CMPS 312

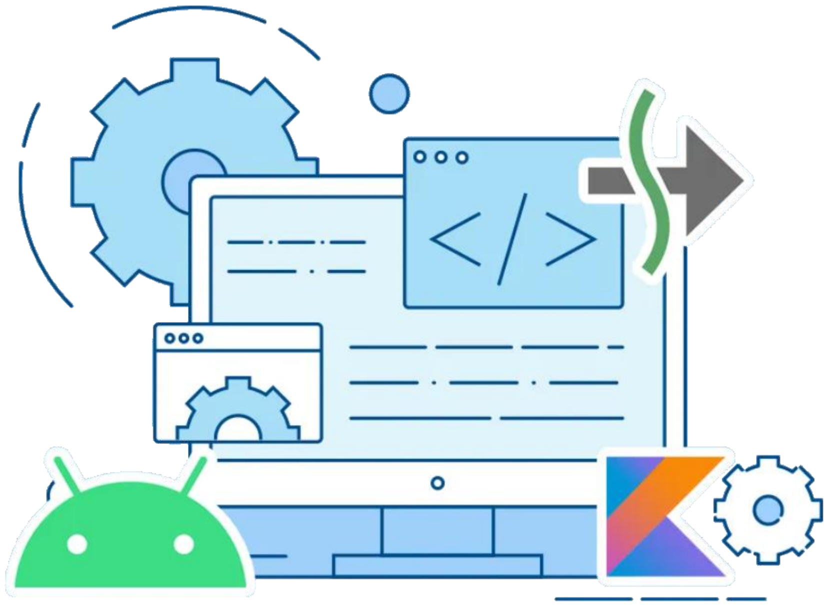
Asynchronous Programming

Dr. Abdelkarim Erradi

CSE@QU

Outline

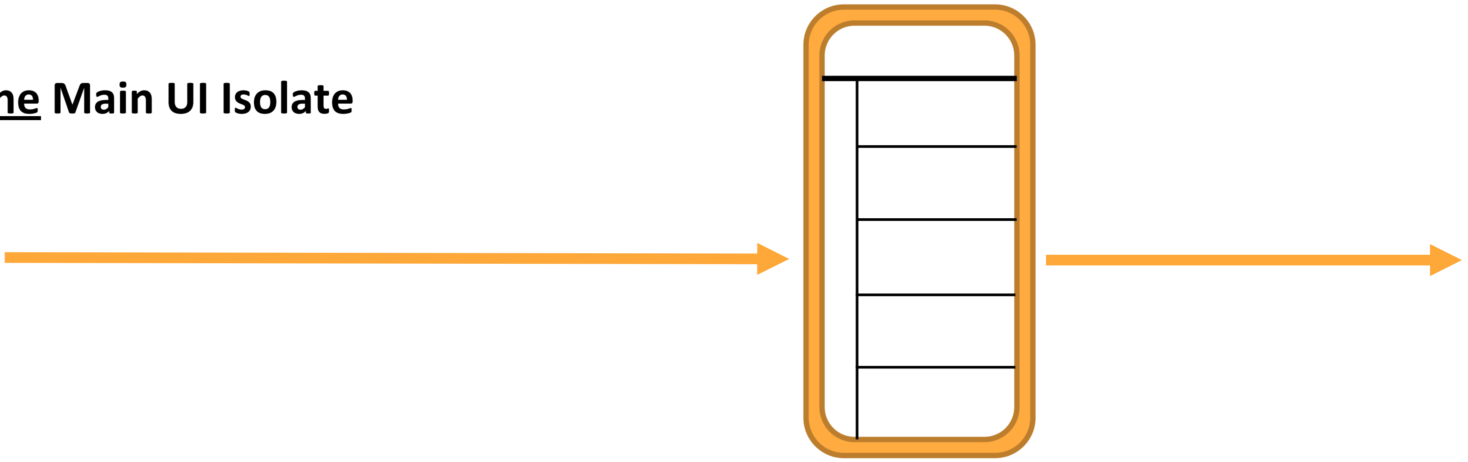
Asynchronous Programming Basics



< Back

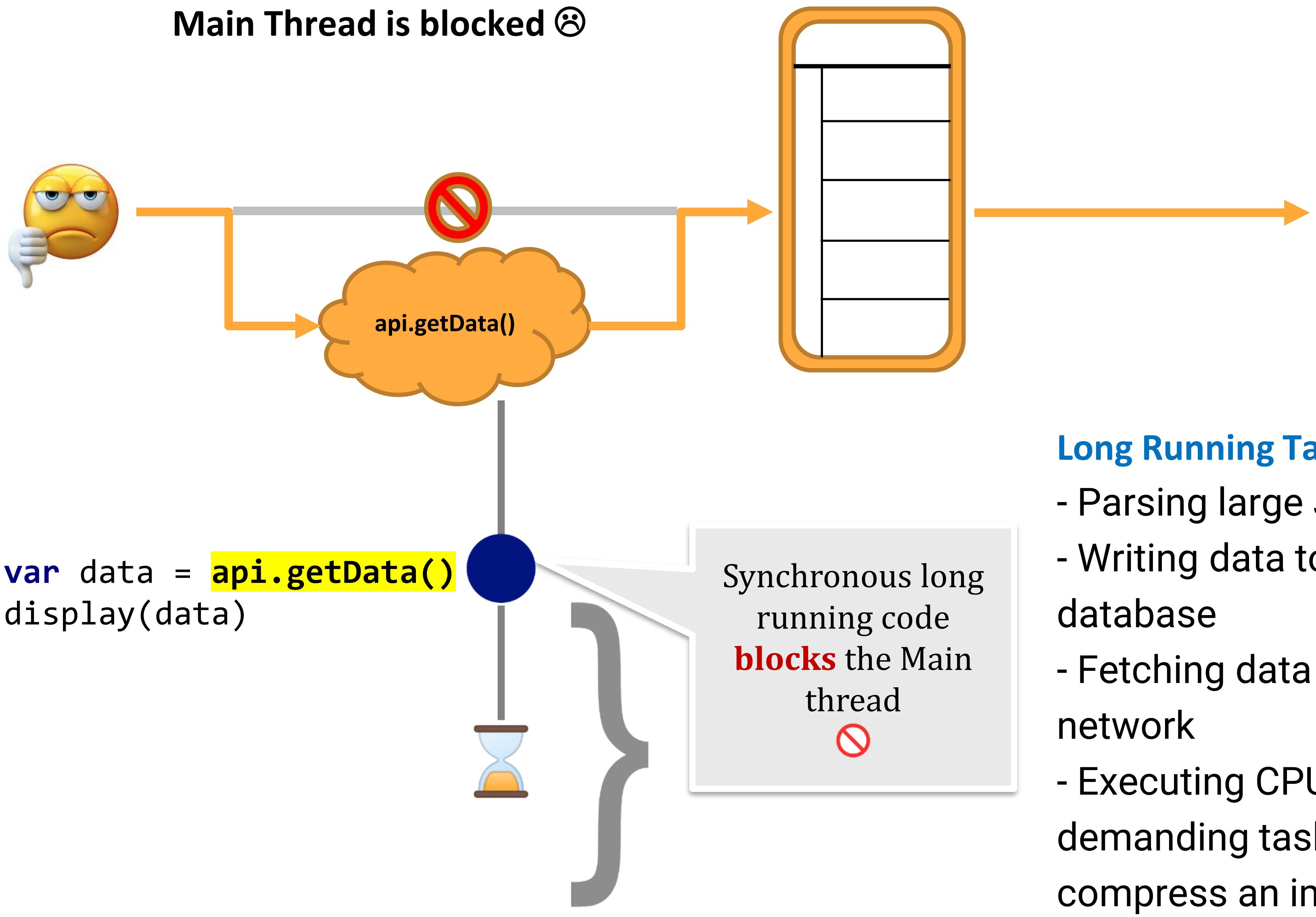
User Interface Running on the Main Isolate

<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 Thread



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

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

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 task?

