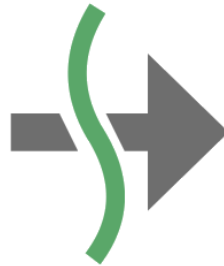# CMPS 312

Asynchronous Programming
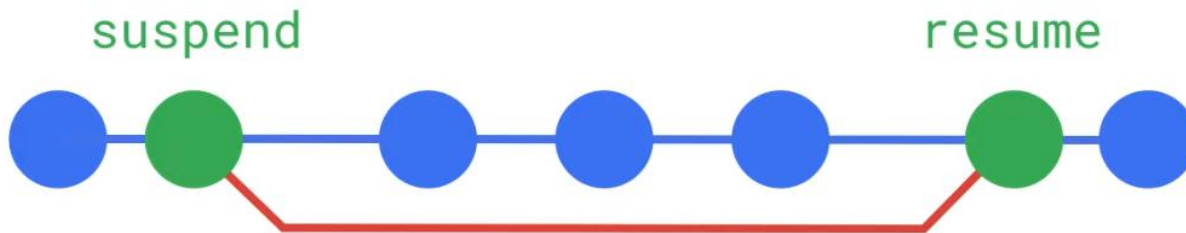
Dr. Abdelkarim Erradi

CSE@QU

Outline

1. Asynchronous Programming

   o Future

   o Stream

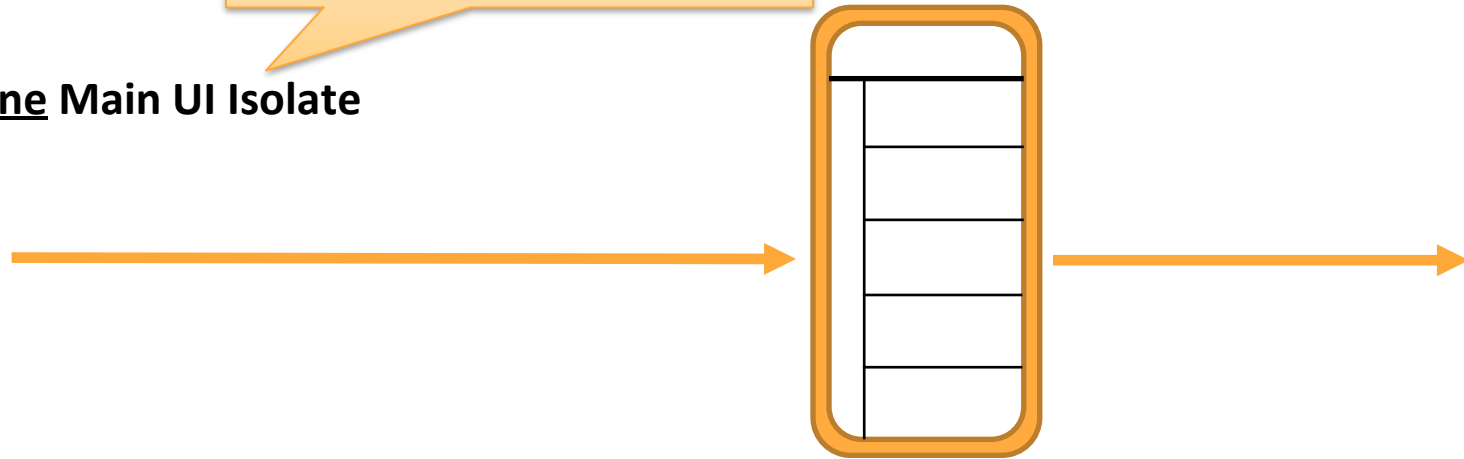2. Isolate for Parallel Execution of Tasks

