

# Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

# Event loop

Concurrency model

Shadi Lahham - Web development

Stack, queue & heap

# Queue

## First in first out - FIFO

### Queue

Items are used or processed in the order they arrive



# FIFO

## First In, First Out

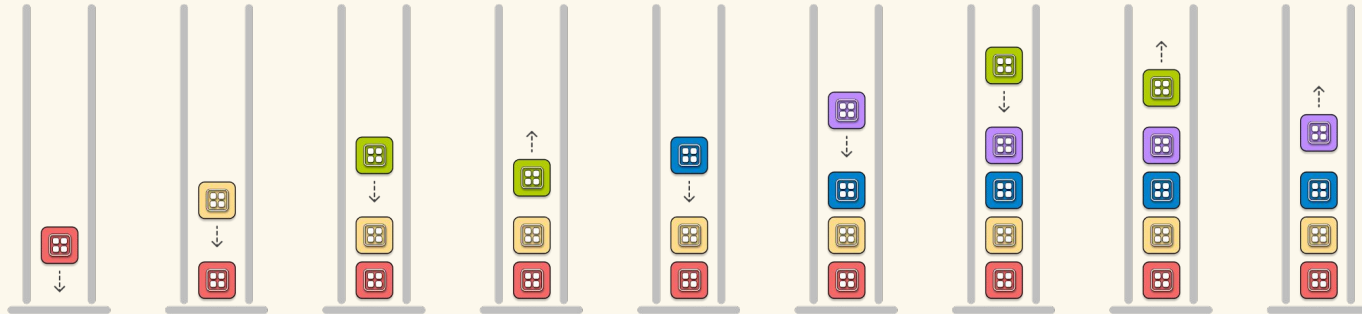
- a method for managing inventory or data
- items or data that enter first are used or processed first
- commonly used in manufacturing, retail, and computing
- prevents newer items from being used before older ones

# Stack

## Last In, First Out - LIFO

### Stack

Items are added to and removed from the top of the stack



# LIFO

## Last In, First Out

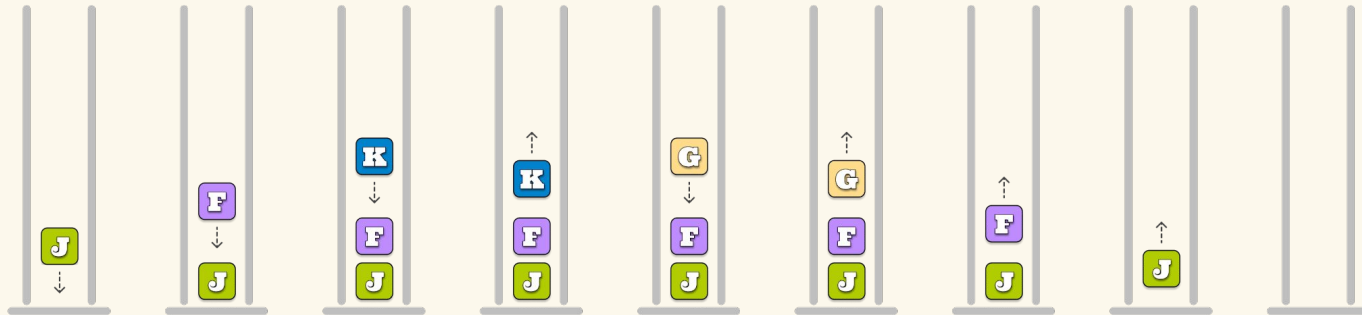
- a method for managing inventory or data
- newest items are used or processed first
- commonly used in accounting, storage systems, and computing
- ensures older items are used after newer ones

# Javascript call stack

## Call stack

### JavaScript call stack

Manages function calls enabling  
execution order and tracking of active functions





