A Journey Into Node.js Internals

Tamar Tena-Stern

Tamar Twena-Stern

- Software Engineer - manager and architect

- Architect @PaloAltoNetworks

- Was a CTO of my own startup
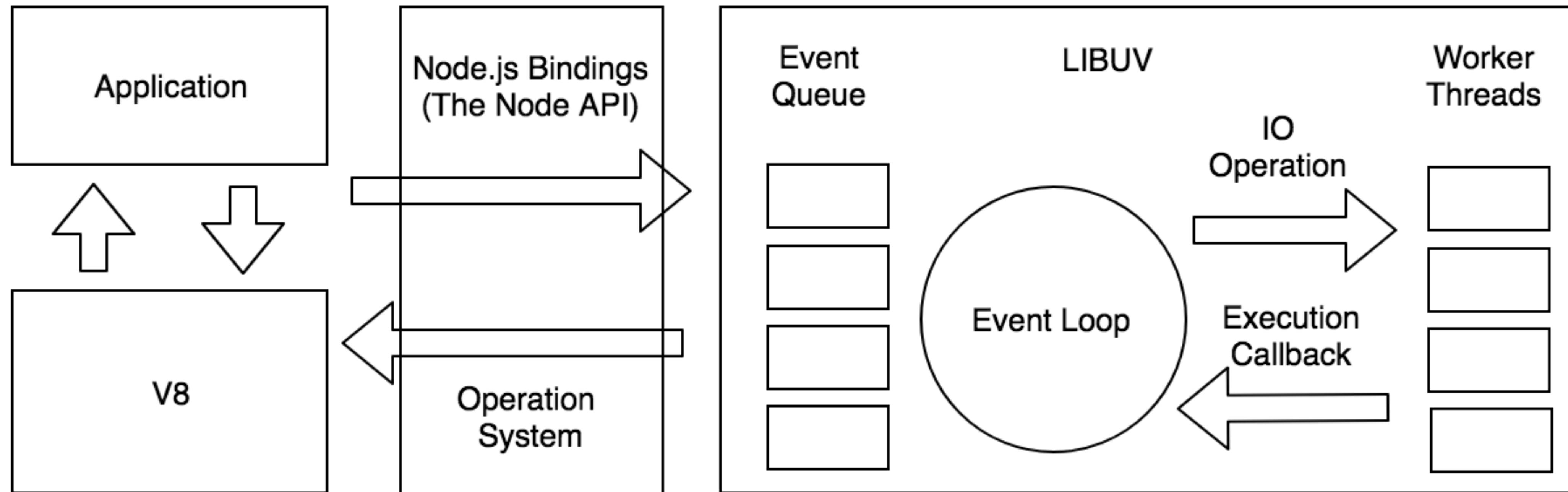
- Passionate about Node.js !

- Twitter: **@SternTwena**

Tamar Tena-Stern

- On Maternity Leave

- Have 3 kids

- Loves to play my violin

Node.js Architecture

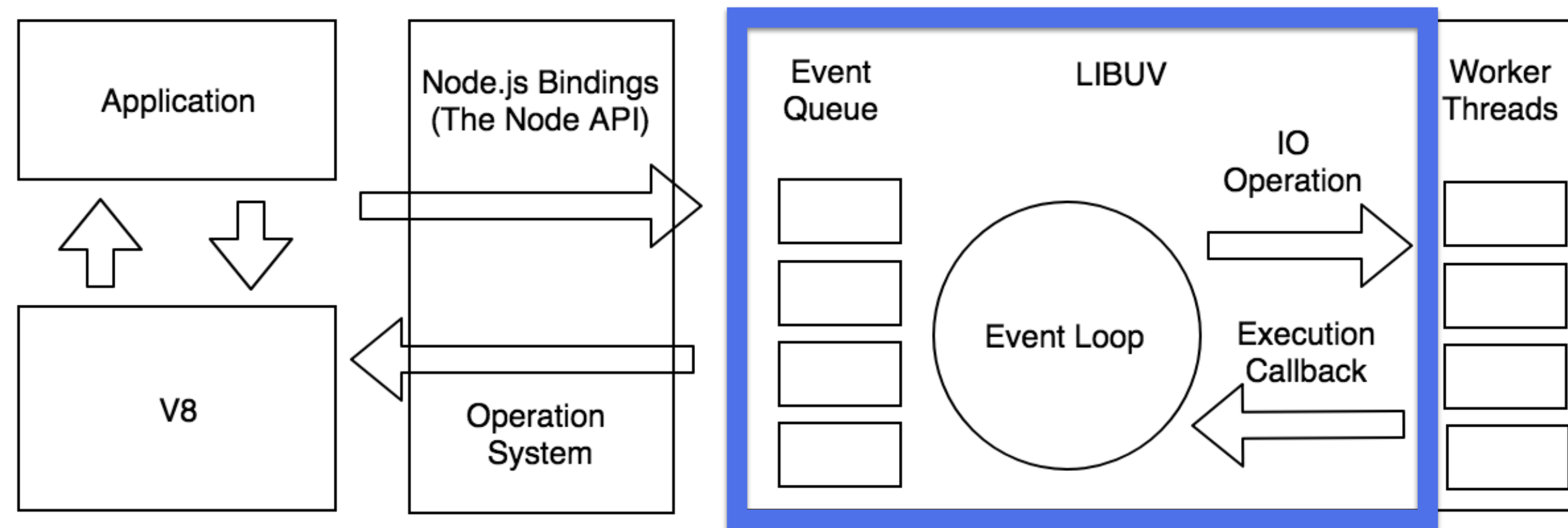# Node.js Architecture - High Level

Now, Lets Get Into The Details

Single Threaded ?

- Not really single threaded

- Several threaded :

  - Event Loop

  - The workers thread pool

Event Loop Thread

- Every request registers a callback which executes immediately

- The event loop execute JavaScript callbacks

- Offloads I/O operations to worker thread pool.

- Handle callbacks for asynchronous I/O operations from multiple requests.
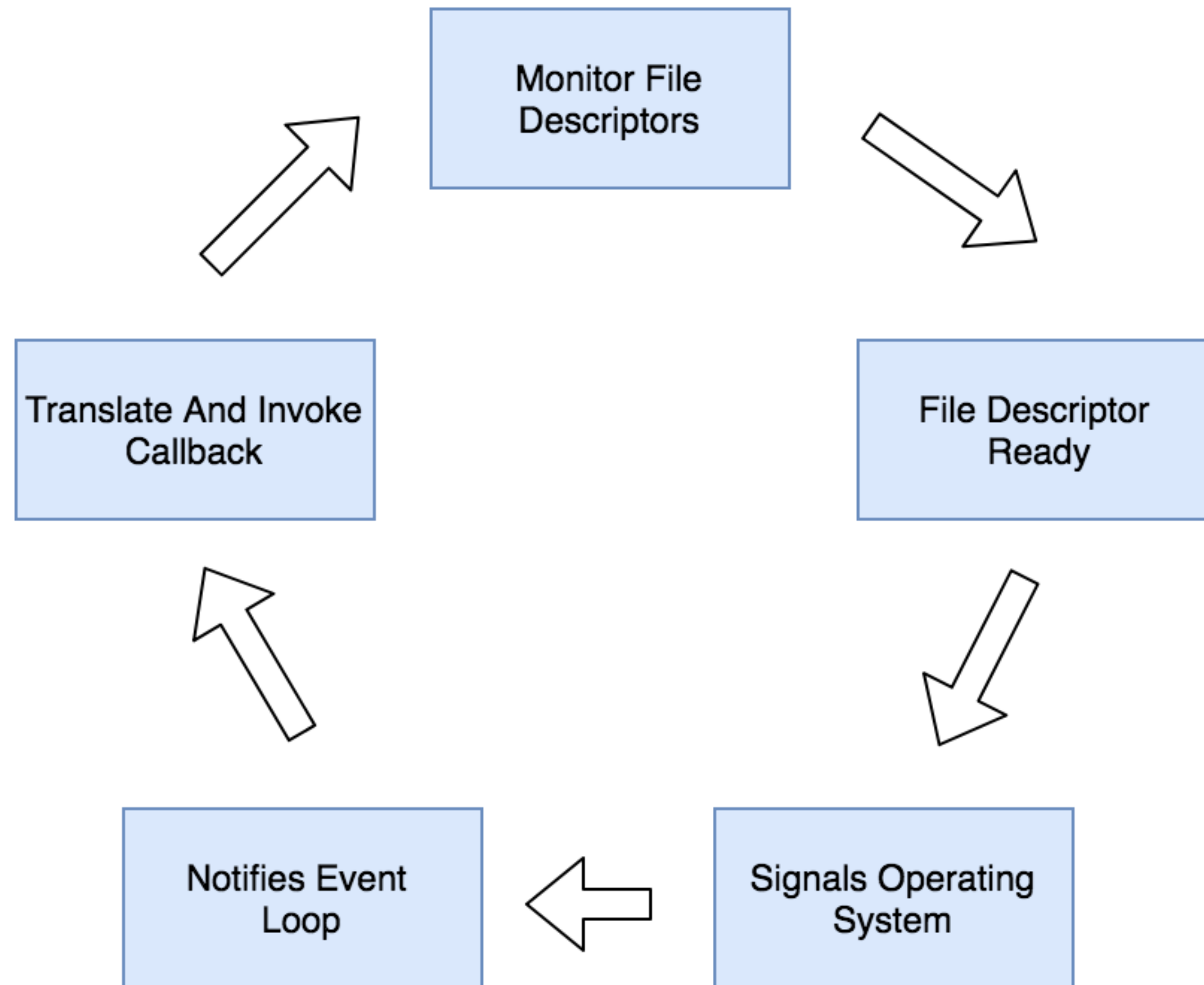
Worker Thread Pool
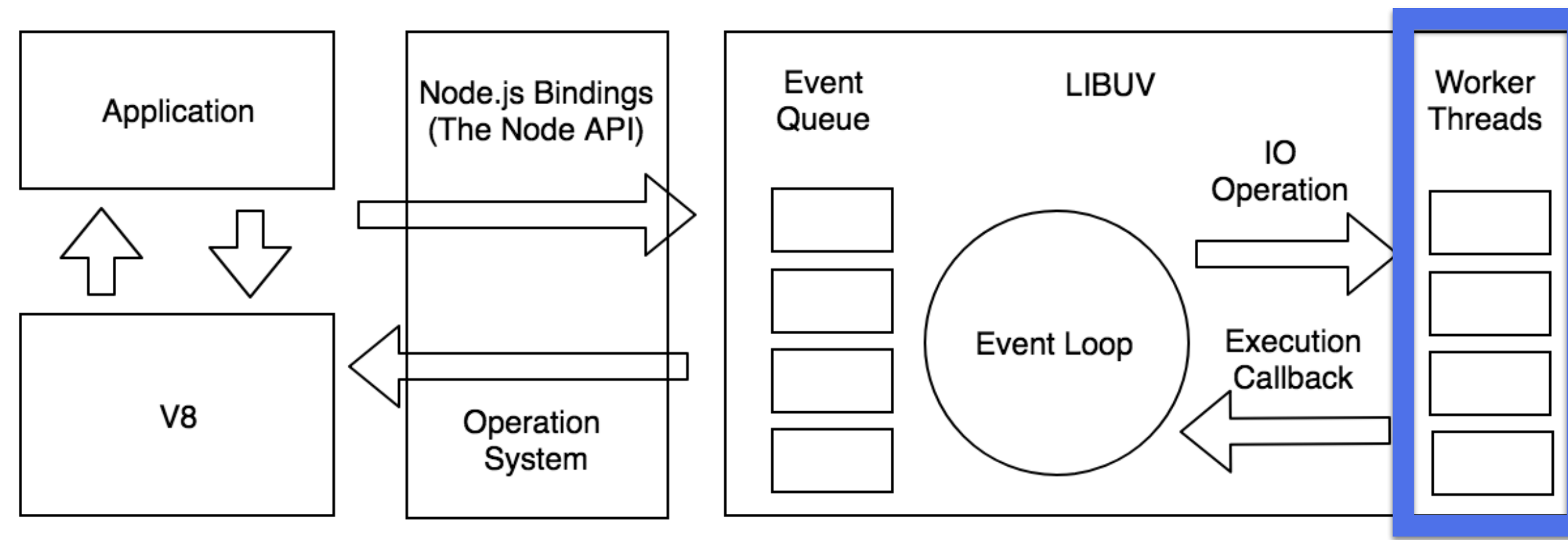
- Thread pool to perform heavy operations

