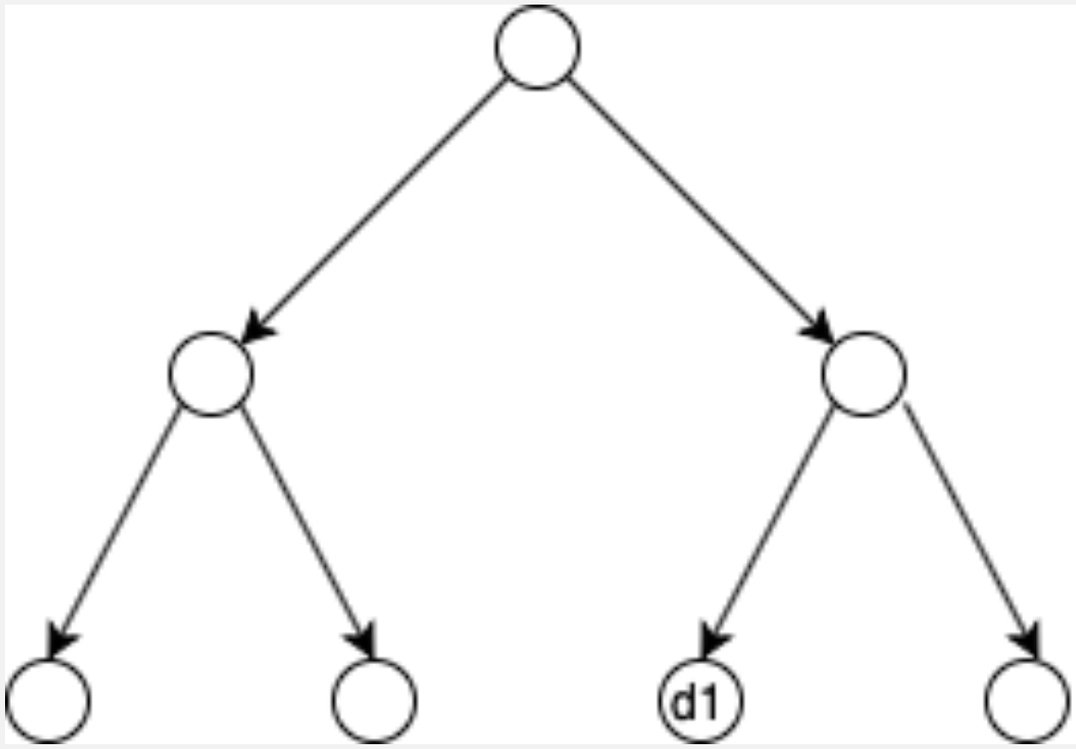
VirtualDOM

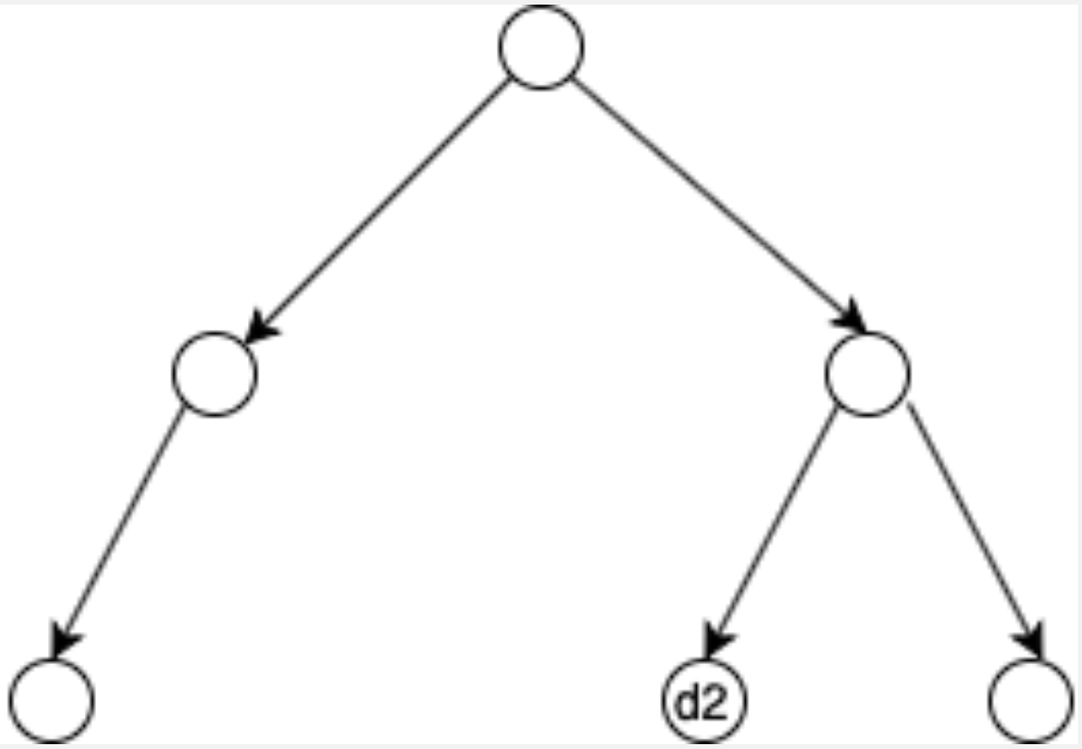
Virtual DOM

- JS object представляющий двойника browser DOM
- Очень быстрый, по сравнению с browser DOM
- Может создавать более 200.000 узлов в секунду
- Создается ПОЛНОСТЬЮ С НУЛЯ при каждом изменении состояния приложения

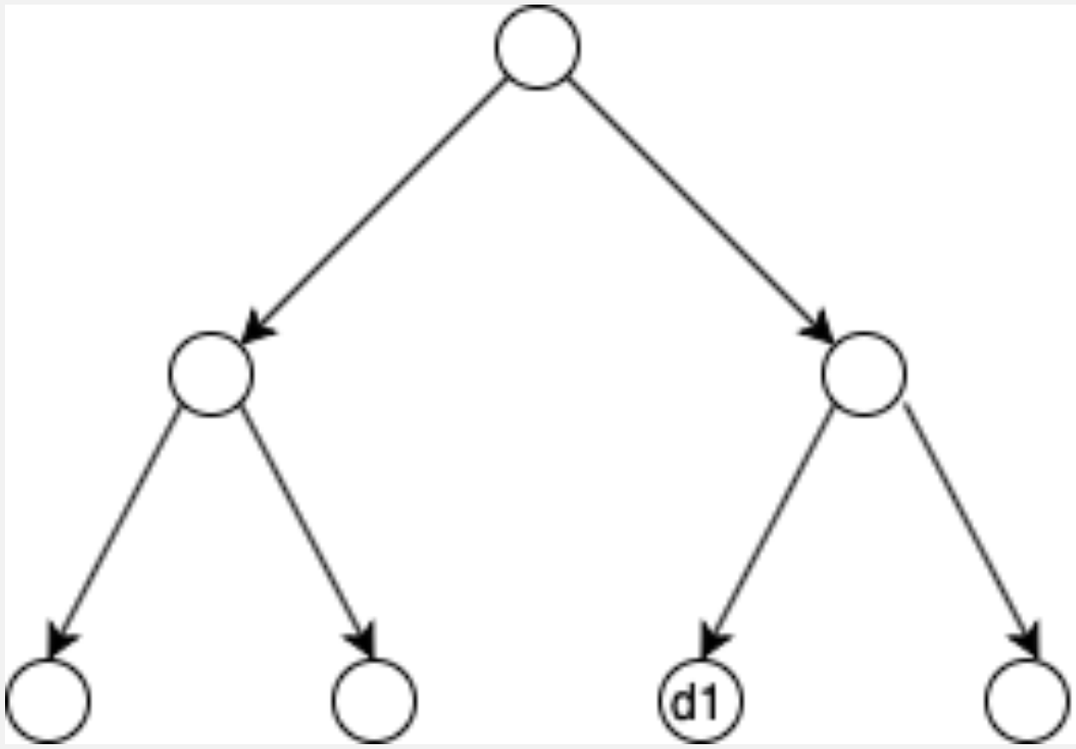
Current state



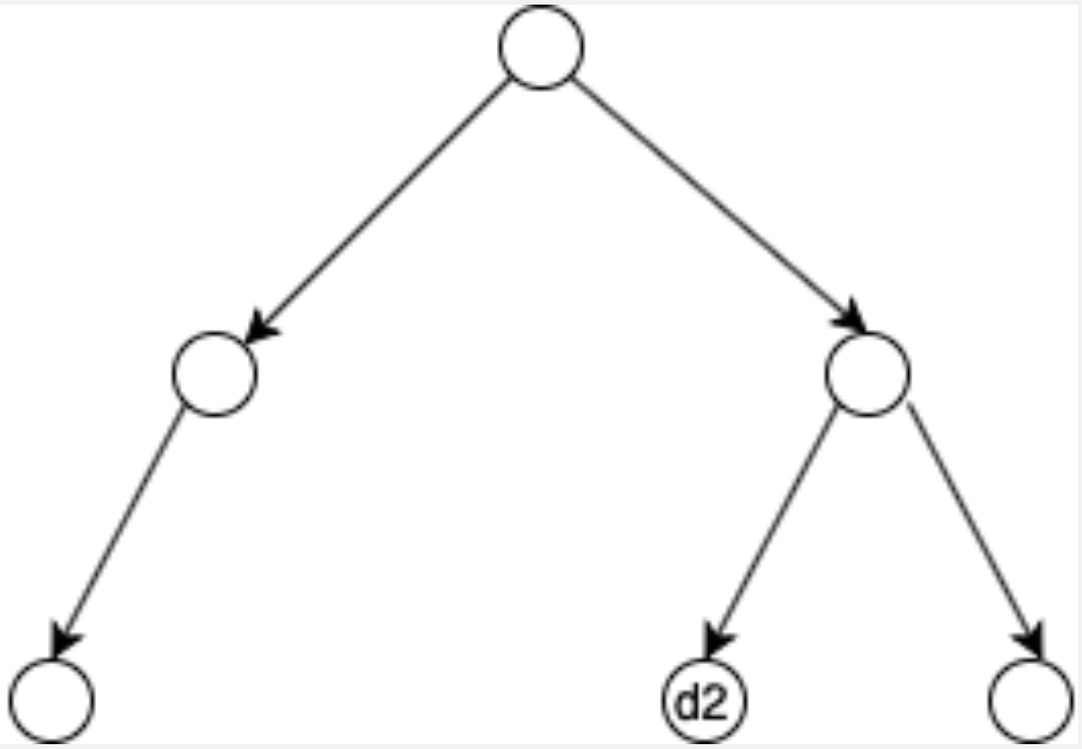
Work in progress state



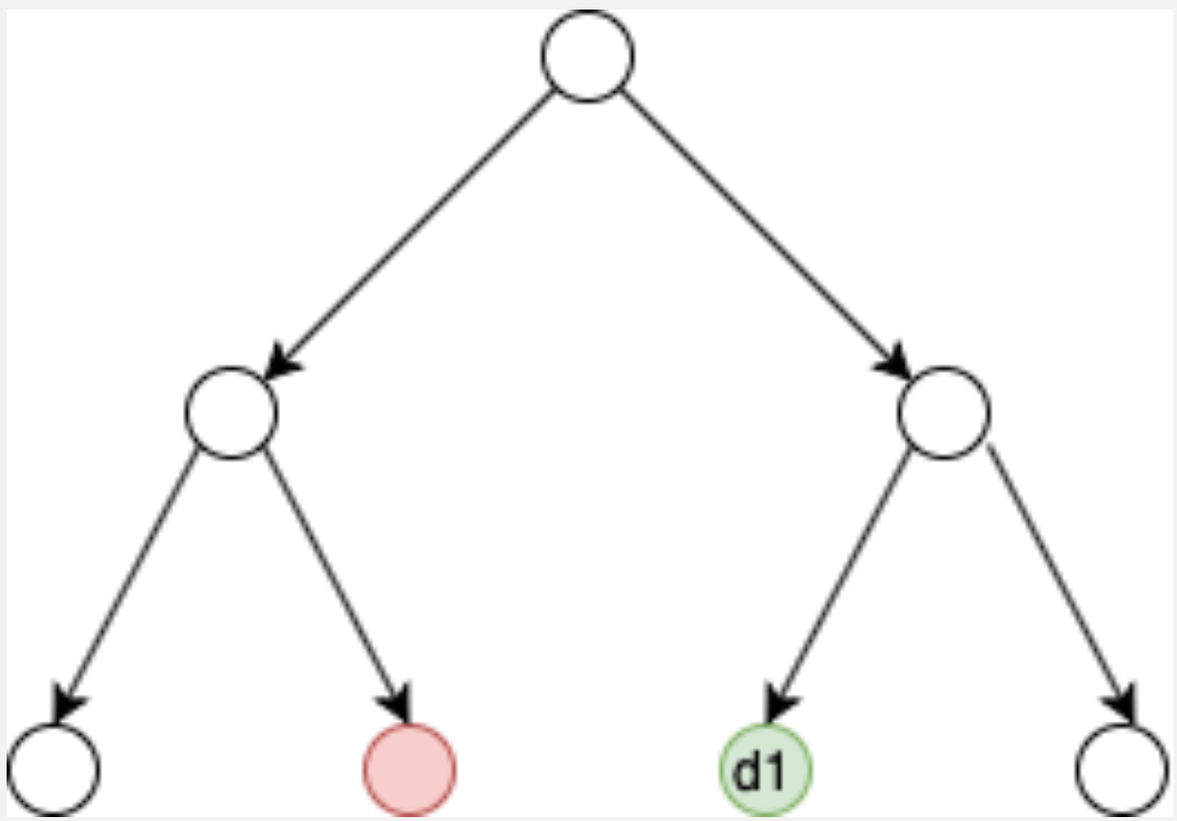
Current state



Work in progress state



Updates to real DOM



Трансформация одного дерева в другое занимает O(n^3). Реакт делает это за O(n) основываясь на двух предположениях.