# JavaScript call stack

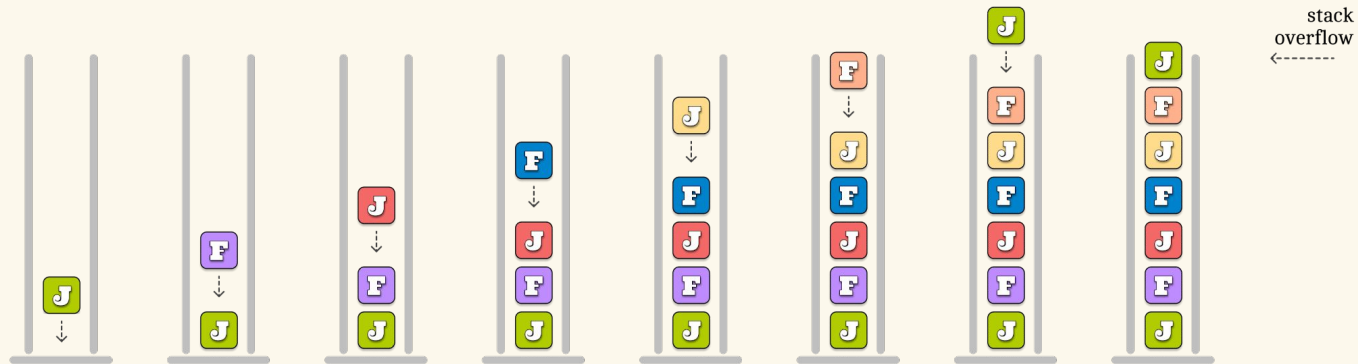
- manages function calls
- follows the Last In, First Out principle
- when a function is called, it's added to the top of the stack
- when a function returns, it's removed from the stack
- the stack helps track the order of function execution

# Stack overflow

## Call stack

### Circular dependency

Functions continuously invoke each other without reaching an endpoint potentially resulting in a stack overflow error



# Stack overflow

- occurs when the call stack exceeds its maximum size
- can happen due to infinite recursion or deeply nested function calls
- results in a runtime error and can crash the program
- can be prevented by optimizing code and avoiding
  - infinite recursion
  - circular dependency
  - deeply nested function calls

# Recursion stack overflow limits

- Google Chrome: Around 10,000 to 17,000 function calls
- Mozilla Firefox: Around 10,000 to 20,000 function calls
- Safari: Around 10,000 to 15,000 function calls
- Microsoft Edge: Around 10,000 to 20,000 function calls

## **note**

numbers valid for browser versions in 2022

# The heap

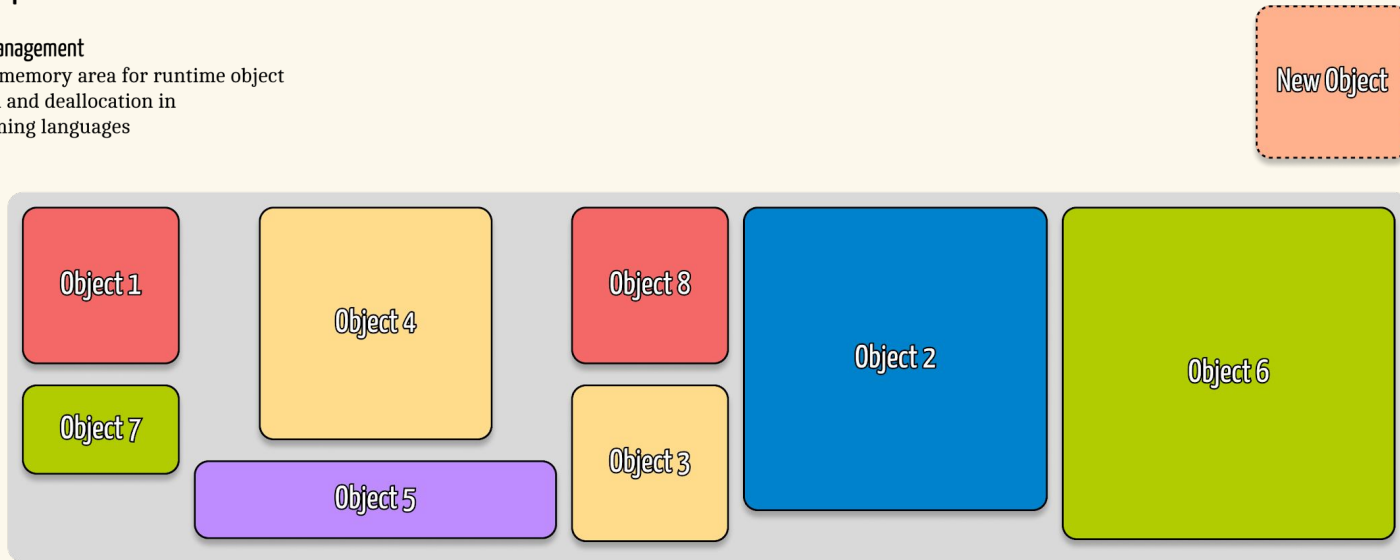
- region of memory for runtime object allocation
- managed by the memory management system
- objects stored on the heap
- memory allocation and deallocation handled by the garbage collector
- enables dynamic memory management in managed languages
- different from the call stack used for function call management