  - I/O

  - CPU intensive operations

- Bounded by fixed capacity

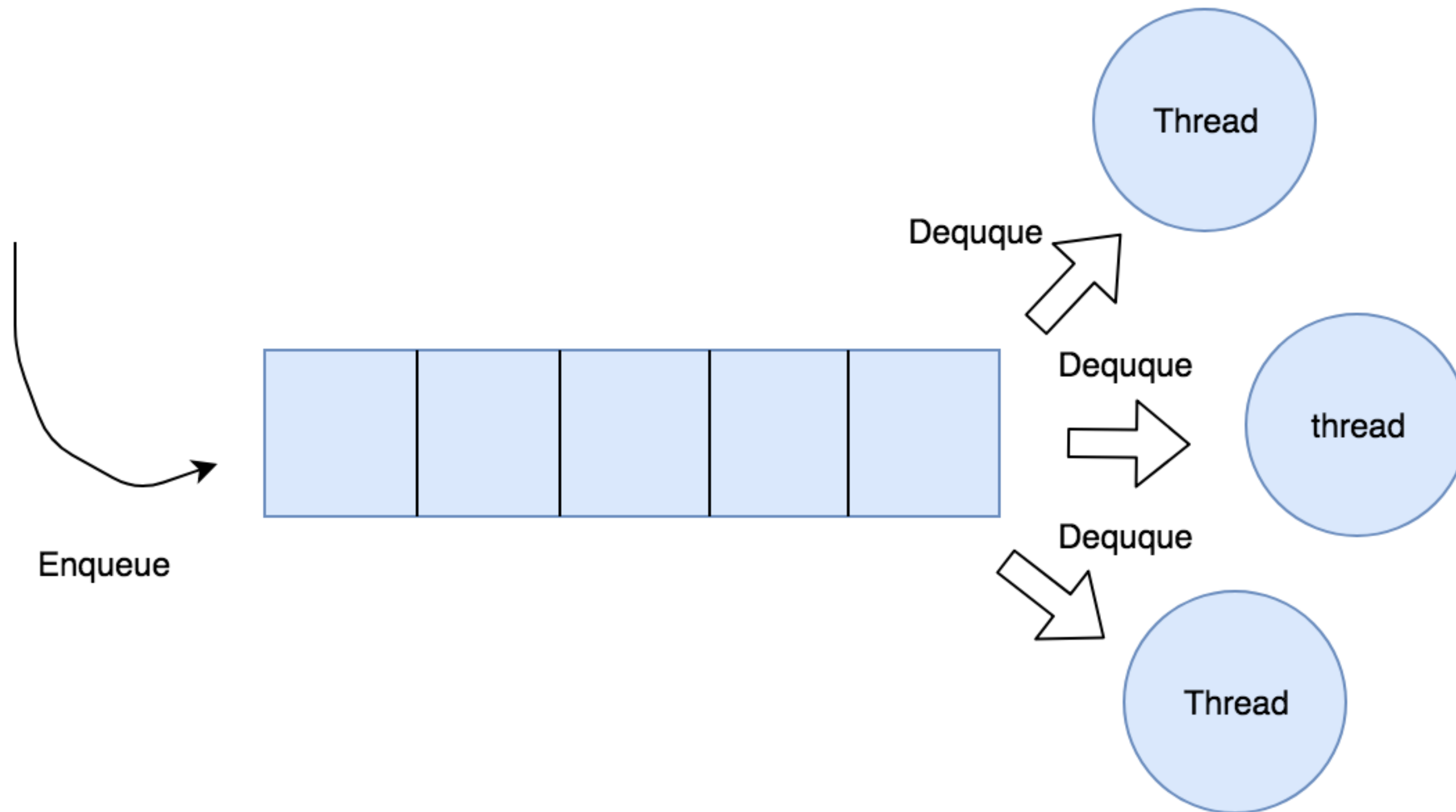- A node module can submit a task to libUV API

Submitting A Request To The Worker Pool

- Use a set of 'basic' modules that work with the event loop

  - Examples:

    - Fs

    - Dns

    - Crypto

    - And more

- Submit a task to libUV using c++ add-on

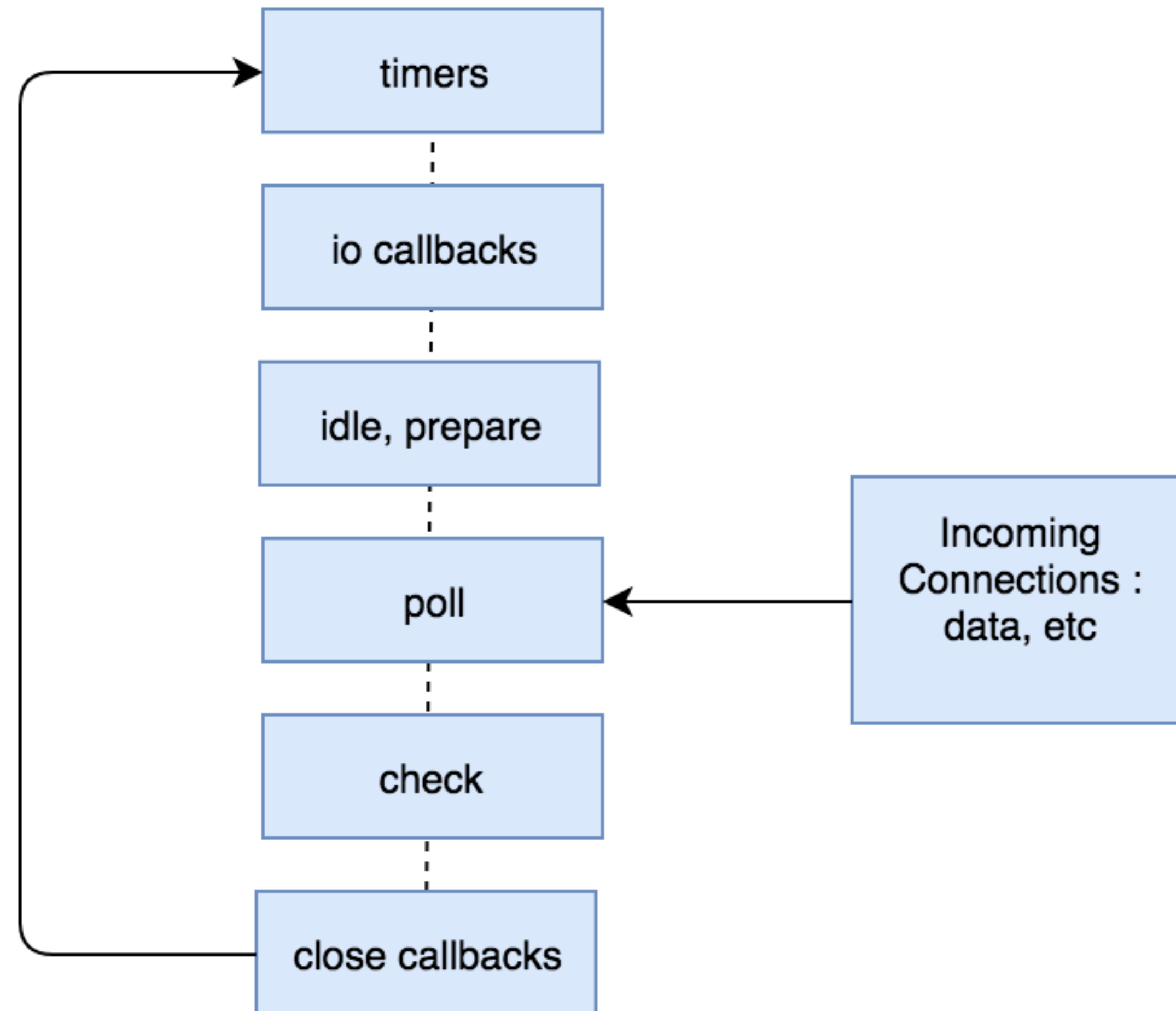# Event Loop Implemented With A Queue ?
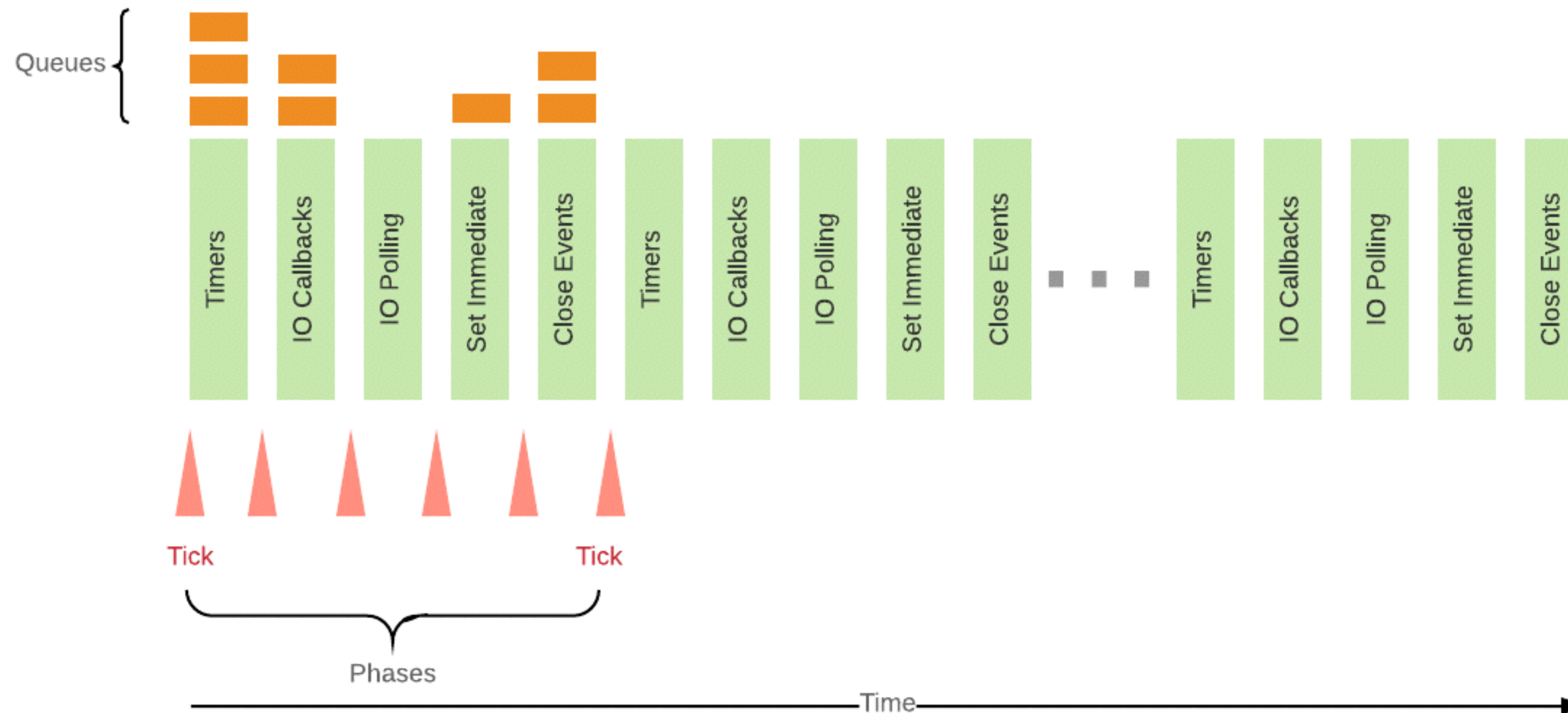
How Worker Pool Implemented ?