- Два элемента с разными типами произведут разные поддеревья
- Разработчик может указать, какие элементы остаются стабильными между рендерами с помощью key

```
render() {
  return items.map(item => <div key={item.id}>{item.data}</div>)
}
```

- Два элемента с разными типами произведут разные поддеревья

```
// render 1
<div>
  <MyClassComponentWithState />
</div>

// render 2
<span>
  <MyClassComponentWithState />
<span>
```

MyClassComponentWithState будет полностью уничтожен и создан заново. Если он имел в стейте данные, отличные от изначальных, то эти данные будут утеряны

- Разработчик может указать, какие элементы остаются стабильными между рендерами с помощью key

```
class List extends React.Component {
  state = {
    data: [1, 2, 3, 4, 5]
  }

  shuffle = () => {
    this.setState({ data: [5, 1, 2, 3, 4] })
  }

  render() {
    return (
      <div>
        {this.state.data.map(it => <MyClassComponent>{it}
</MyClassComponent>)}
        <button onClick={this.shuffle}>shuffle data</button>
      </div>
    )
  }
}
```

- Разработчик может указать, какие элементы остаются стабильными между рендерами с помощью key

```
class List extends React.Component {
  state = {
    data: [1, 2, 3, 4, 5]
  }
  shuffle = () => {
    this.setState({ data: [5, 1, 2, 3, 4] })
  }

  render() {
    return (
      <div>
        {this.state.data.map(it => <MyClassComponent>{it}
</MyClassComponent>)}
        <button onClick={this.shuffle}>shuffle data</button>
      </div>
    )
  }
}
```

После нажатия на кнопку КАЖДЫЙ MyClassComponent будет удален и создан вместо него новый, тк реакт сравнивает «первый» с «первым», «второй» со «вторым» и тд. Любой внутри них будет утерян

- Разработчик может указать, какие элементы остаются стабильными между рендерами с помощью key

```
class List extends React.Component {
  state = {
    data: [1, 2, 3, 4, 5]
  }
  shuffle = () => {
    this.setState({ data: [5, 1, 2, 3, 4] })
  }

  render() {
    return (
      <div>
        {this.state.data.map(it => <MyClassComponent key={it}>{it}
</MyClassComponent>)}
        <button onClick={this.shuffle}>shuffle data</button>
      </div>
    )
  }
}
```

Сравниваться будут элементы с одинаковыми ключами. Поэтому MyClassComponent не будут удалены, их стейт сохранится

JavaScript

JavaScript

What are you?

I am

- Single threaded
- Non-blocking
- Asynchronous
- Concurrent

language

I have

- a call stack
- an event loop
- a callback queue
- and other APIs

JS

I am

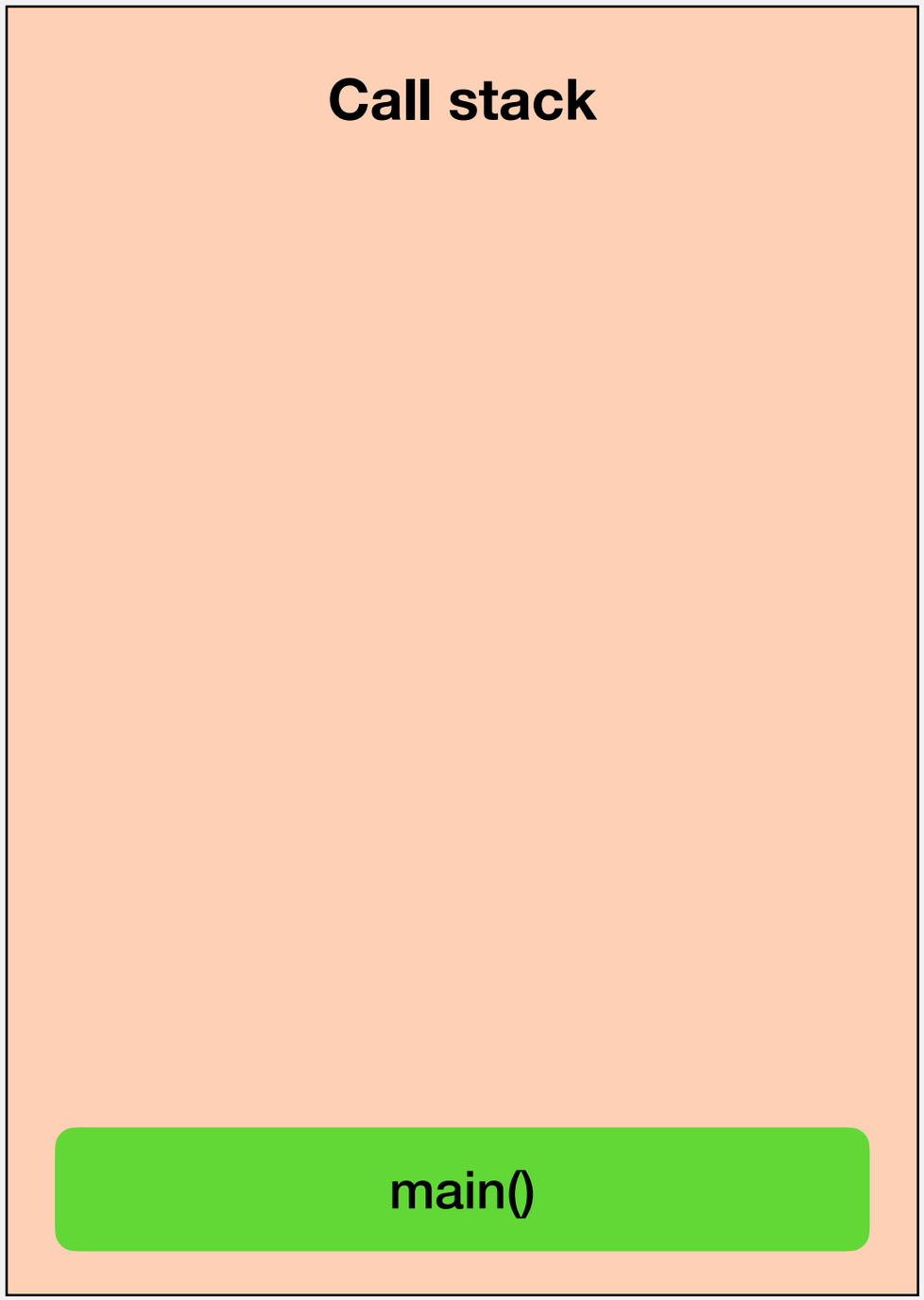
- **Single threaded**
- Non-blocking
- Asynchronous
- Concurrent

language

JS

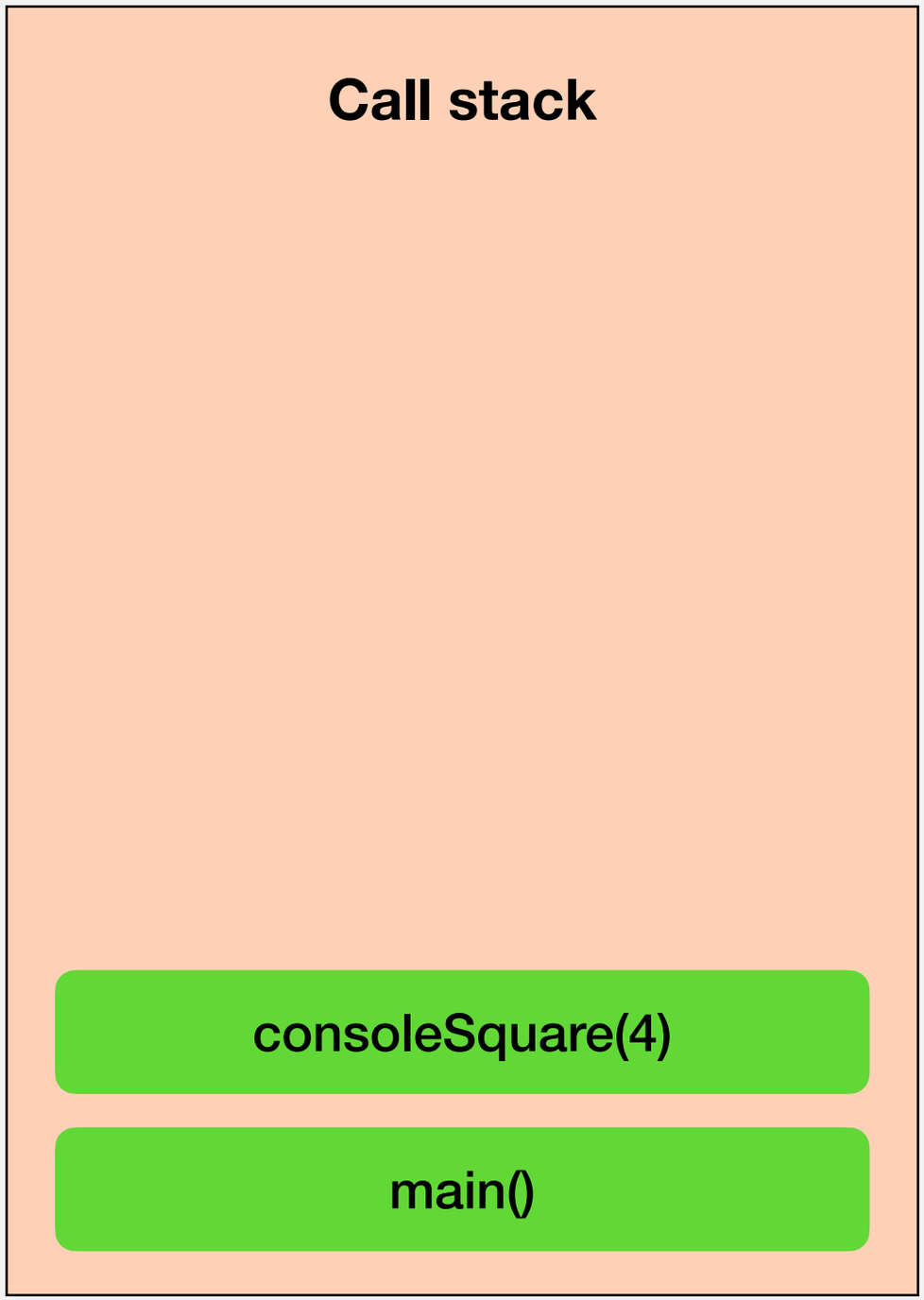
One thread ==
one call stack ==
one thing at time

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



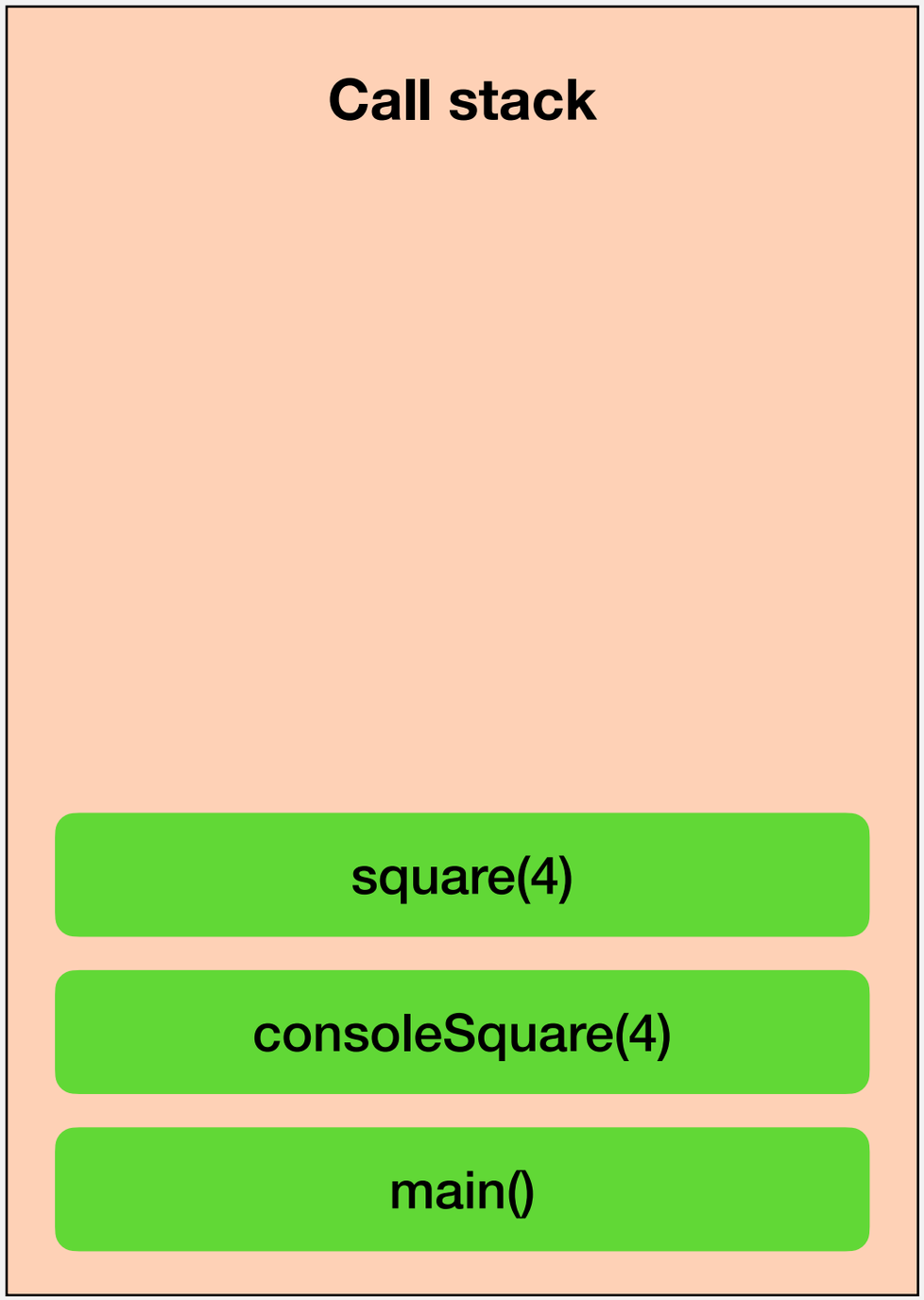
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