# The heap

## The heap

### Memory management

Dynamic memory area for runtime object allocation and deallocation in programming languages

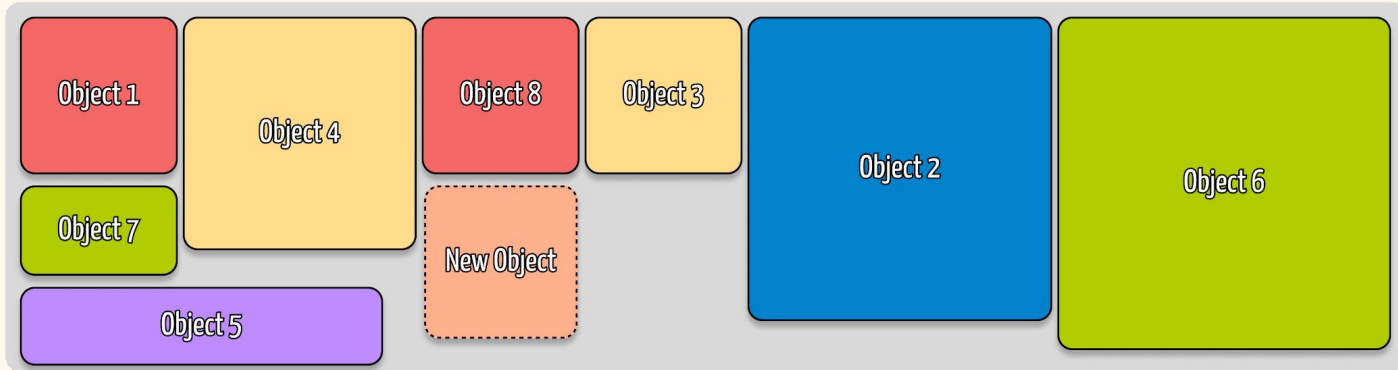


# The heap

## The heap

### Heap fragmentation

Dispersed memory blocks hinder allocation despite sufficient total space availability



# Heap management

Options when there's no contiguous space on the heap

- trigger garbage collection to free up memory
- expand heap by requesting additional memory from the operating system
- move already existing allocated memory to create contiguous space



# Stack vs heap

## Stack

- structured memory space for function calls and local variables
- fixed size determined by the operating system
- faster access due to its LIFO structure
- managed automatically by the compiler or runtime system
- suitable for managing predictable lifetimes of variables and function calls

# Stack vs heap

## Heap

- memory space for dynamic memory allocation and storage of objects
- dynamically grows and shrinks during program execution
- access speed might be slower due to dynamic allocation
- requires explicit allocation and deallocation
- suitable for dynamic size requirements and longer-lived objects

codice asincrono --> il codice viene eseguito in momenti differenti da diversi fattori setTimeout(), e non per forza in ordine di riga di codice

codice sincrono --> SONO FUNZIONI NORMALI SENZA TIMEOUT, VERRANNO ESEGUITE PRIMA DI ESSE, FINO A QUANDO NON VERRANNO ESEGUITE (POTREBBERO RIMANERE BLOCCATE), NON VERRANNO ESEGUITI IL CODICE ASINCRONO

# Event loop

# Single and multi-threaded

## Single-threaded

- a process that executes tasks sequentially
- can only perform one operation at a time
- limited in utilizing system resources efficiently

## Multithreaded

- a process can execute multiple tasks concurrently
- utilizes multiple threads for parallel execution
- allows for efficient resource utilization
- potential for faster execution of tasks

# Event loop

The event loop enables JavaScript to handle asynchronous operations and provide the illusion of being multithreaded despite being single-threaded

- being single-threaded, JavaScript can only perform one operation at a time
- examples of asynchronous operations that are handled by the event loop
  - DOM events for user input
  - fetching data with `fetch()`
  - promises, and `async/await`
  - timer events such as `setTimeout()` and `setInterval()`
- with asynchronous code, statements are not necessarily executed in the order they appear