- Libuv in a nutshell :
  - Multi platform C library
  - Provides support for async I/O based on event loop
- Supports :
  - Epoll (Linux)
  - Kqueue (OSX)
  - Windows IOCP
  - Solaris event ports

The Event Loop - The Different Phases

# Event Loop Phases Overview

Phase General Mechanism

Timers Phase

- setTimeout, setInterval

- Timer's callback will run as soon as they can be scheduled after the threshold

- Timer's callback scheduling controlled by the "poll" phase

I/O Callback Phase

- Executes system error callbacks

  - Example : TCP socket connection error.

- Normal I/O operation callbacks are executed in the poll phase.

```
socket.on('error', function(exception) {
  console.log('SOCKET ERROR');
  socket.destroy();
})
```

Poll phase
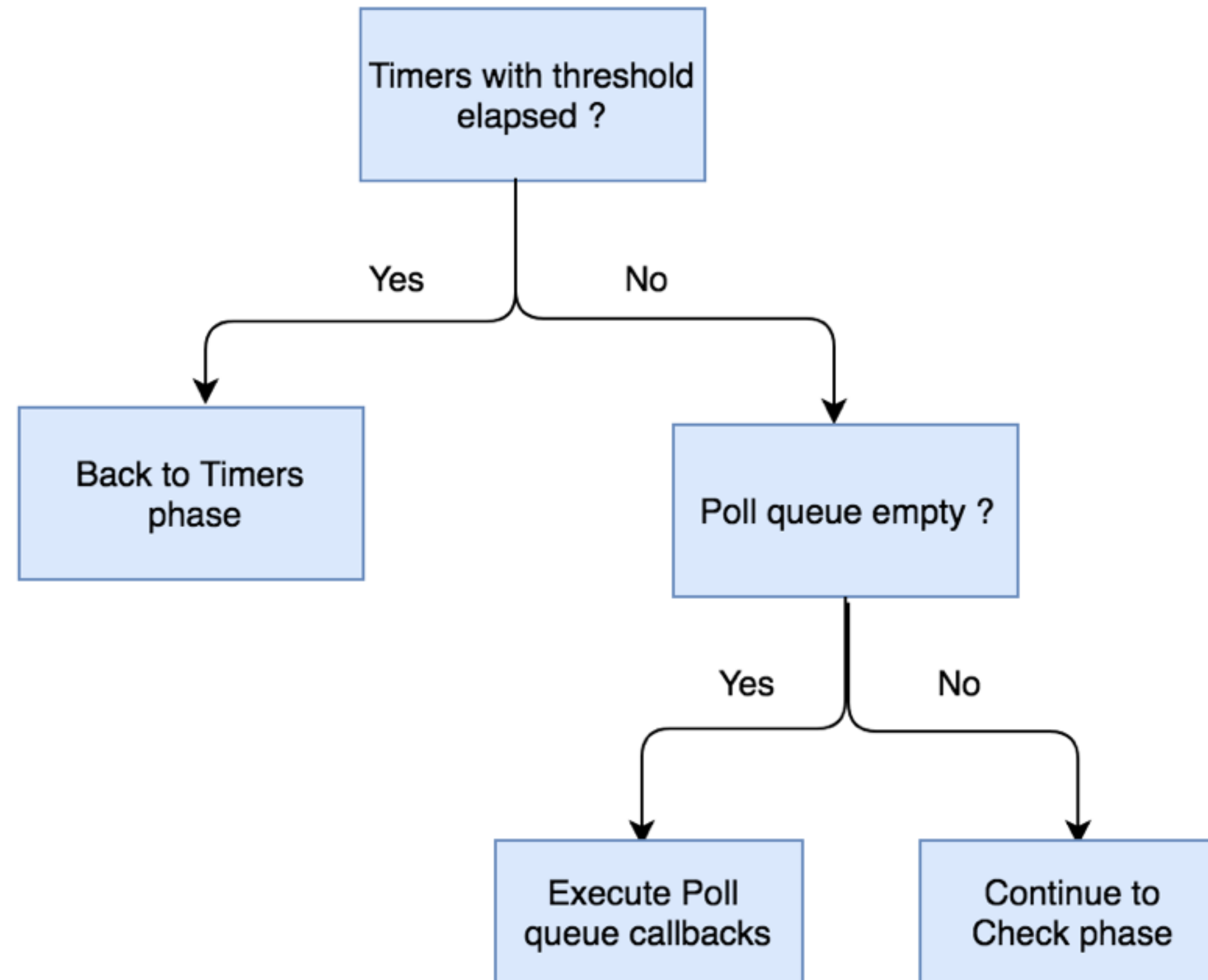
Check Phase And Close Phase

- Check Phase - Execute callbacks for setImmediate timers

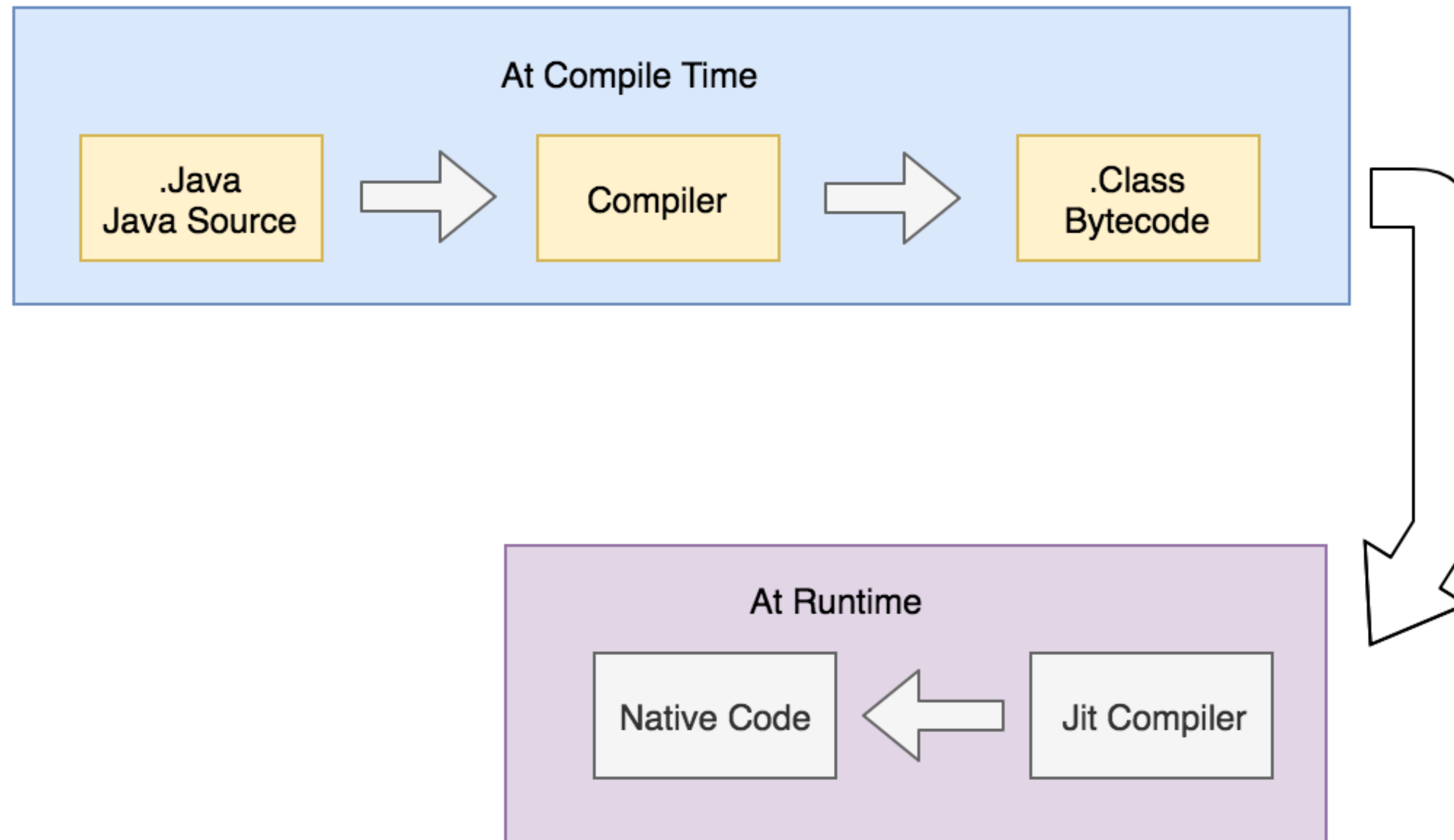- Close Phase – Handles an abruptly close of a socket or a handle

Lets profile some code

The JIT Compiler And V8 Engine

What Is Just-In-Time Compilation ?

