What is the difference between micro and macro tasks?

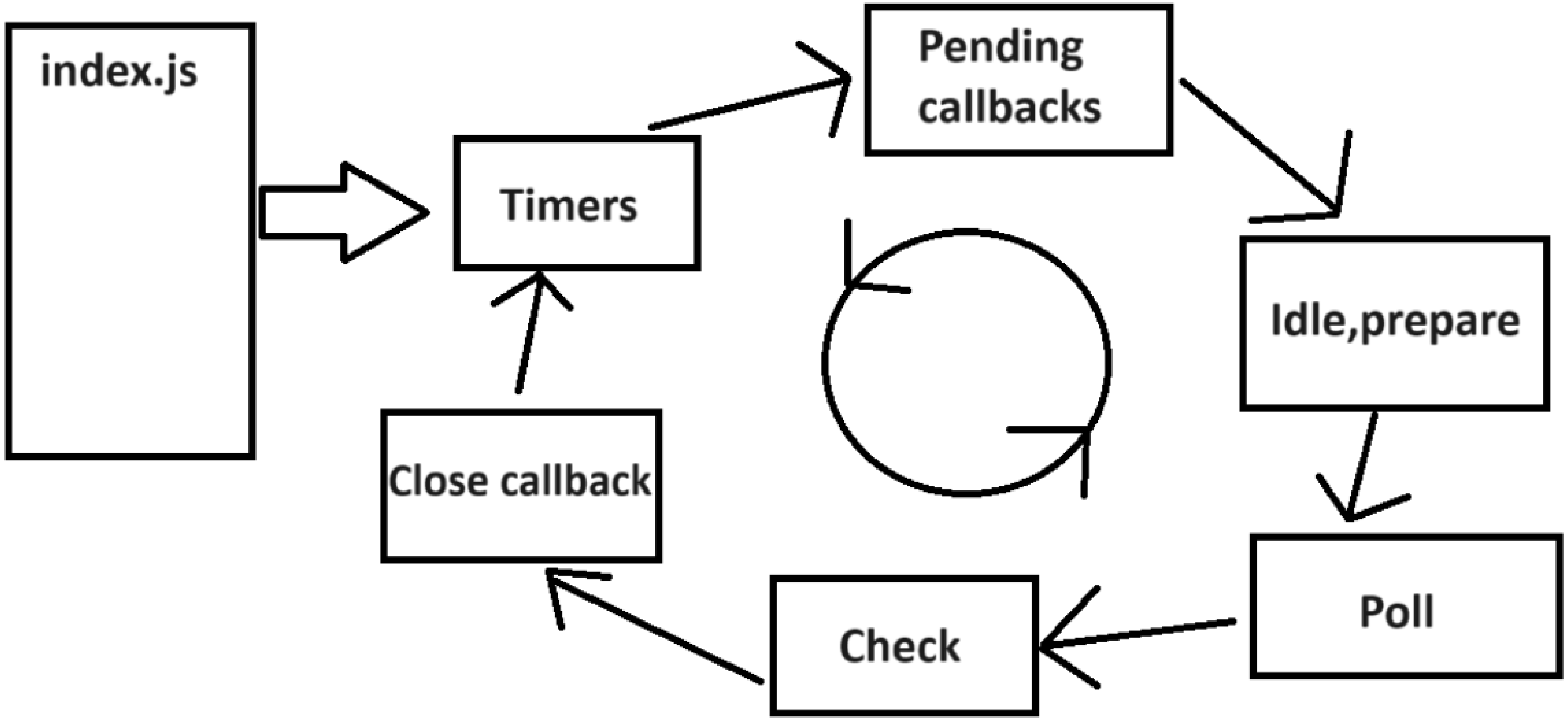
In Node.js, asynchronous tasks are divided into two main categories: microtasks and macrotasks. The distinction between these two is based on their priority and when they are executed in the event loop.

macroTasks: setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering

microTasks: process.nextTick, Promises, queueMicrotask, MutationObserver

Only after tasks in **microTasks** are completed/ exhausted, event loop will next pick up one task from **macroTasks**. And this repeats. As a result, **microtasks** are higher in priority than **macrotasks** so **microtasks** are processed before **macrotasks** in the event loop .So be careful when using macrotasks with microtasks.

