Purva Dalvi Follow
Sep 3, 2021 · 6 min read · Listen

Microtasks and Event loop in Dart

Event queue: To manage code execution, Dart uses something called *Event Loop*. It is the queue of actions that need to be performed.

Event loop:

Need of event loop:

- Imagine the life of an app stretched out on a timeline. The app starts, the app stops, and in between a bunch of events occur : I/O from the disk, finger taps from the user… all kinds of stuff.
- Your app can't predict when these events will happen or in what order, and it has to handle all of them with a single thread that never blocks.
- So, the app runs an *event loop*.

Order of execution of events:

- It grabs the oldest event from its event queue, processes it, goes back for the next one, processes it, and so on, until the event queue is empty.
- *The whole time the app is running — you're tapping on the screen, things are downloading, a timer goes off — that event loop is just going around and around, processing those events one at a time.*

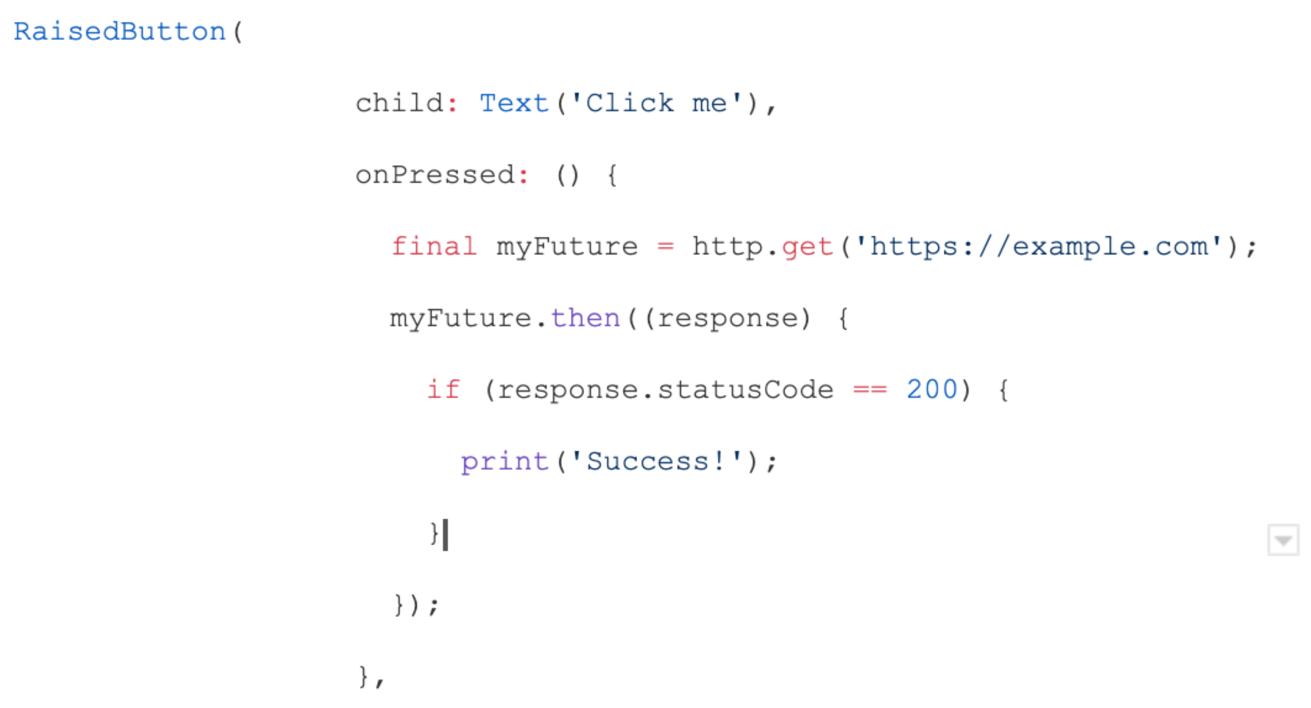
What happens when the event loop has no event:

- When there's a break in the action, the thread just hangs out, waiting for the next event. It can trigger the garbage collector, get some coffee, whatever.

