CMPS 312

Asynchronous Programming

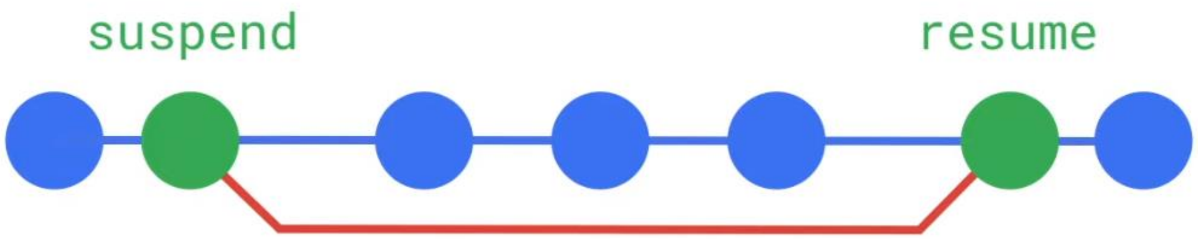
Dr. Abdelkarim Erradi

CSE@QU

Outline

1. Asynchronous Programming Basics
2. Future
3. Stream
4. Isolate

Asynchronous Programming Basics



< Back

Dart Single-thread Model

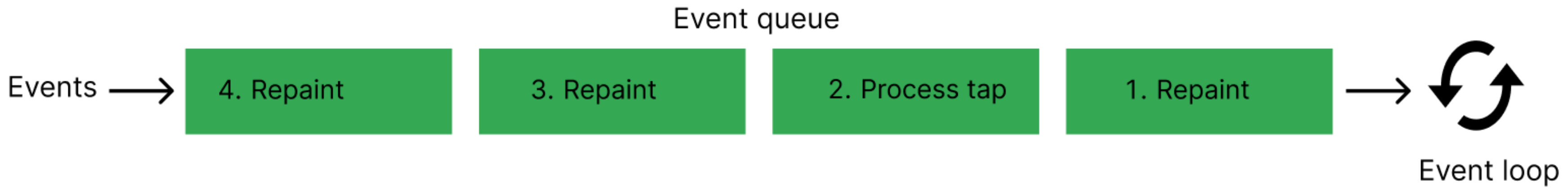
- Dart code runs on a single thread within a single isolate
  - An isolate is a unit of execution with its own memory, and it doesn't share data directly with other isolates
  - This structure helps keep Flutter apps responsive, as each isolate can manage tasks independently without blocking others
- Event loop within each isolate process tasks sequentially
  - This loop ensures that tasks are executed in order, keeping the app responsive

Difference Between Thread and Isolate

- A thread shares memory with other threads in the same process, which can lead to data inconsistencies
- Isolate: An isolate has its own memory, preventing data conflicts.
  - Communication between isolates is through message-passing, not shared memory.
- For tasks needing high computation additional isolates can be created, but they won’t interfere with the main UI isolate