Asynchronous Programming

User Interface Running on the Main Isolate
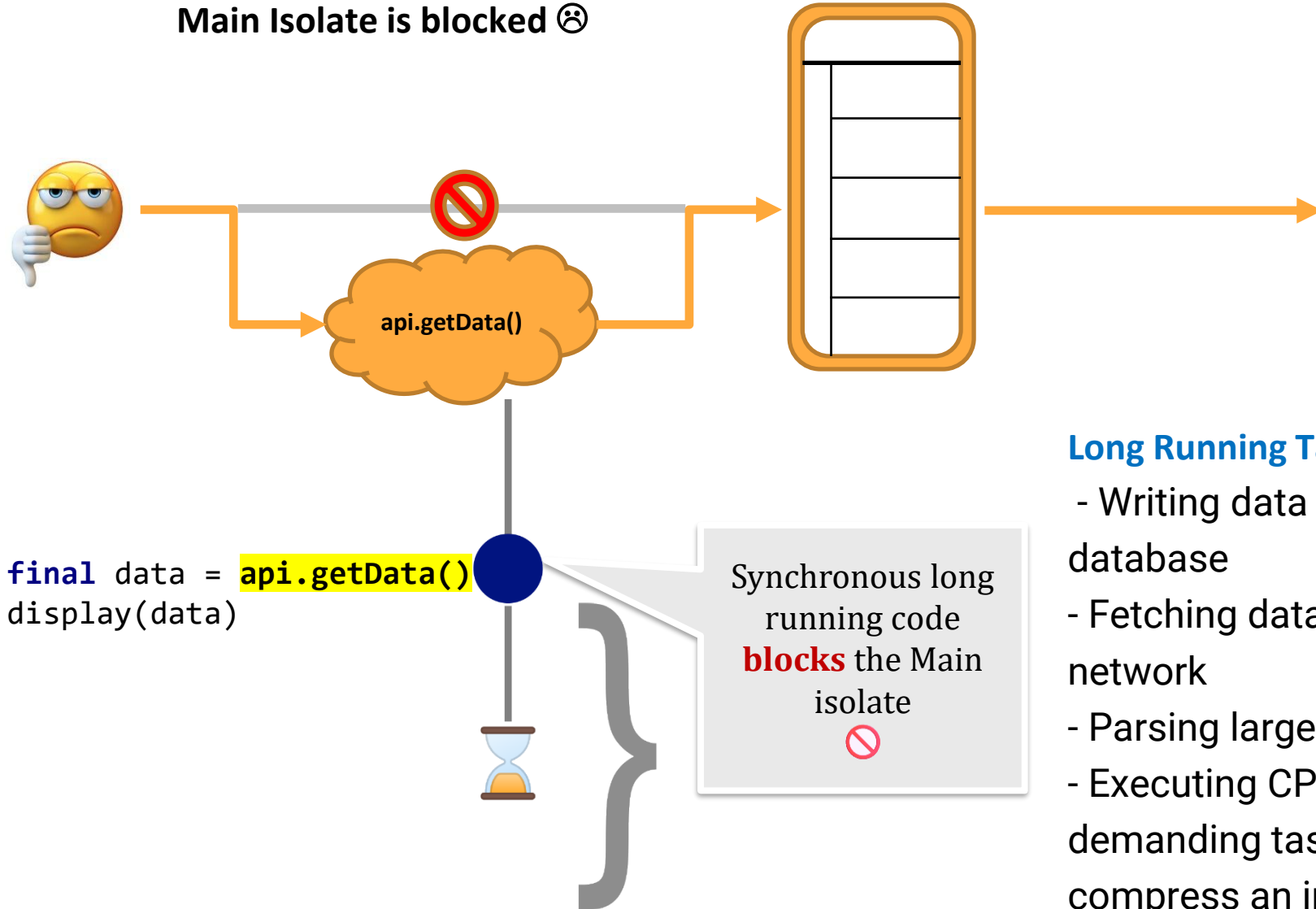
Isolate = an execution unit with its own memory

One Main UI Isolate

To guarantee a great user experience, it's essential to **avoid blocking the main isolate** as it used to handle UI updates and UI events

Long Running Task on the Main Isolate

Main Isolate is blocked ☹

api.getData()

```
final data = api.getData()
display(data)
```

Synchronous long running code **blocks** the Main isolate 🚫

Long Running Tasks include:
- Writing data to a database
- Fetching data from the network
- Parsing large JSON file
- Executing CPU demanding task (e.g., compress an image)

Concurrency vs Parallelism in Dart

- **Concurrency:** ability to handle **multiple tasks over time** without blocking the main Isolate
  - Achieved using **async/await with Future & Stream** on the **same isolate**
  - Use for Network requests (HTTP calls), File I/O, Database queries & any **I/O-bound** task (waiting for external resources)

- **Parallelism:** ability to run **multiple tasks at the same time** on **different isolates**
  - Achieved using **Isolate** (which is a lightweight thread with its own memory) to offloads heavy work
  - Use for **CPU-intensive tasks** (e.g., image compression, encryption, large JSON parsing)