- Compilation during run time

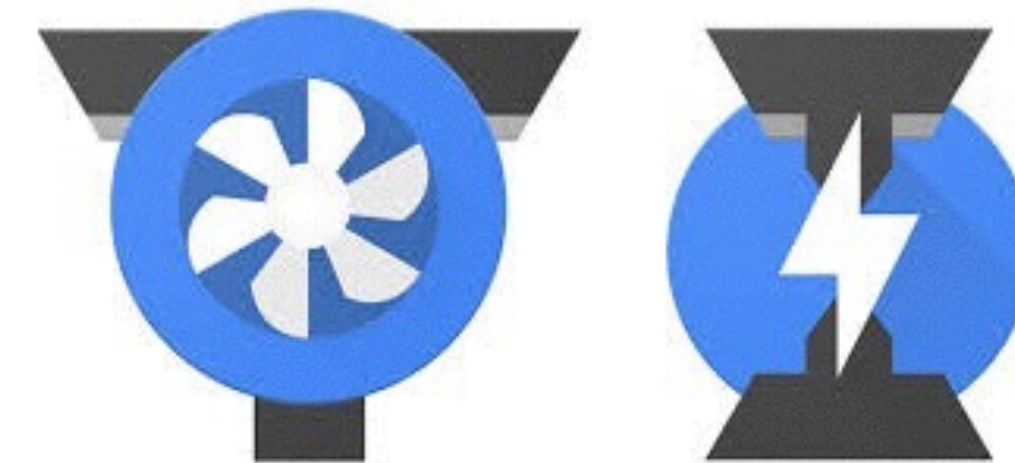- Combines two approaches :

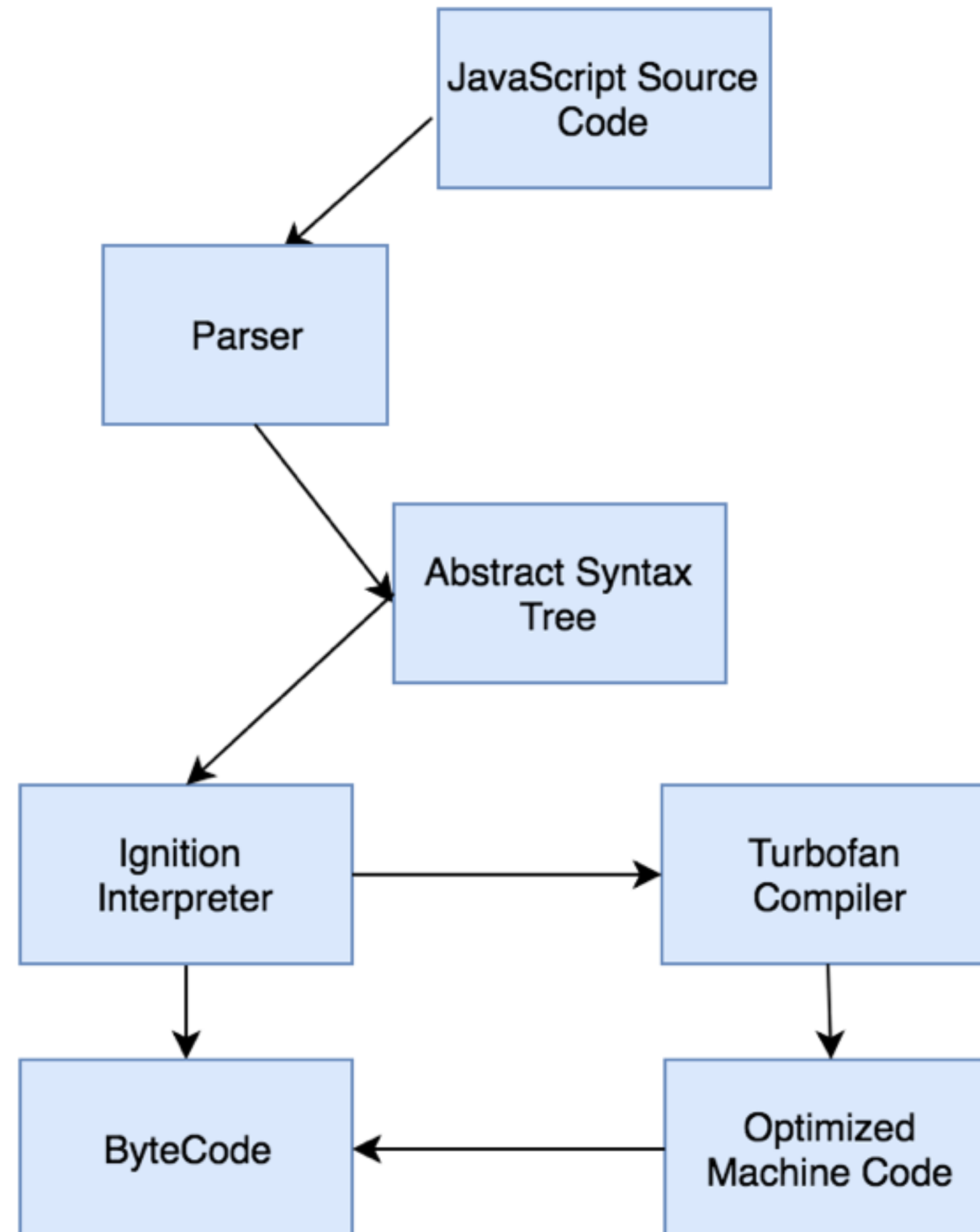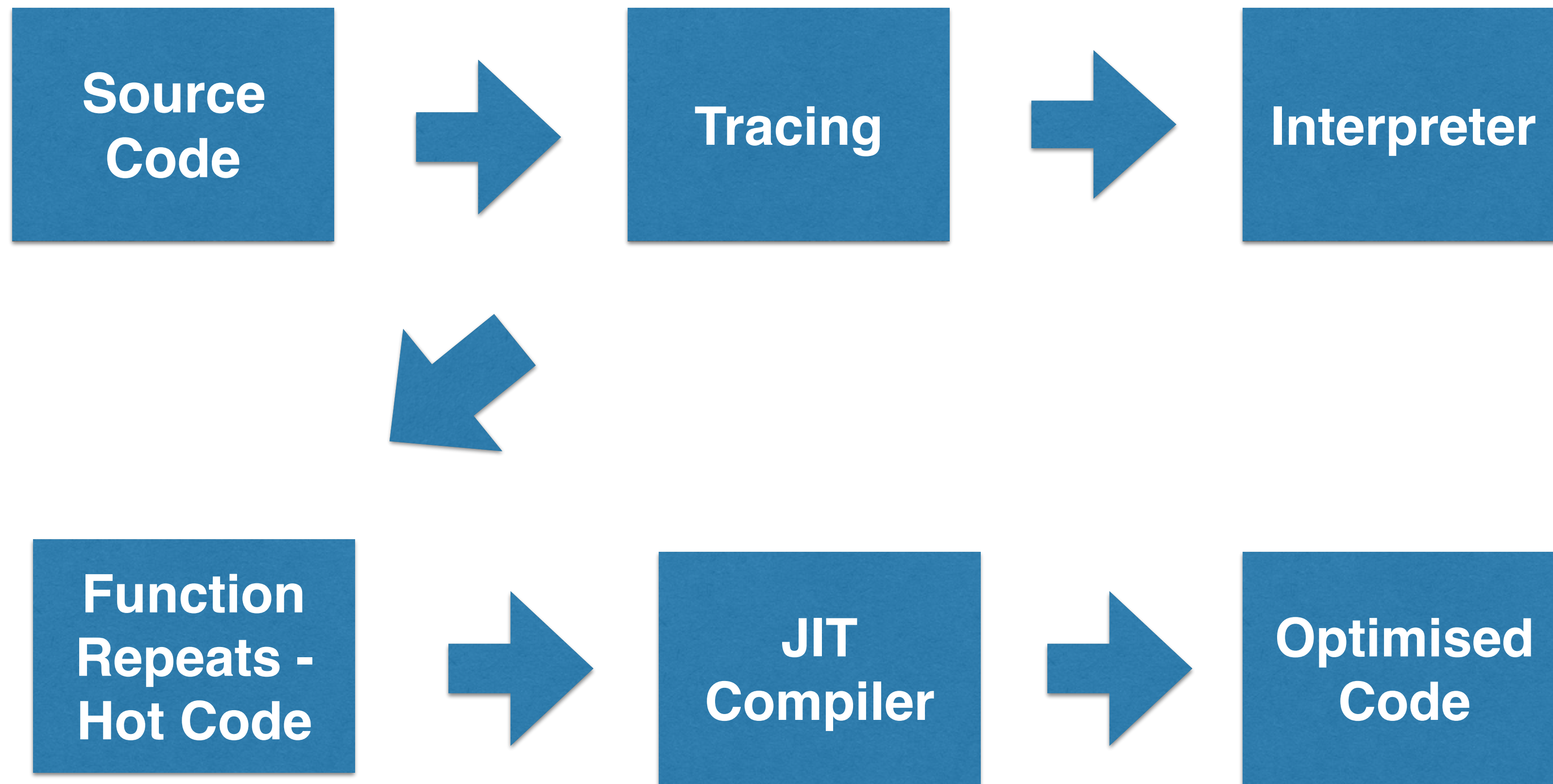 - Ahead of compilation

