

Concurrency Model and Event Loop



Samer Buna

@samerbuna www.jscomplete.com

Event Model

Nodejs

libuv

Ruby

Event Machine

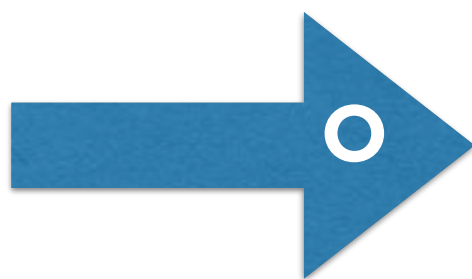
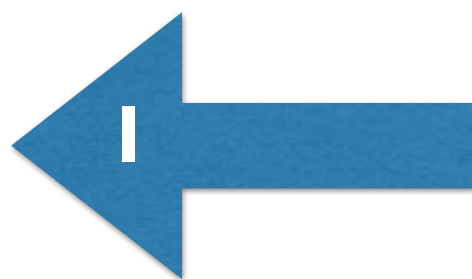
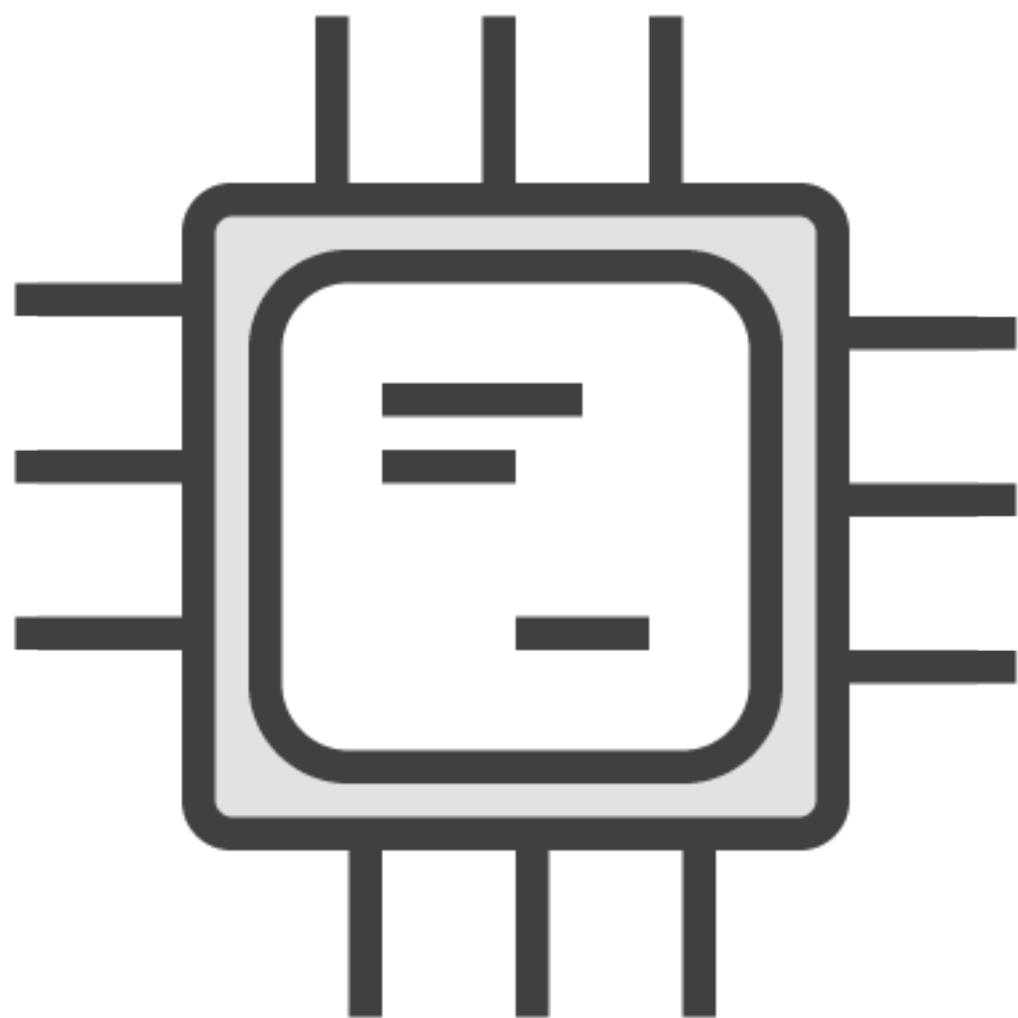
Python

Twisted

The Event Loop

Slow I/O operations

What Is I/O Anyway?