Future
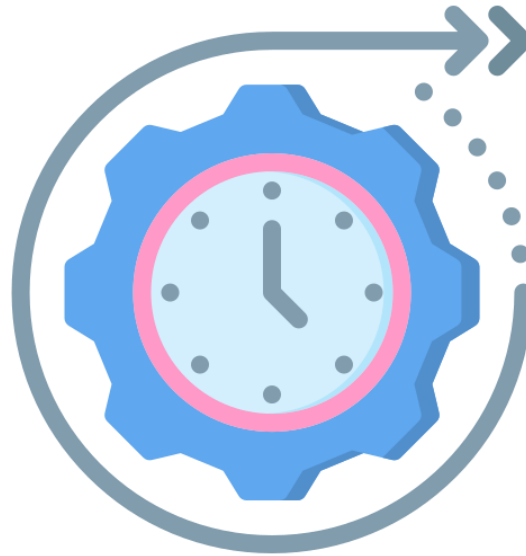
Future

- Future represents a **value** or **error** that will be available later

- Purpose: Used for asynchronous operations

- E.g., `fetchUserOrder` returns a Future

```
final url =
Uri.parse('https://api.example.com/users/$userId/order');
final response = await http.get(url);
```

- The event loop
  - Starts the HTTP request, keeps the app responsive by processing other tasks,
  - Later delivers the response to your code when it's ready

Async Function and Event Loop

- An **async function** can **pause at await and later resume execution** to avoid blocking the isolate

- **Event Loop** ensures non-blocking by:

  o When an async function encounters await:

    - The function is paused, and its state (local variables, execution point) is saved, and remaining work is scheduling in the microtask queue to resume later

    - The Event Loop continues processing other queued tasks (UI rendering, user input handling), keeping the **UI responsive**

  o When the awaited Future completes:

    - The Event Loop resumes the function from where it left off

Event Loop: How it Avoids UI Blocking
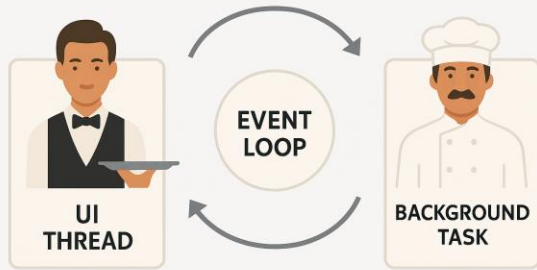
- **UI thread** (analogy **Waiter**)
  - Takes orders (user actions) and serves customers (updates UI)
  - Never goes to the kitchen → stays responsive to guests

- **Background task** (analogy **Chef**)
  - Prepares meals (long-running tasks: network calls, file I/O)
  - Notifies waiter when the meal is ready

- **Event Loop:**
  - **Schedules** tasks and **delivers results** when ready
    - Waiter writes down the order and moves on (doesn't wait in the kitchen)
    - Chef works in the background; notifies the waiter when done
  - This keeps the UI (restaurant) running smoothly without delays

FutureProvider

- Riverpod FutureProvider is used to handle asynchronous operations, like fetching data from an API or database queries

  - **UI rebuilds when the future is completed**: it listens to a Future and triggers a UI rebuild once the operation completes and data is received

  - Handles the **loading**, **error**, and **data** states in a structured manner, e.g.:

    - **loading**: show a spinner until data is available

    - **error**: display error message if something fails

    - **data**: show the received data

FutureProvider Example

```dart
final weatherProvider = FutureProvider<String>((ref) async {
  await Future.delayed(const Duration(seconds: 2)); // Simulate network call
  return "Sunny";  // Data returned from API
});

class WeatherScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final weatherAsync = ref.watch(weatherProvider);

    return Scaffold(
      appBar: AppBar(title: const Text('Weather Forecast')),
      body: weatherAsync.when(
        loading: () => const CircularProgressIndicator(),  // Loading state
        error: (err, stack) => Text('Error: $err'),  // Error state
        data: (weather) => Text('Weather: $weather'), // Success state
      ),
    );
  }
}
```

Stream

Stream

- Stream is used to handle asynchronous data that arrives over time, such as continuous data updates from a remote service

- A Stream allows you to listen and react to events as they arrive, without blocking the main UI

- **`Stream.periodic`** or **`async*`** can be used for asynchronous generator functions that produce a stream of values over time

Stream Example 1

```dart
Stream<int> temperatureStream() {
  return Stream.periodic(Duration(seconds: 1),
      (count) => 25 + count % 5);
}


void main() {
  final tempUpdates = temperatureStream();

  tempUpdates.listen((temp) {
    print("Current temperature: $temp°C");
  });
}
```