- All of the high-level APIs and language features that Dart has for asynchronous programming — futures, streams, async and await — they're all built on and around this simple loop.

Complete Example:

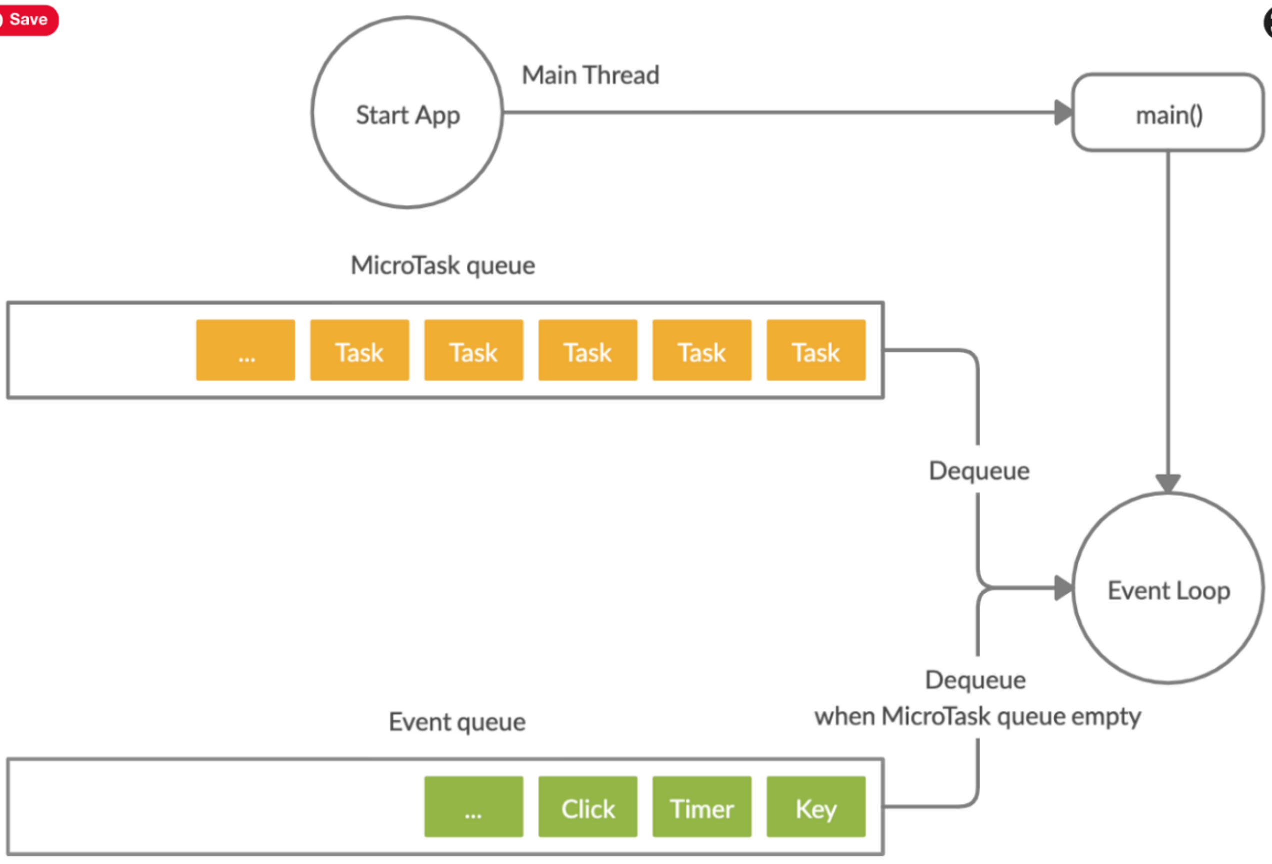
Say you have a button that initiates a network request, like this one:



```
RaisedButton(
                    child: Text('Click me'),
                    onPressed: () {
                      final myFuture = http.get('https://example.com');
                      myFuture.then((response) {
                        if (response.statusCode == 200) {
                          print('Success!');
                        }
                      });
                    },
```