# Event loop and timer events

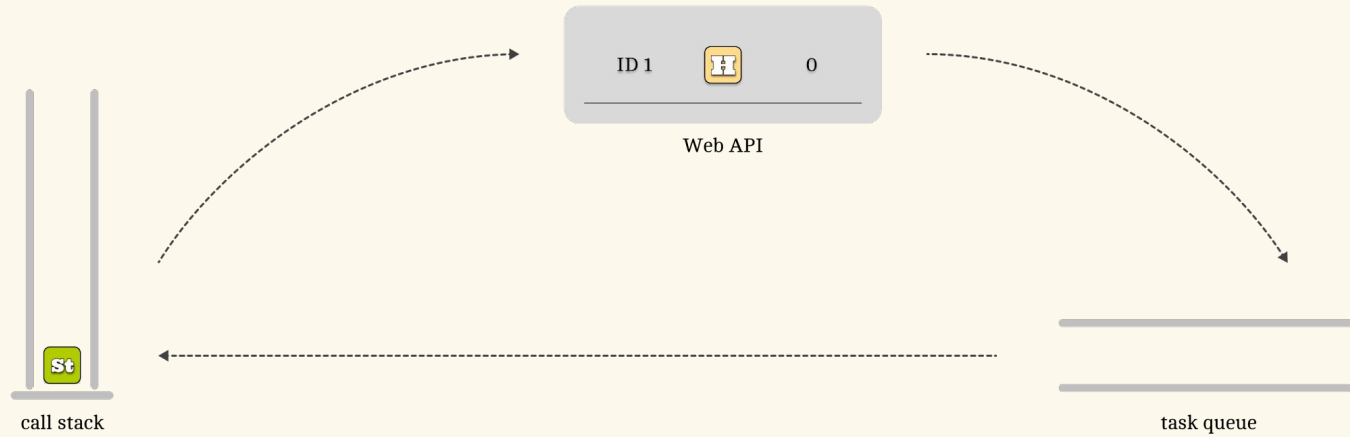
- timer events are triggered by `setTimeout()` and `setInterval()`, which schedule code execution after a specified time
- execution of Javascript continues with other tasks instead of waiting while the browser Web API keeps track of the timer and the callback function
- when a timer expires, the callback function is added to the "task queue"
- the event loop constantly monitors the queue
- when the main thread is idle and the call stack empty, the event loop gets tasks from the queue and runs their callback functions

# Timer events example

```
// define functions  
let f = () => console.log('first');  
let h = () => console.log('hi');  
let b = () => console.log('bye');  
  
// schedule calls  
setTimeout(h, 0);  
setTimeout(b, 400);  
setTimeout(b, 1000);  
  
// main  
f();
```