- **Asynchronous Programming**: is a programming paradigm that allows certain tasks to run separately from the main execution thread
    - Enabling your app to execute other tasks in the meantime
    - Particularly useful in operations that involve waiting, such as **network requests, file I/O**, **DB read/write**, or time-consuming computations
- Dart:
    - Future
    - Stream
    - Isolate (thread)

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

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
  - 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);
  });
```

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);
  }
}
```

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);
  }
}
```

Summary

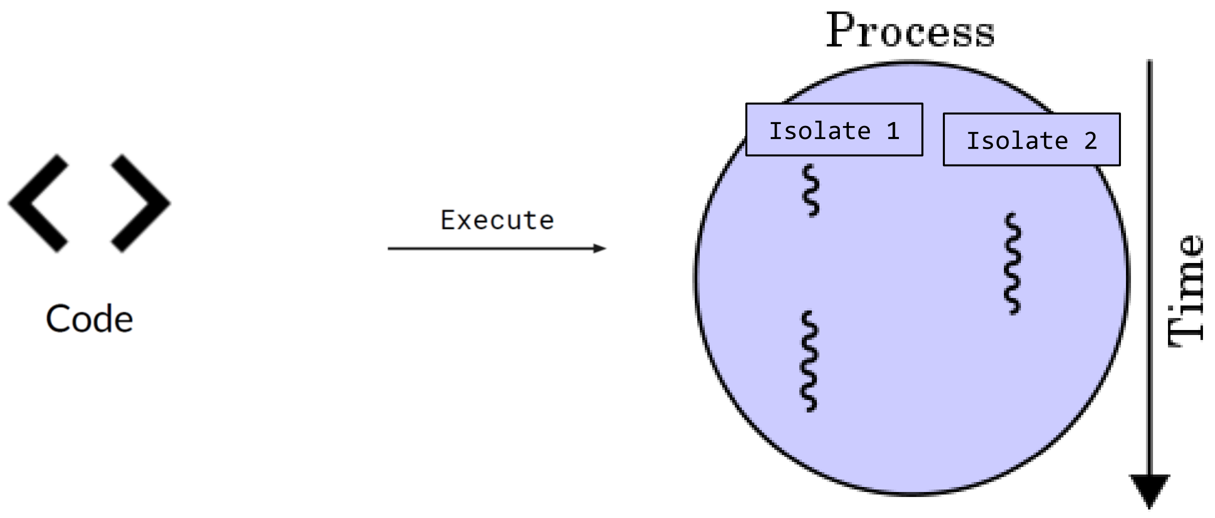
- · async is used for asynchronous functions that return a single Future or value.
- · async* is used for asynchronous generator functions that produce a sequence of values over time using a Stream.
- async is for functions that perform asynchronous operations and return a single result, while async* is for functions that generate multiple results over time in an asynchronous manner

How to address problem of long-running task?

- How to execute a long running tasks without blocking the Main isolate?

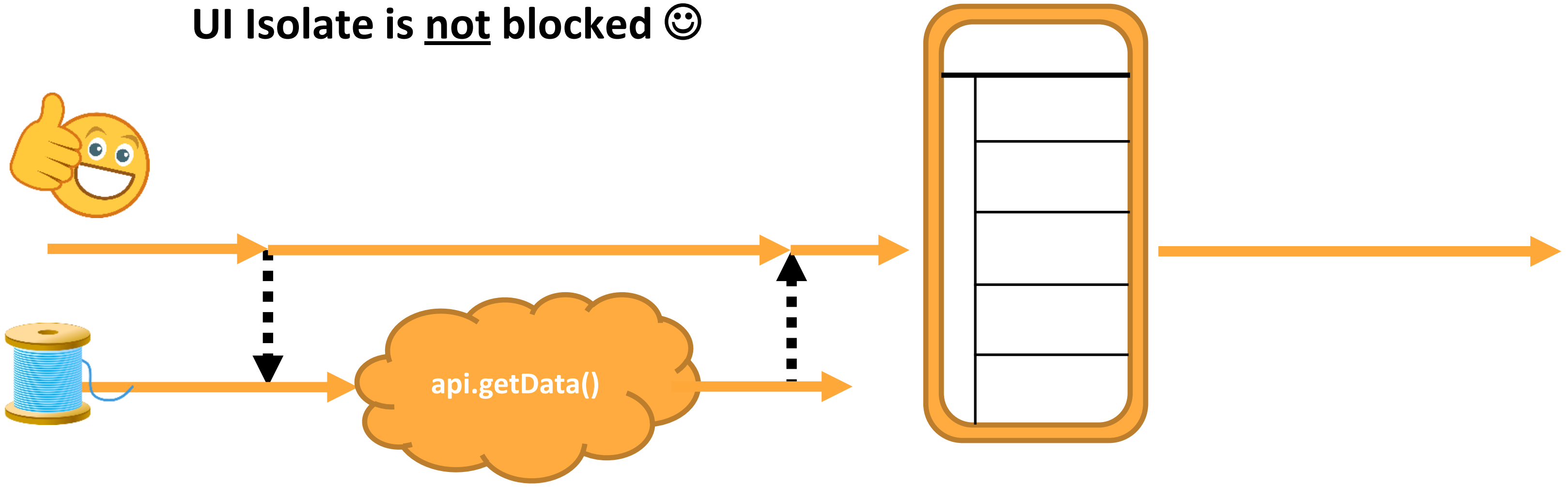
  => Solution 1: **Use isolates**

- An isolate is the **unit of execution** within a process
  - It allows **concurrent** execution of tasks within an App



Solution 1 – Run Long Running tasks on a background isolate

UI Isolate is <u>not</u> blocked ☺

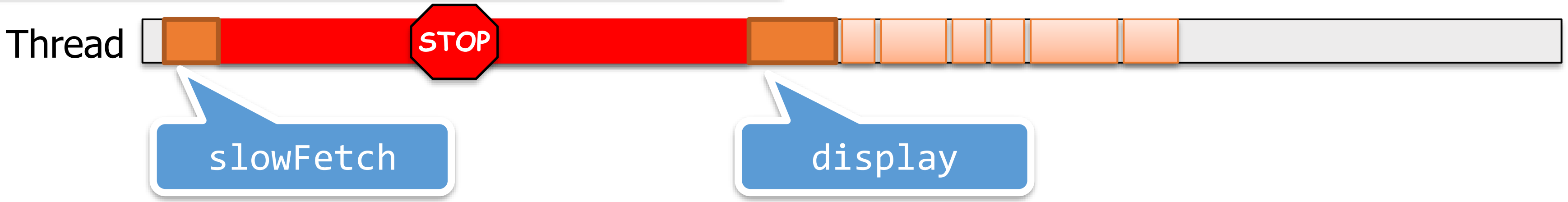