1. When you run your app, Flutter builds the button and puts it on screen.
2. Then your app waits.
3. Your app's event loop just sort of idles, waiting for the next event.
4. Events that aren't related to the button might come in and get handled, while the button sits there waiting for the user to tap on it. Eventually they do, and a tap event enters the queue.
5. That event gets picked up for processing. Flutter looks at it, and the rendering system says, "Those coordinates match the raised button," so Flutter executes the onPressed function. That code initiates a network request (which returns a future)

Flutter order of work:



- First it initializes two queues: *Microtask* Queue and *Event* Queue.
- Then, it executes the *main*() method
- And after main() exits, it creates an *Event Loop*. The Event Loop is responsible for controlling code execution by reading the queues.
- During the whole life of the thread, a single internal and invisible process, called the “Event Loop”, will drive the way your code will be executed and in which sequence order, depending on the content of both MicroTask Queue and Event queues.

To summerise, Dart has :

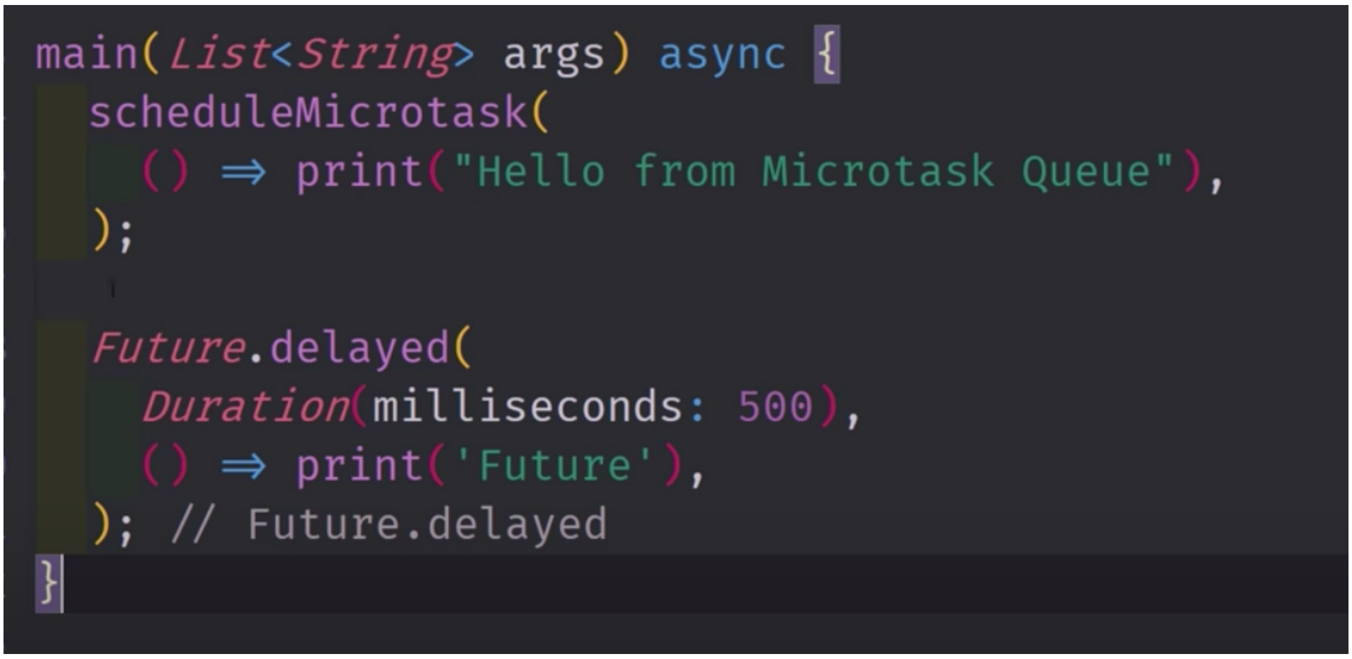
- 2 queues(Event queue, microtask queue)

- Handles outside events. So, side effects like input/output, drawings, timer, messages from isolates and mouse events.
- Only one item in the Event Queue can be executed on every event loop after there is no available *microtask* to run. Because dart is a single threaded language, once it starts executing something it continues to execute until it finishes. In other words, a dart function cannot be interrupted by other dart code
- To add an item to the end of the Event Queue, create a new Future.