Handling Slow I/O

Synchronous

fork()

Threads

Event Loop

The Event Loop

The Event Loop

The entity that handles external events and converts them into callback invocations

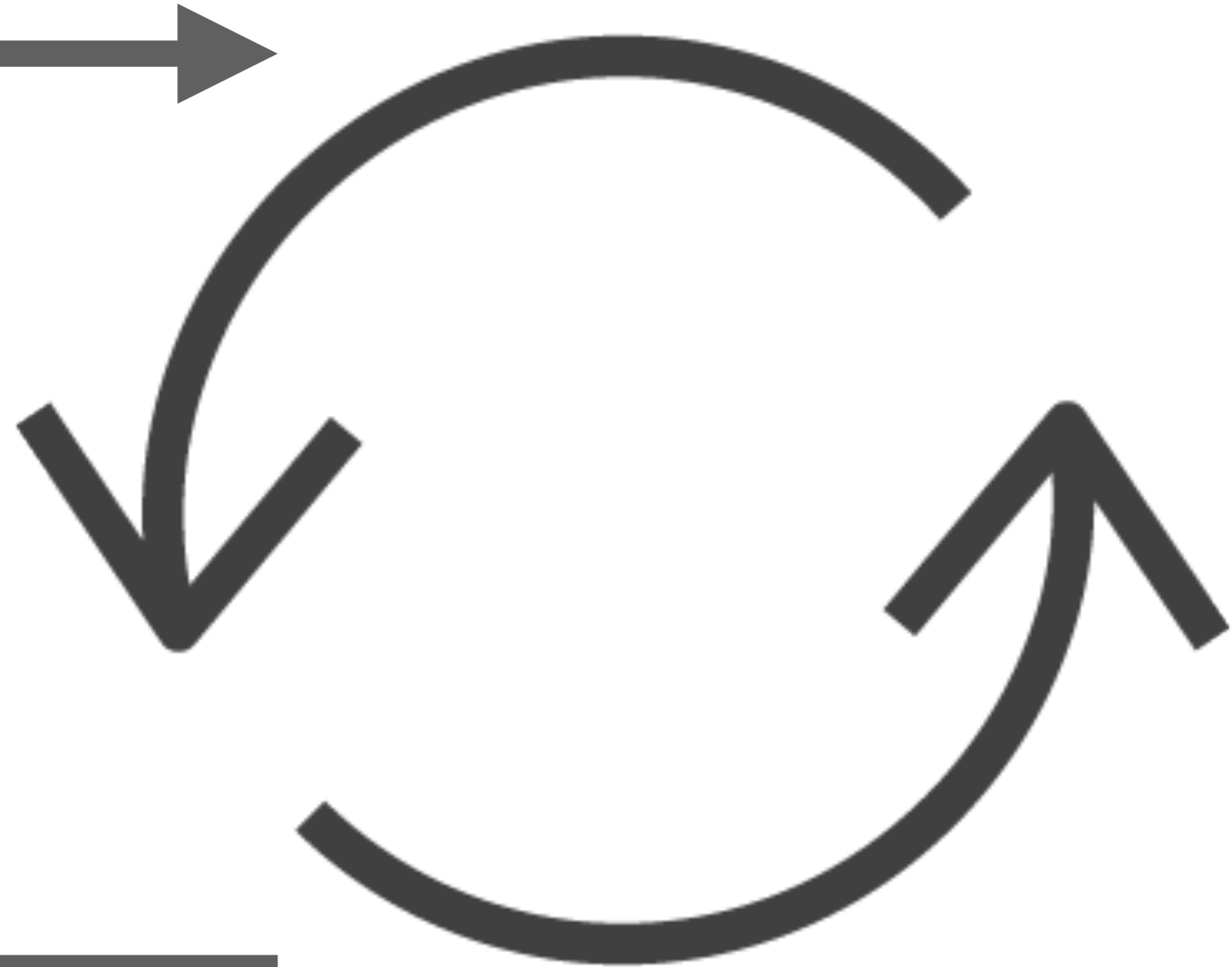


The Event Loop

A loop that picks events from the event queue and pushes their callbacks to the call stack