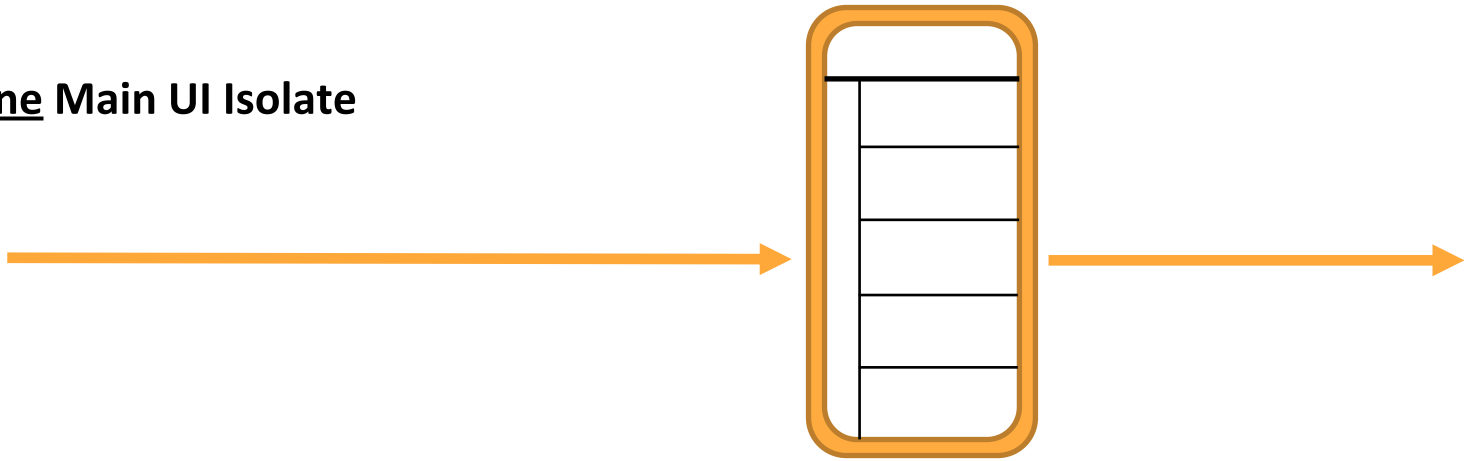
Event Loop

- Dart's runtime model is based on an event loop
  - The event loop is responsible for executing the program code, collecting and processing events
  - As the application runs, all events are added to a queue, called the event queue
  - Events can be anything from requests to repaint the UI, to user taps and keystrokes, to I/O from the disk
  - The event loop processes events in the order they're queued, one at a time



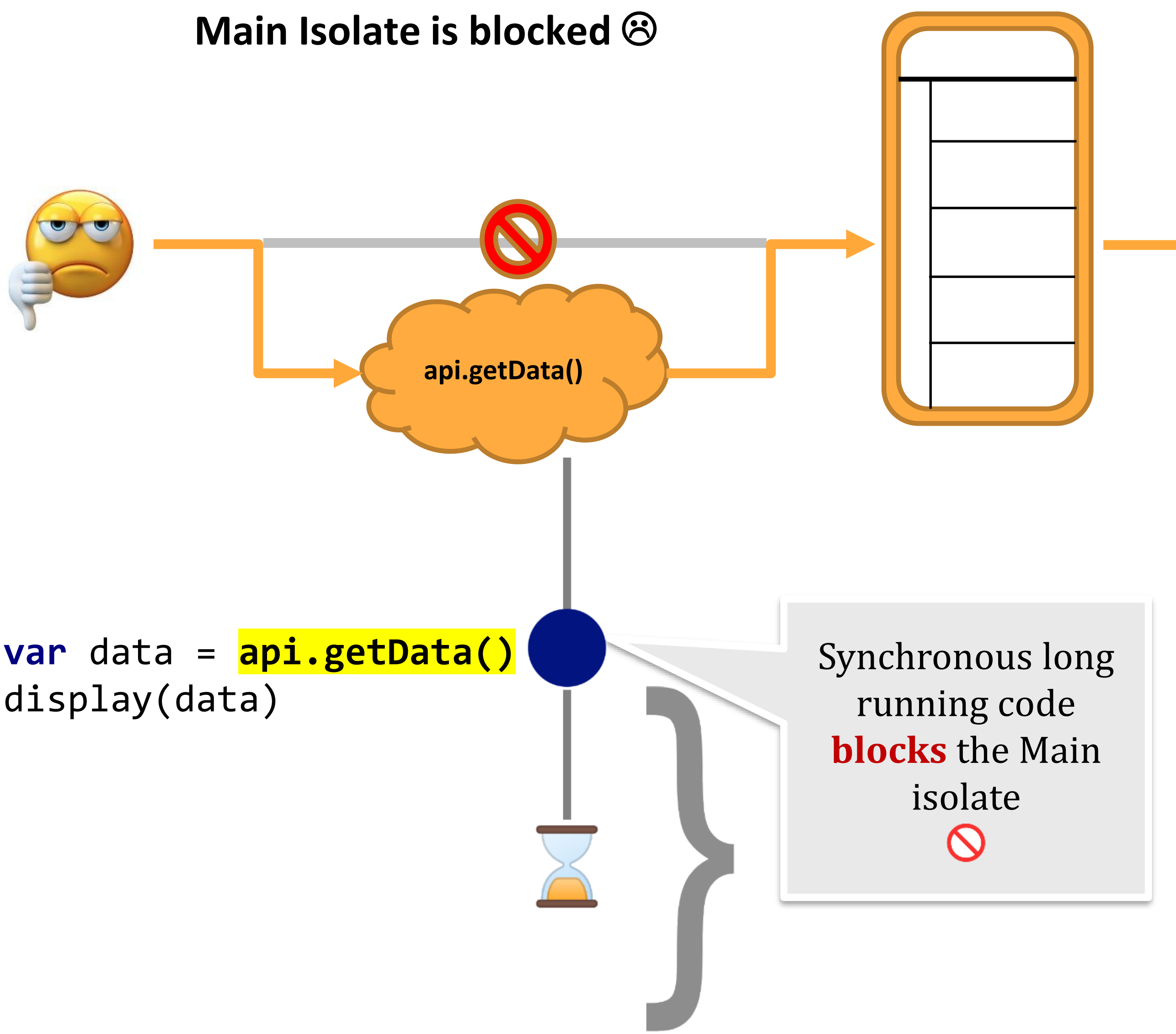
User Interface Running on the Main Isolate

<u>One</u> 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



Long Running Tasks include:

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

How to address problem of long-running tasks?