# Timer events example

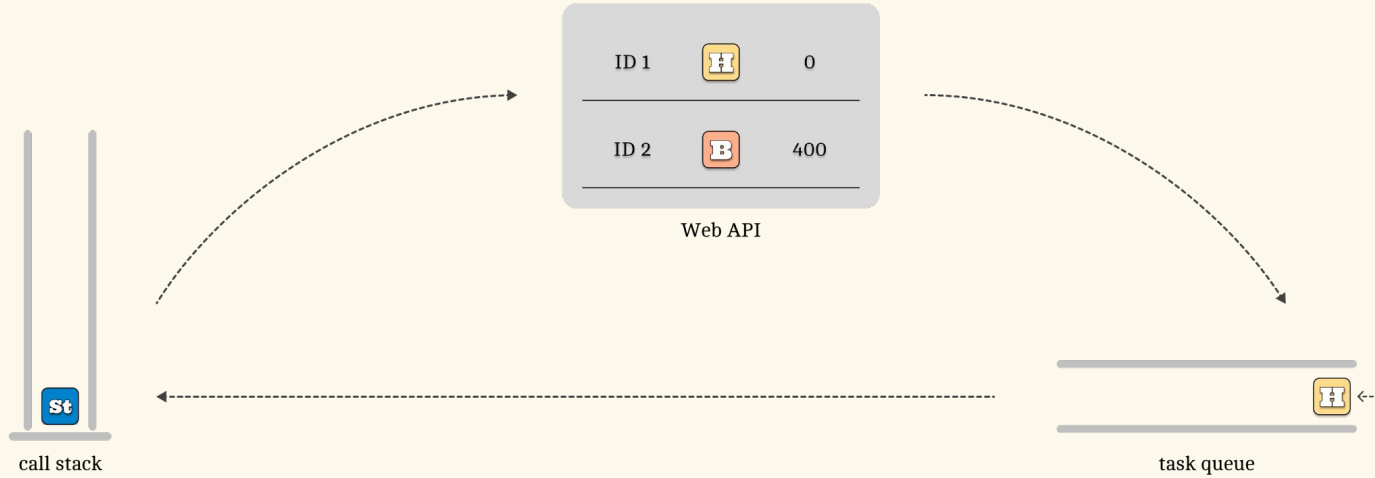
## Event loop





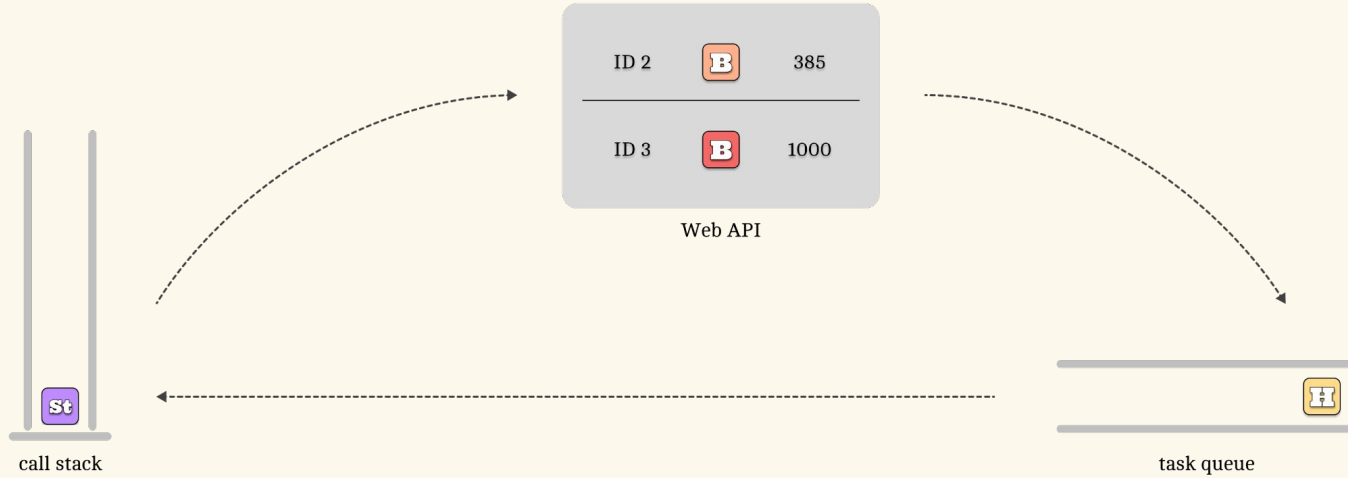
# Timer events example

## Event loop



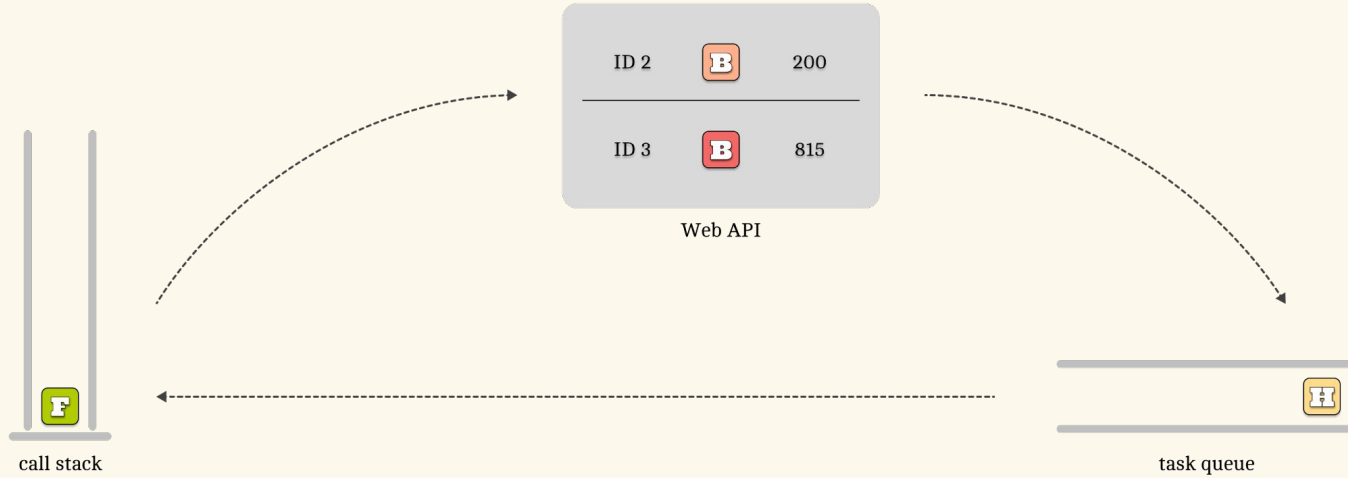
# Timer events example

## Event loop



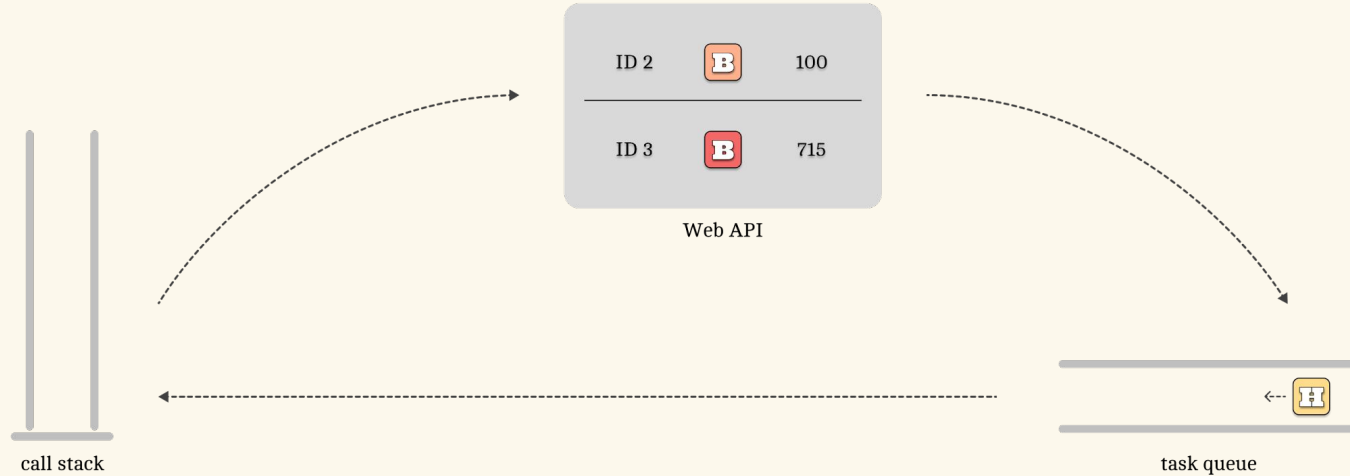
# Timer events example

## Event loop



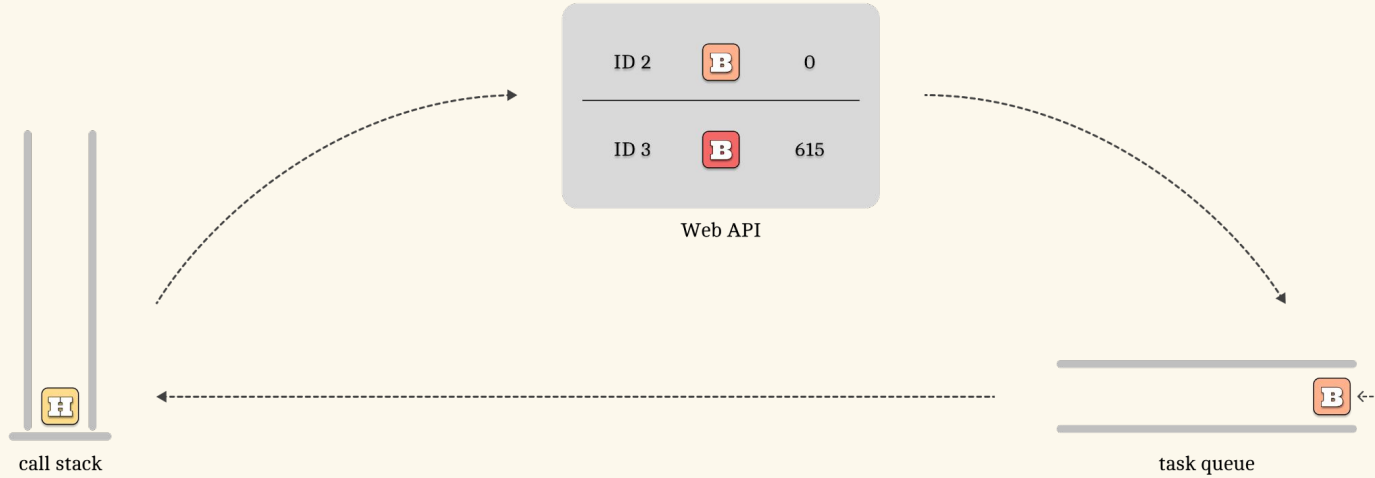
# Timer events example

## Event loop



# Timer events example

## Event loop



# setTimeout minimum delay

setTimeout guarantees a minimum delay before executing the callback function, the actual delay may be longer due to factors such as the presence of other tasks in the task queue, the workload on the main thread, and system performance. This means that setTimeout does not provide an exact delay, but rather a minimum delay