```
var result = await Isolate.run(() => {
        api.getData()
})
```

Synchronous vs. Asynchronous Functions

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

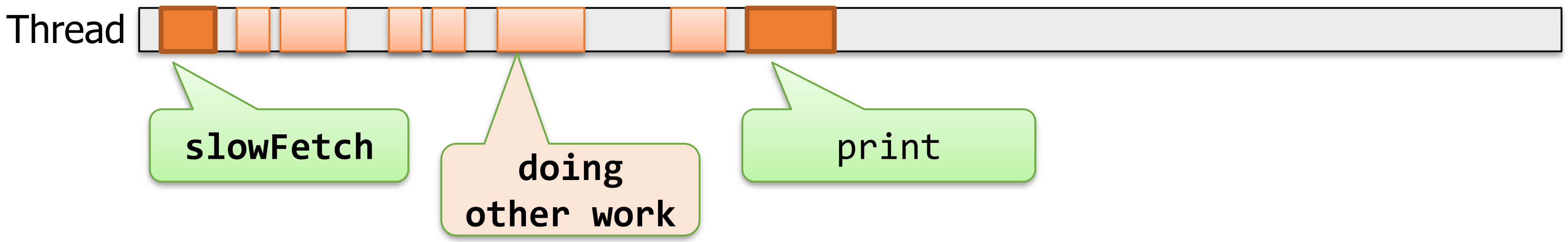
Synchronous (i.e. **Blocking**) → Wait for result before returning