- **Asynchronous Programming**: is a programming paradigm that allows certain tasks to run separately from the main execution flow
  - Enabling the app to execute other tasks in the meantime
  - Particularly useful in operations that involve waiting, such as **network requests, file Input/Output**, **Database read/write**, or time-consuming computations
  - Ensures the UI remains responsive even when waiting for asynchronous tasks like network requests

Why Asynchronous Programming?

Most mobile apps typically need:

Call Web API (Network Calls)	Database Operations (read/write to DB)	Complex Calculations (e.g., image processing)

Can use Asynchronous Programming to offload long-running computations or Asynchronous I/O operations without blocking the main UI thread

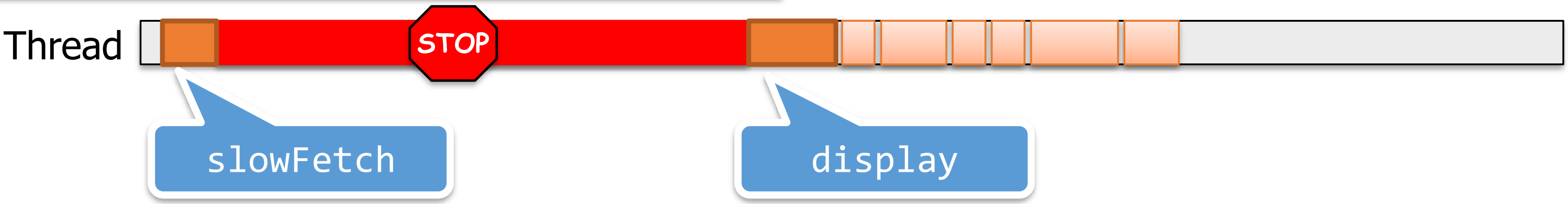
Dart asynchronous programming features

- Future & Stream: used to manage **Concurrency** , which means handling multiple tasks over time
  - Futures represent a single result (e.g., a network request), while Streams handle a series of asynchronous events (e.g., a continuous data feed)
  - Enables non-blocking multitasking
- Isolate: used for **Parallelism**, allowing simultaneous execution by running tasks in separate memory spaces (isolates)
  - Ideal for CPU-intensive tasks
  - Enables Flutter to offload heavy work from the main UI isolate without blocking the app’s responsiveness

Synchronous vs. Asynchronous Functions

```
val result = slowFetch(...) // UI Thread
display(result)   // UI Thread
```

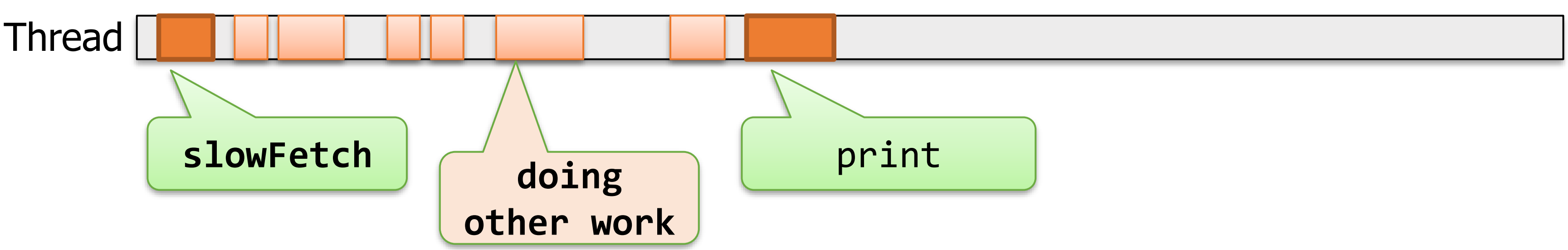
Synchronous (i.e. **Blocking**) → Wait for result before returning




```
// Slow request with callbacks
void makeNetworkRequest() {
      // The slow network request runs on another isolate
      var result = await Isolate.run(() => {
            api.slowFetch()
      });
      // When the result is ready, print it
      print(result);

}
```

Asynchronous (i.e. **Non-Blocking**) → do an **asynchronous** call to slowFetch using backgroud thread, then later update then UI with the result is ready



UI not blocked

Blocking vs. Non-Blocking (async/suspendable task)

- **Salat** **vs.** **Reading a Book**

Mum: 📢 "Fatima comedown dinner ready!"