Microtask queue:

- Built on top of event queue for different parts that need that specific type of functionality.
- scheduleMicrotask():

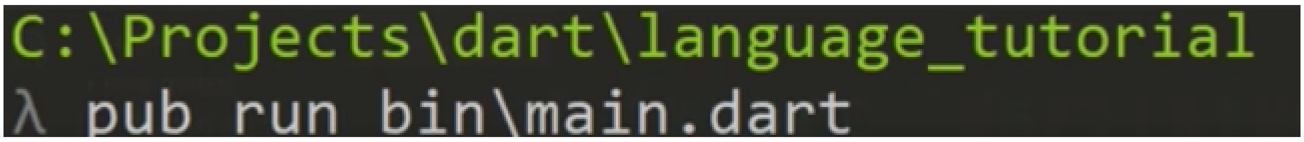
1)



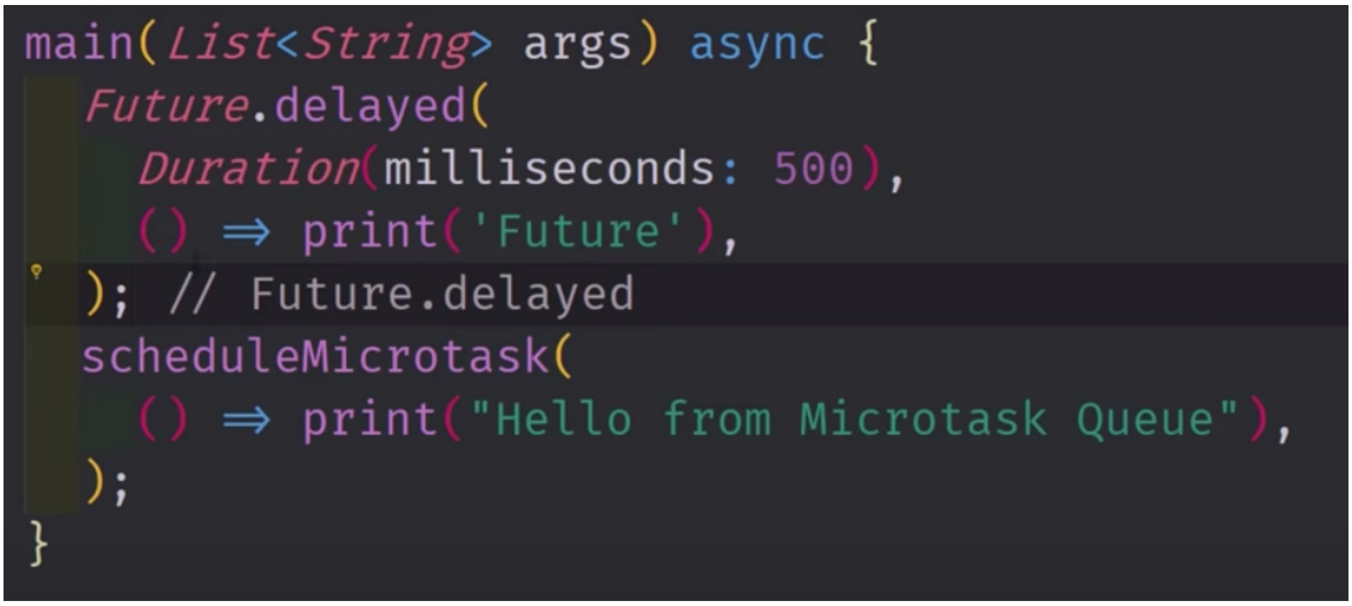
```
main(List<String> args) async {
  scheduleMicrotask(
    () ⇒ print("Hello from Microtask Queue"),
  );

  Future.delayed(
    Duration(milliseconds: 500),
    () ⇒ print('Future'),
  ); // Future.delayed
}
```