  - Interpreter

JIT Compiler In Java

At Compile Time

| .Java Java Source | → | Compiler | → | .Class Bytecode |

At Runtime

| Native Code | ← | Jit Compiler |

Chrome V8

- Open source JavaScript engine

- Developed originally for Google Chrome and chromium

- Also used for

  - Couchbase

  - MongoDB
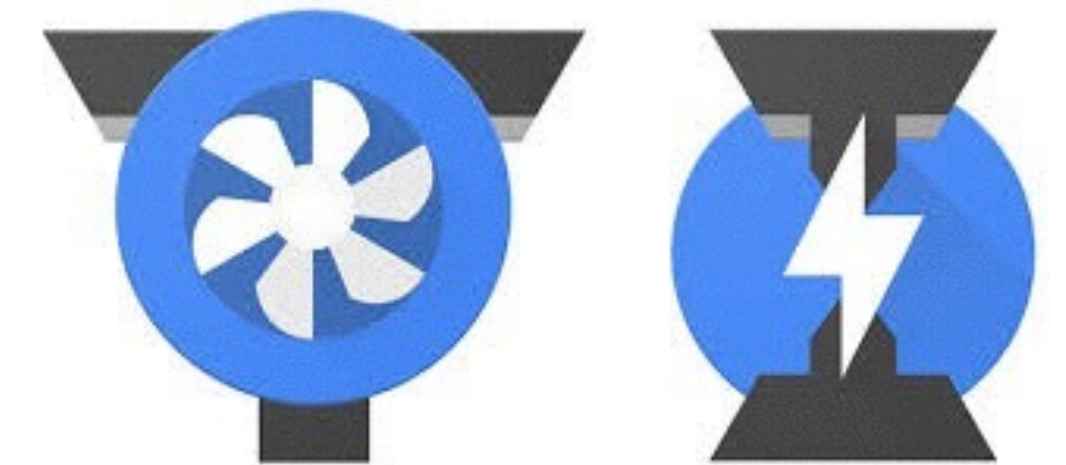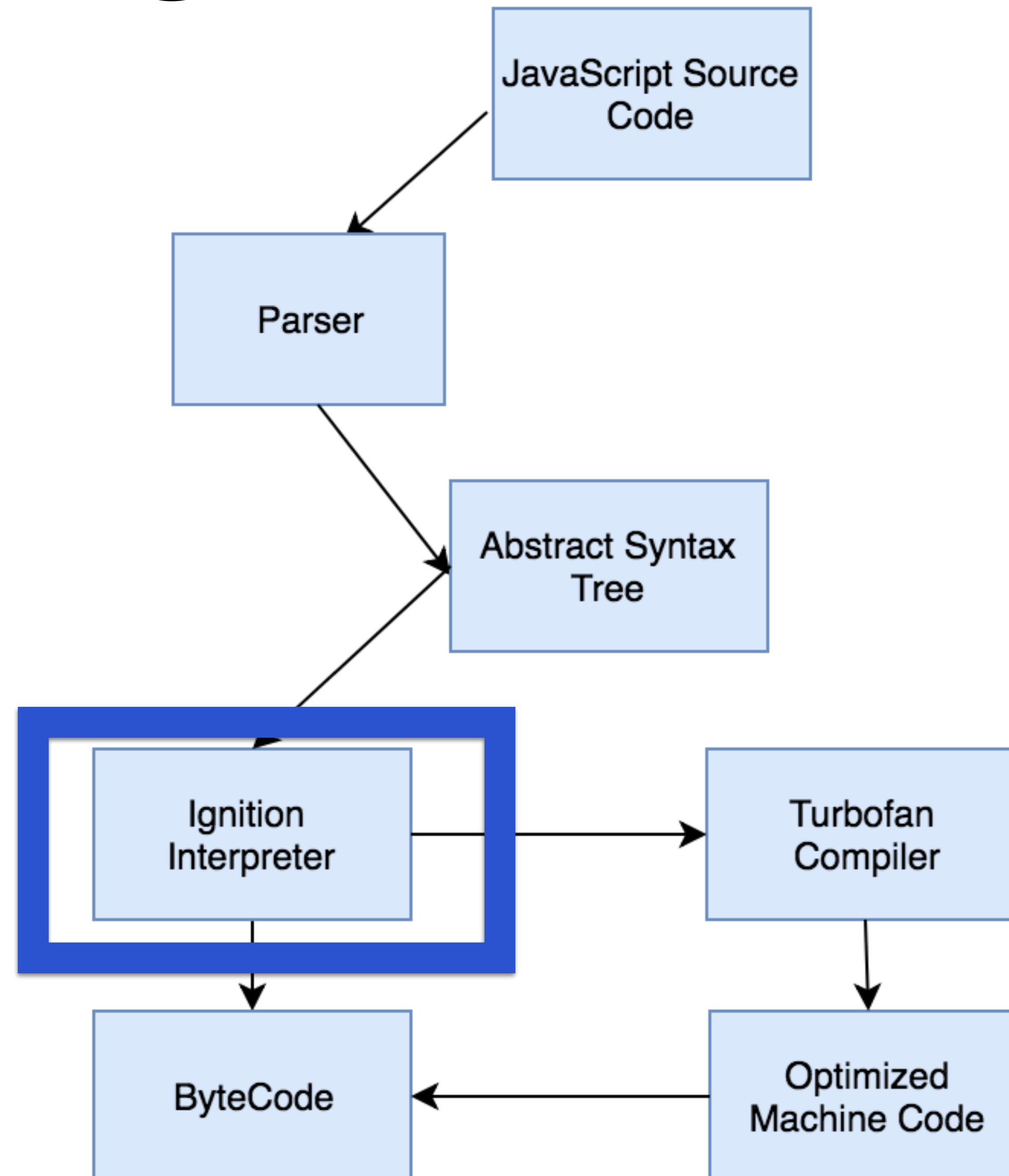
  - Node.js

V8 Architecture

The JIT Compilation In V8

Source Code → **Tracing** → **Interpreter**

**Function Repeats - Hot Code** → **JIT Compiler** → **Optimised Code**

What Is An Optimised Compiler?

- When a code is hot – it is worth doing multiple optimisations

- Tracing will send it to optimising compiler

- Creates an even faster version of the code

- Tracing pulls it when the function code runs

Ignition Interpreter

Turbofan Compiler

JavaScript Source Code

Parser

Abstract Syntax Tree

Ignition Interpreter

Turbofan Compiler

ByteCode

Optimized Machine Code
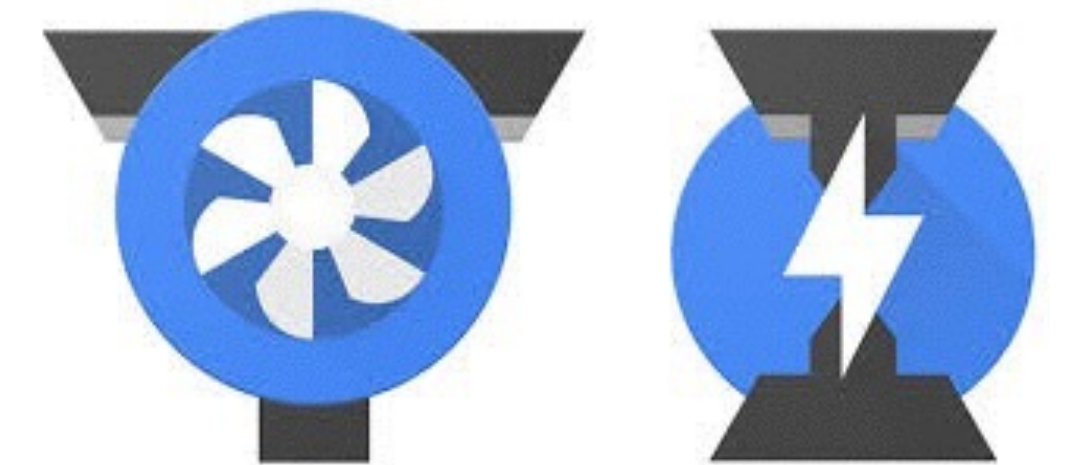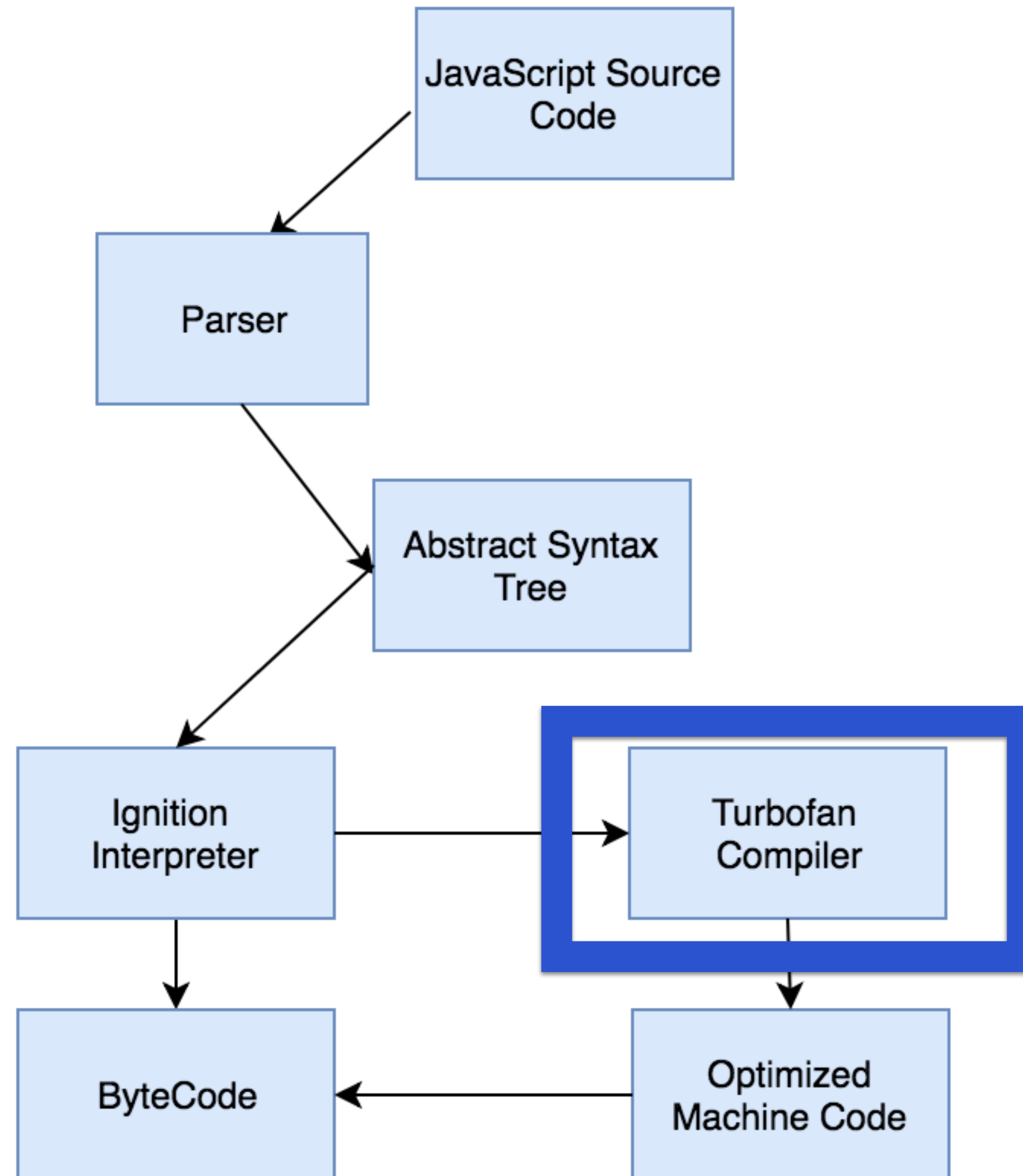
How To Help Turbofan Optimise Hot Code ?

• The fewer function input type variations lead to smaller and faster resulting code.

• Keeping your functions *monomorphic* or at least *polymorphic*

  • *Monomorphic*: one input type

  • *Polymorphic*: two to four input types

  • *Megamorphic*: five or more input types

Optimisation And De-optimisation

- **Optimisation** - All assumptions fulfilled – Compiled code runs.

- **Deoptimisation** – Not all assumptions fulfilled – Compiled code erased

| Assumptions Fulfilled	→	Optimisation	→	Assumptions Break	→	De-Optimisation
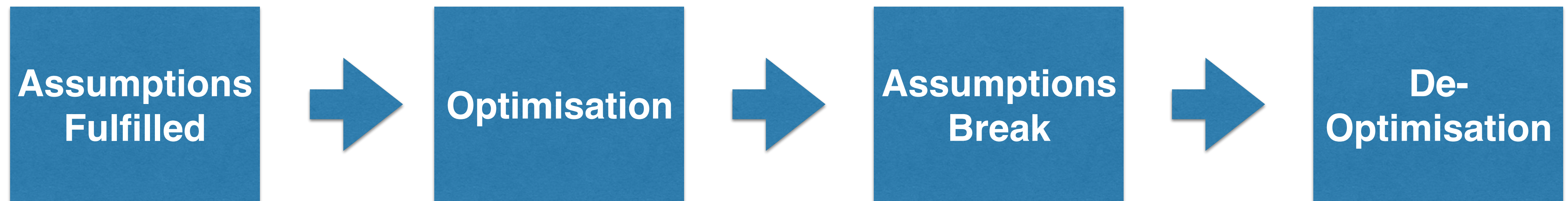
Avoid De-optimisation

- When a code is optimised and de-optimised – it ended up being slower then just use the baseline compiled version

- Most browsers and engines will stop trying after several iterations of optimising and de-optimizing