```
// 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



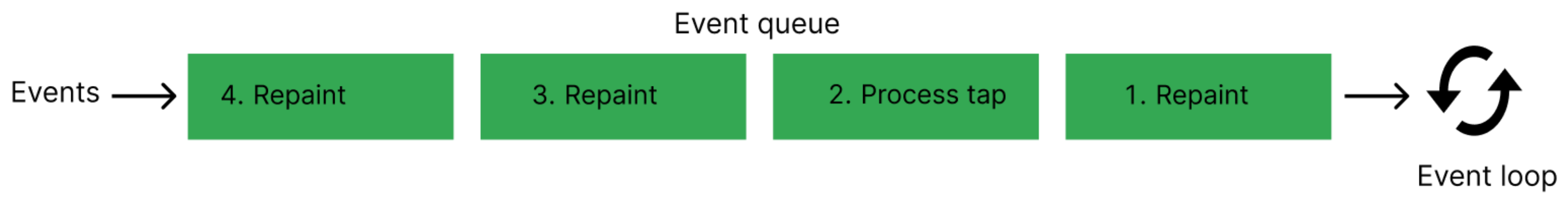
UI not blocked

Isolates

- All Dart code runs in isolates, starting in the default main isolate, and optionally expanding to whatever subsequent isolates you explicitly create
    - When you spawn a new isolate, it has its own isolated memory, and its own event loop. The event loop is what makes asynchronous and concurrent programming possible in Dart.

Event Loop

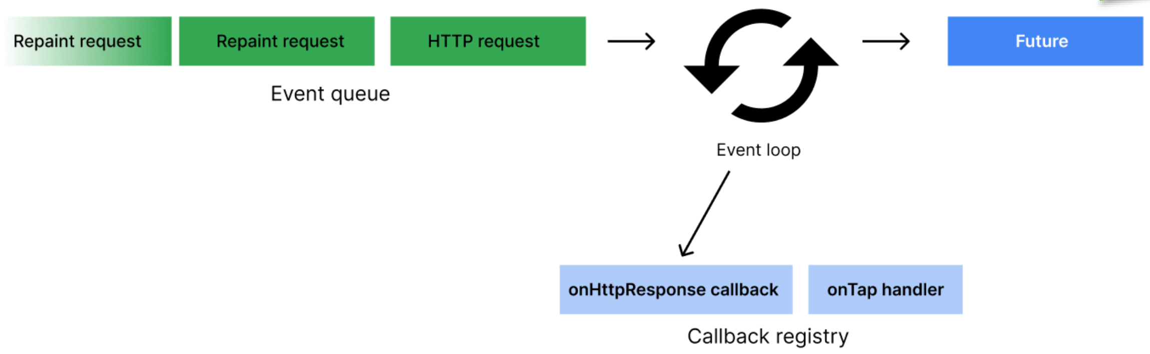
- Dart’s runtime model is based on an event loop. The event loop is responsible for executing your program's code, collecting and processing events
  - As your application runs, all events are added to a queue, called the event queue
  - Events can be anything from requests to repaint the UI, to user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