Output:

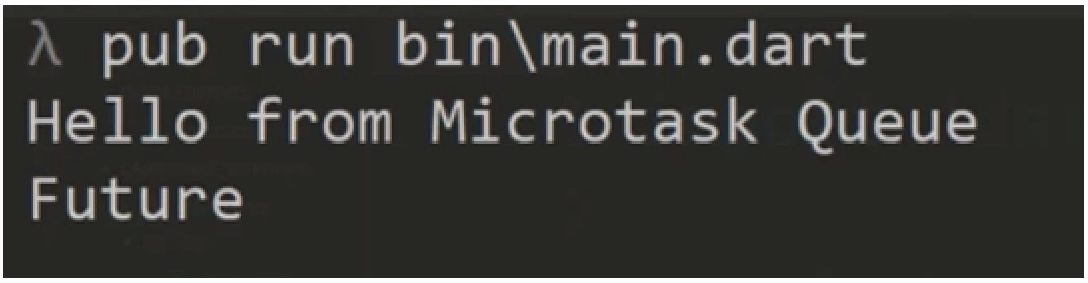


```
C:\Projects\dart\language_tutorial
λ pub run bin\main.dart
```



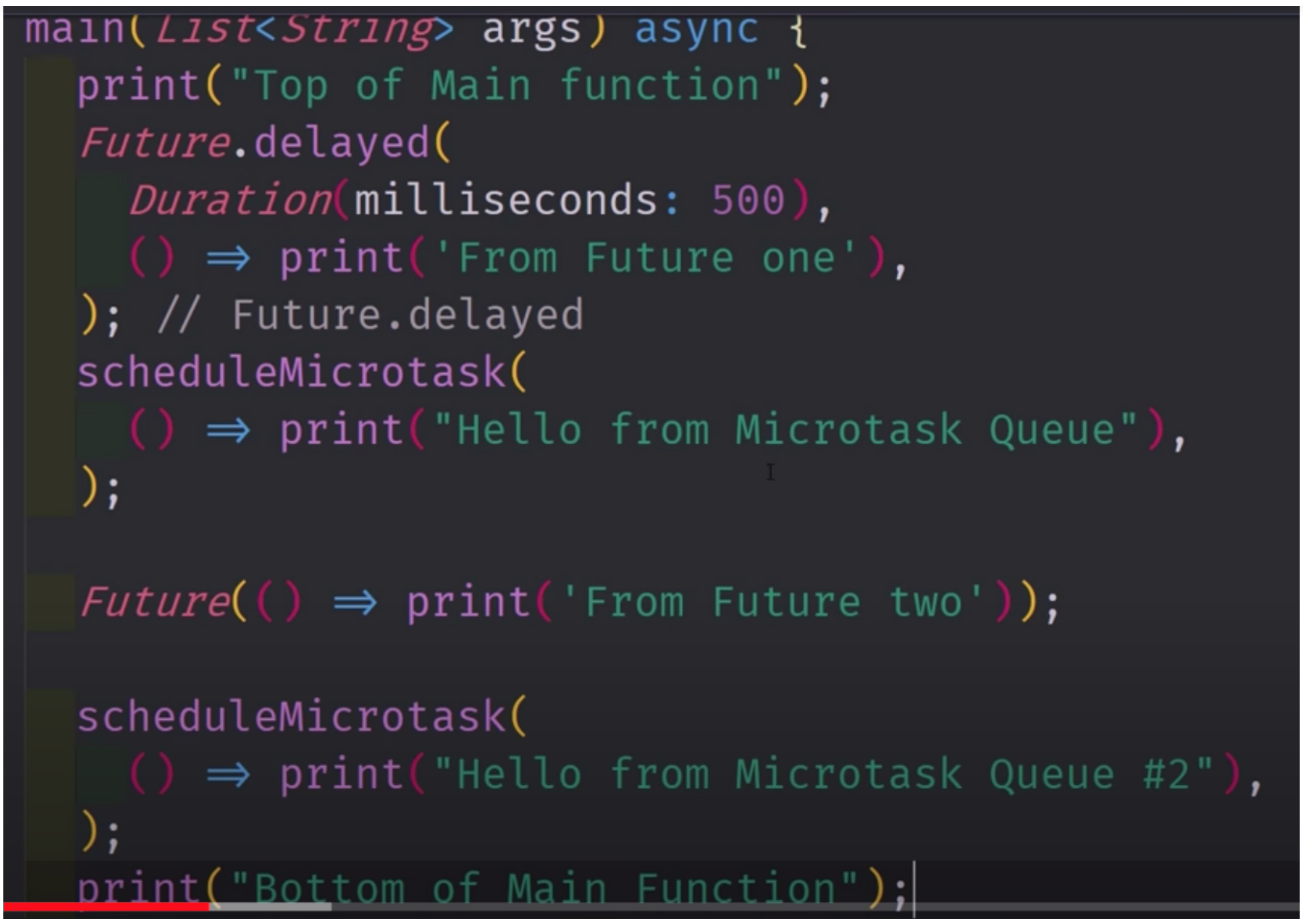
```
main(List<String> args) async {
  Future.delayed(
    Duration(milliseconds: 500),
    () => print('Future'),
  ); // Future.delayed
  scheduleMicrotask(
    () => print("Hello from Microtask Queue"),
  );
}
```