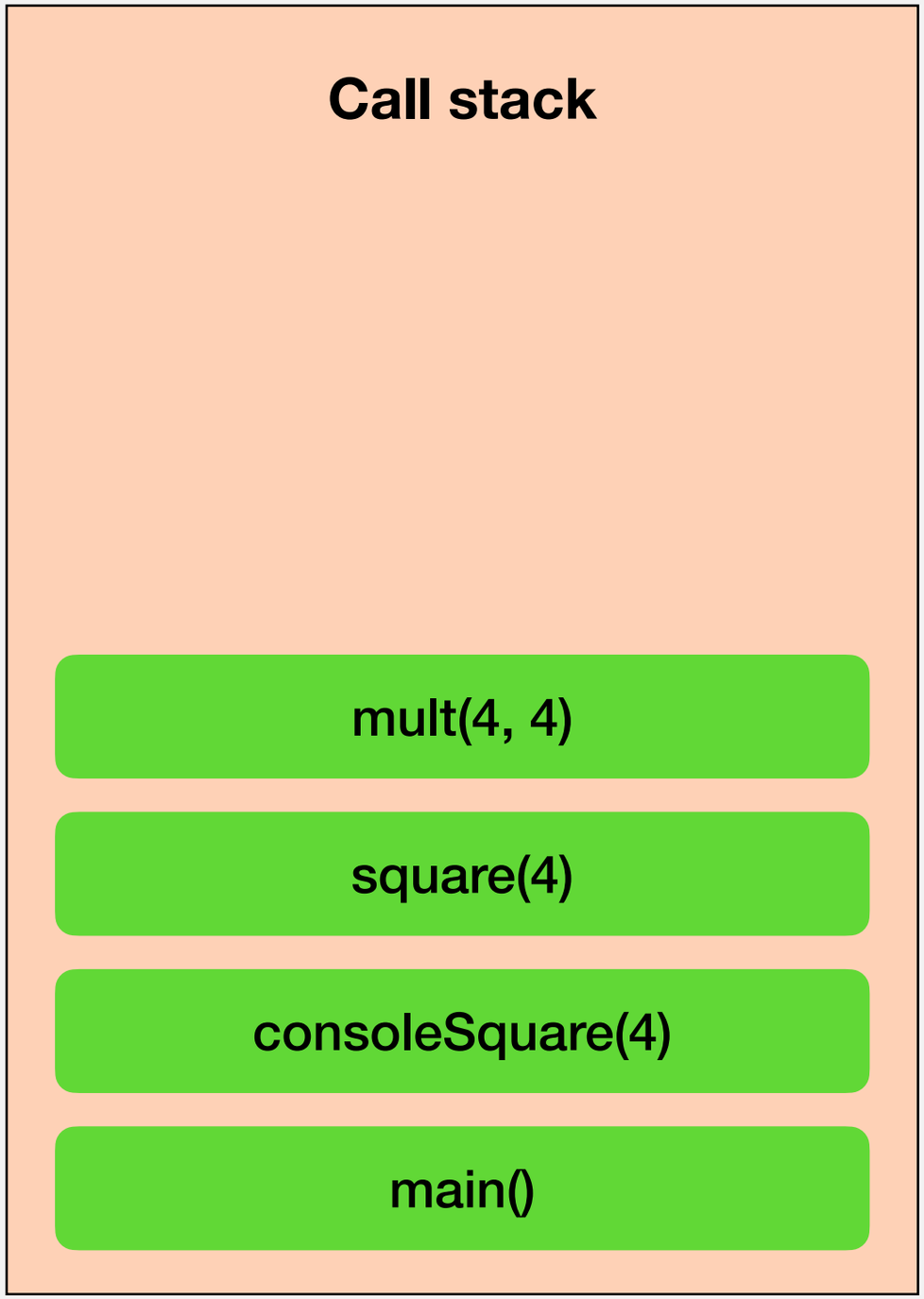
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



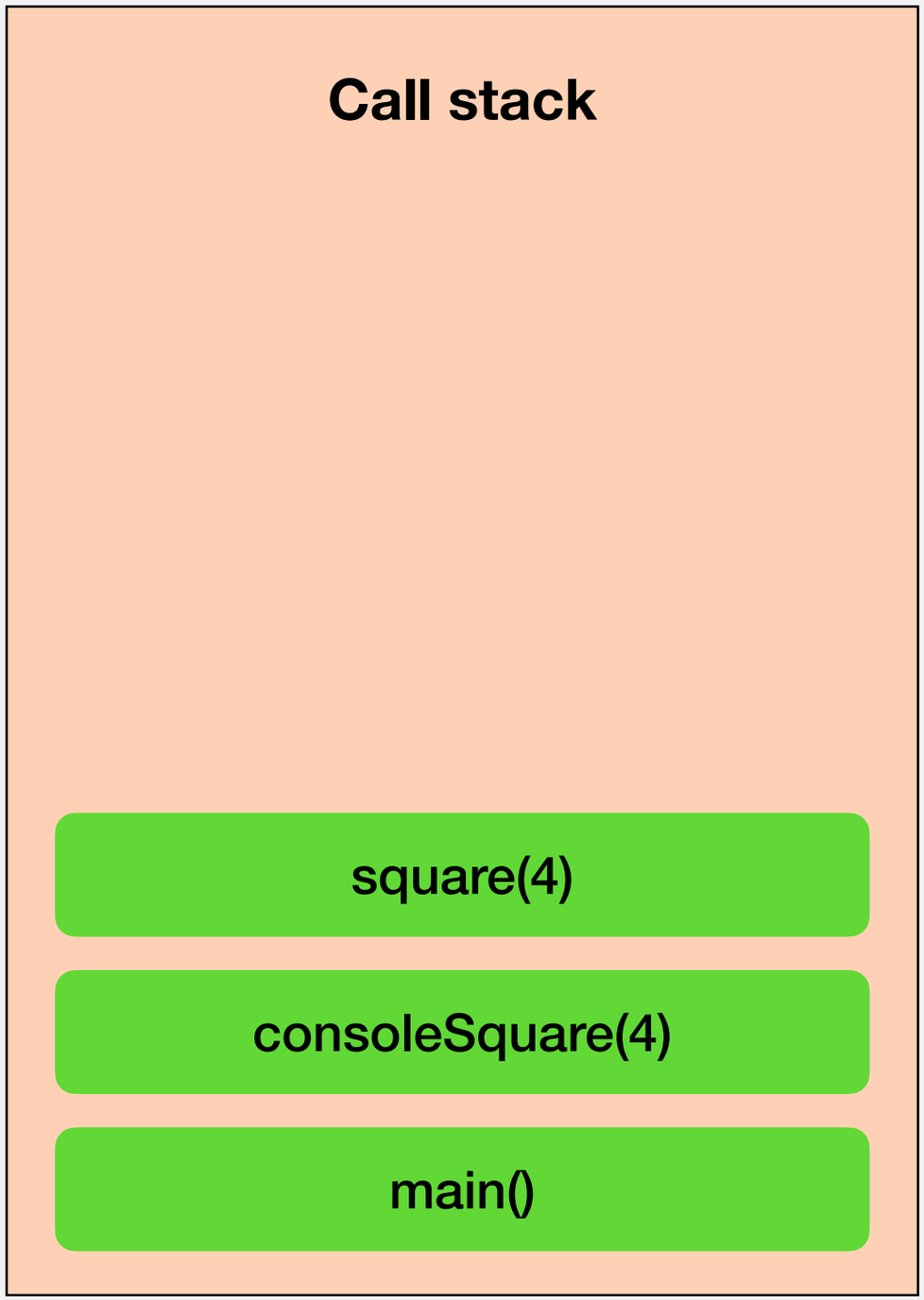
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



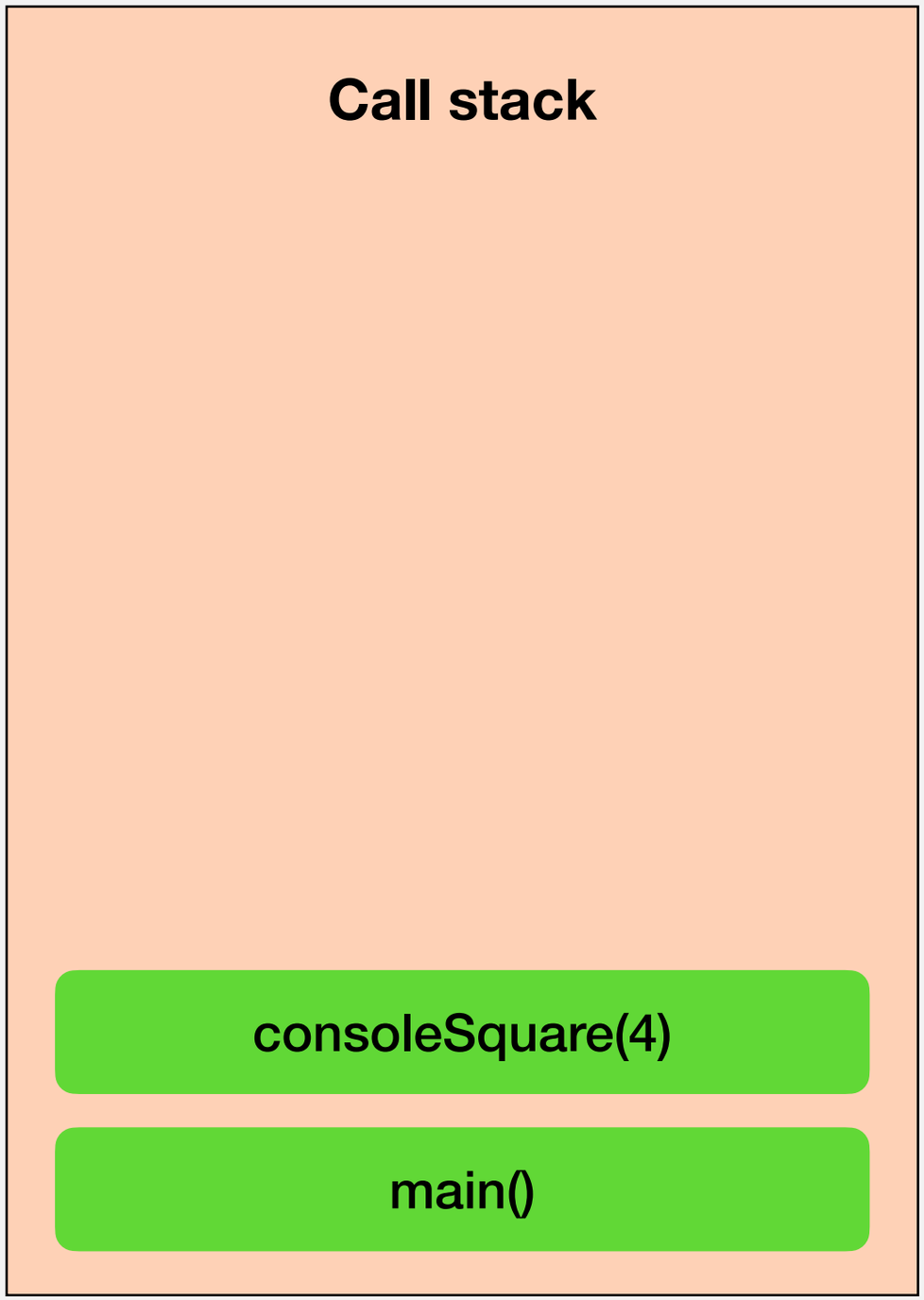
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