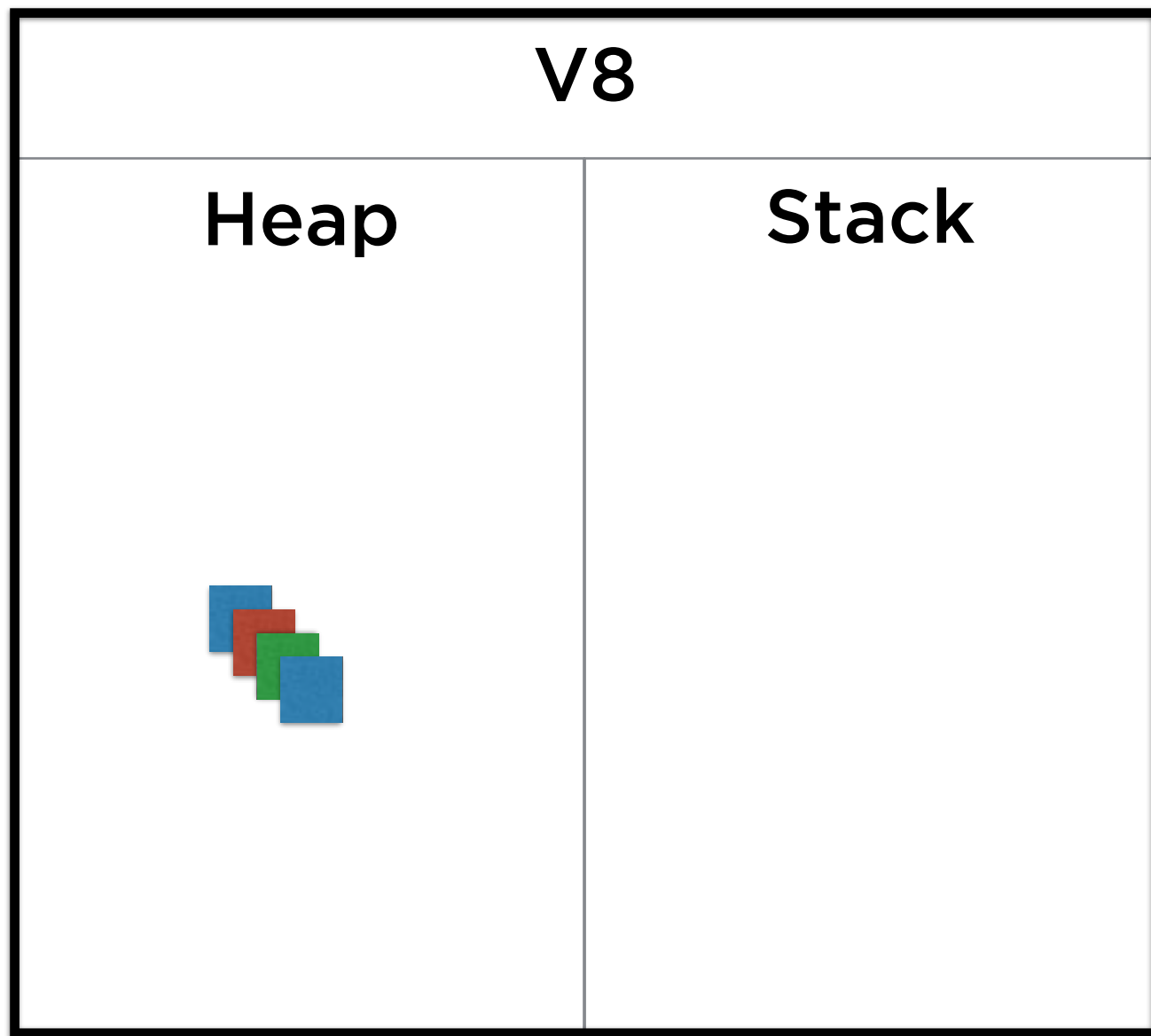


npm start



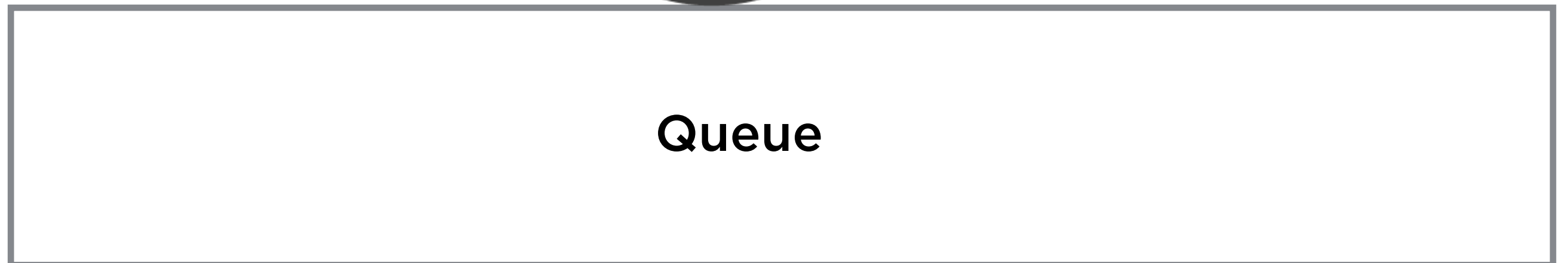
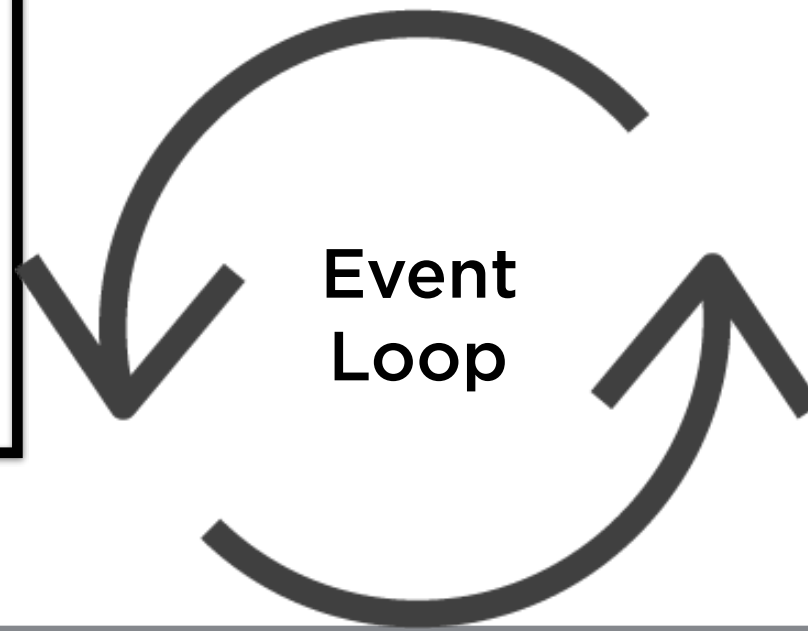
process.exit()





Node API

```
setTimeout()  
fs.readFile()  
emitter.on()  
....
```



The Call Stack



```
const f1 = () => { f2(); };
```

```
const f2 = () => { f3(); };
```

```
const f3 = () => { f4(); };
```

```
const f4 = () => { f4(); };
```

Call Stack

f4()

f4()

f3()

f2()

f1()

```
const add = (a, b) => a + b;
```

```
const double = a =>  
  add(a, a);
```

```
const printDouble = a => {  
  const output = double(a);  
  console.log(output);  
};
```

```
printDouble(9);
```

Call Stack

add(9,9)

double(9) (18)

printDouble(9)

