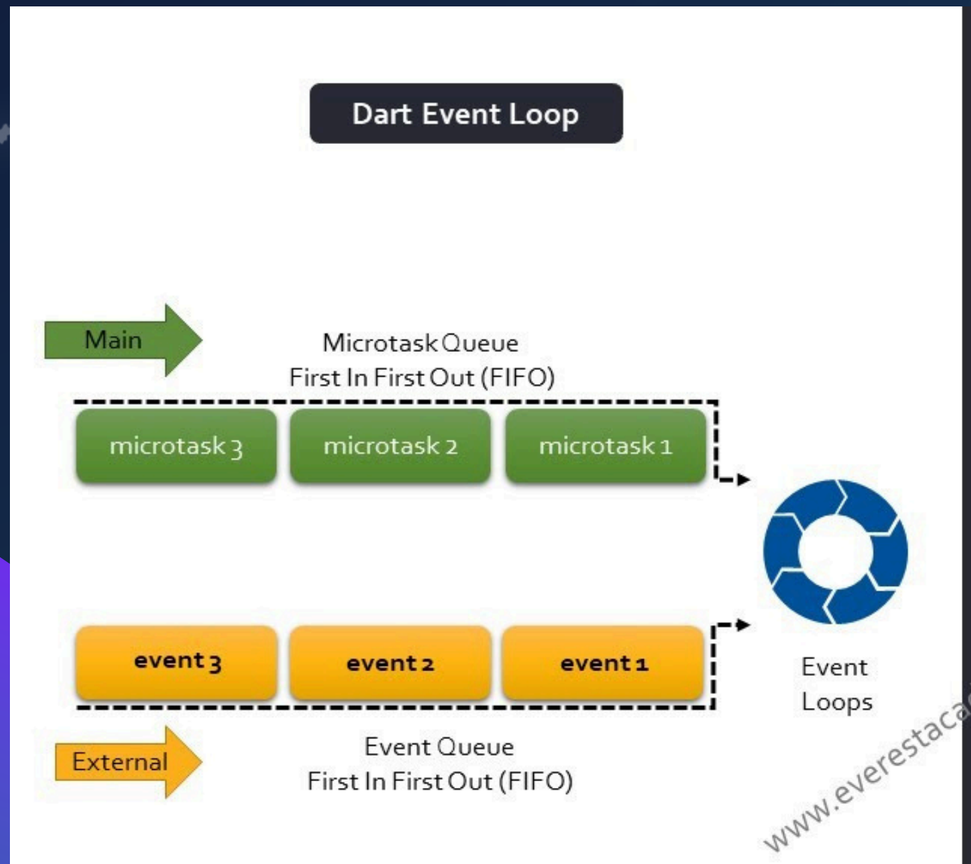
WHAT IS AN ISOLATE

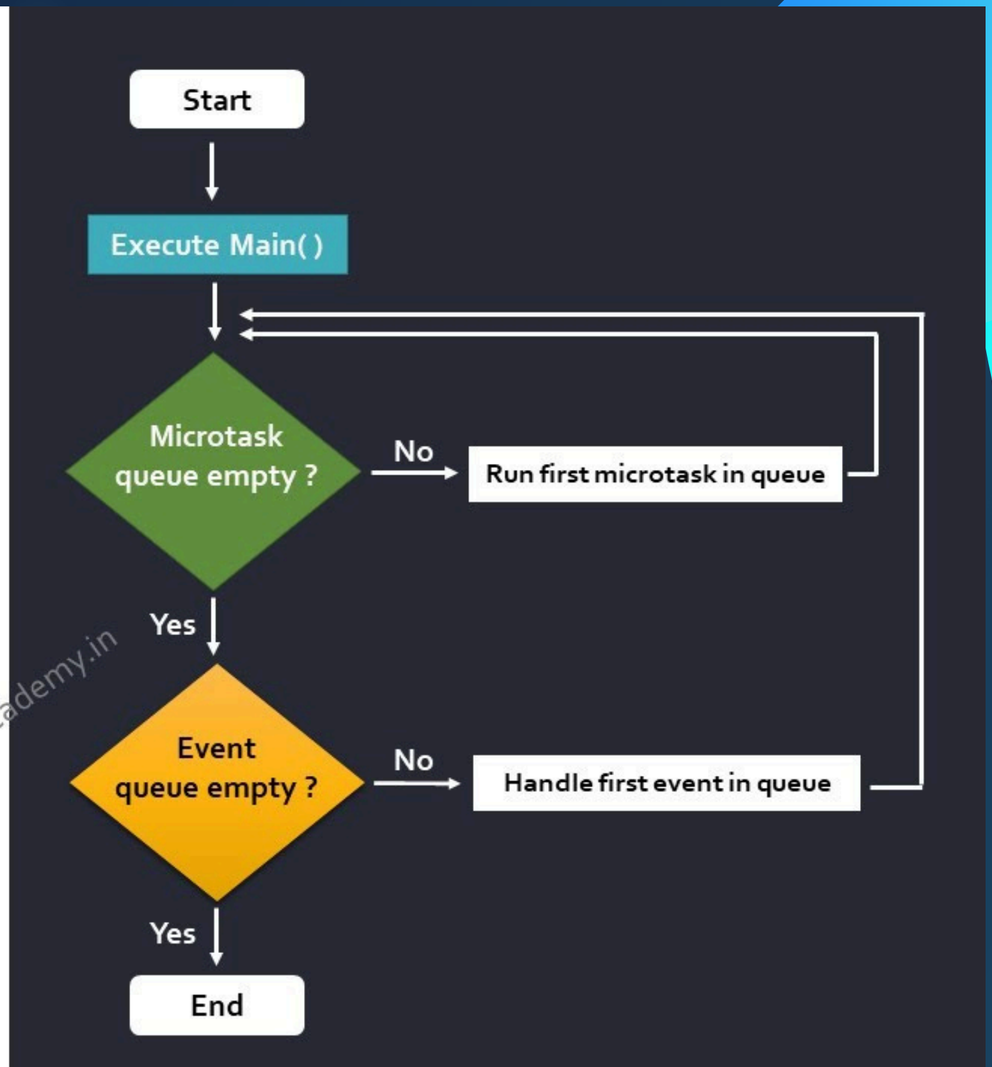
One thing to note before we talk about it is that Dart programs run in an isolate by default **main isolate**. Think of an isolate in Dart as that small room within a coffee kiosk where activities like making coffee drinks and ordering coffee take place. Isolate is like a little space in a machine, with its memory (a system that stores information that will be needed soon on a short-term basis) and a single thread running an event loop that processes the code.

DIFFRENCE
THREAD VS ISOLATE

Feature	Isolates	Threads
Memory Sharing	Isolates do not share memory.	Threads share the same memory.
Communication	Message passing via ports.	Direct access to shared memory.
Concurrency	Isolates run concurrently.	Threads run concurrently.
Synchronization	No need for locks or synchronization.	Requires synchronization (e.g., mutex
Safety	More secure, no data races.	Data races can occur without proper locking.

EVENT LOOP IN FLUTTER





EVENT LOOP
HOW DOES IT WORK ?

ASYNCHRONOUS OPERATIONS

When you use Dart’s Future, async, and await keywords, you’re allowing long-running tasks (like network requests or reading files) to run asynchronously. These tasks are added to the event queue, and once the task completes, the event loop picks it up and processes the result.

WHY ISOLATES ?

Dart is a **single-threaded** language, which can lead to performance issues when running heavy computations asynchronously. To avoid application lag, Dart provides isolates, which allow tasks to be executed on separate threads.

However , a major limitation of standard Dart isolates is that they cannot interact with Flutter plugins, as Flutter’s platform plugin system is tightly coupled with the main application isolate.

WHY ISOLATES ?

The **flutter_isolate** package solves this issue by introducing the FlutterIsolate class, which enables spawned isolates to communicate with Flutter plugins. This allows developers to perform background tasks efficiently while maintaining full access to Flutter's plugin ecosystem, overcoming the restrictions of native Dart isolates.