De-optimisation Demo

V8 Memory Management

V8 Memory Structure
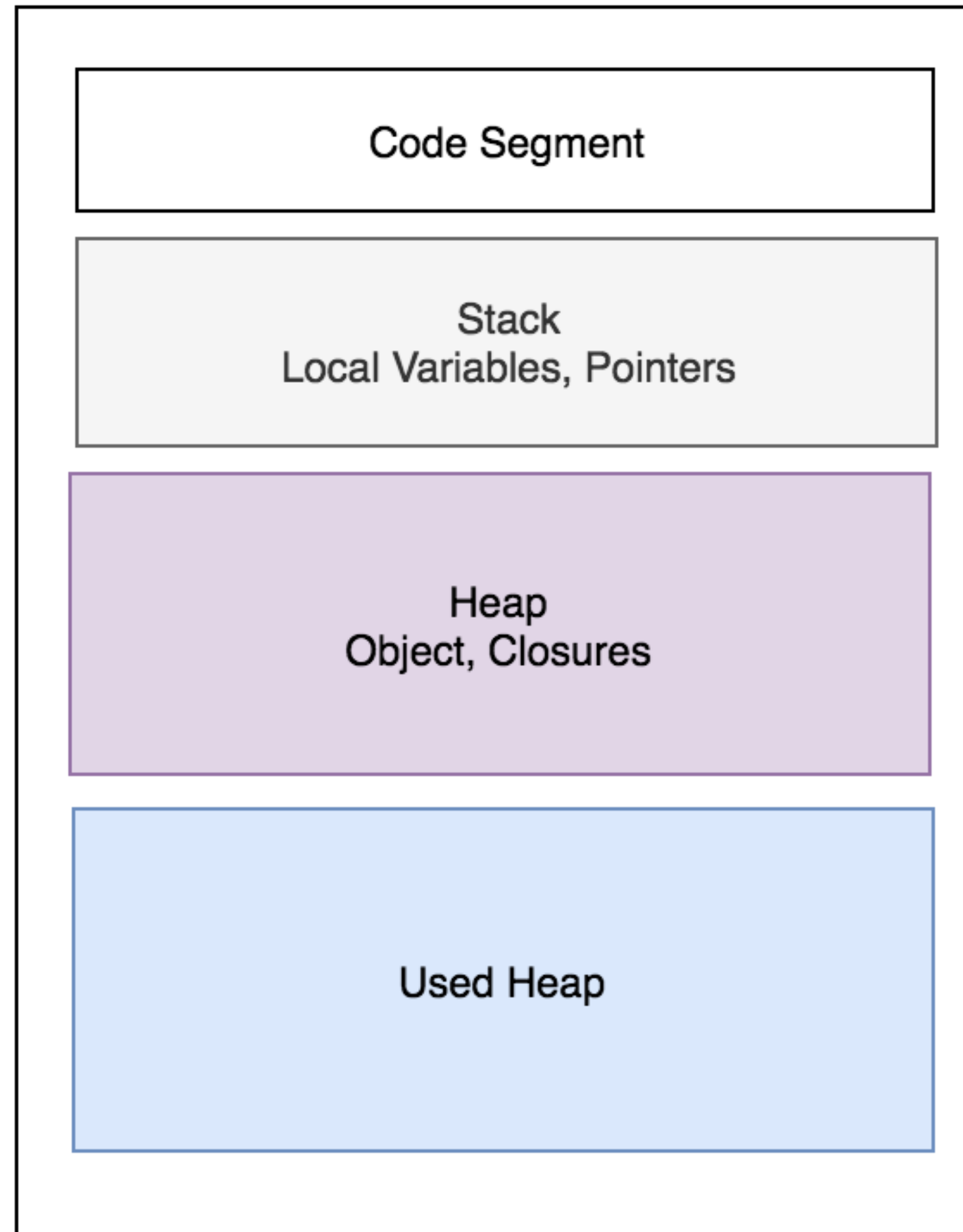
The Stack

- Every executing function pushes its local variables and arguments

- Maintains two pointers – Stack Pointer and Base Pointer

- Divided into stack frames

- No Garbage Collection - self cleaning

- Hold the key to the garbage collection process - Starts from active objects that has pointers to the stack.
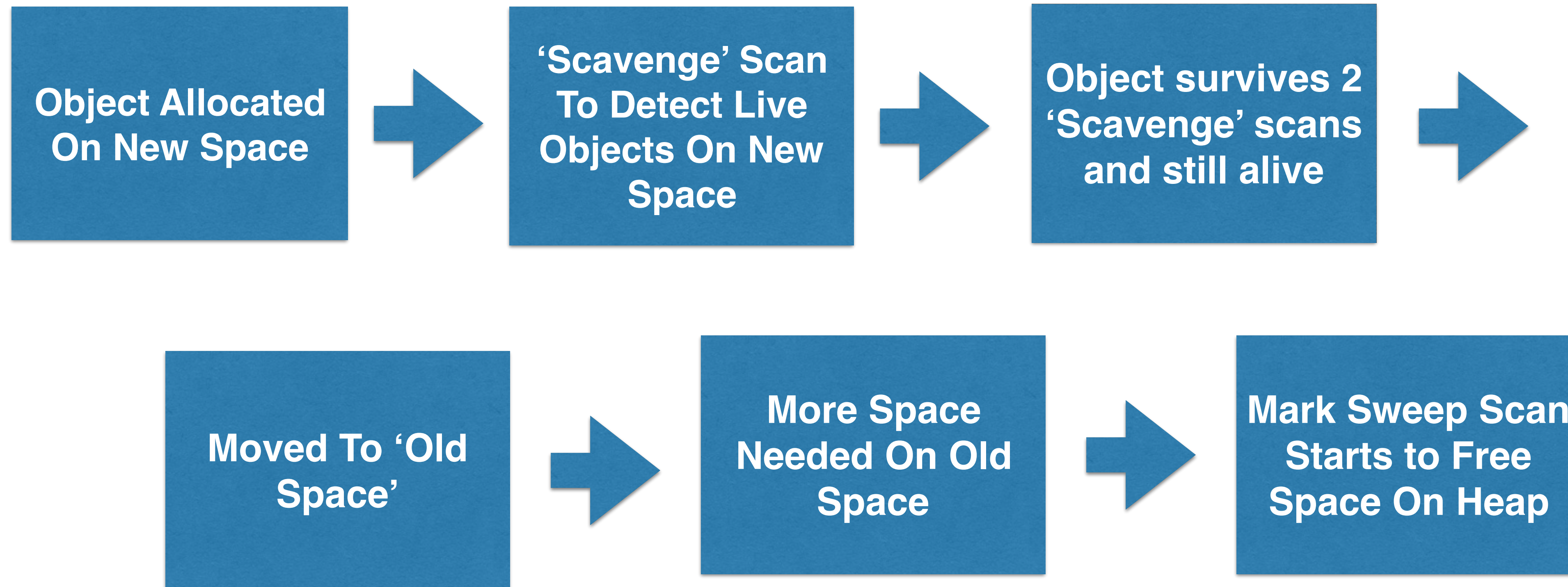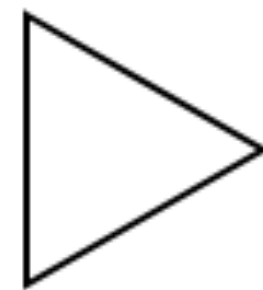
The Heap Structure
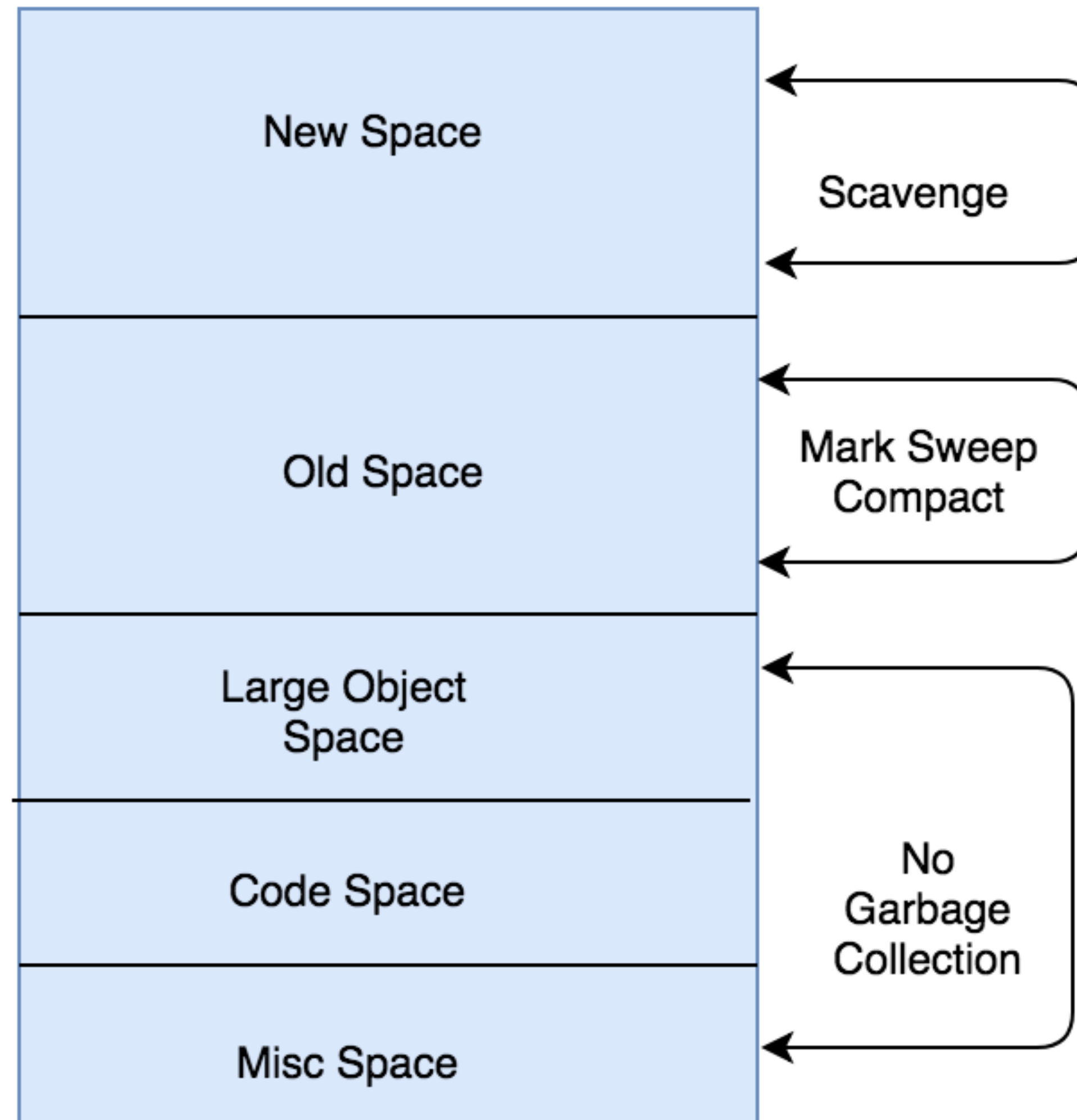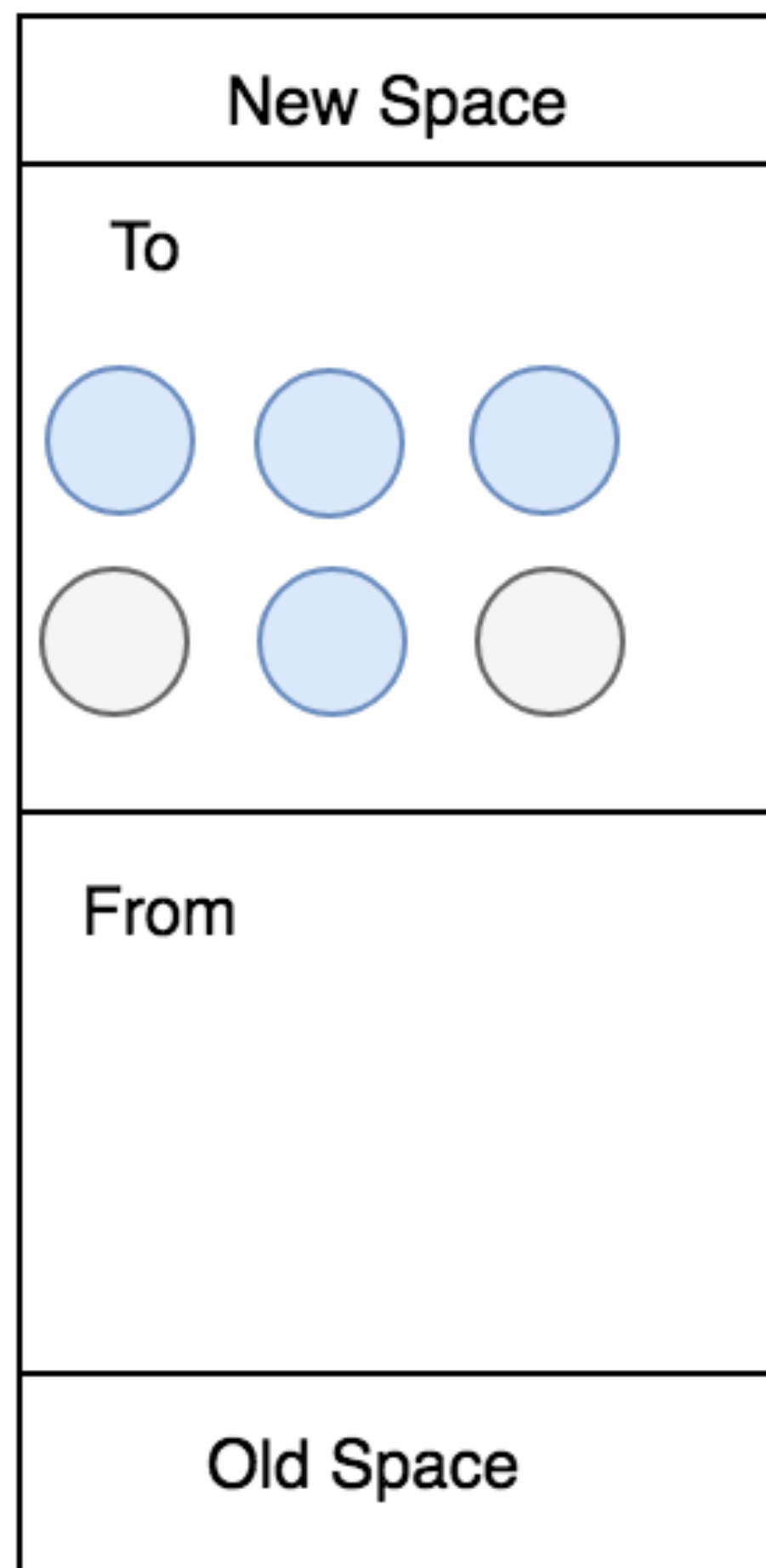
Generational GC System