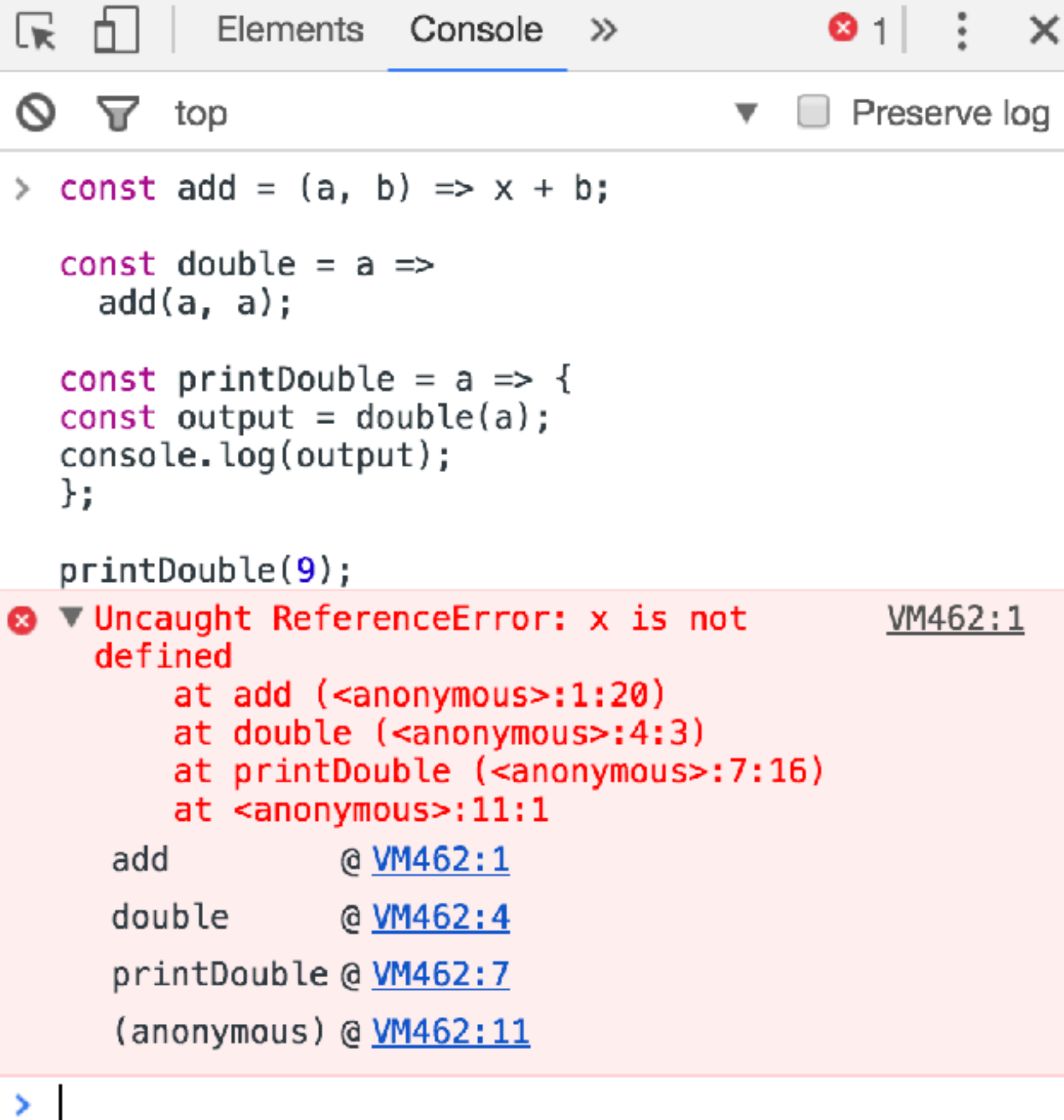
anonymous()

```
const add = (a, b) => x + b;
```

```
const double = a =>  
  add(a, a);
```

```
const printDouble = a => {  
  const output = double(a);  
  console.log(output);  
};
```

```
printDouble(9);
```