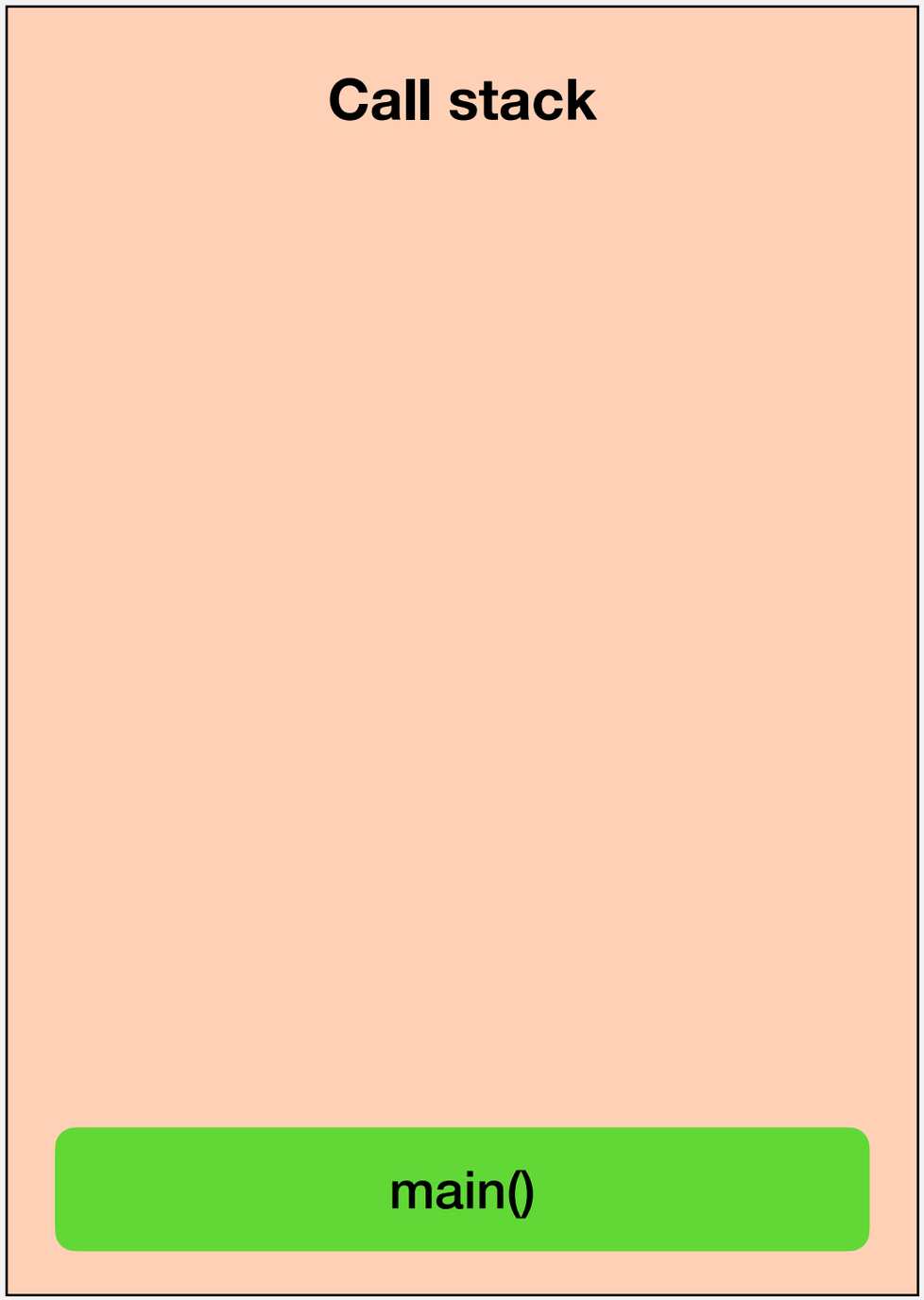
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



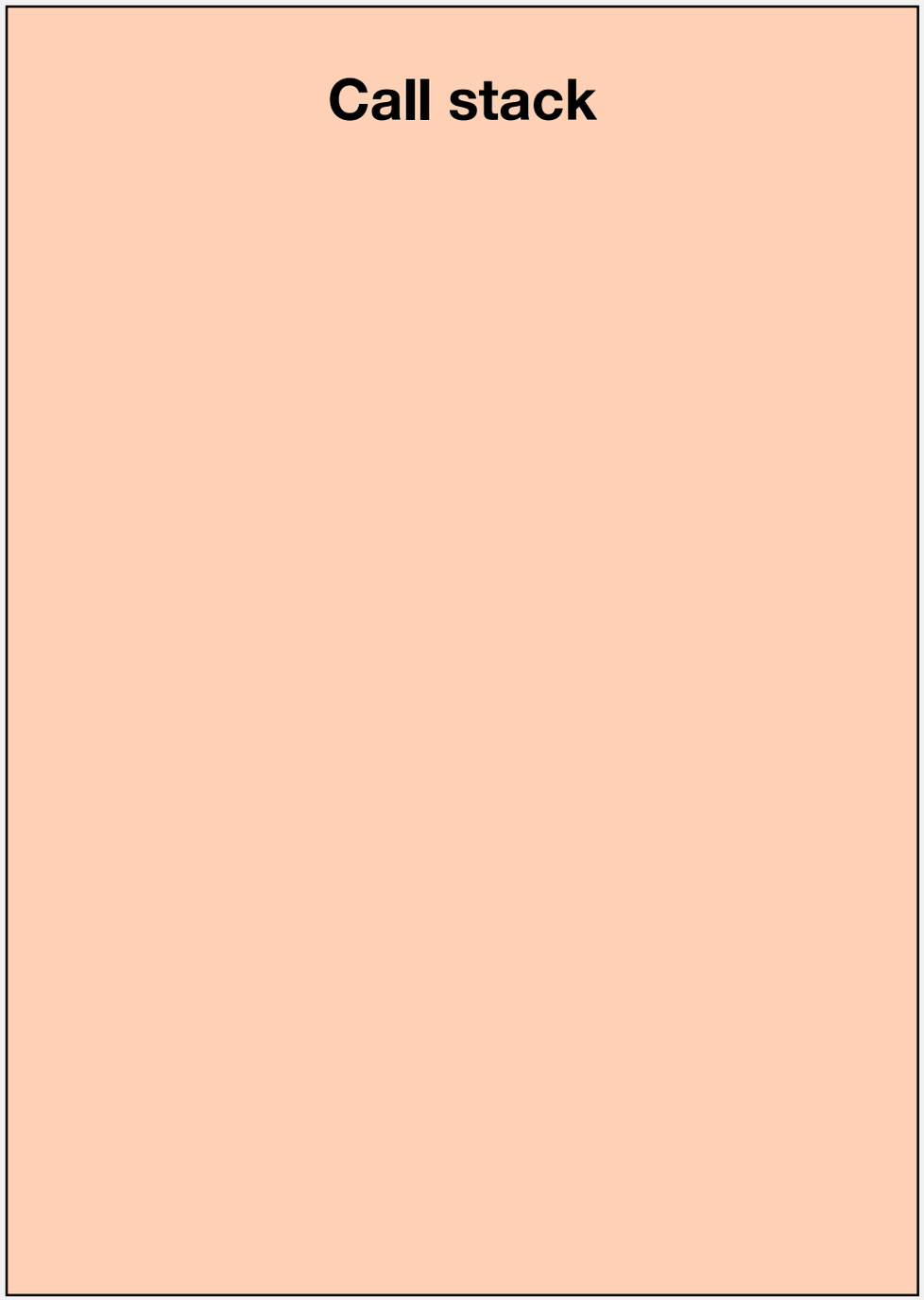
JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```



JS

```
function mult(a, b) {
  return a * b;
}


function square(n) {
  return mult(n, n);
}

function consoleSquare(n) {
  const squared = square(n);
  console.log(squared);
}

consoleSquare(4);
```

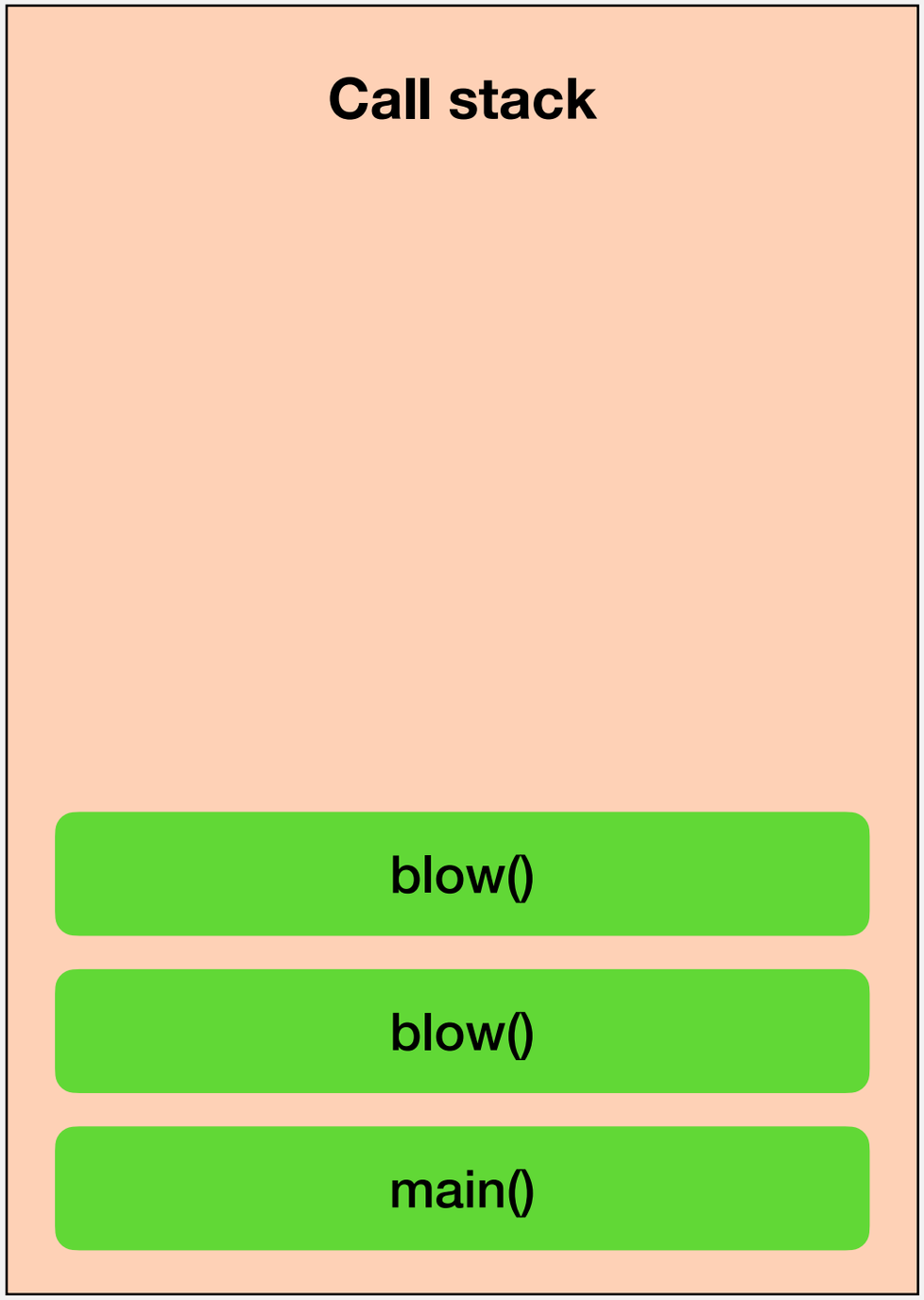


JS

```
function blow( ) {
  return blow( );
}

blow( );
```

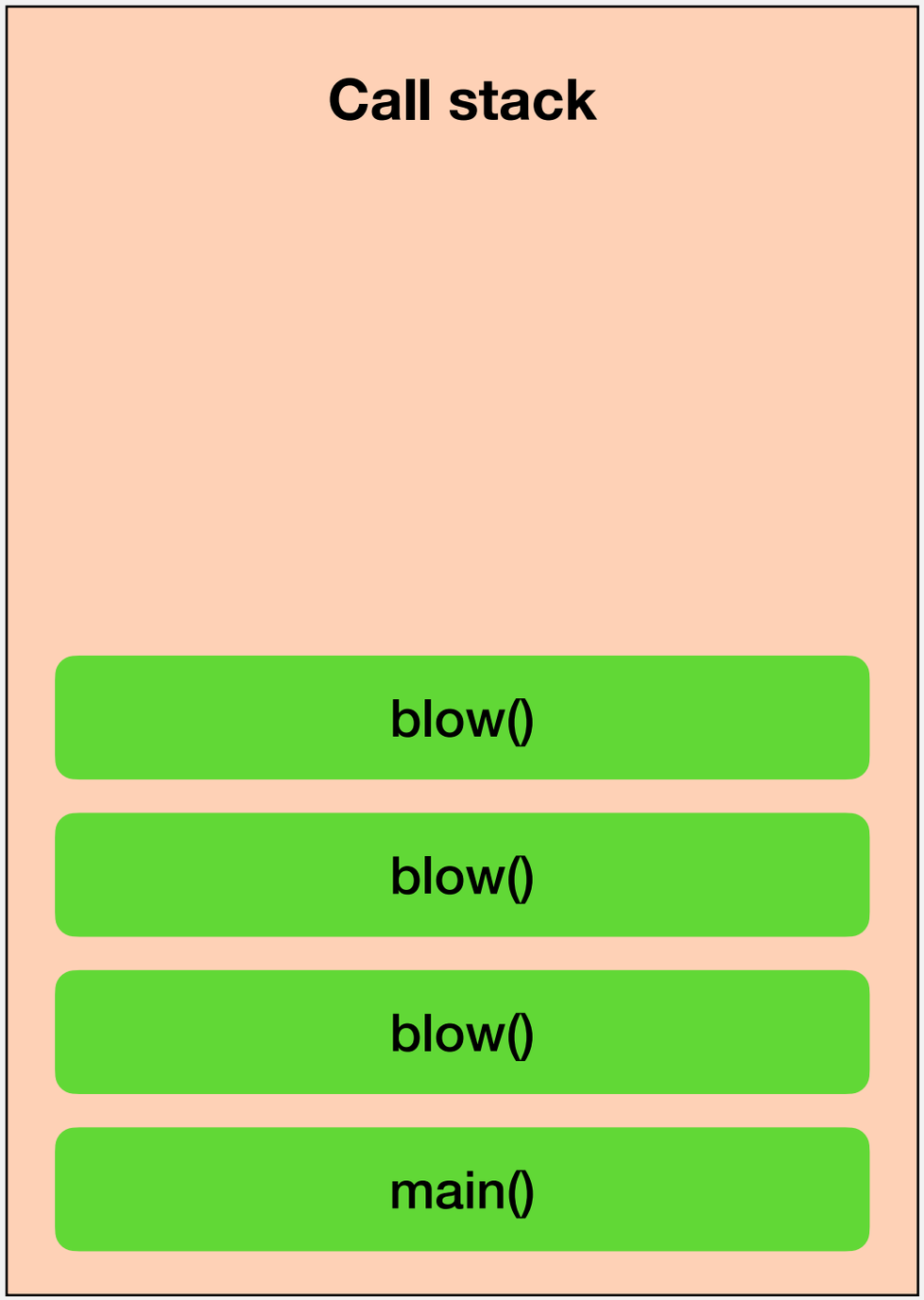
JS

```
function blow( ) {
  return blow( );
}

blow( );
```