Stream Example 2

```dart
Stream<String> symbolsStream() async* {
  yield "🌊";
  await Future.delayed(Duration(milliseconds: 500));
  yield "⚽";
  await Future.delayed(Duration(milliseconds: 300));
  yield "🎉";
}

void main() async {
  await for (var symbol in symbolsStream()) {
    print("Receiving $symbol");
  }
}
```

StreamProvider

- Riverpod StreamProvider is used to listen to asynchronous data streams

    o 🌊 It returns a stream of values produced incrementally over time, allowing for live updates (e.g., receiving updates from a database or Web API to refresh the UI)

    o It provides the latest emitted value from the stream to update widgets when new data arrives

    o Ideal for real-time data, such as stock prices, chat messages, or sensor readings

    o Handles the **loading**, **error**, and **data** states in a structured manner

StreamProvider Example

```dart
final stockPriceProvider = StreamProvider<double>((ref) async* {
  // Simulate fetching stock prices from an API.
  await Future.delayed(const Duration(seconds: 1));
  yield 150.0; // Initial price
  await Future.delayed(const Duration(seconds: 2));
  yield 152.5; // New price update
  await Future.delayed(const Duration(seconds: 2));
  yield 151.0; // Another update
});

class StockPriceScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final stockPriceAsync = ref.watch(stockPriceProvider);

    return Center(
        child: stockPriceAsync.when(
          loading: () => const CircularProgressIndicator(),
          error: (err, stack) => Text("Error: $err"),
          data: (price) => Text("Stock Price: \$${price}"),
    );
}}
```

# Isolate for Parallel Execution of Tasks
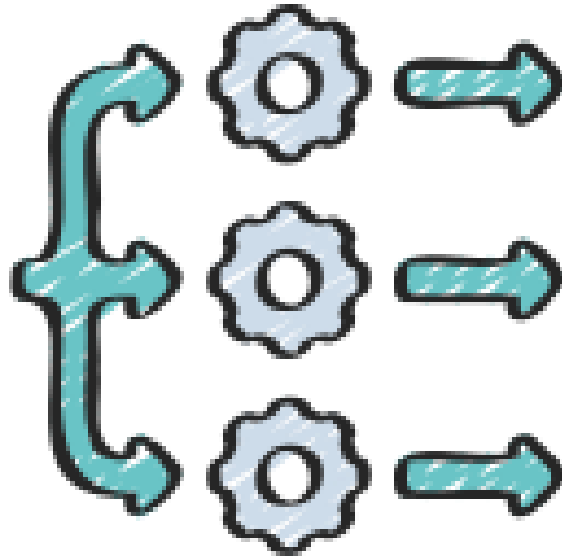
Use Isolate for Parallel Execution of Tasks

- Creates a **separate isolate** to execute a CPU-intensive task without blocking the Main isolate. **Use when:**
  - CPU-intensive tasks (e.g., image compression, encryption, large JSON parsing, heavy computations) that would block the main isolate if run there
  - **Parallel** execution of tasks