# Long function execution example

```
// function intentionally wastes time
let f = () => {
  let count = 0;
  // waste time
  while (count < 1e10) {
    count++;
  }
  console.log('first');
};
let h = () => console.log('hi');
let b = () => console.log('bye');

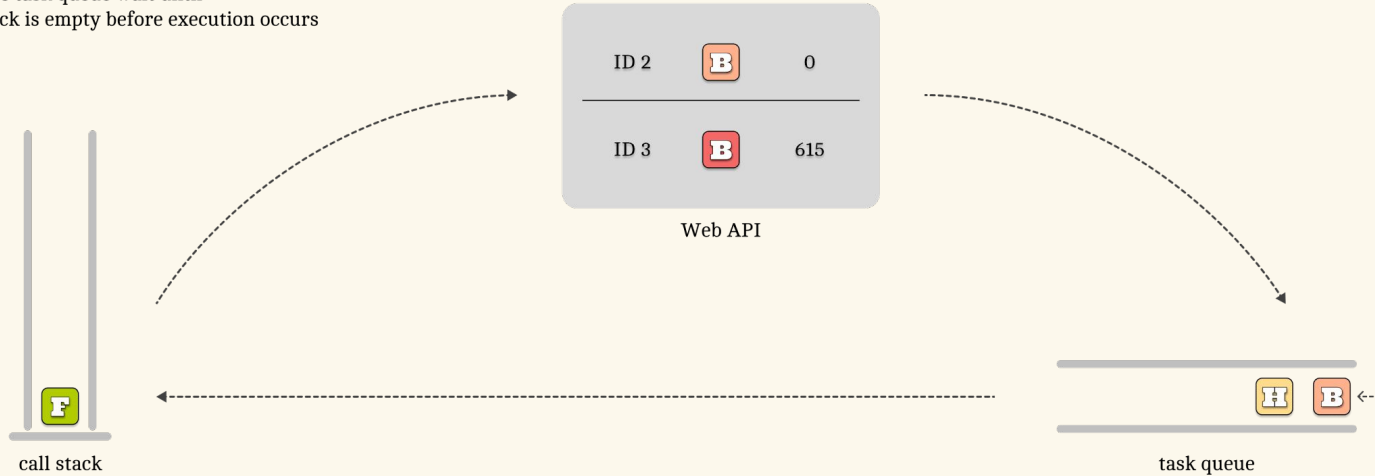
// schedule functions and invoke f
setTimeout(h, 0);
setTimeout(b, 400);
setTimeout(b, 1000);
f();
```

# Long function execution example

## Event loop

Call stack not empty

Tasks in the task queue wait until  
the call stack is empty before execution occurs