JS

```
function blow( ) {
  return blow( );
}

blow( );
```



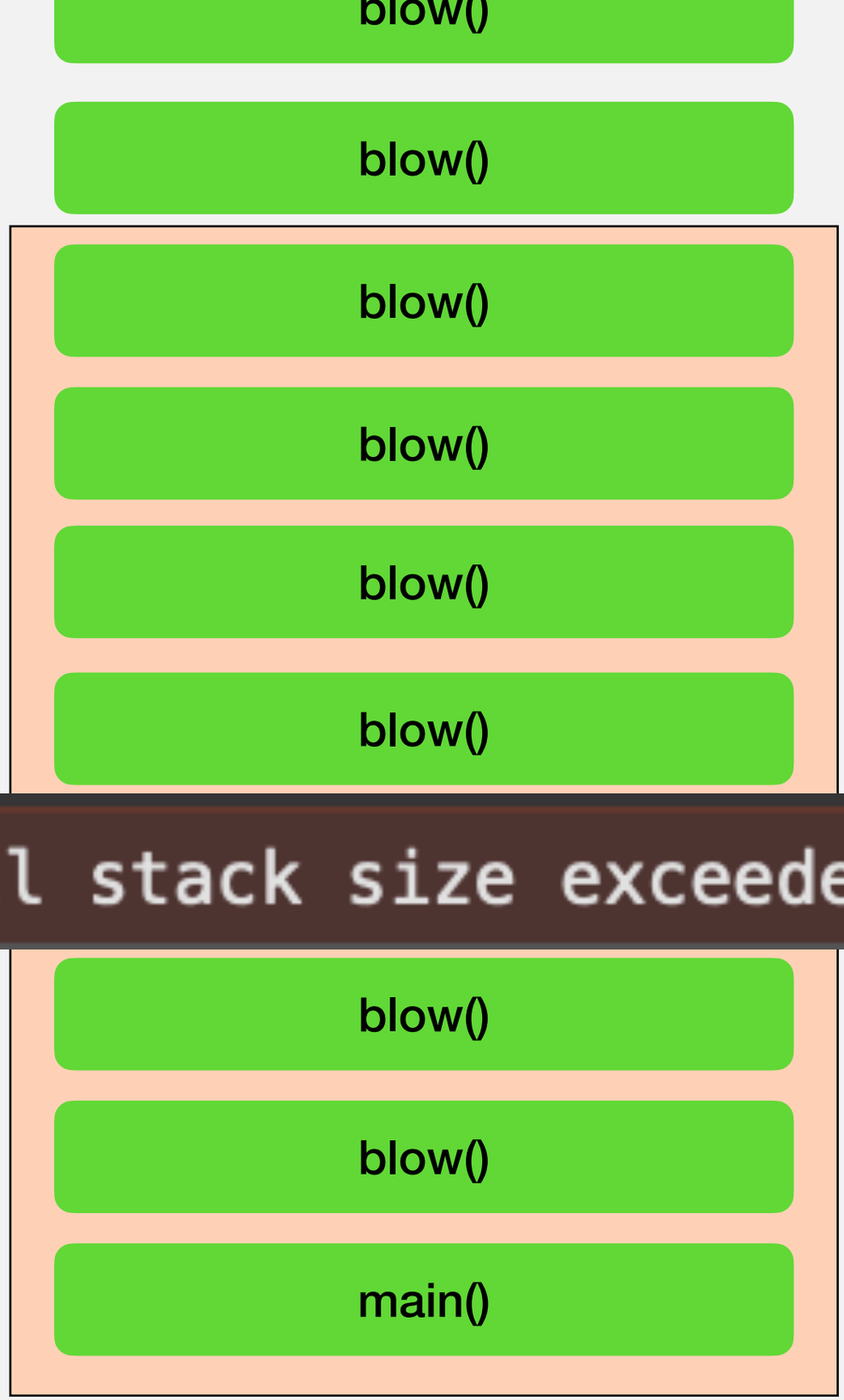
JS

```
function blow( ) {
  return blow( );
}

blow( );
```

JS

```
function blow( ) {
  return blow( );
}

blow( );
```

```
RangeError: Maximum call stack size exceeded.
```



JS

I am

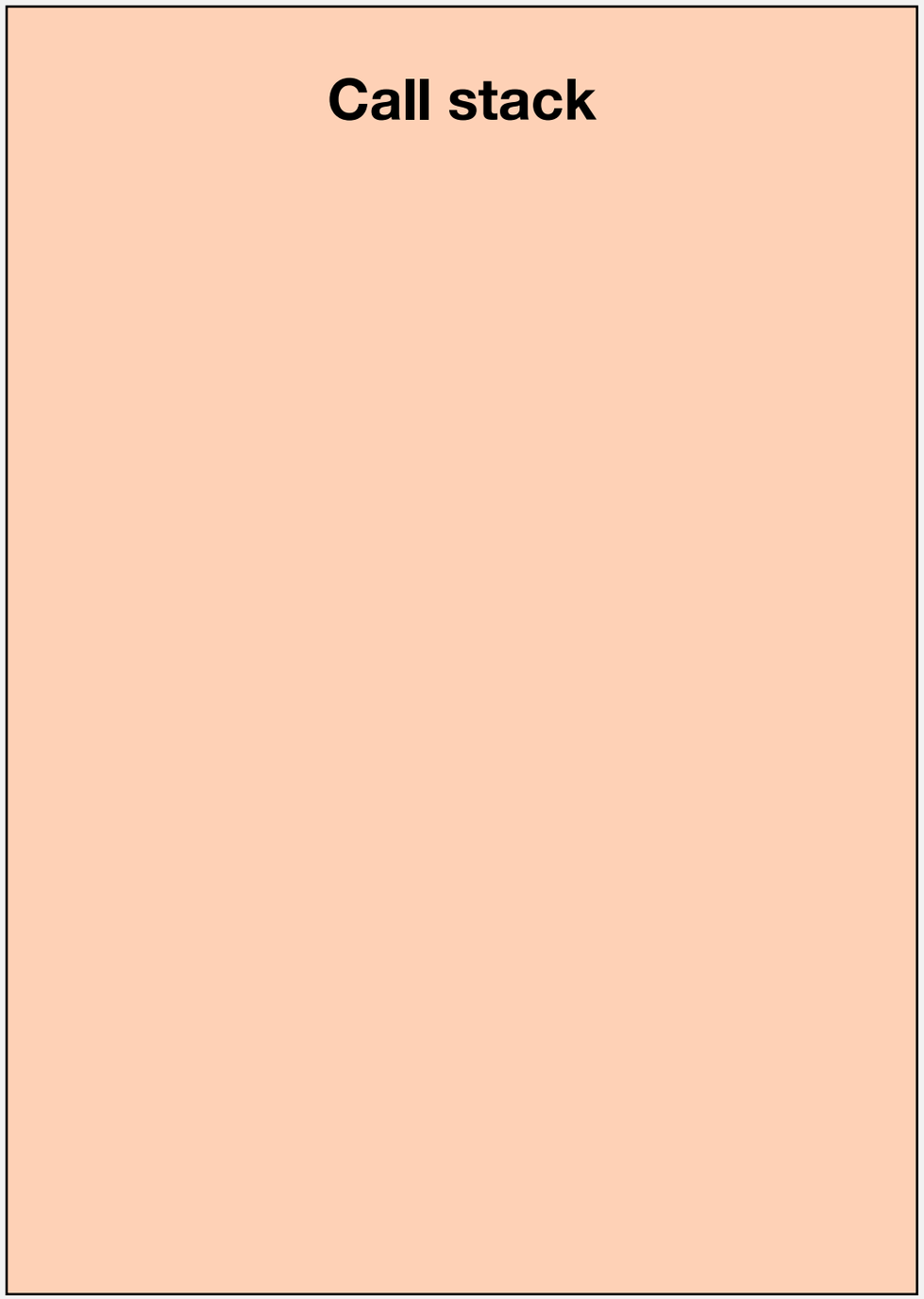
- ~~Single threaded~~
- **Non-blocking**
- **Asynchronous**
- Concurrent

What is blocking?

What is blocking?

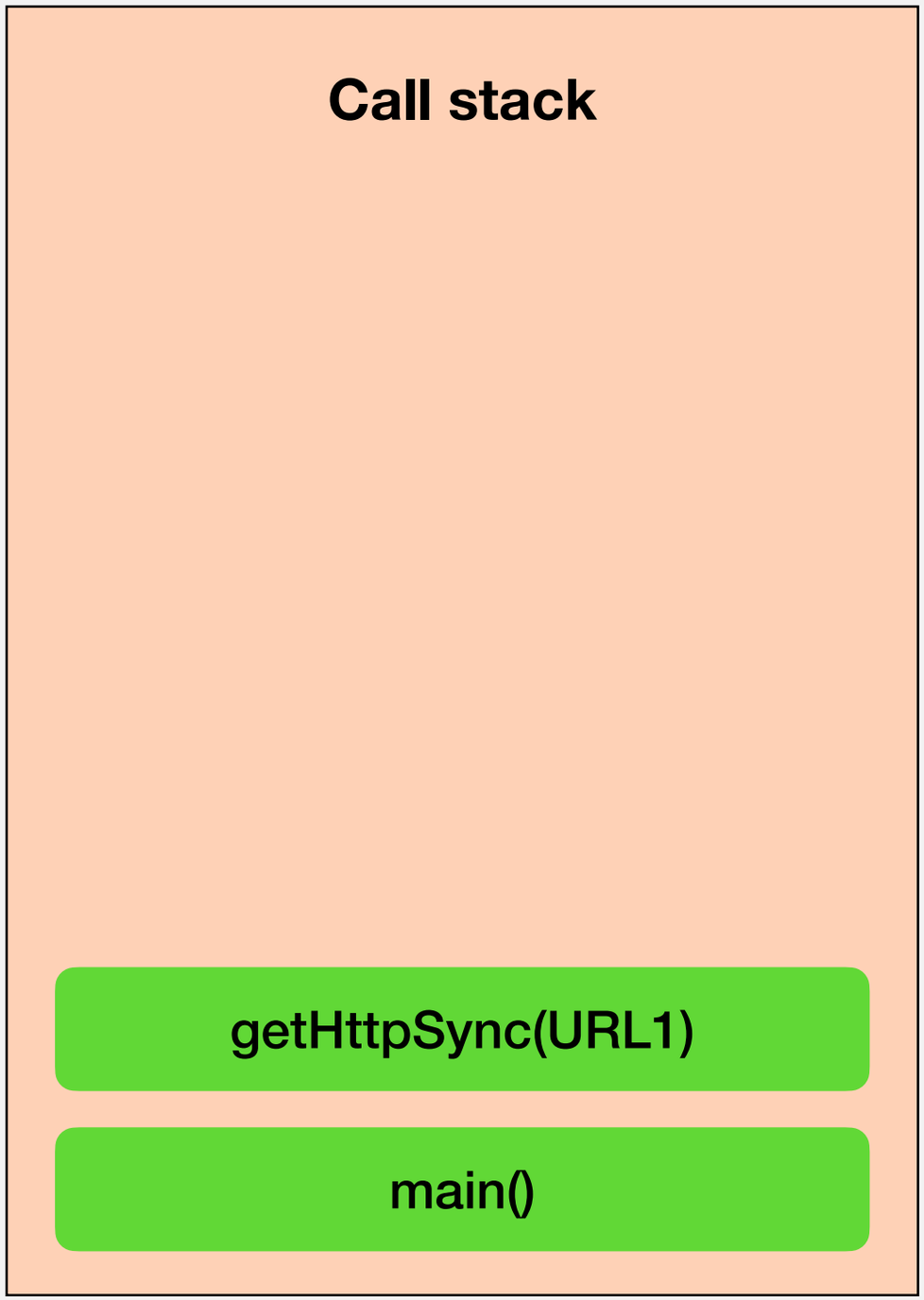
Slow function calls (such as loop from 1 to 1 billion, image processing, networking etc.) on a call stack that block other function calls

```
const foo = getHttpSync(URL1);
const baz = getHttpSync(URL2);
const bar = getHttpSync(URL3);

console.log(foo, baz, bar);
```