=> Salat is a **bocking** task. The caller needs to **wait** for Salat to complete to get an answer

=> Reading a book is a non-blocking task than can be **suspended** then **resumed**: add a bookmark then suspend reading, when ready resume reading from the bookmark

Summary

- Async functions implements computation that can be suspended then resumed
- Easier **asynchronous** programming
  - Replace callback-based code with sequential code to handle asynchronous long-running tasks without blocking
  - Structure of asynchronous code is the same as synchronous code

Future

Back

Future

- A Future represents a potential value, or error, that will be available at some time in the future
  - a promise that there will be a value or an error at some point
- Futures are used for asynchronous operations
  - E.g., `fetchUserOrder` returns a Future that completes with a string after a delay of 2 seconds

```
Future<String> fetchUserOrder() {
  // Imagine that this function is more complex and slow
  return Future.delayed(const Duration(seconds: 4),
              () => 'Large Latte');
}
```

Working with Futures: Then and CatchError

- You can handle the result of a Future using **`then`** and errors using **`catchError`**
  - But can lead to deeply nested code
  - Dart offers async and await to write asynchronous code that looks synchronous

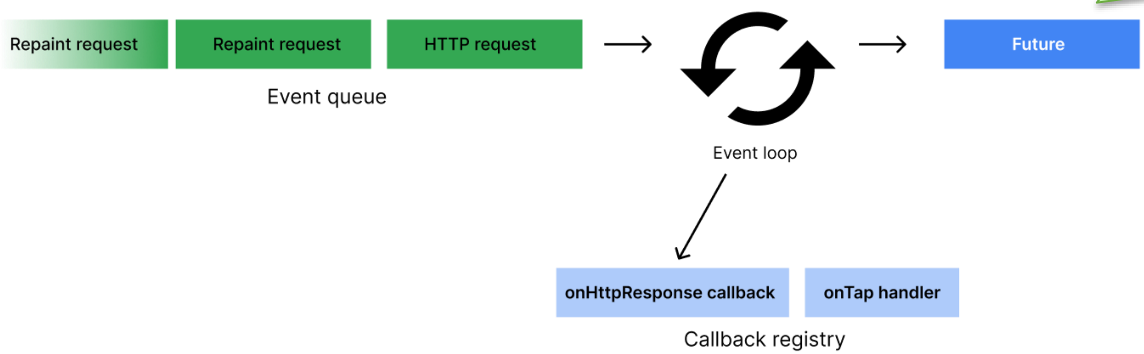
```
var order = fetchUserOrder();

fetchUserOrder().then((order) {
    print(order);
  }).catchError((error) {
    print(error);
  });
```

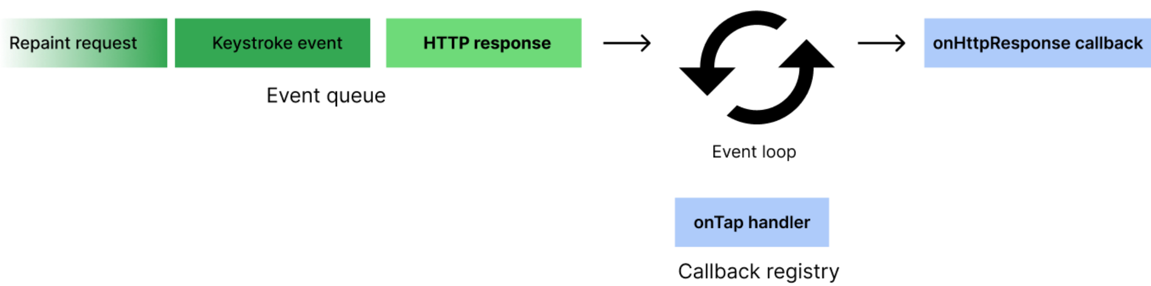
Future is executed using the Event Loop

```
http.get('https://example.com').then((response) {
  if (response.statusCode == 200) {
    print('Success!');
  }
});
```

- When this code reaches the event loop, it immediately calls **http.get**, and returns a Future
- It also tells the event loop to hold onto the **callback** in the **then()** clause until the HTTP request resolves



Some time later....