- Lifetime of objects determines their place on the heap

- New Space – short lived Objects

- Old Space – long lived Objects

- Scan on 'New Space' called 'Scavenge'

  - Very fast – takes less then a millisecond

- Scan on 'Old Space' called 'Mark Sweep'

  - Slower scan

Object Life On The Heap

Object Allocated On New Space → **'Scavenge' Scan To Detect Live Objects On New Space** → **Object survives 2 'Scavenge' scans and still alive** →

Moved To 'Old Space' → **More Space Needed On Old Space** → **Mark Sweep Scan Starts to Free Space On Heap**
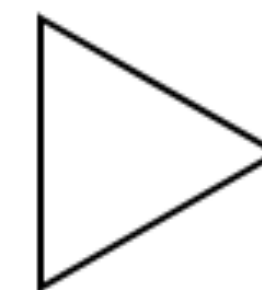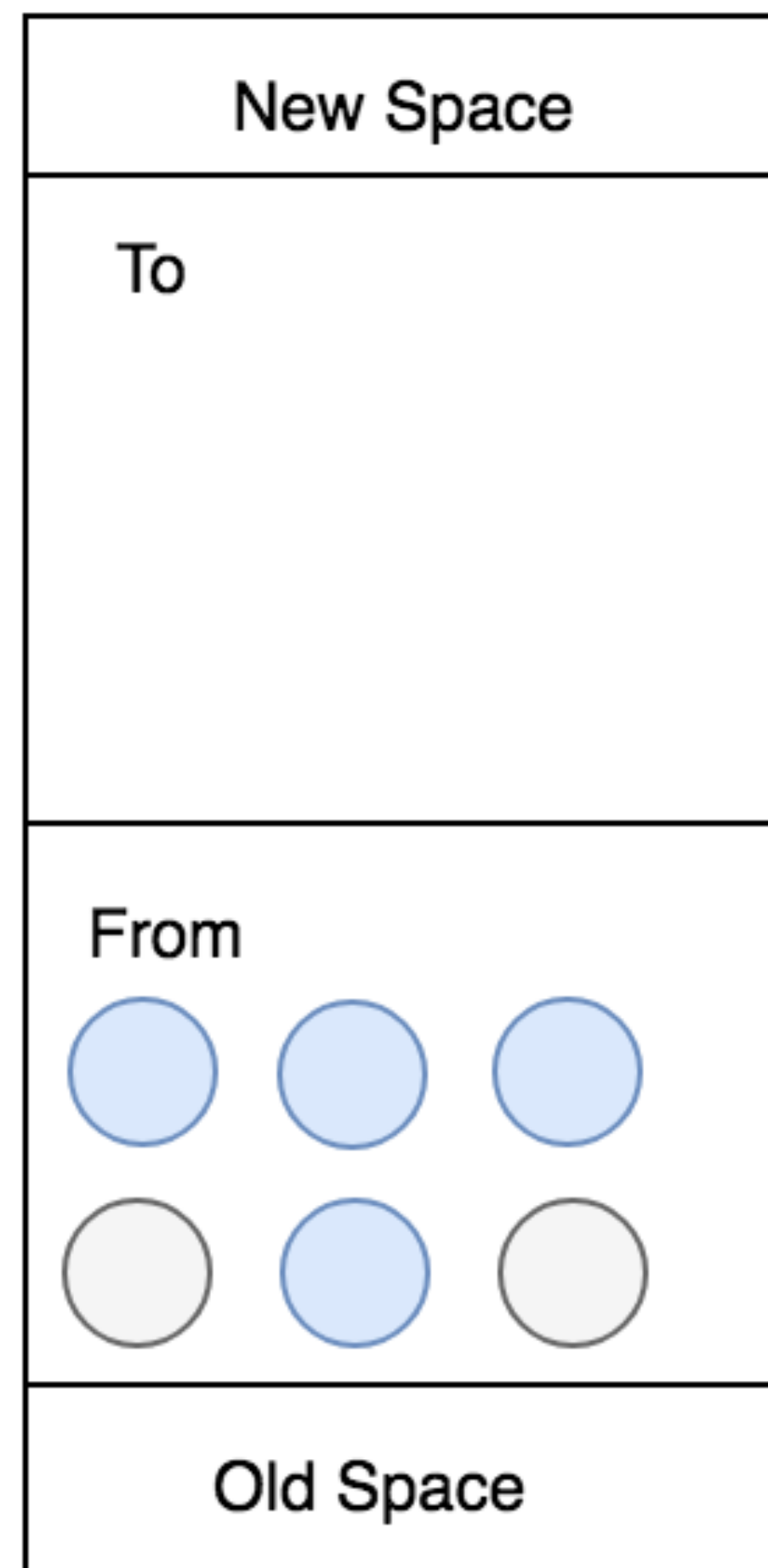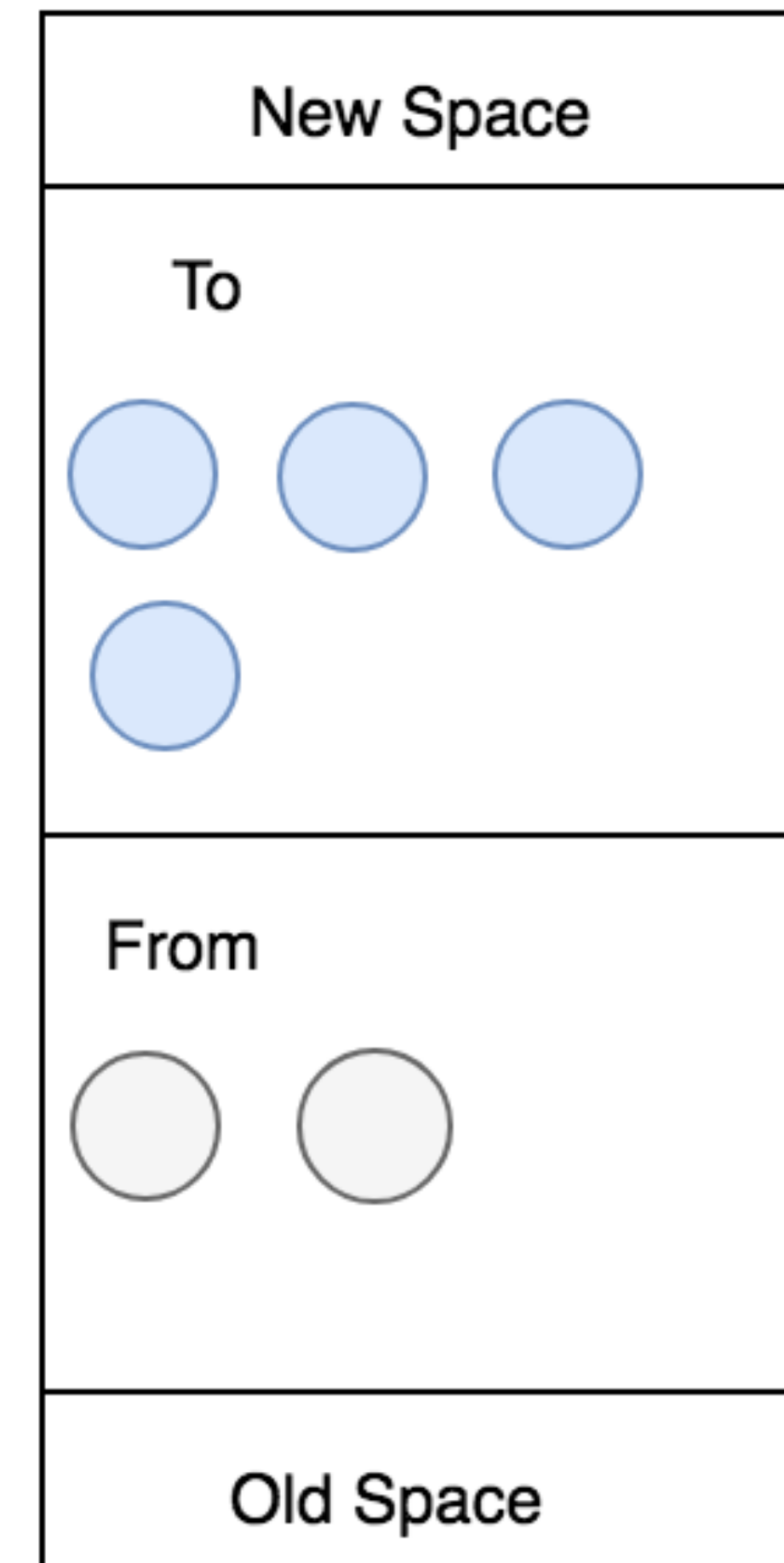
Scavenge Scan Explained