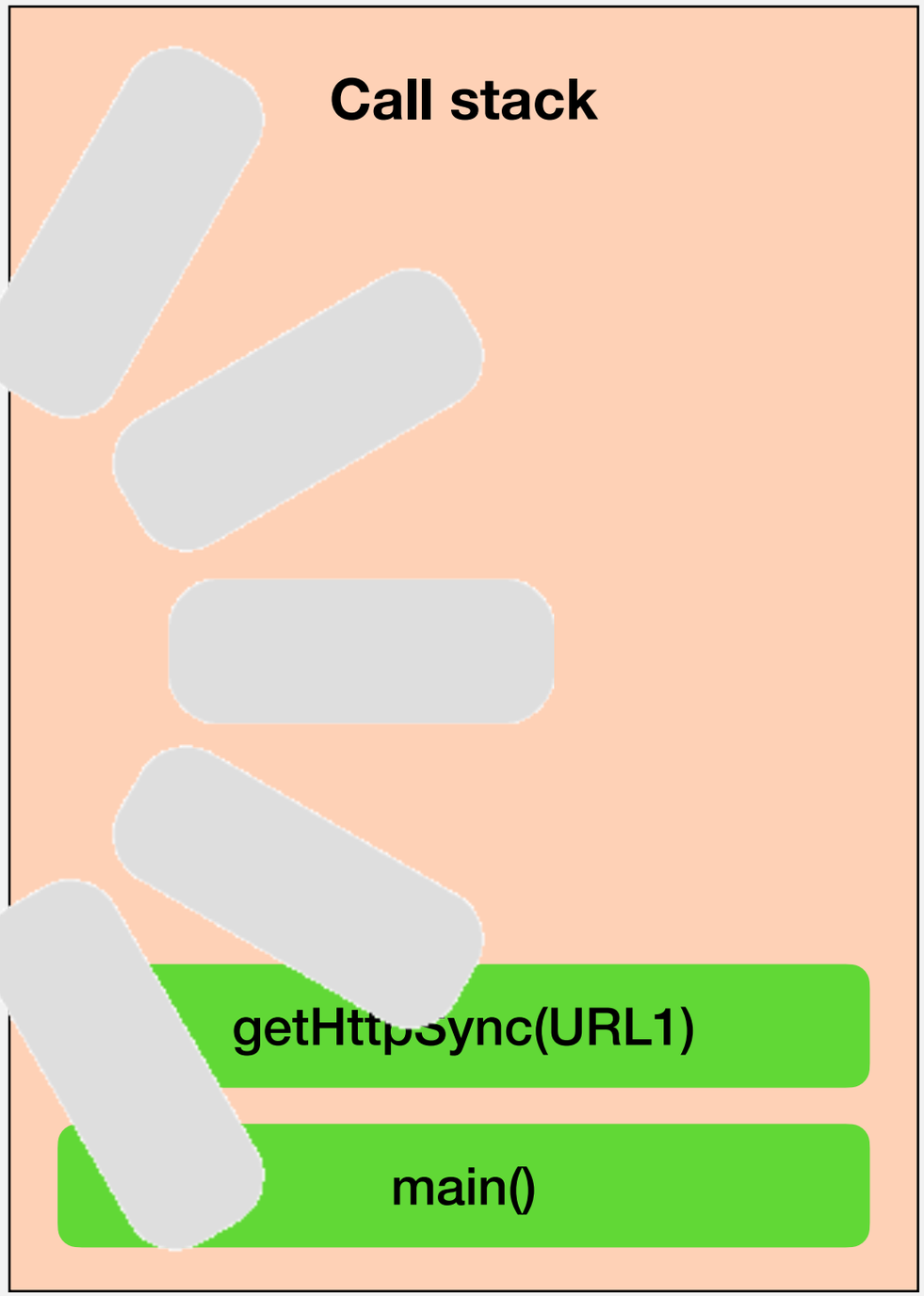
JS

```
const foo = getHttpSync(URL1);
const baz = getHttpSync(URL2);
const bar = getHttpSync(URL3);

console.log(foo, baz, bar);
```



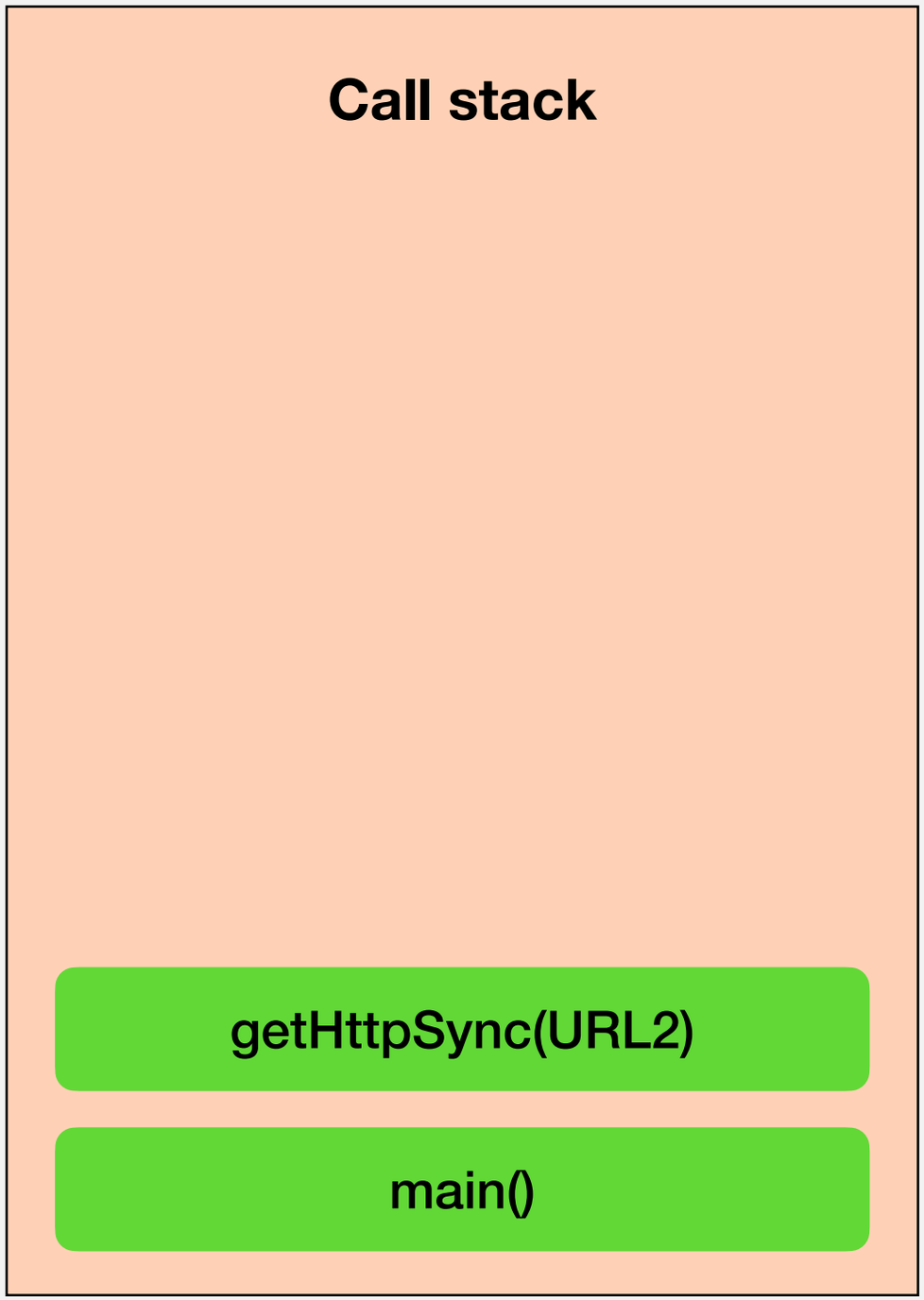
JS

```
const foo = getHttpSync(URL1);
const baz =                c(URL2);
const bar =                c(URL3);

console.log(foo                ar);
```



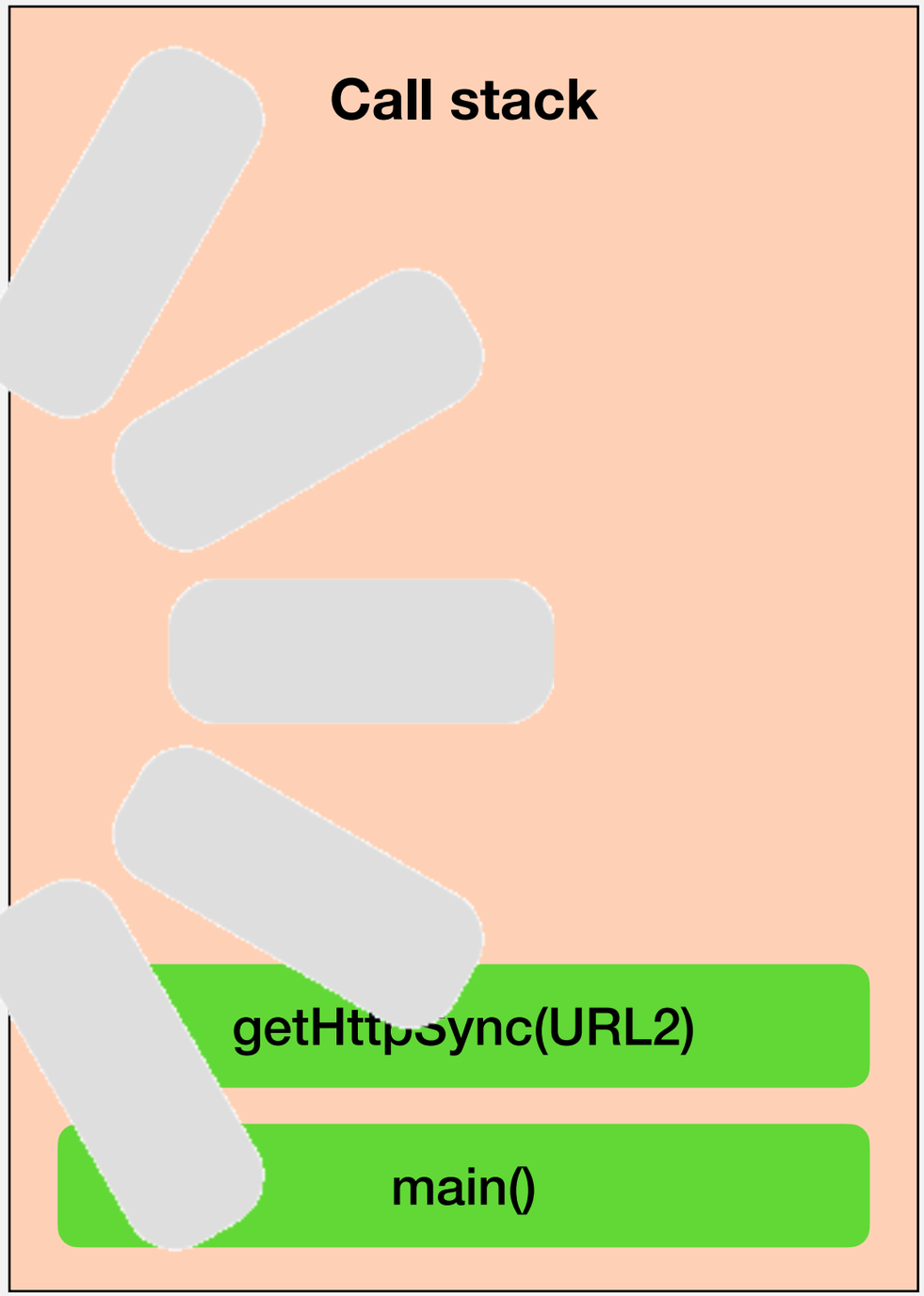
JS

```
const foo = getHttpSync(URL1);
const baz = getHttpSync(URL2);
const bar = getHttpSync(URL3);

console.log(foo, baz, bar);
```



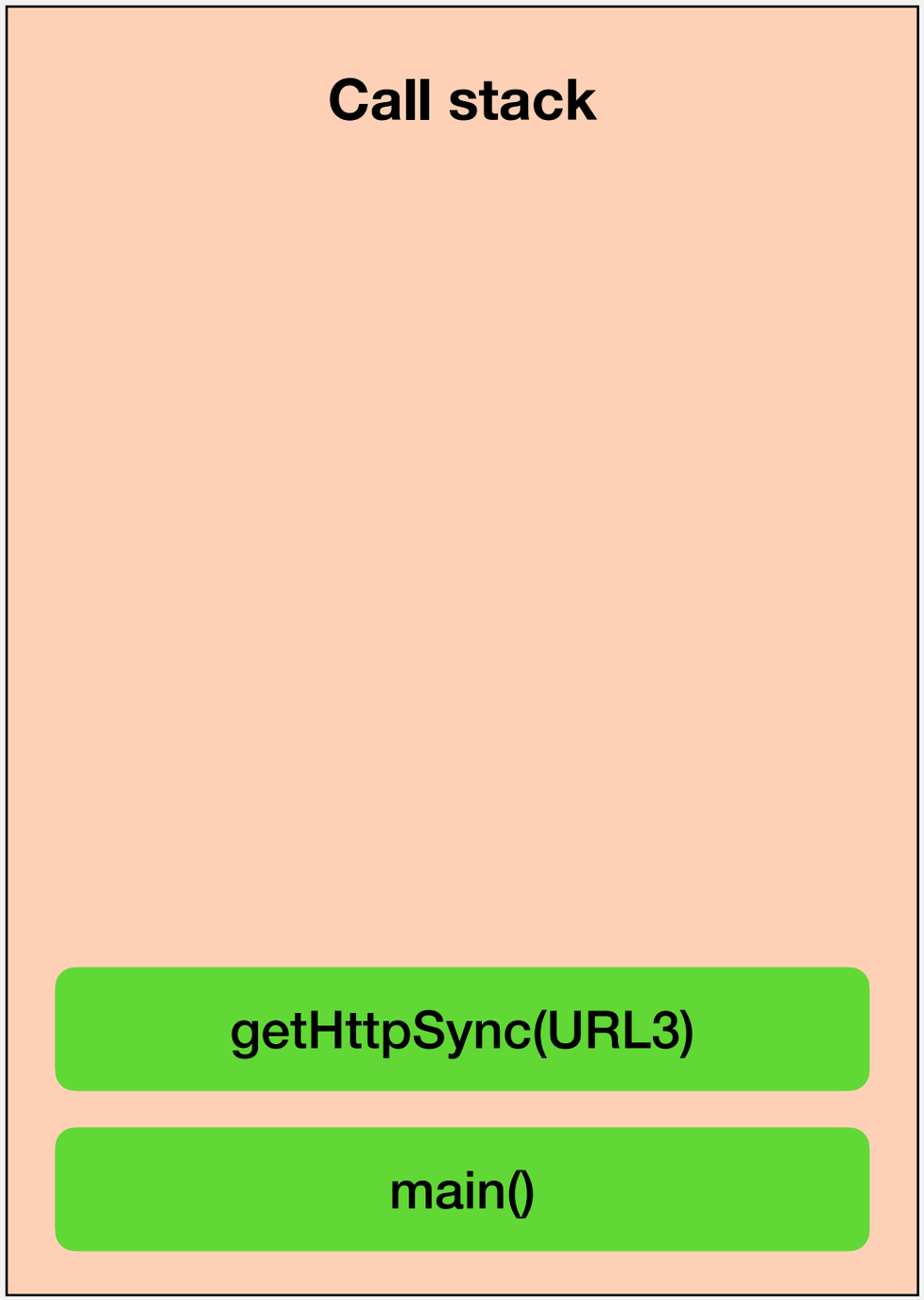
JS

```
const foo = getHttpSync(URL1);
const baz =  c(URL2);
const bar =  c(URL3);

console.log(foo  ar);
```