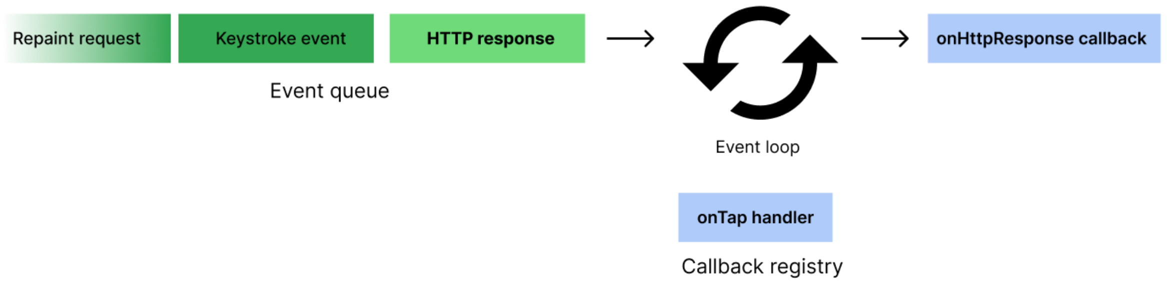
Event Loop Example

```
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

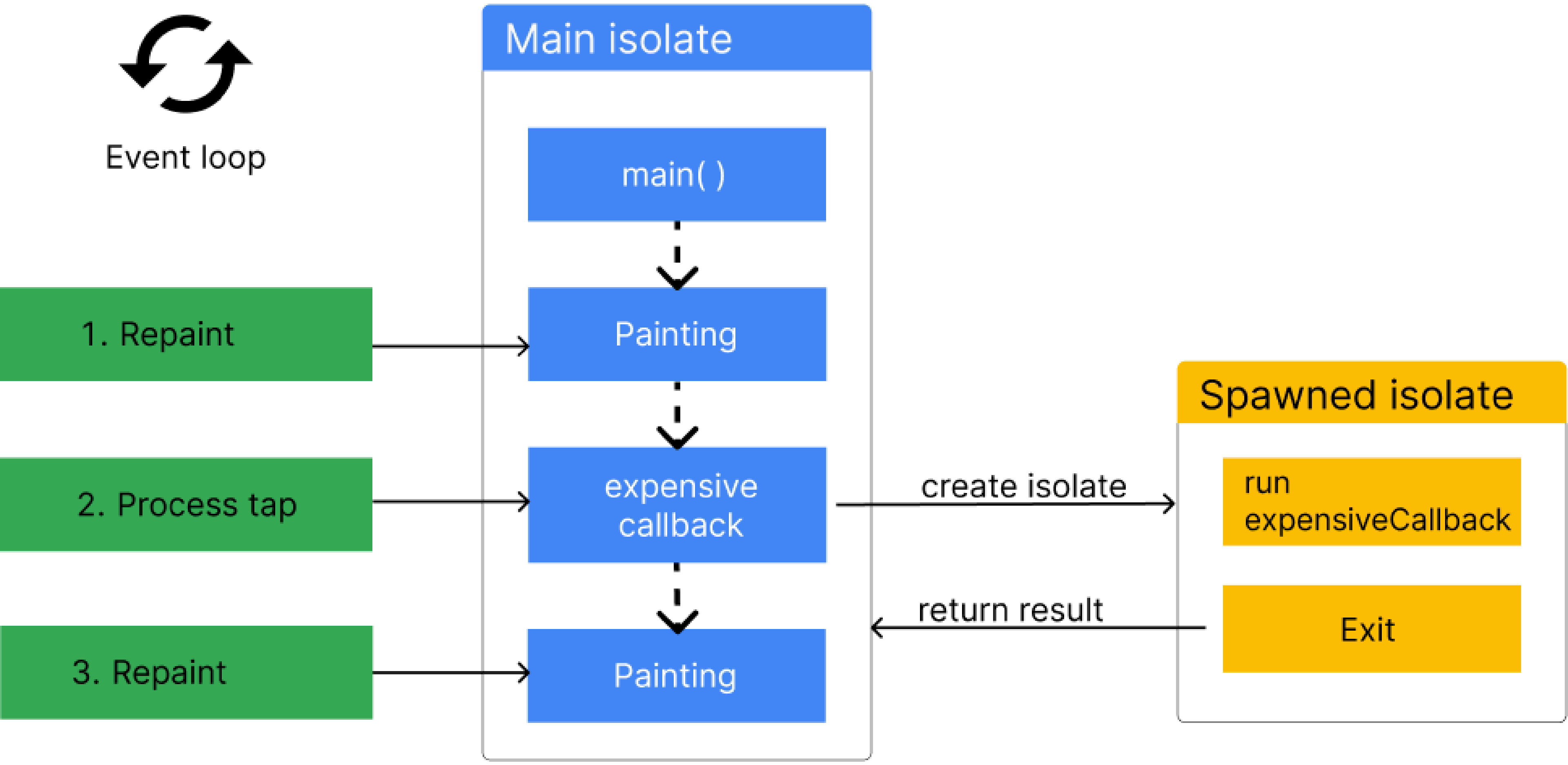


Some time later....

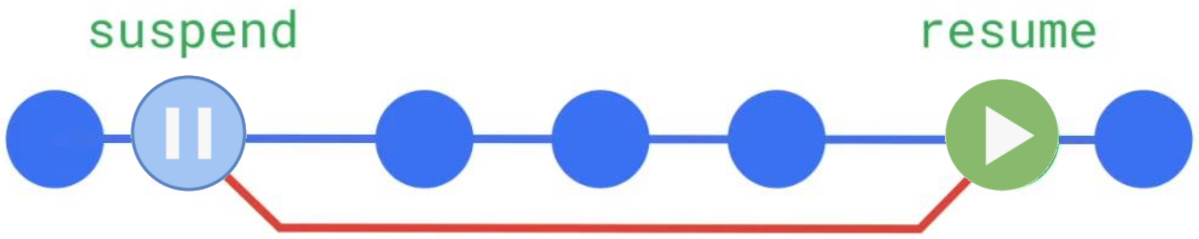


- When HTTP request resolves, the Event loop executes the callback, passing the result of the request as an argument

Background workers



Asynchronous Programming Programming Model



Back

Blocking vs. Non-Blocking (suspendable task)

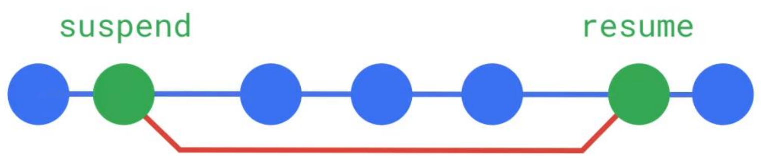
- **Salat** **vs.** **Reading a Book**

Mum: 📢 "Fatima comedown dinner ready!"

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

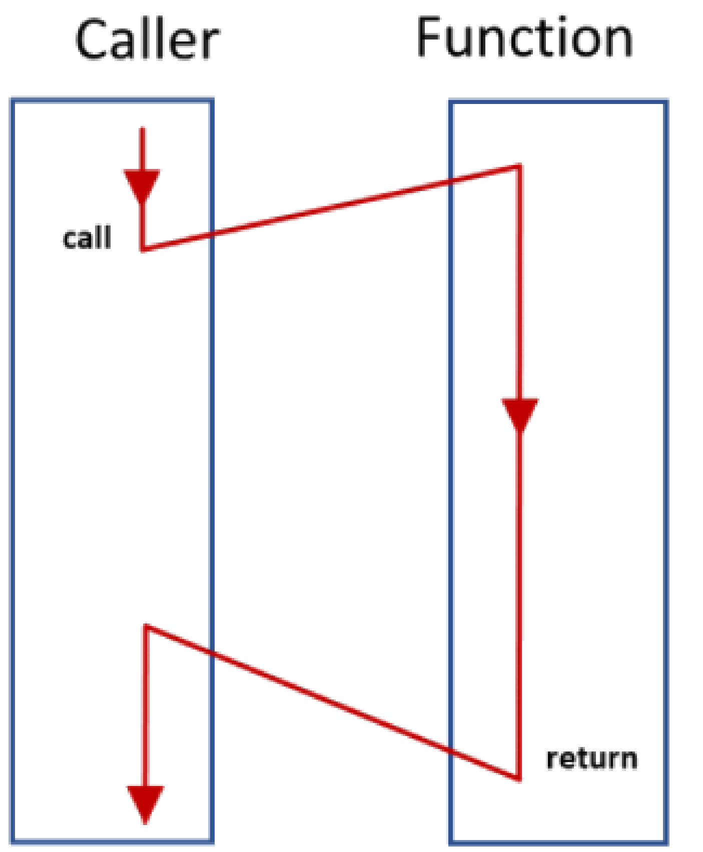
=> Reading a book is a non-blocking task than can be **suspended** then **resumed**: add a bookmark then suspend reading, when ready resume reading from the bookmark