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following JavaScript code:

```
> const add = (a, b) => x + b;  
  
const double = a =>  
  add(a, a);  
  
const printDouble = a => {  
  const output = double(a);  
  console.log(output);  
};  
  
printDouble(9);
```

Below the code, an error message is displayed in red text:

```
✖ Uncaught ReferenceError: x is not defined VM462:1  
    at add (<anonymous>:1:20)  
    at double (<anonymous>:4:3)  
    at printDouble (<anonymous>:7:16)  
    at <anonymous>:11:1  
add @ VM462:1  
double @ VM462:4  
printDouble @ VM462:7  
(anonymous) @ VM462:11
```

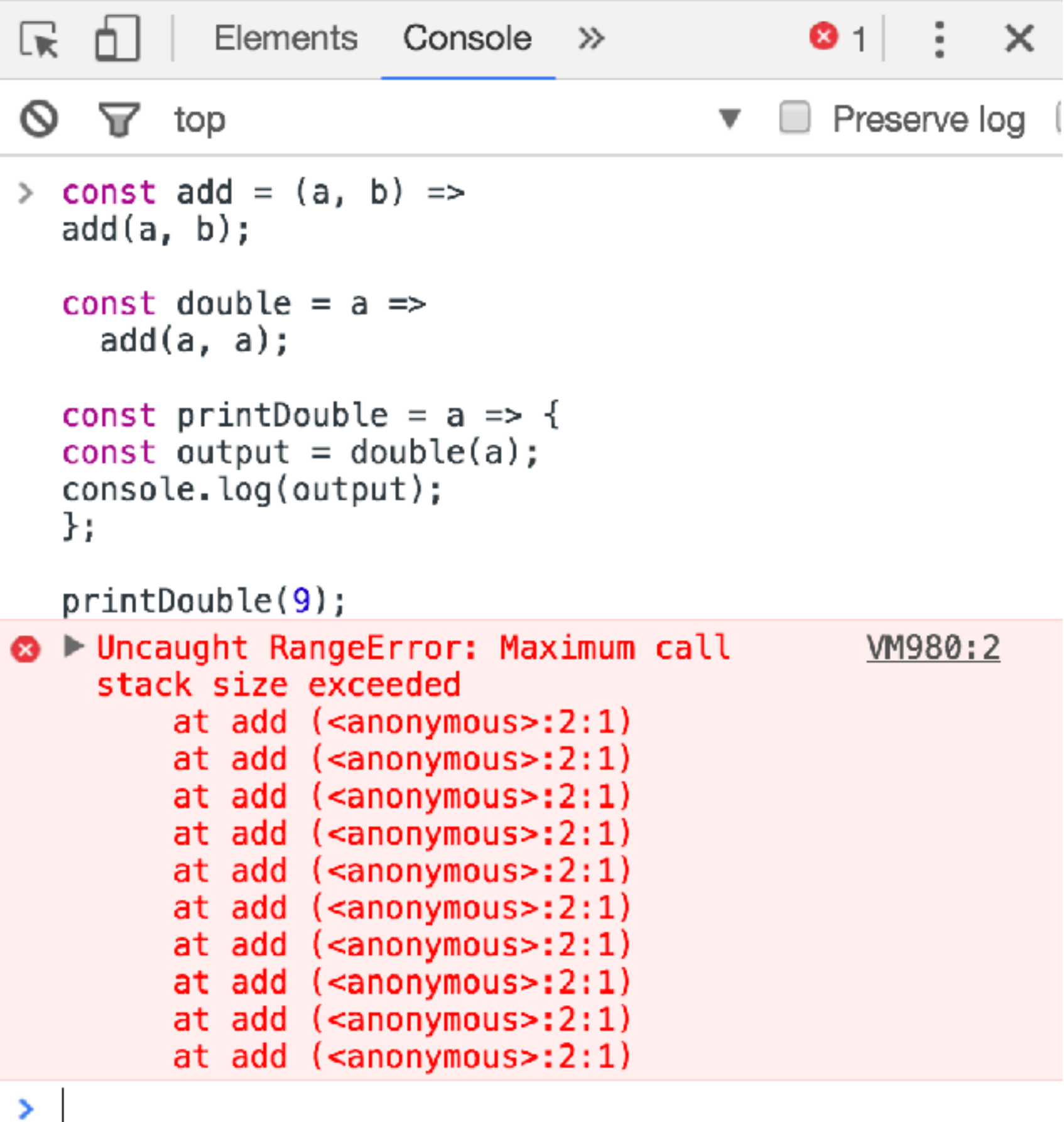
The error message indicates that the variable `x` is not defined when the `add` function is executed. The stack trace shows the error originates from the `add` function at line 1, column 20, and propagates through the `double` function, the `printDouble` function, and finally to the anonymous function at line 11, column 1.


```
const add = (a, b) =>
  add(a, b);
```

```
const double = a =>
  add(a, a);
```

```
const printDouble = a => {
  const output = double(a);
  console.log(output);
};
```

```
printDouble(9);
```