- When HTTP request resolves, the Event loop executes the callback, passing the result of the request as an argument

async - await

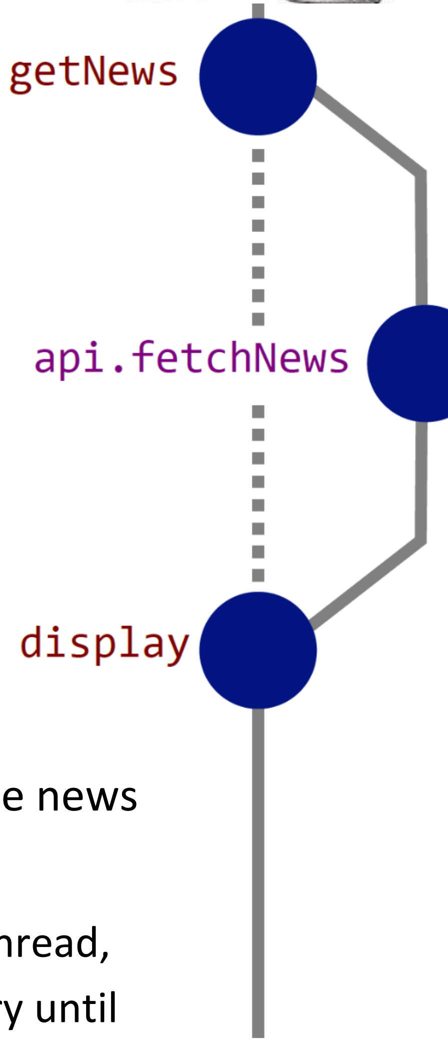
- Mark a function as **async** to use await within it
  - **await** pauses the function until the Future completes
  - This code is cleaner and easier to understand compared to chaining **then** and **catchError**

```
Future<void> displayUserOrder() async {
  try {
    String order = await fetchUserOrder();
    print(order);
  } catch (error) {
    print(error);
  }
}
```

Async Non-blocking calls

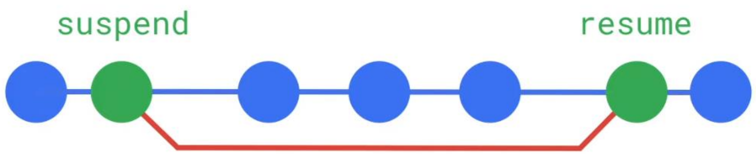
```
ElevatedButton(
    onPressed: () {
      getNews();
    },
    child: const Text("Get News")
)

Future<NewsItem> getNews() async {
    return await api.fetchNews();
}
```



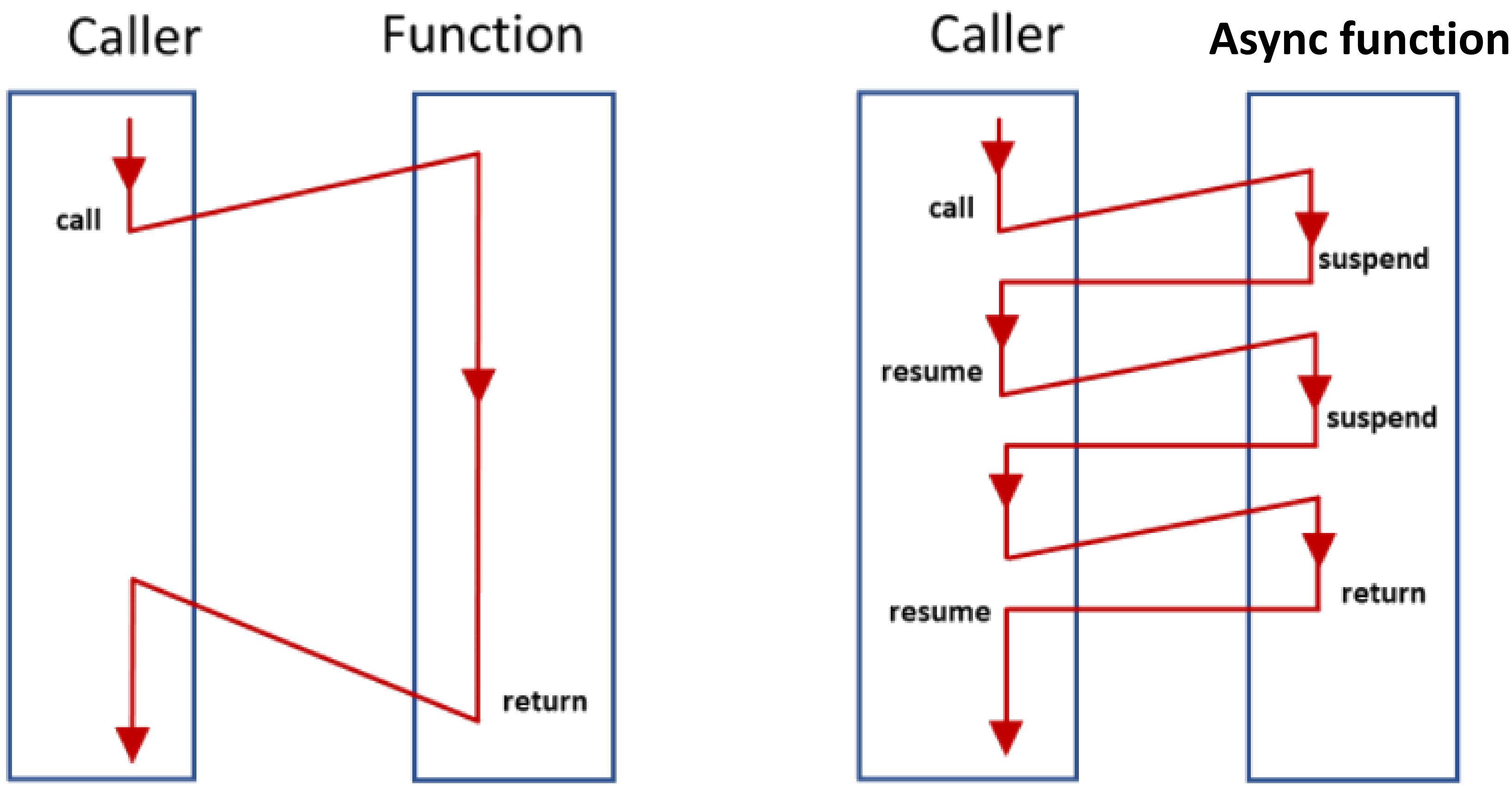
- When getNews async function is waiting for the result from the remote news service it does NOT block instead the runtime:
  - suspends the execution of **getNews()** function, removes it from the thread, and stores the state and the remaining function statements in memory until the result is ready then resumes the function execution where it left off