HOW ??
COMMUNICATE BETWEEN TWO ISOLATES

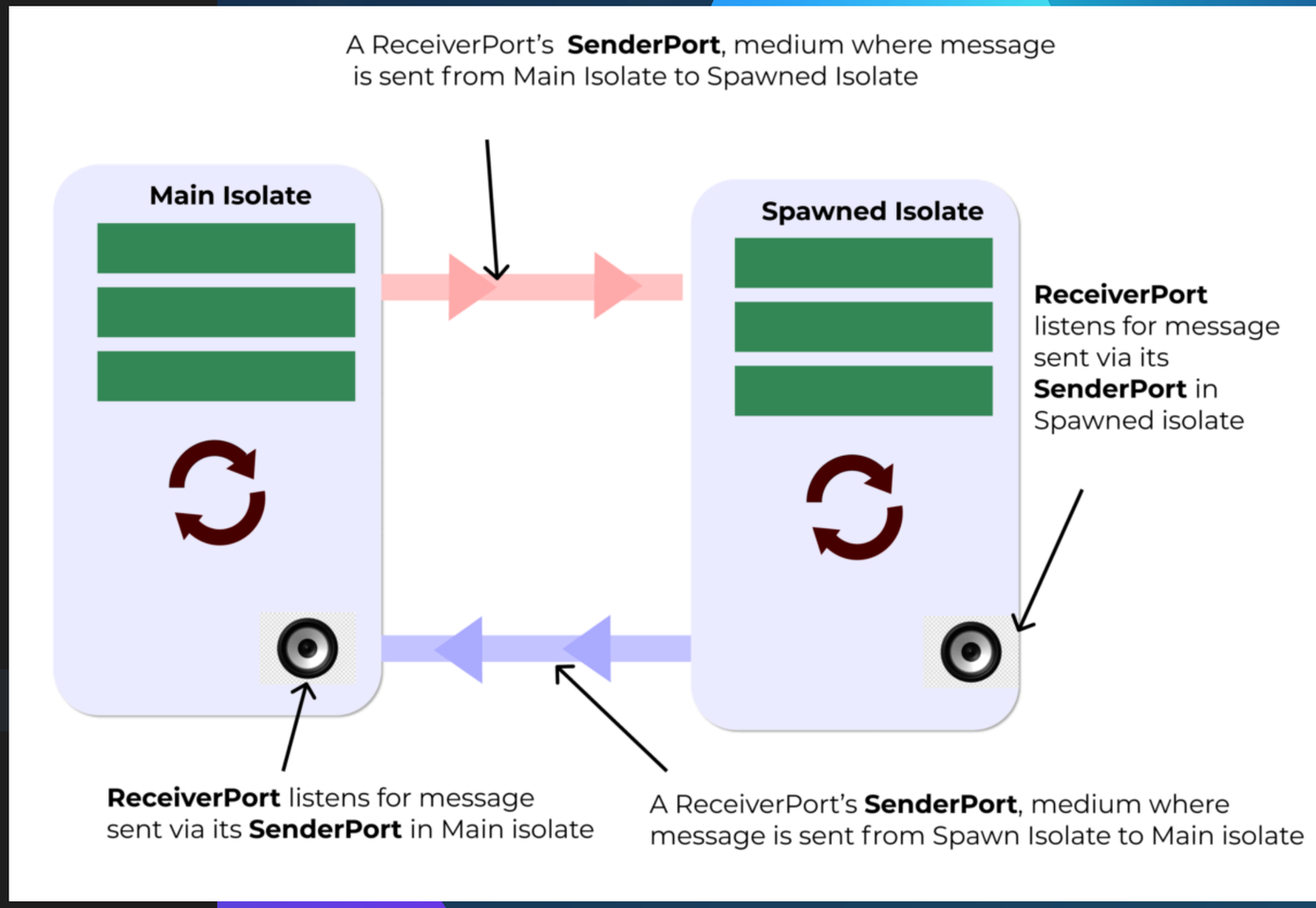
Communication between two isolates can be accomplished by sending messages or values through the ports (**ReceivePort** and **SendPort**).

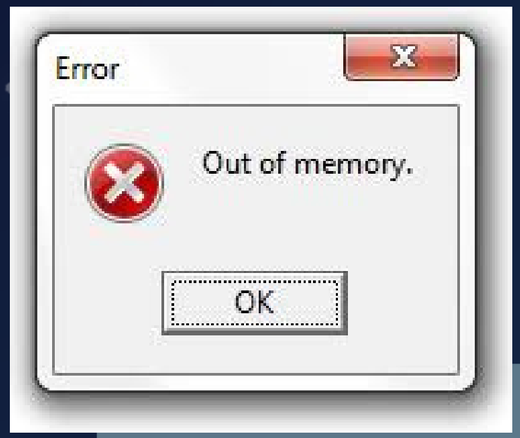
These ports work similarly to Stream, in fact, ReceivePort implements the Stream abstract class. SendPort can be created from ReceivePort by calling ReceivePort's sendPort getter method.