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following JavaScript code:

```
> const add = (a, b) =>
  add(a, b);

const double = a =>
  add(a, a);

const printDouble = a => {
  const output = double(a);
  console.log(output);
};

printDouble(9);
```

Below the code, a red error message is displayed:

Uncaught RangeError: Maximum call stack size exceeded VM980:2

The error message is followed by a list of stack frames, all pointing to the `add` function:

```
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
at add (<anonymous>:2:1)
```

The console interface includes standard navigation icons (back, forward, search, etc.) and a 'top' filter button. A 'Preserve log' checkbox is also visible.

Handling Slow Operations

```
const slowAdd = (a, b) => {  
  for(let i=0; i<9999999999; i++) {}  
  return a + b;  
};
```

```
const a = slowAdd(3, 3);
```

```
const b = slowAdd(4, 4);
```

```
const c = slowAdd(5, 5);
```

```
console.log(a);
```

```
console.log(b);
```

```
console.log(c);
```

Call Stack



slowAdd(4, 4)

anonymous()

How Callbacks Actually Work

```
const slowAdd = (a, b) => {  
  setTimeout(() => {  
    console.log(a+b);  
  }, 5000);  
};
```

```
slowAdd(3, 3);
```

```
slowAdd(4, 4);
```

Call Stack

setTimeout()

slowAdd(3, 3)

console.log(8)

```
const slowAdd = (a, b) => {  
  setTimeout(cb, 5000);  
};
```

```
slowAdd(3, 3);
```

```
slowAdd(4, 4);
```

Call Stack

cb2

Node

Timer()

cb1

cb2

Queue

```
const slowAdd = (a, b) => {
  setTimeout(() => {
    console.log(a+b);
  }, 5000);
};

slowAdd(3, 3);

slowAdd(4, 4);
```

When the call stack gets empty:

While the queue is not empty:
 event = dequeue an event
 if there is a callback:
 call the event's callback


Call Stack


setTimeout(cb2, 4000)

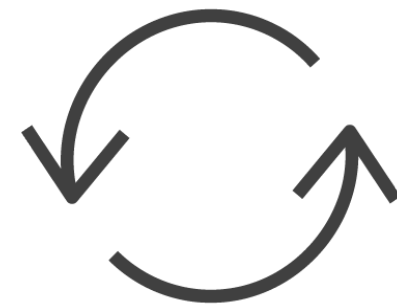
slowAdd(4, 4)

anonymous()

Node

timer
 cb1

timer
 cb2



cb1

cb2

Queue

setImmediate & process.nextTick

```
const slowAdd = (a, b) => {  
  setTimeout(() => {  
    console.log(a+b);  
  }, 0);  
};  
  
slowAdd(3, 3);  
  
slowAdd(4, 4);
```

Call Stack

setTimeout(cb2, 0)

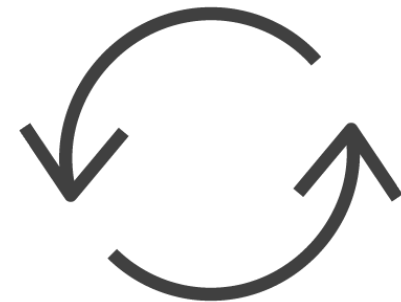
slowAdd(4, 4)

anonymous()