Async function



- **Async** function is a function that can be paused (i.e., suspended) and **resumed** later
- When an async function reaches an await (where it needs to wait for a result), it **doesn't block** the isolate. Instead, the Dart runtime:
  - **suspends** the function execution, removes it from the isolate, and saves its current state (including the point of suspension and any variables) in memory
  - **resumes** the function execution where it left off, once their awaited result is available
- While it's suspended waiting for a result, the Event Loop processes other tasks (such as handling user interactions) so the isolate isn't idle

Function vs. Async Function



Async function can **suspend** at some points and later **resume** execution when the return value is ready

Combining Multiple Futures

- Run multiple asynchronous functions and wait for all of them to complete using **Future.wait**

```
Future<void> displayAllData() async {
  try {
    var results = await Future.wait([
      fetchUserData(),
      fetchAnotherData()
    ]);
    print(results[0]); // User data loaded
    print(results[1]); // Another data loaded
  } catch (e) {
    print(e);
  }
}
```

For Flutter use 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

< Back

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

```
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");
  }
}
```

For Flutter use StreamProvider

- Riverpod StreamProvider is used to listen to asynchronous data streams
  - 🌊 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)
  - It provides the latest emitted value from the stream to update widgets when new data arrives
  - Ideal for real-time data, such as stock prices, chat messages, or sensor readings
  - Handles the **loading**, **error**, and **data** states in a structured manner

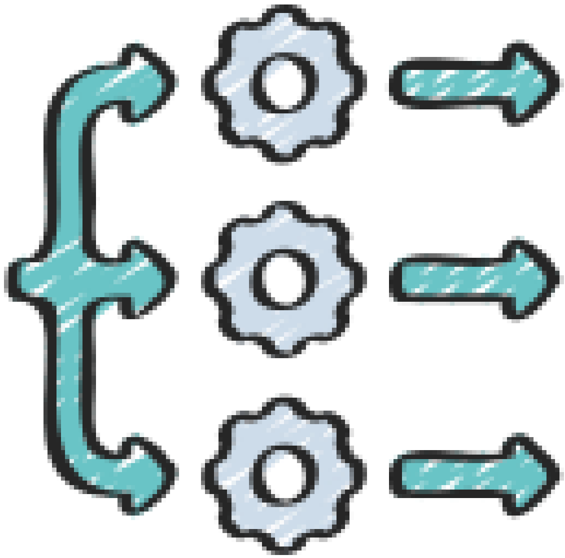
StreamProvider Example

```
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}"),
    );
}}
```