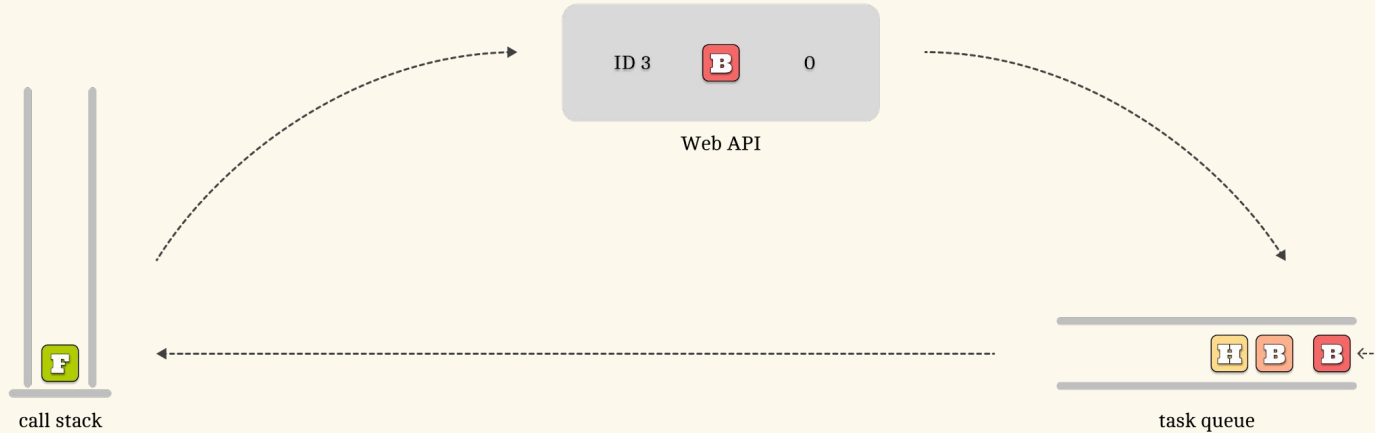


# Long function execution example

## Event loop

Call stack not empty

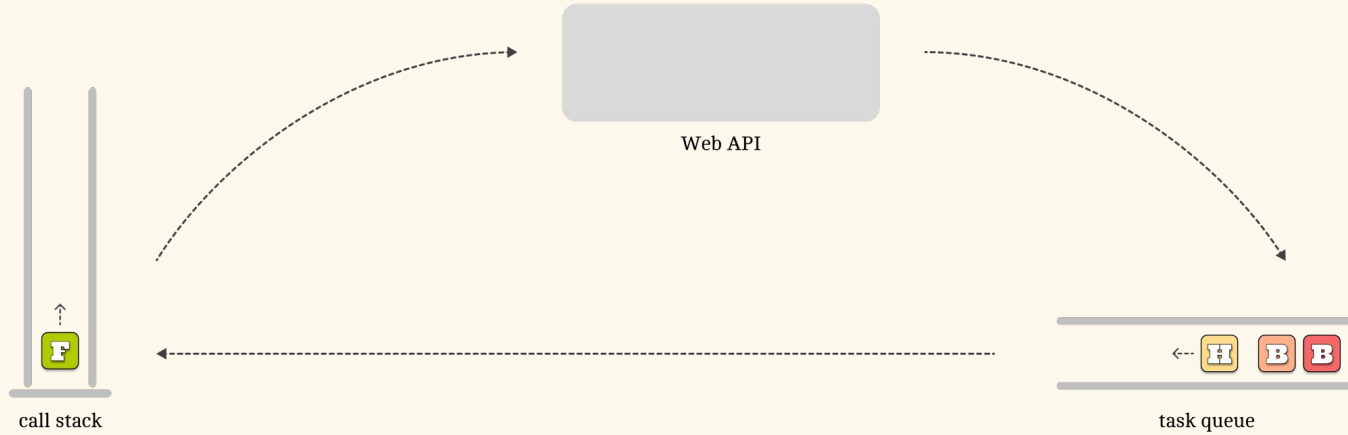
Function F takes a long time to execute



# Long function execution example

## Event loop

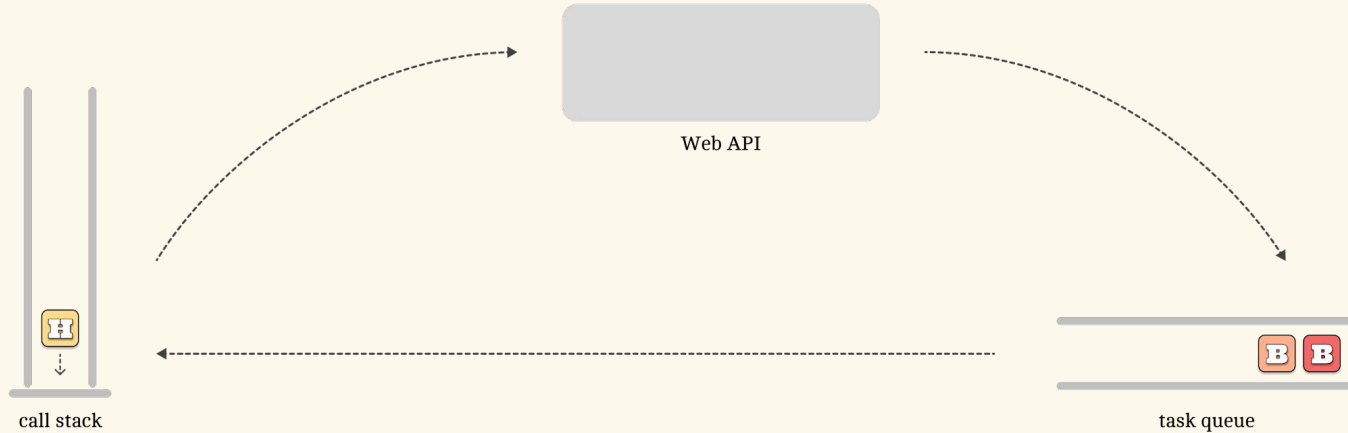
Call stack empty  
Function F terminates



# Long function execution example

## Event loop

Task queue processed  
Function H executes



# References

[Call stack - MDN](#)

[The event loop - MDN](#)

[Web API, Task Queue and Event Loop](#)

[Understanding event loop in JavaScript](#)

[Javascript Event Loop Explained](#)