To space filled, Scavenge triggered
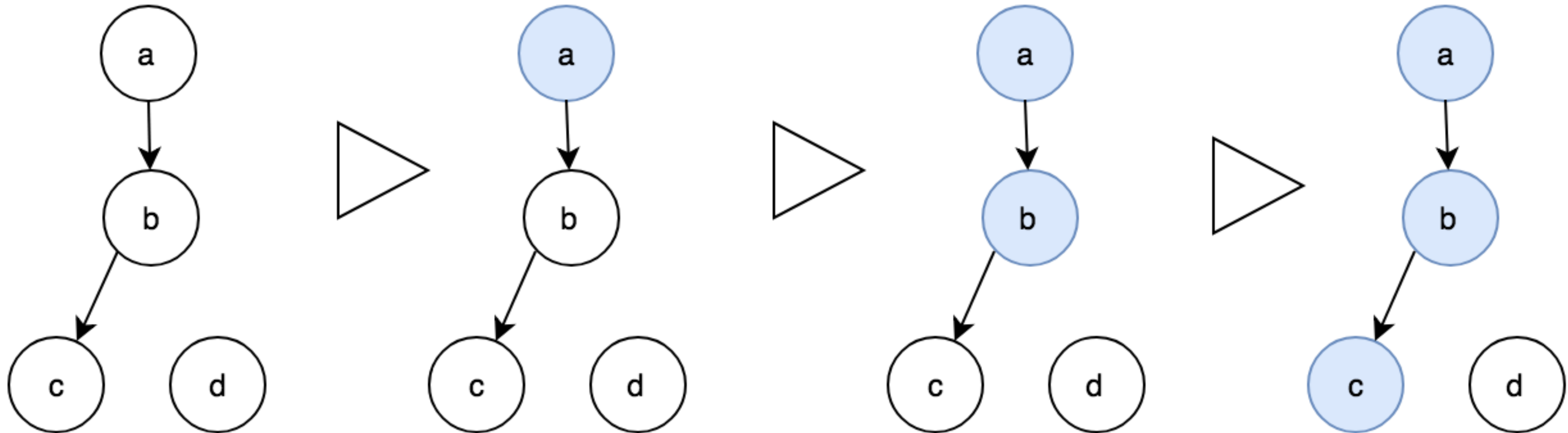
All objects move to from space
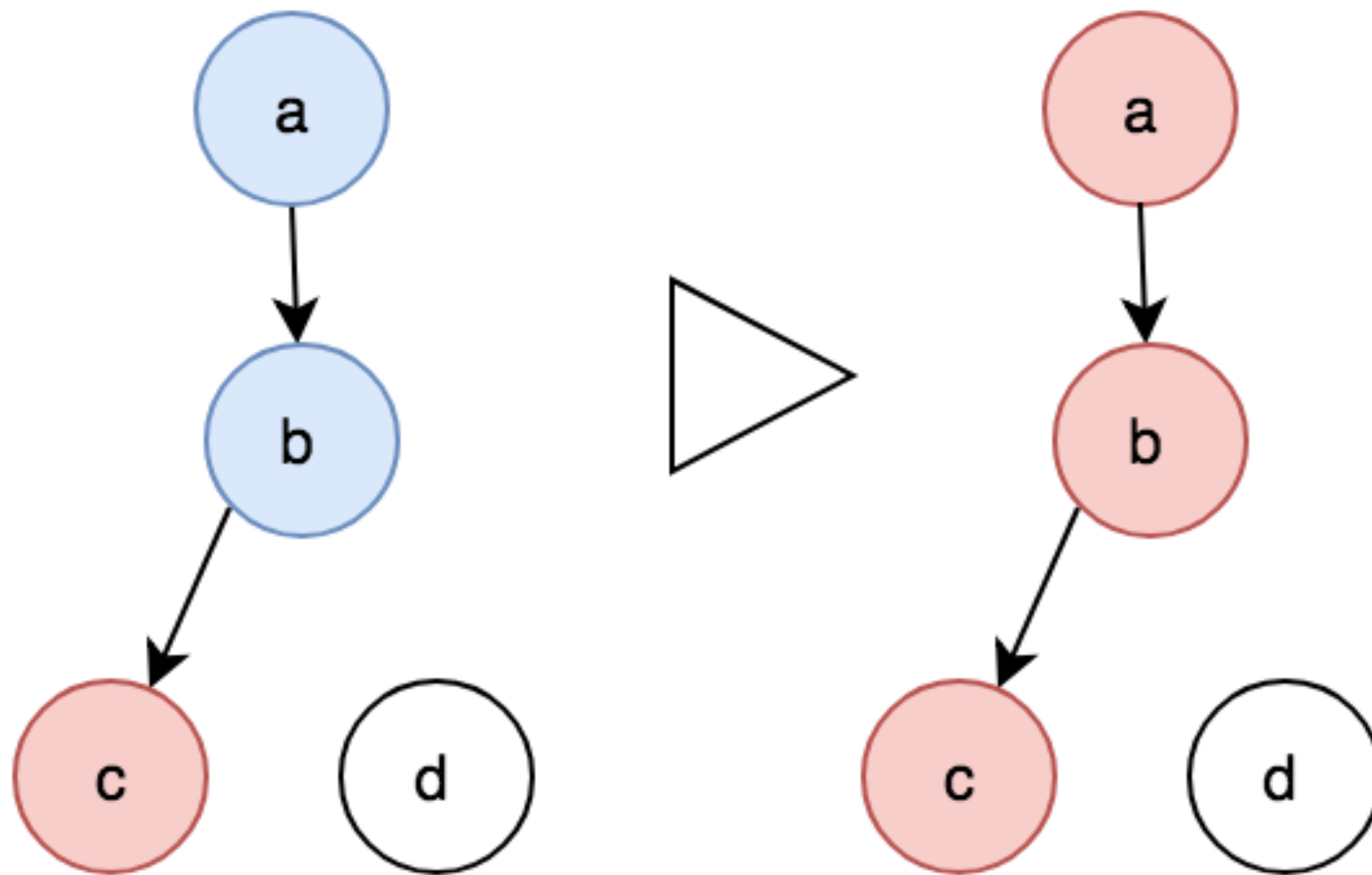
All live objects move back to 'To Space'

Mark Sweep For The Following Code

```
const a = {};
a.b = {};
a.b.c = [1, 2];
a.b.d = [3, 4];
a.b.d = null;
```
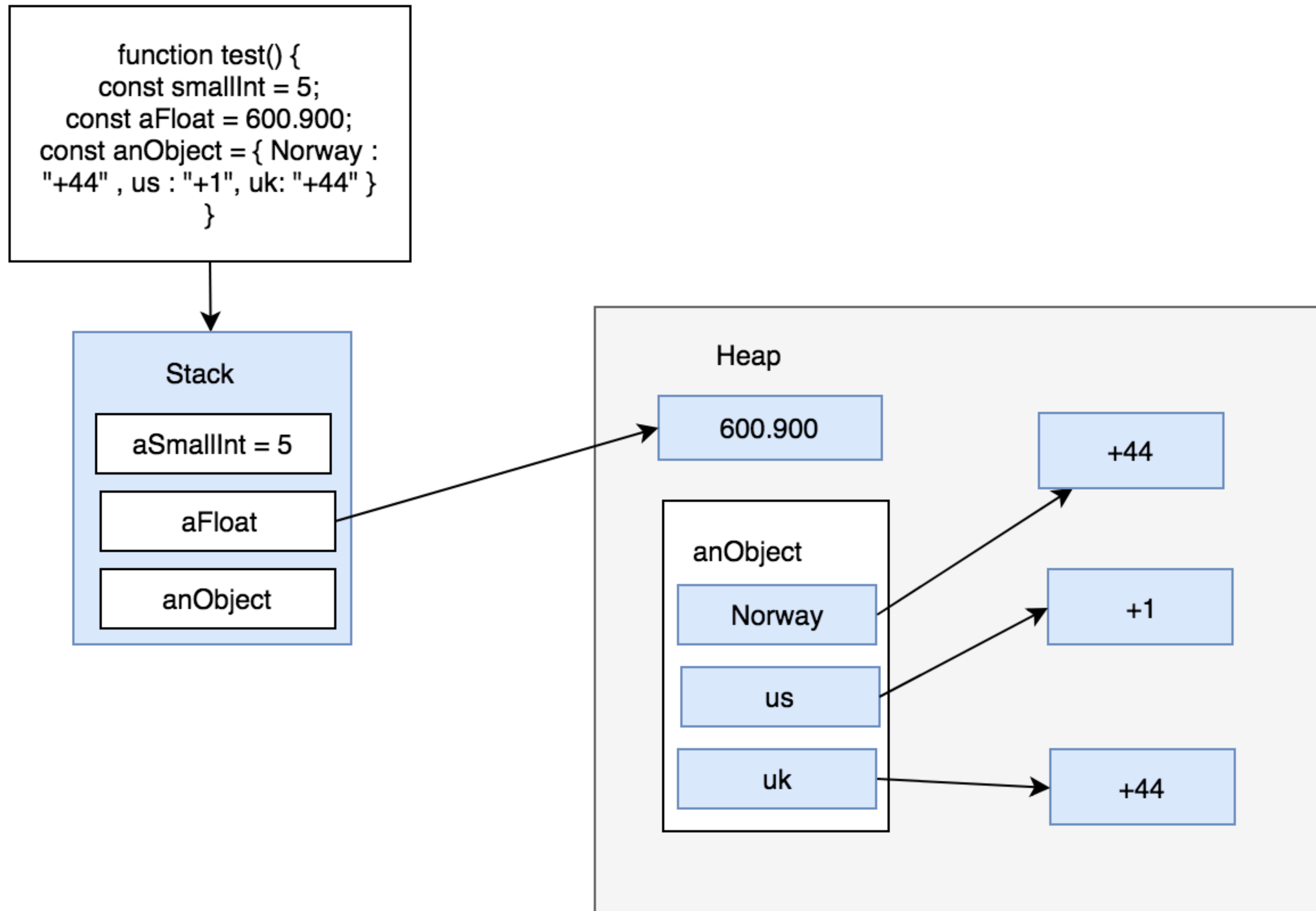
Mark Sweep - DFS

Mark Sweep - DFS

Whats Allocated Where ?

function test() {
    const smallInt = 5;
    const aFloat = 600.900;
    const anObject = { Norway :
    "+44" , us : "+1", uk: "+44" }
}

Stack

aSmallInt = 5

aFloat

anObject

Heap

600.900

+44

anObject

Norway

us

+1

uk

+44

Lets Find Some Memory Leaks !

- Twitter: **@SternTwena**