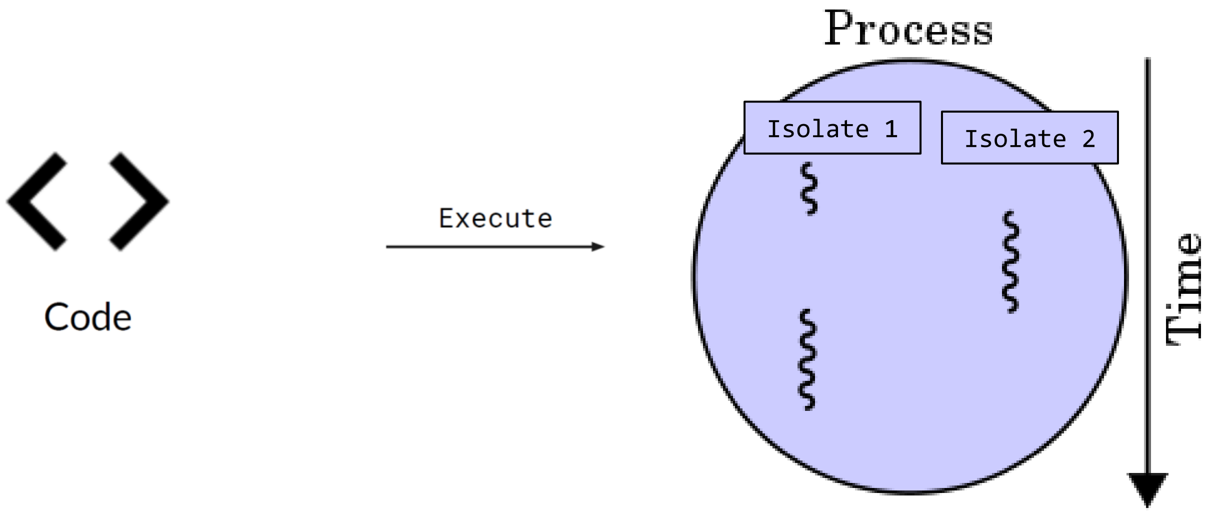
Isolate



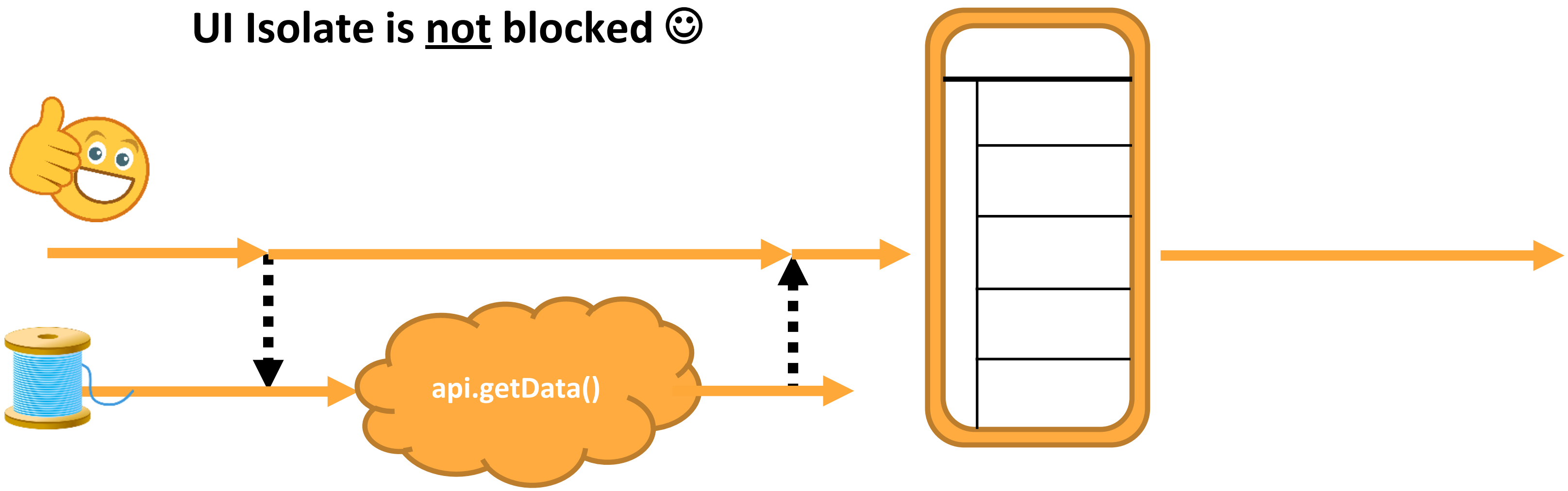
< Back

How to address problem of long-running task?