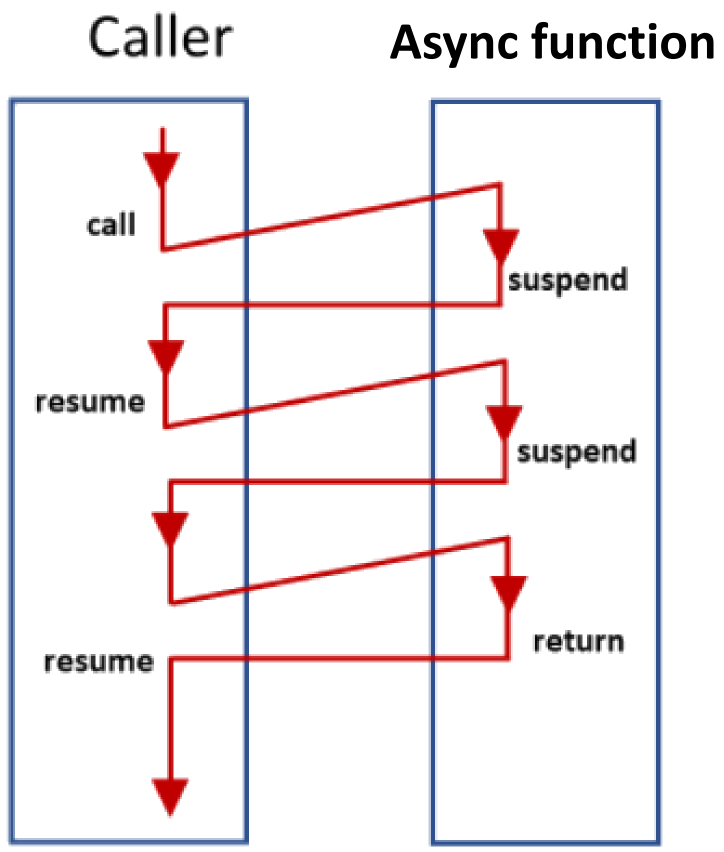
Async function



- **Async** function is a function that can be **suspended** and **resumed**
- When an async function needs to wait for a result it does NOT block instead the runtime:
  - **suspends** the function execution, 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
- While it's suspended waiting for a result, **it unblocks the thread that it's running on**, so that the thread is free to be used for other tasks

Function vs. Async Function



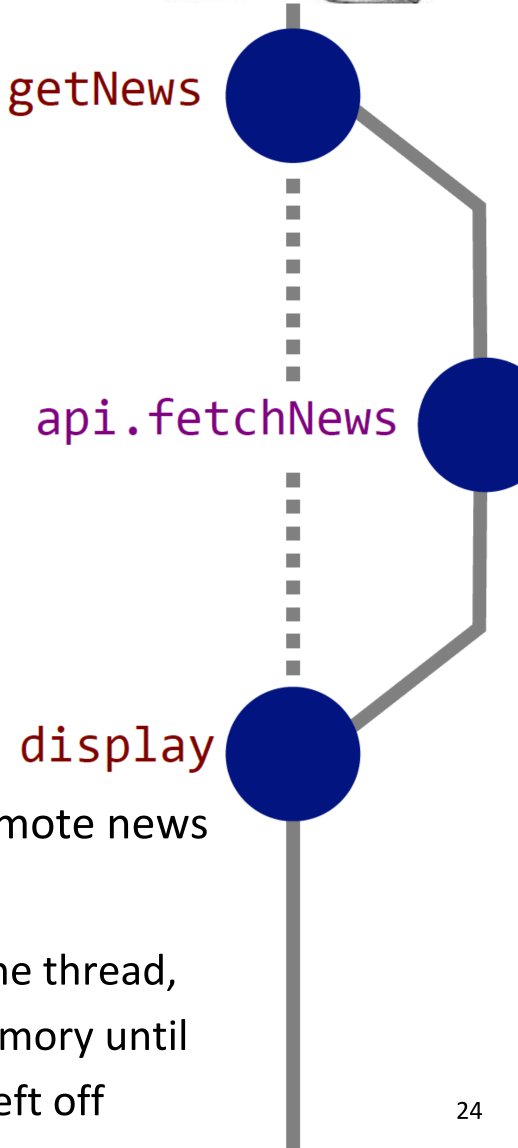


Async function can **suspend** at some points and later **resume** execution when the return value is ready

Async Non-blocking calls with Coroutines

```
val coroutineScope = rememberCoroutineScope()
Button(
    onClick = {
        coroutineScope.launch {
            newsStateVar = getNews()
        }
    }) {
    Text(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

Parallel Execution of Async Functions

- Coroutines can be **executed in parallel** (concurrently) using **Async** or **Launch**
    - Parallelism is about doing lots of things simultaneously
- Async can await for the results (i.e. suspend until results are ready)

```
val deferred1 = async { getStockQuote("Apple") }
val deferred2 = async { getStockQuote("Google") }

val quote1 = deferred1.await()
println(">> ${quote1.name} (${quote1.symbol}) = ${quote1.price}")

val quote2 = deferred2.await()
println(">> ${quote2.name} (${quote2.symbol}) = ${quote2.price}")
```

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

Resources

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

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