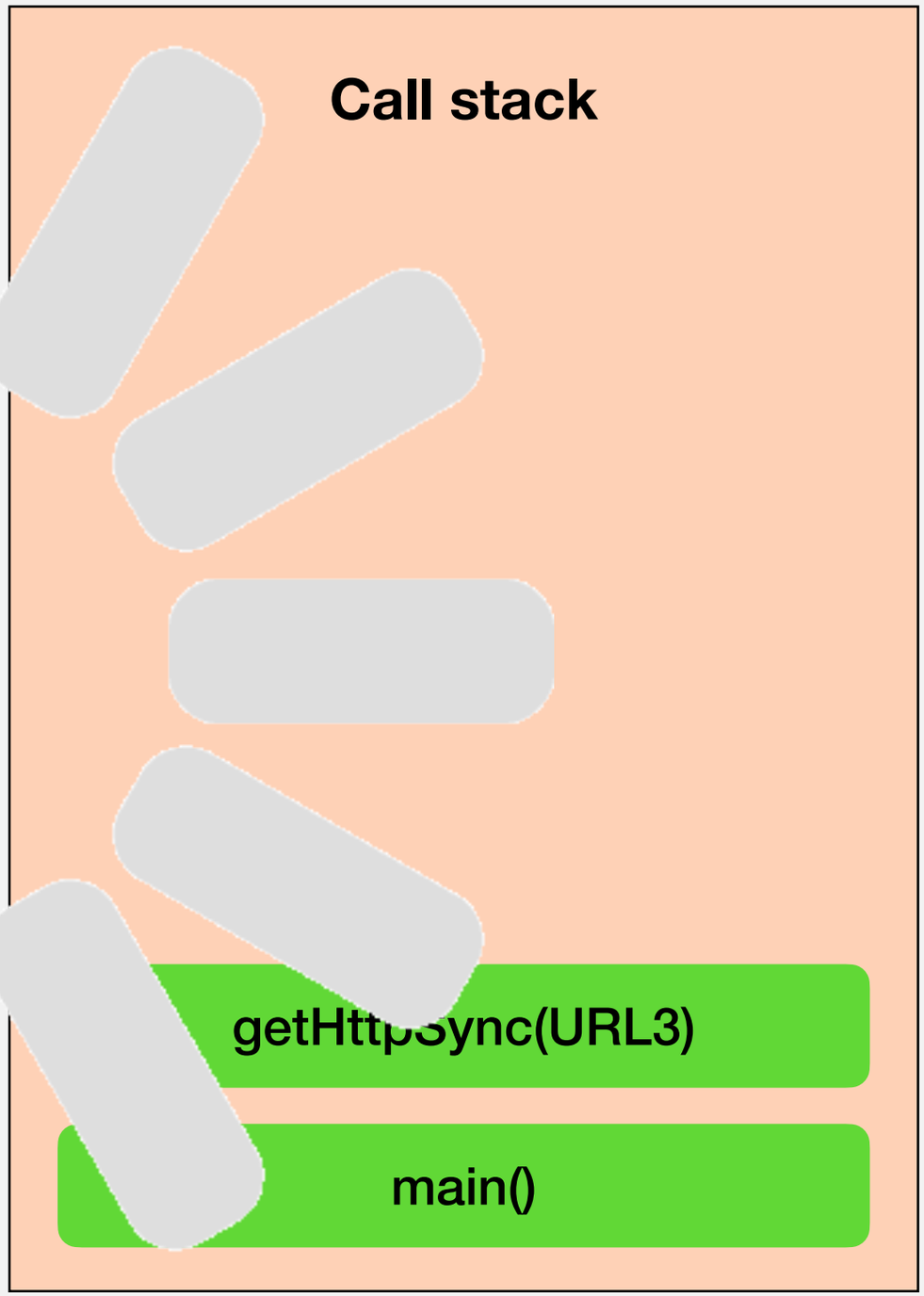
JS

```
const foo = getHttpSync(URL1);
const baz = getHttpSync(URL2);
const bar = getHttpSync(URL3);

console.log(foo, baz, bar);
```



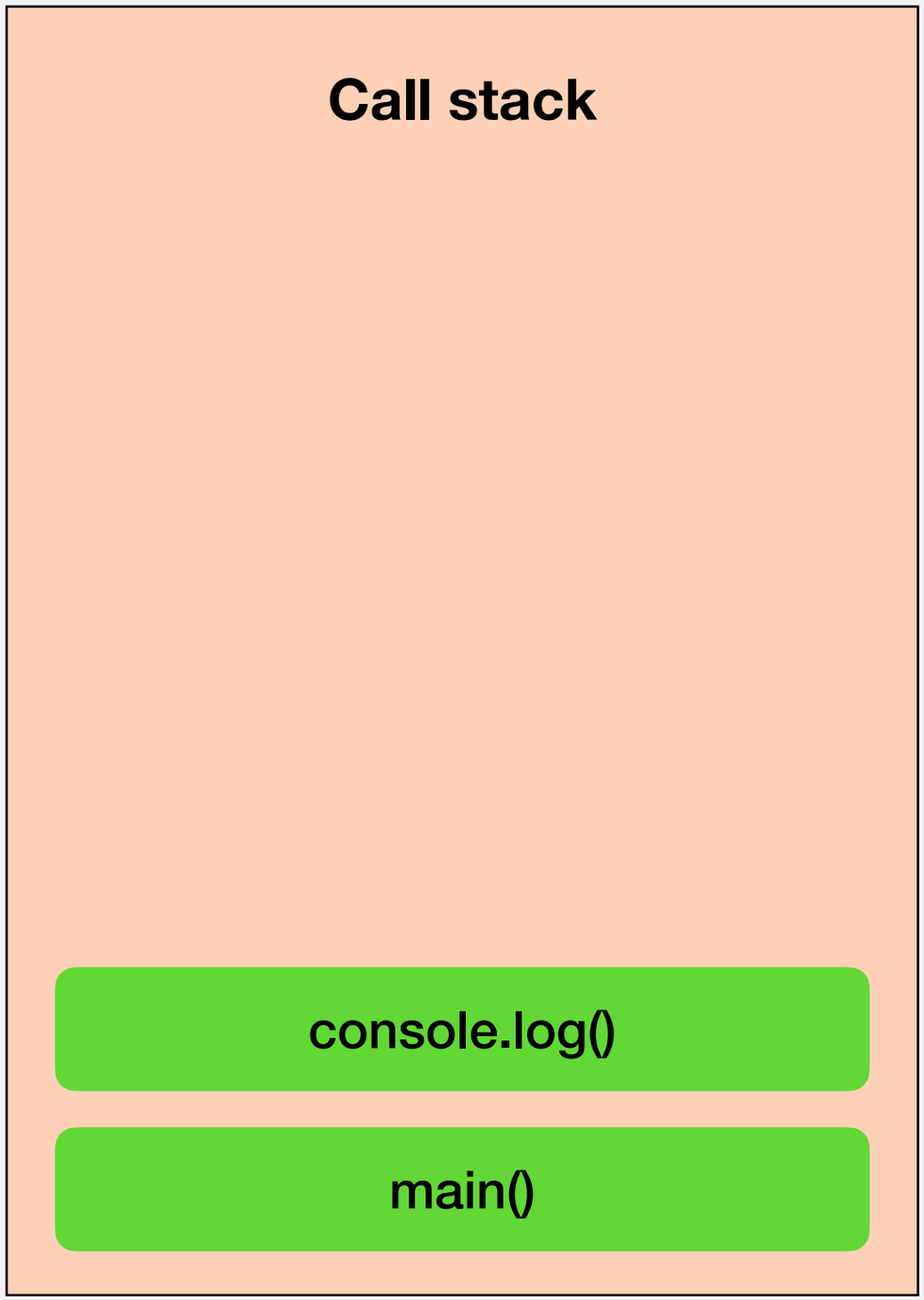
JS

```
const foo = getHttpSync(URL1);
const baz =                c(URL2);
const bar =                c(URL3);

console.log(foo                ar);
```



JS

```
const foo = getHttpSync(URL1);
const baz = getHttpSync(URL2);
const bar = getHttpSync(URL3);

console.log(foo, baz, bar);
```



JS

Solution?

Solution?

Asynchronous callbacks.

...

Call me maybe?

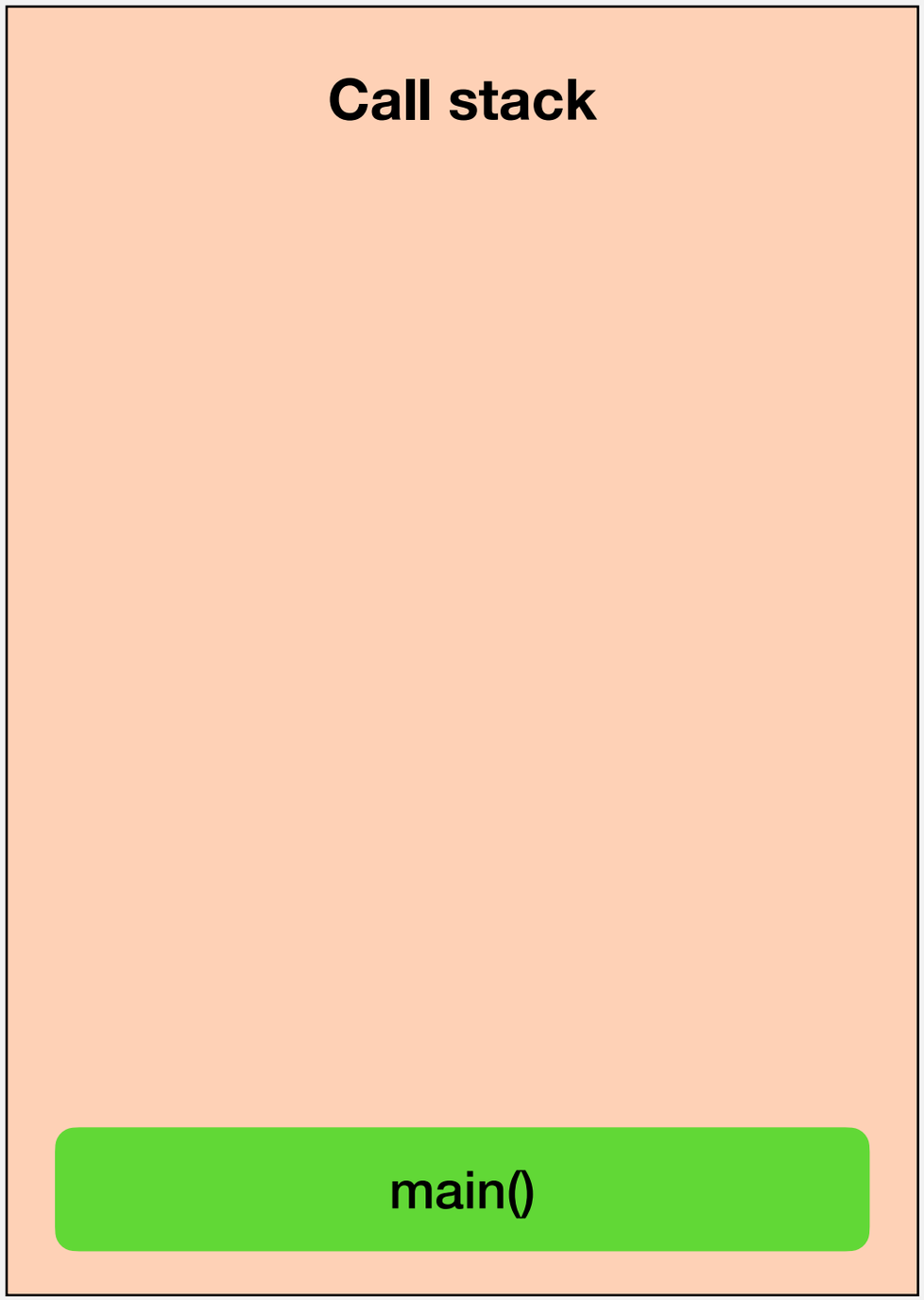
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