Node

timer
⌘ cb1

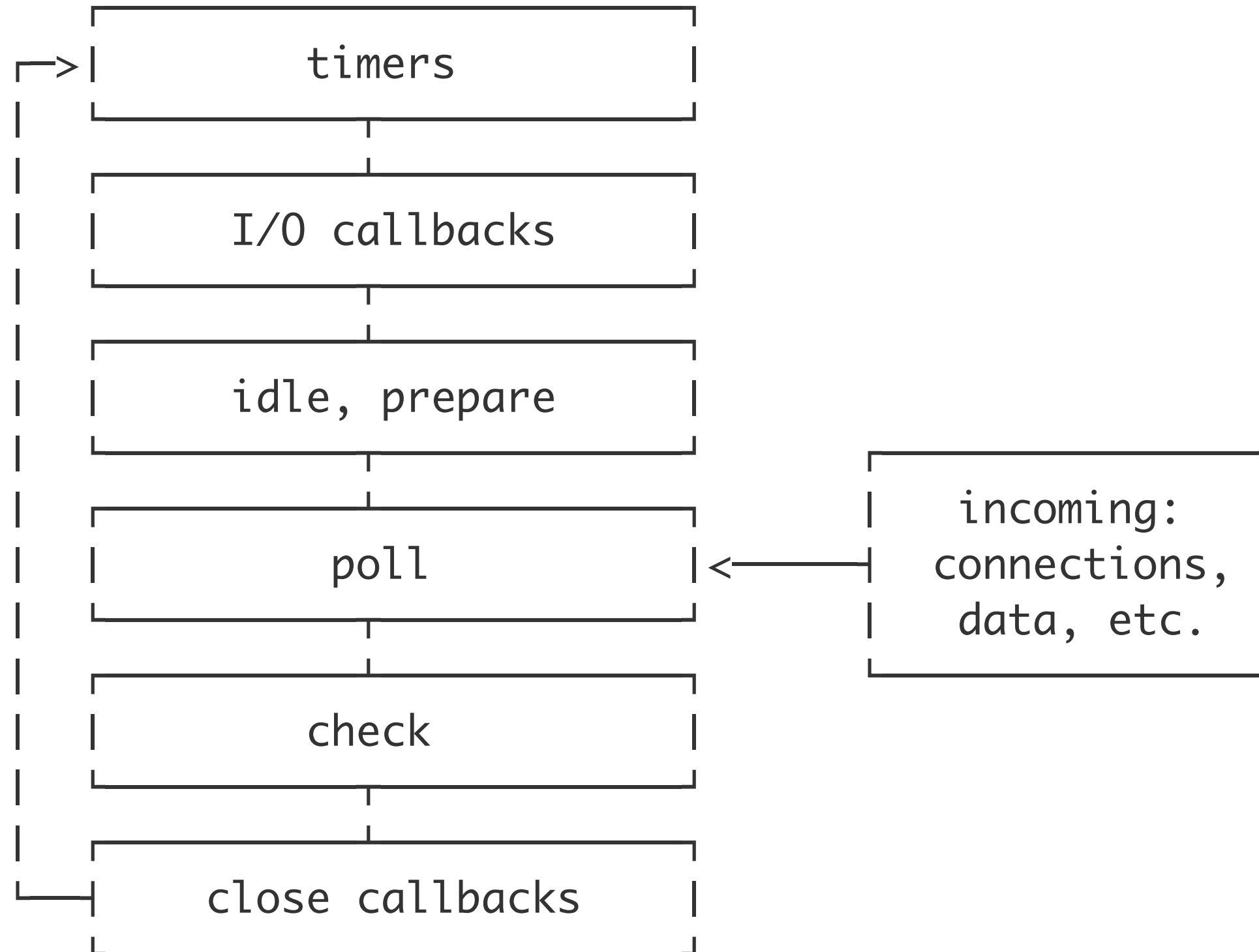
timer
⌘ cb2



cb1

cb2

Queue



Who Named These?

setTimeout

setImmediate

process.nextTick

Summary

The event loop

V8's call stack

Slow I/O operations

setTimeout

setImmediate

process.nextTick