This is the medium through which messages are being sent to ReceivePort.

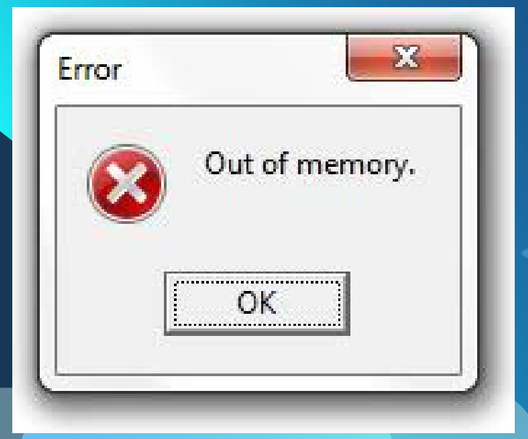
The ReceivePort, on the other hand, listens to messages from its SendPort.

```dart
Future<void> _startCommunication() async {
  /// Create a port to receive messages
  final receivePort = ReceivePort();
  /// Spawn the isolate and send the port
  await Isolate.spawn(workerIsolate, receivePort.sendPort);

 /// Listen for messages from the worker isolate
  receivePort.listen((message) {
    setState(() {
      _message = "Worker Isolate says: $message";
    });
  });
}

static void workerIsolate(SendPort sendPort) {
  /// Simulate some work
  String result = "Hello from Worker Isolate!";

  /// Send the result back to the main isolate
  sendPort.send(result);
}
```

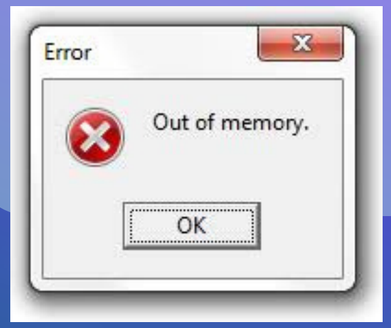


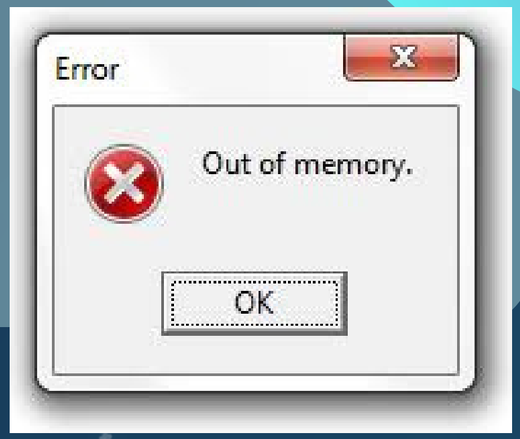


KEEP IN MIND !



It is also important to understand that the transmitted data is essentially copied. Sending data to an isolate, at the moment it is simultaneously in the memory of the isolate and in the memory of the main thread. This can lead to an out of memory (Out-Of-Memory Risk). A crude example: if we want to pass 2 GB of data, on iPhone X (3 GB of RAM) we cannot complete the message transfer operation. Thus, it is best to use isolates for resource-intensive operations that return small amount of data.





HOW ??
ENABLING INTERACTION WITH FLUTTER PLUGIINS

Platform plugins in background isolates (As of Flutter 3.7,)

✅ What it does:

- It allows spawned isolates to access Flutter platform channels (plugins like path_provider, shared_preferences, etc.).
- Works within Dart's default isolate system.
- Requires manual setup: you must pass RootIsolateToken to the new isolate and call BackgroundIsolateBinaryMessenger.ensureInitialized().

🛑 Limitations:

- You still need to manage isolates manually .
- It only fixes platform plugin access; it doesn't simplify isolate management.

HOW ??
ENABLING INTERACTION WITH FLUTTER PLUGIINS

flutter_isolate (Third-Party Plugin)

✅ What it does:

- It extends Dart's isolate system by making it easier to create isolates that work seamlessly with Flutter plugins.
- It automatically binds isolates to the main isolate's platform messaging, so you don't have to pass RootIsolateToken manually.
- Provides an easy API for spawning and managing isolates.

🛑 Limitations:

- Requires an external dependency (flutter_isolate package).

REAL LIFE
USE CASES FOR ISOLATES

- Image Processing (Compression, Cropping, Filters)
- Parsing Large JSON Files
- Reading large files
- Encrypting and decrypting data in the background