- Use Isolate to execute a long running task without blocking the Main isolate
- An isolate is the **unit of execution** within a process
  - It allows **concurrent** execution of tasks within an App
  - Each isolate has its own isolated memory, and its own event loop

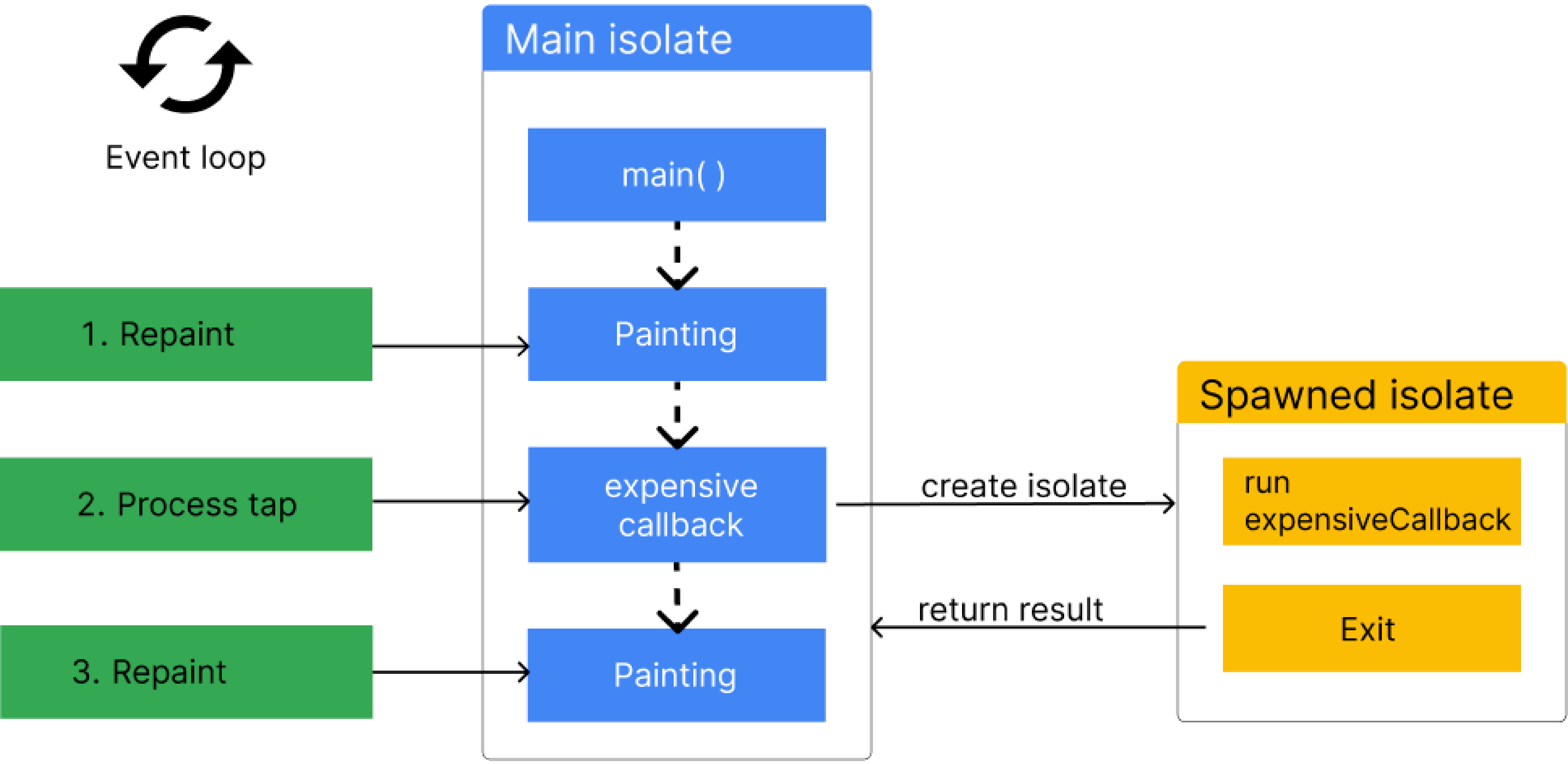


Run Long Running tasks on an isolate




```
var result = await Isolate.run(() => {
        api.getData()
})
```

Isolate = Background worker



Summary

- async functions enable **non-blocking** code execution
  - Event loop manages task sequencing and resumes suspended functions when their results are ready
- Dart's async programming ensure non-blocking, performant code
  - **Concurrency (Future & Stream)**: Manages multiple tasks, one at a time, in a non-blocking way
    - Future (for single async tasks) and Stream (for multiple async events over time)
  - **Parallelism (Isolate)**: Allows tasks to execute in parallel on separate threads, using message-passing for communication

Resources

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

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