Output:



```
λ pub run bin\main.dart
Hello from Microtask Queue
Future
```

2)

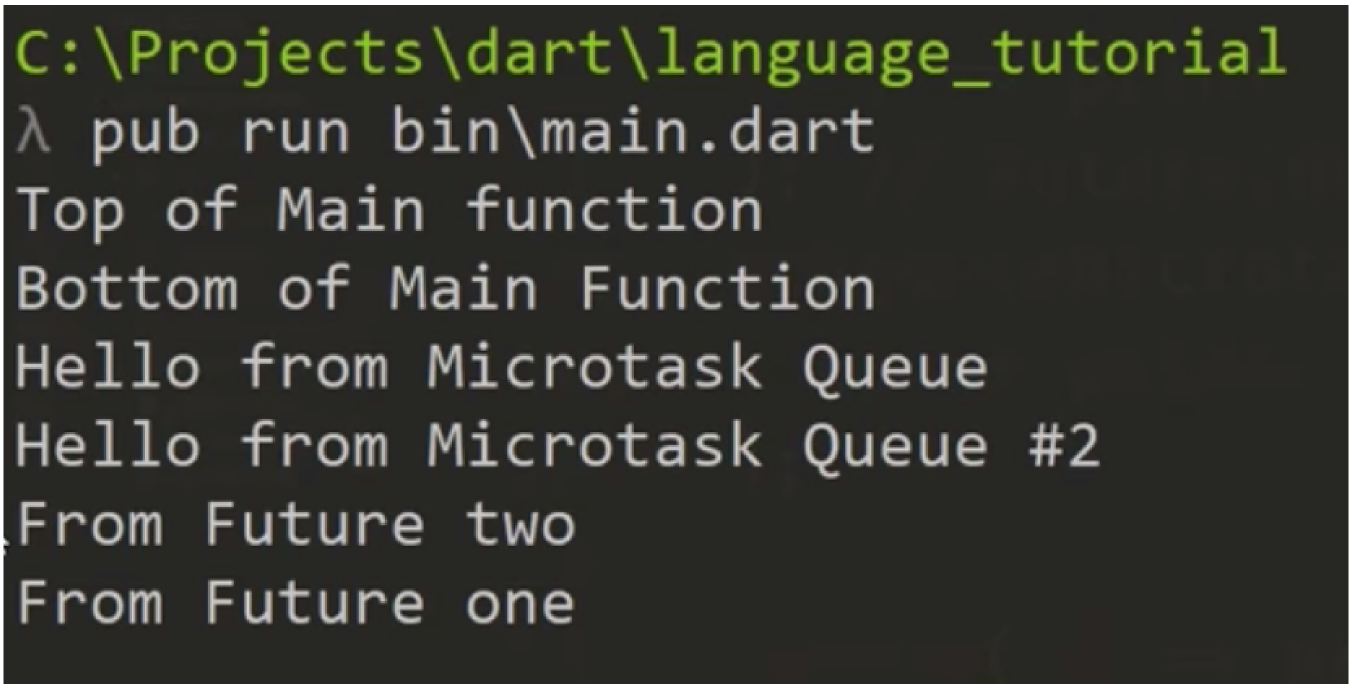


```
main(List<String> args) async {
  print("Top of Main function");
  Future.delayed(
    Duration(milliseconds: 500),
    () => print('From Future one'),
  ); // Future.delayed
  scheduleMicrotask(
    () => print("Hello from Microtask Queue"),
  );

  Future(() => print('From Future two'));

  scheduleMicrotask(
    () => print("Hello from Microtask Queue #2"),
  );
  print("Bottom of Main Function");
```