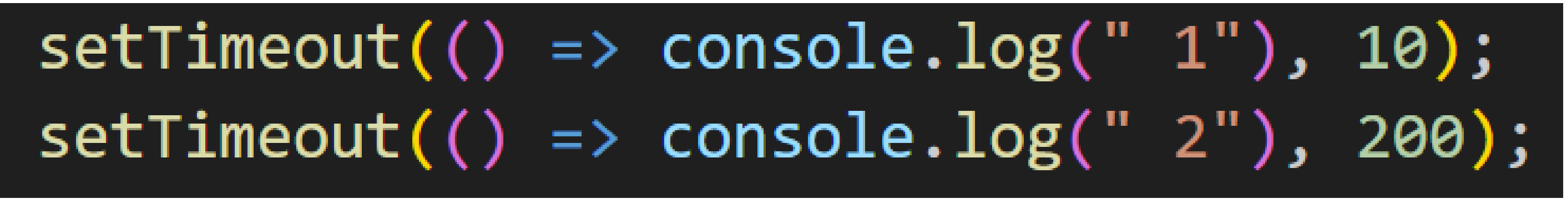
What are the different phases of an event loop?



Six Phases of the event loop(**Note**: There's a callback queue in each phase that stores callbacks to be executed in that phase):

The Event Loop is composed of the following six phases, which are repeated for as long as the application still has code that needs to be executed. These six phases create one cycle, or loop, which is known as a **tick**. A Node.js process exits when there is no more pending work in the Event Loop, or when process.exit() is called manually.

Timers: This is the first phase in the event loop. This phase executes callbacks scheduled by setTimeout() and setInterval().It finds expired timers in every iteration (also known as Tick) and executes the timers callbacks created by **setTimout** and **setInterval**. For example below, **1** is printed first, because the first timer expires in 10 milliseconds before 200 milliseconds, however **2** may be printed in another iteration because the second timer expires after 200 milliseconds.



```
setTimeout(() => console.log(" 1"), 10);
setTimeout(() => console.log(" 2"), 200);
```

Pending callbacks: This phase is responsible for executing I/O-related callbacks deferred to the next loop iteration. This includes callbacks from networking operations, file system operations, and other asynchronous tasks.

Idle, prepare: **Idle** phase is used for executing callbacks that were deferred to the next loop iteration. The **Prepare** phase is related to preparing the system for new events.

Poll: This is the phase where the event loop waits for events to occur. If there are no timers or I/O events, it will block and wait for events to happen. If there are any I/O related **callbacks** in the queue from the previous phases, they will be executed in this phase.

Check: The **setImmediate()** callbacks are executed in this phase. It is typically used to execute callbacks after the poll phase, before the next cycle of the event loop.

Close Callbacks: Callbacks registered via close events, such as those in socket.on('close', ...) or server.on('close', ...), are executed in this phase. This is the final phase in the event loop cycle.

What is the output of 1.js file?
What is the output of the code in 2.js file in the folder called 2?

When to use setImmediate over setTimeout?

The main advantage to using setImmediate() over setTimeout() is setImmediate() will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present. To understand this better answer this question:

What is the output in the 3.js file ?

What is the output in the 4.js file?

What is process.nextTick()?

Every time the event loop takes a full trip, we call it a tick. When we pass a function to **process.nextTick()** , we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts. **process.nextTick()** is not technically part of the event loop. It does not care about the phases of the event loop. So all callbacks passed to process.nextTick() will be resolved before the event loop continues.

How does the microtask queue work?

The microtask queue stores microtasks callbacks created by Promises **then() and catch()** that handle resolved and rejected promises and **process.nextTick** callbacks and **queueMicrotask** callbacks.

process.nextTick() callbacks will always be executed before the **Promise** callbacks**.**

What is the output of 5.js in the folder called 5?

What is the output of 6.js?

What is the output of 7.js?

What is the output of 8.js?

use **setImmediate**() in all cases instead of **process.nextTick** because it's easier to reason about.

Can nextTick be dangerous?

Yes, if we recursively run process.nextTick(), the event loop will never reach the timers queue, and the corresponding callbacks in the timers queue will never be executed or any other phase of the event loop.

What is the output of the code in 9.js?

What is the output of the code in 10.js?

What is the difference between process.nextTick(), setTimeout and setImmediate?

process.nextTick() will always execute before **setTimeout** and **setImmediate**.

Do promises run on a separate thread?

No. The .then, .catch, and .finally callbacks are added to the microtask queue.