- An isolate is the **unit of execution** (like a lightweight thread) within a process
  - Each isolate has its own isolated memory, and its own event loop
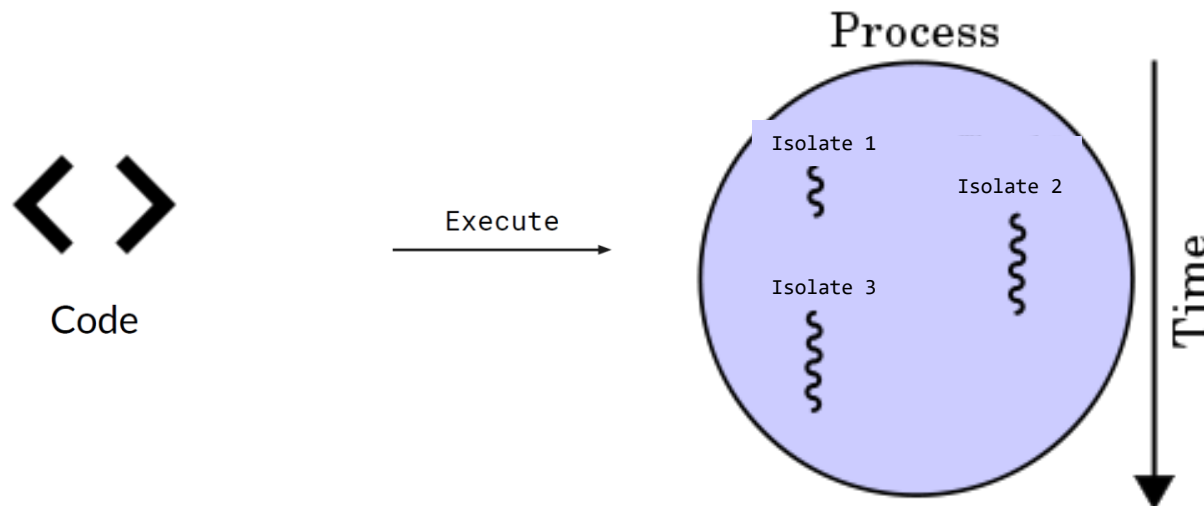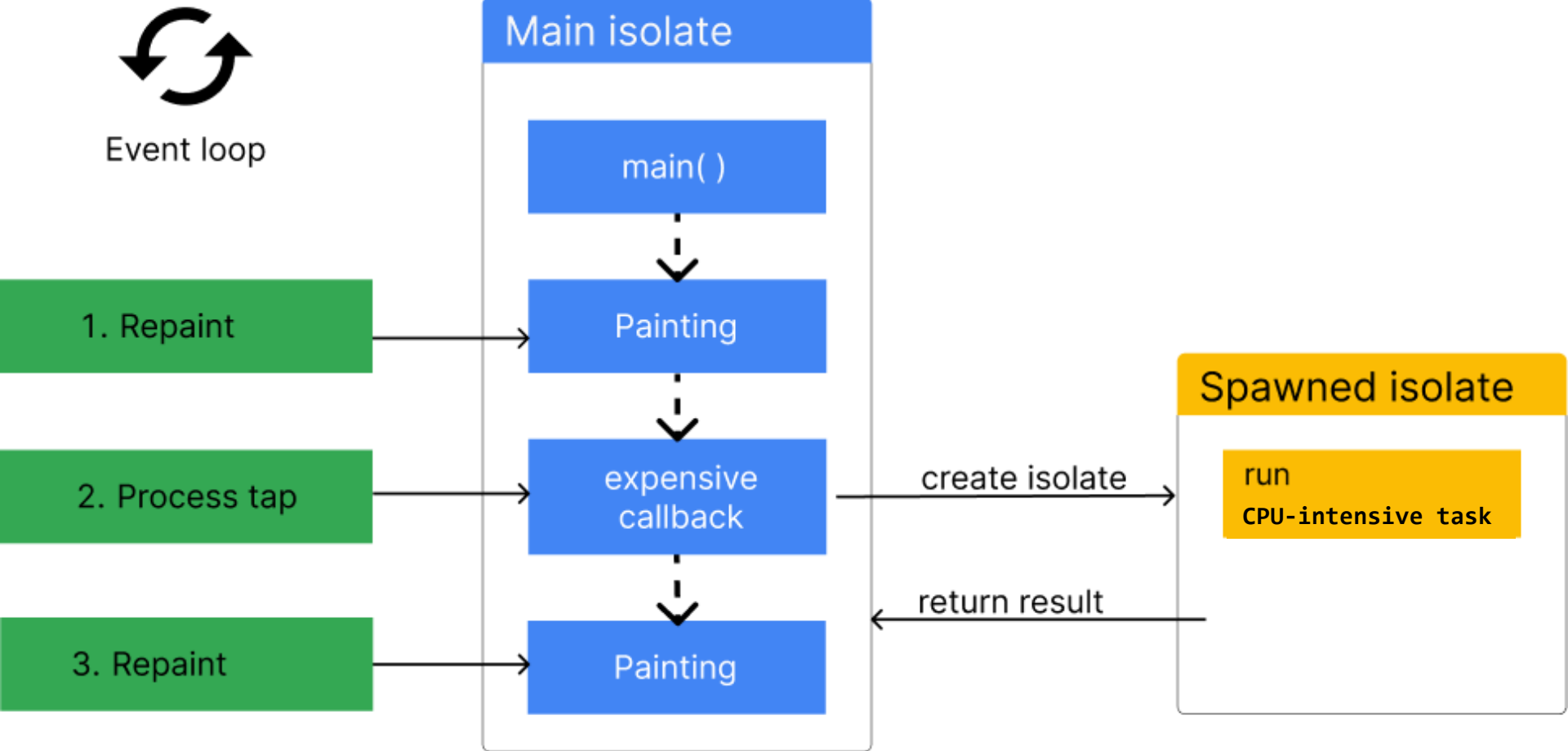
Run Long Running tasks on an isolate