Output:



```
C:\Projects\dart\language_tutorial
λ pub run bin\main.dart
Top of Main function
Bottom of Main Function
Hello from Microtask Queue
Hello from Microtask Queue #2
From Future two
From Future one
```

2. Tasks from microtask queue

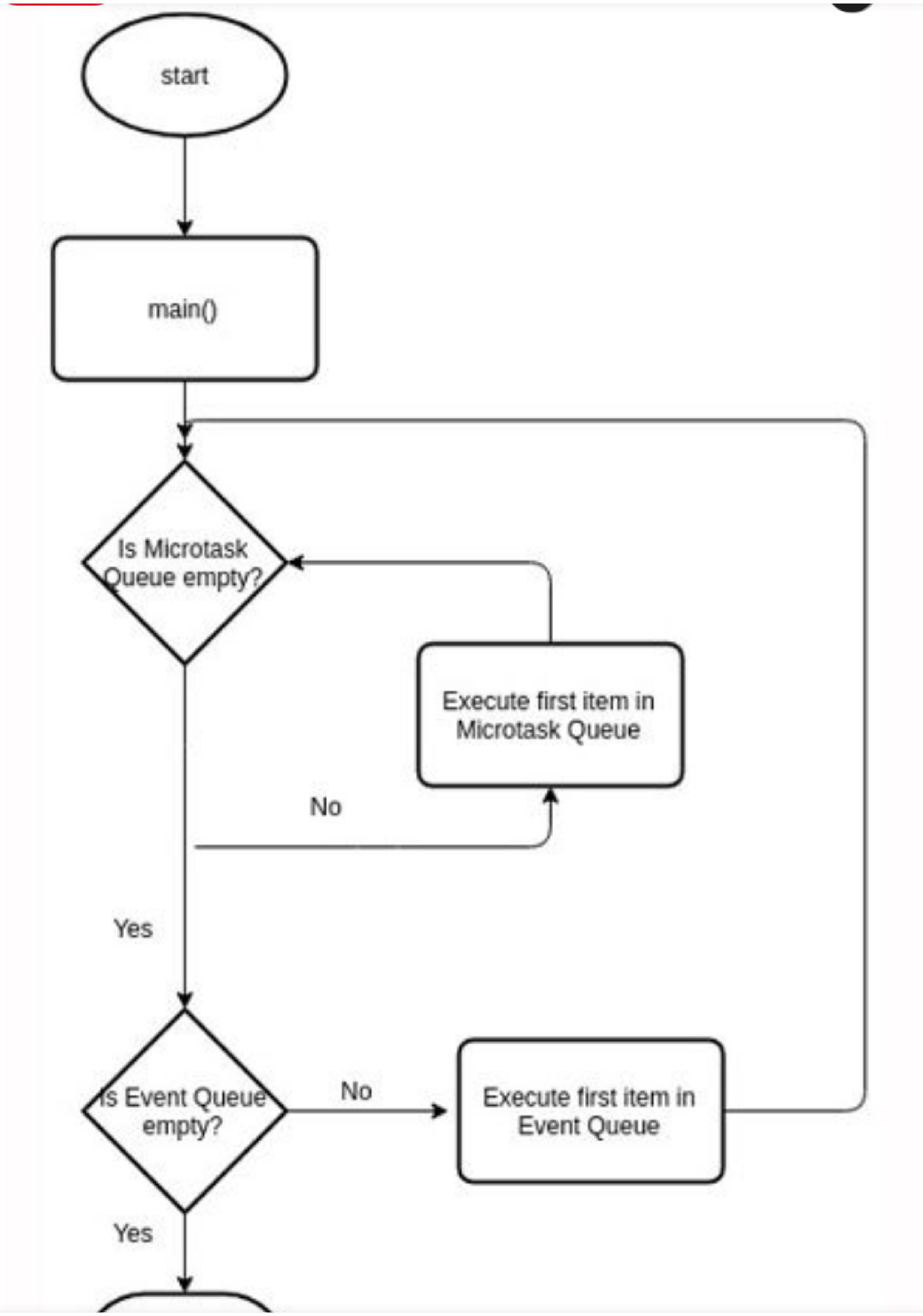
3. Tasks from event queue

Note:

> All the calls in the main() function execute synchronously, start to finish. -
>
> First main() calls print(), then scheduleMicrotask(), then new Future.delayed(), then new Future(), and so on.
>
> Only the callbacks — the code in the closure bodies specified as arguments to scheduleMicrotask(), new Future.delayed(), and new Future() — execute at a later time.

Microtask queue exists as long as it has any entry inside of it. The microtask queue is created the first time a microtask is scheduled and it only exists for as long as a microtask exists

Microtask queue tasks are executed before event queues tasks: The MicroTask Queue is used to store some very short asynchronous internal actions. All of the actions in Microtask Queue will be executed before the Event Queue turn


start
main()
Is Microtask Queue empty?
Execute first item in Microtask Queue
No
Yes
Is Event Queue empty?
No
Execute first item in Event Queue
Yes

- *As you can see on the figure, on every Event Loop, first it fetches all microtasks from Microtask Queue. All Microtasks will be executed and dequeued.*
- *After the Microtask queue is empty, it executes only the first item in the Event Queue. Both Microtask Queue and Event Queue use FIFO (First In First Out) order.*
- *As soon as there is no longer any micro task to run, the Event Loop considers the first item in the Event Queue and will execute it*. It is very interesting to note that Futures are also handled via the Event queue.
- To add an item to the Microtask Queue, use scheduleMicrotask function.
- The **Future.value()** constructor completes in a microtask

Microtask task use cases:

1. Microtask is usually created if we need to complete a task later, but before returning control to the event loop. The microtask queue is necessary because event-handling code sometimes needs to complete a task later, but before returning control to the event loop. For example, when an observable object changes, it groups several mutation changes together and reports them asynchronously. The microtask queue allows the observable object to report these mutation changes before the DOM can show the inconsistent state.
2. To detect that user is no longer authenticated and redirect user to login: Wrap it in Future.microtask. This will schedule it to happen on the next async task cycle (i.e. after build is complete).

```
Future.microtask(() => Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => LoginView())
));
```

3. Suppose that I have a very long loop to execute. It would be nice to keep the user informed about the progress, right? I would print for example the number of loops that have been executed so far and how many are remaining. The problem is that , those

of a handler to the browser to execute tasks and events whenever I want to and keep running the loop.

Answer:

Contributor 1: An approach would be a method that returns after a chunk of work and gets recalled repeatedly in a loop until it returns for example true for done (false for not yet). When you call this method using scheduleMicrotask(doChunk) or new Timer(() => doChunk()) other tasks get some air (import 'dart:async';) each time before the method gets actually called.

Contributor 2: Probably better to use new Future(() { … }) rather than scheduleMicrotask(), as scheduleMicrotask() can also starve the browser's main event loop

Ref:

1. https://web.archive.org/web/20170704074724/https://webdev.dartlang.org/articles/performance/event-loop
2. https://oleksandrkirichenko.com/blog/delayed-code-execution-in-flutter/

Open in app

Get started