UI Isolate is <u>not</u> blocked ☺

compressImage
(imagePath)
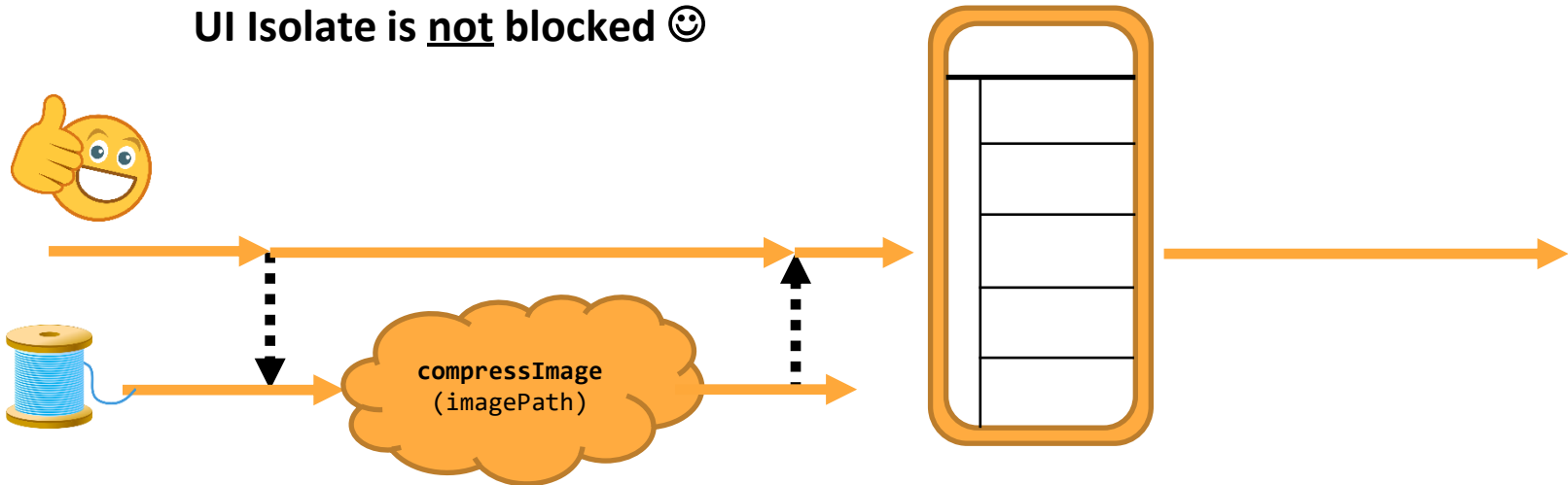
```
// UI remains responsive while compressing the image
var result = await Isolate.run(() => {
      compressImage(imagePath)
})
```

Isolate = Background worker

Async/Await vs Isolate

Feature	Async/Await (Future & Stream)	Isolate
Execution Model	Runs on main isolate using the **event loop**	Runs on a **separate isolate** (parallel lightweight thread)
Memory Sharing	Shares memory with main isolate	Own memory, communicates via messages
Best For	**I/O-bound tasks** (network, DB, file I/O)	**CPU-bound tasks** (image compression, heavy computation)
UI Responsiveness	Keeps UI responsive by suspending tasks	Keeps UI responsive by offloading heavy work
Parallelism	No true parallelism, tasks scheduled sequentially	**True parallel execution** on multiple isolates
Example	`final response = await http.get(url);`	`final result = await Isolate.run(() => compressImage(path));`

Summary

- Use **async/await** for tasks that wait on external resources (network, disk)

  - Manages multiple tasks, one at a time, enabling concurrency without blocking

  - Works with Futures (for single async result) and Streams (for multiple async results over time)

  - Event loop manages tasks sequencing and delivers results when ready

- Use **Isolate** for CPU-heavy tasks that would block the main Isolate

  - Allows tasks to execute in parallel on separate Isolate and prevent blocking the main Isolate

Resources

- Concurrency in Dart
  - https://dart.dev/language/concurrency

- Asynchronous programming tutorial: futures, async, await, and streams
  - https://dart.dev/libraries/async/async-await