Start program
finish program
I am in callback
Second timeout

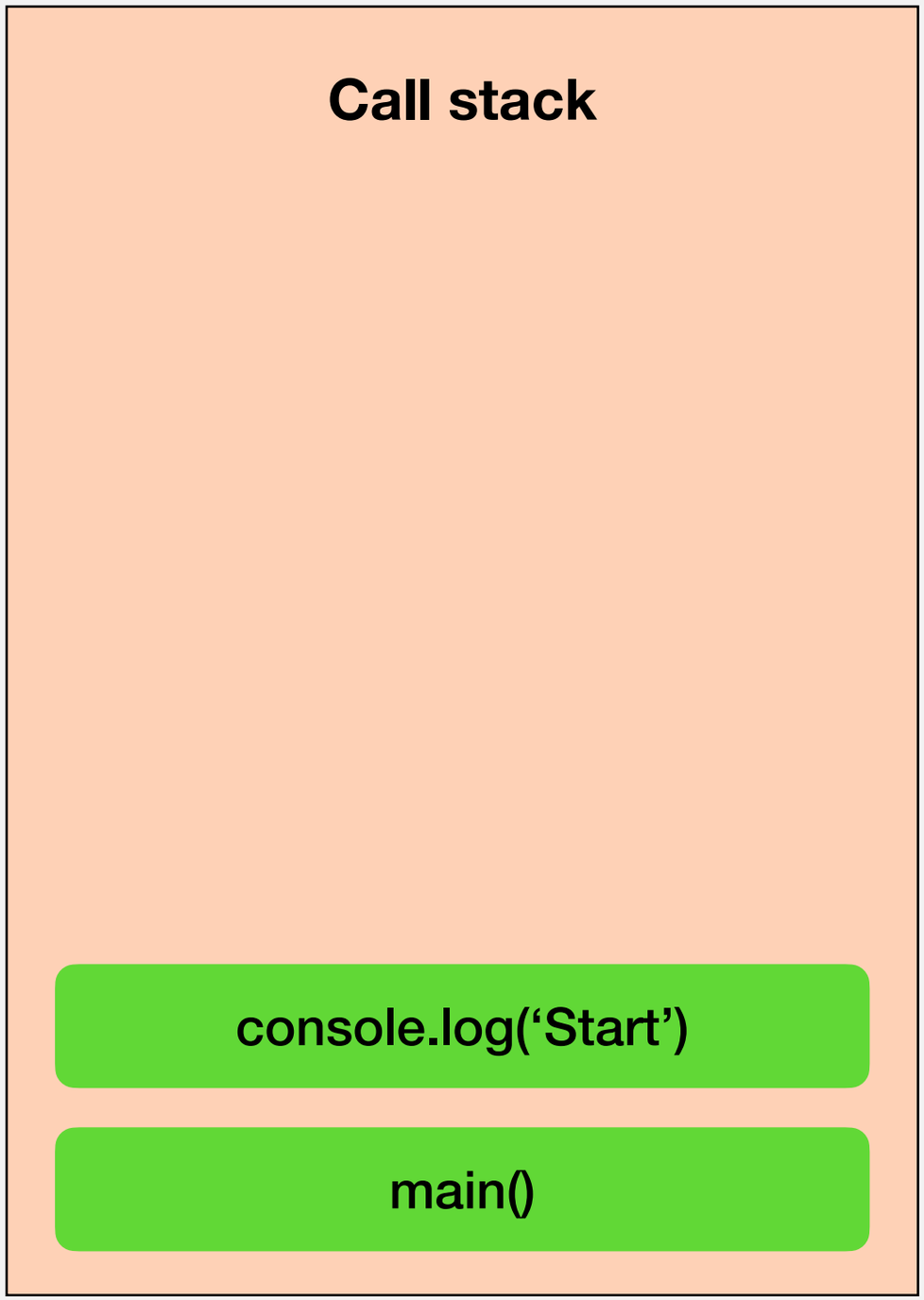
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



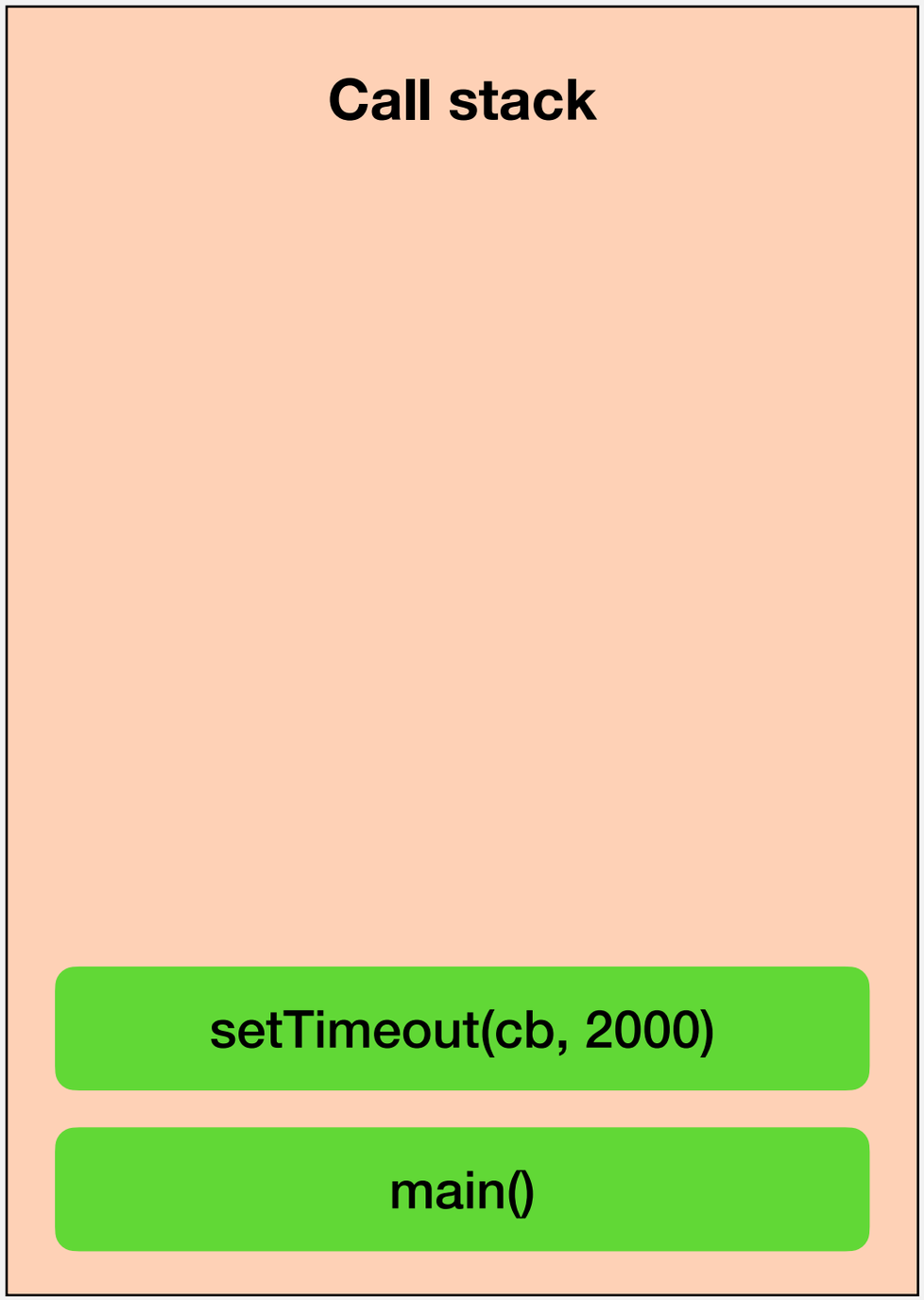
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



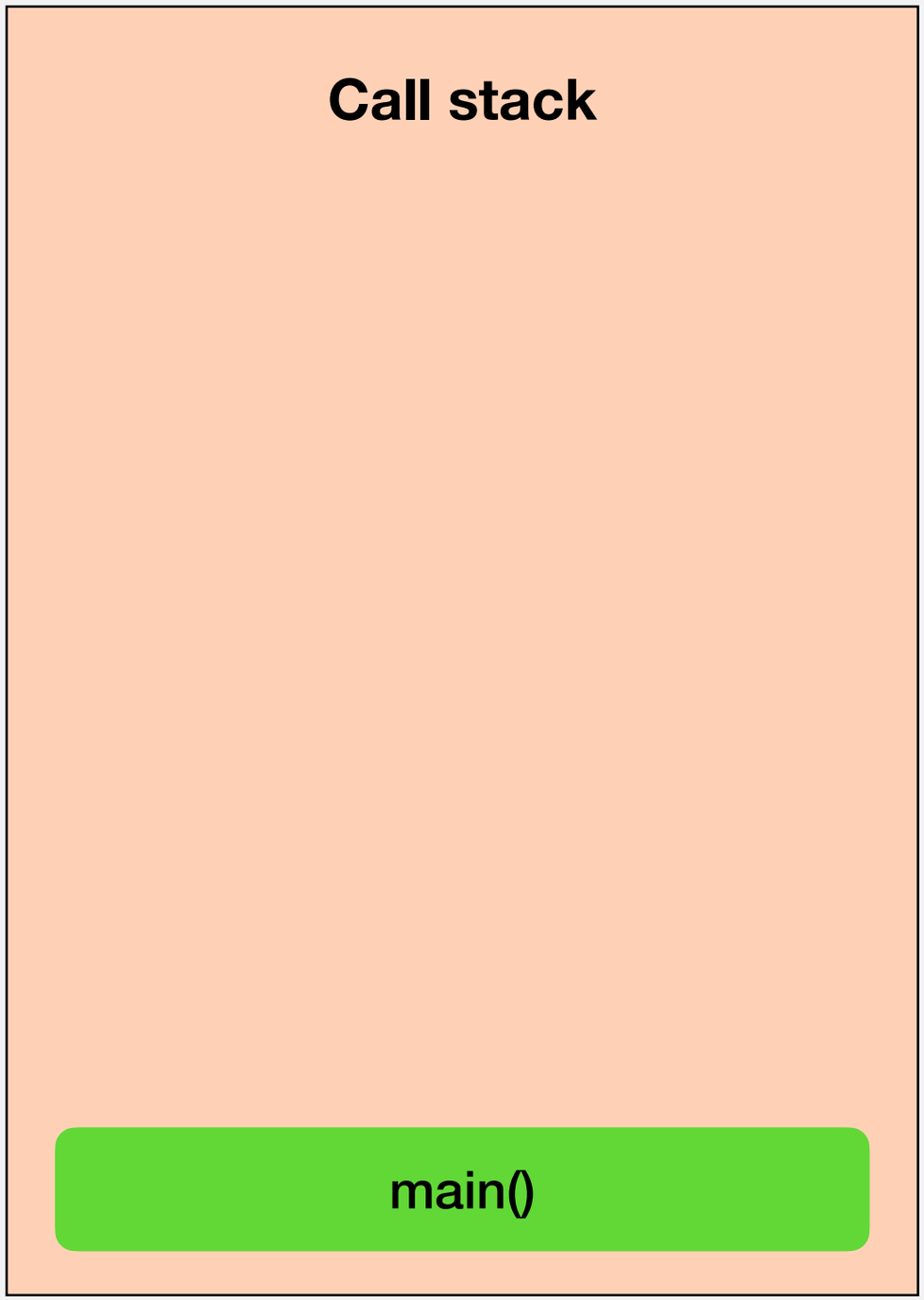
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



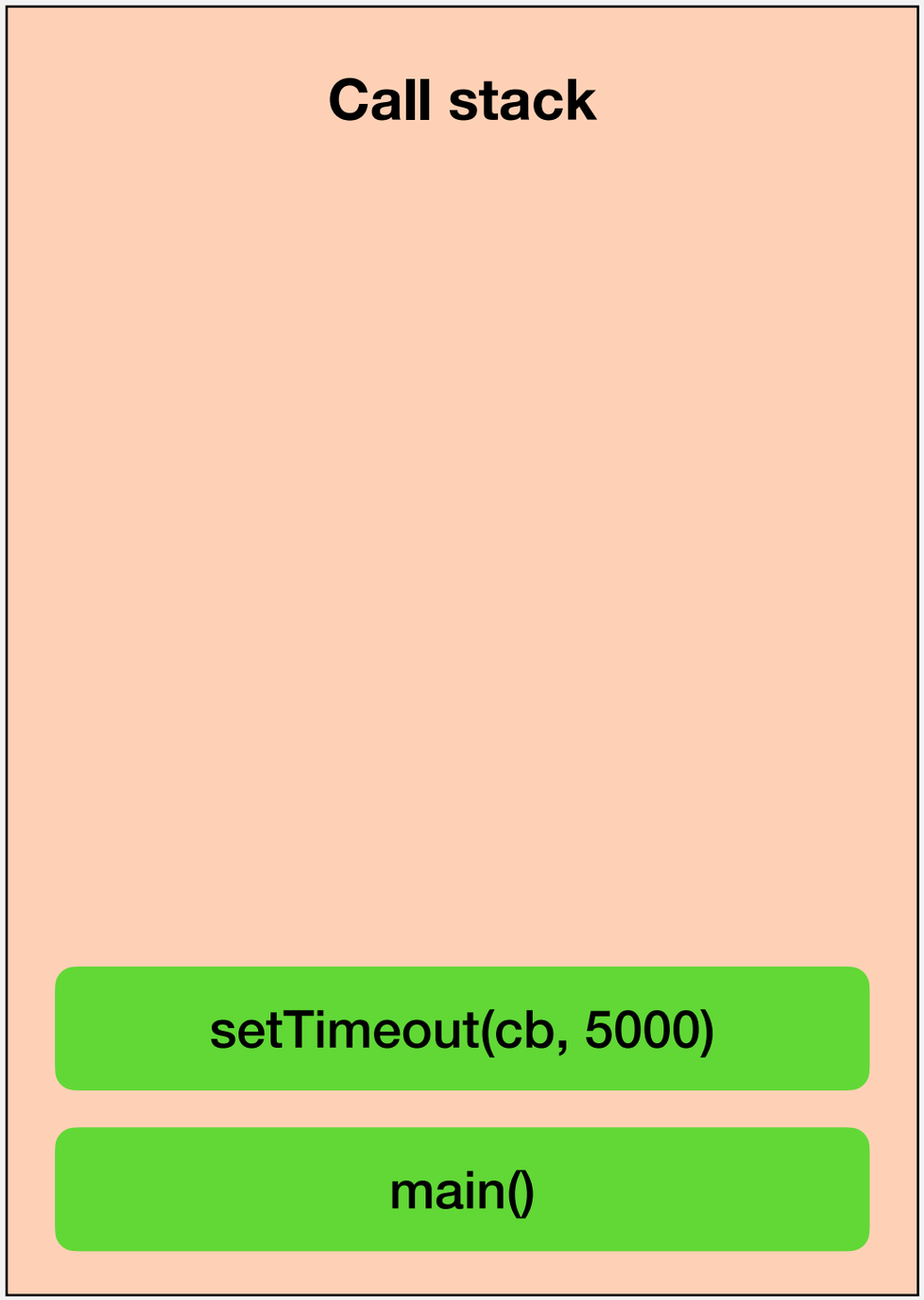
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



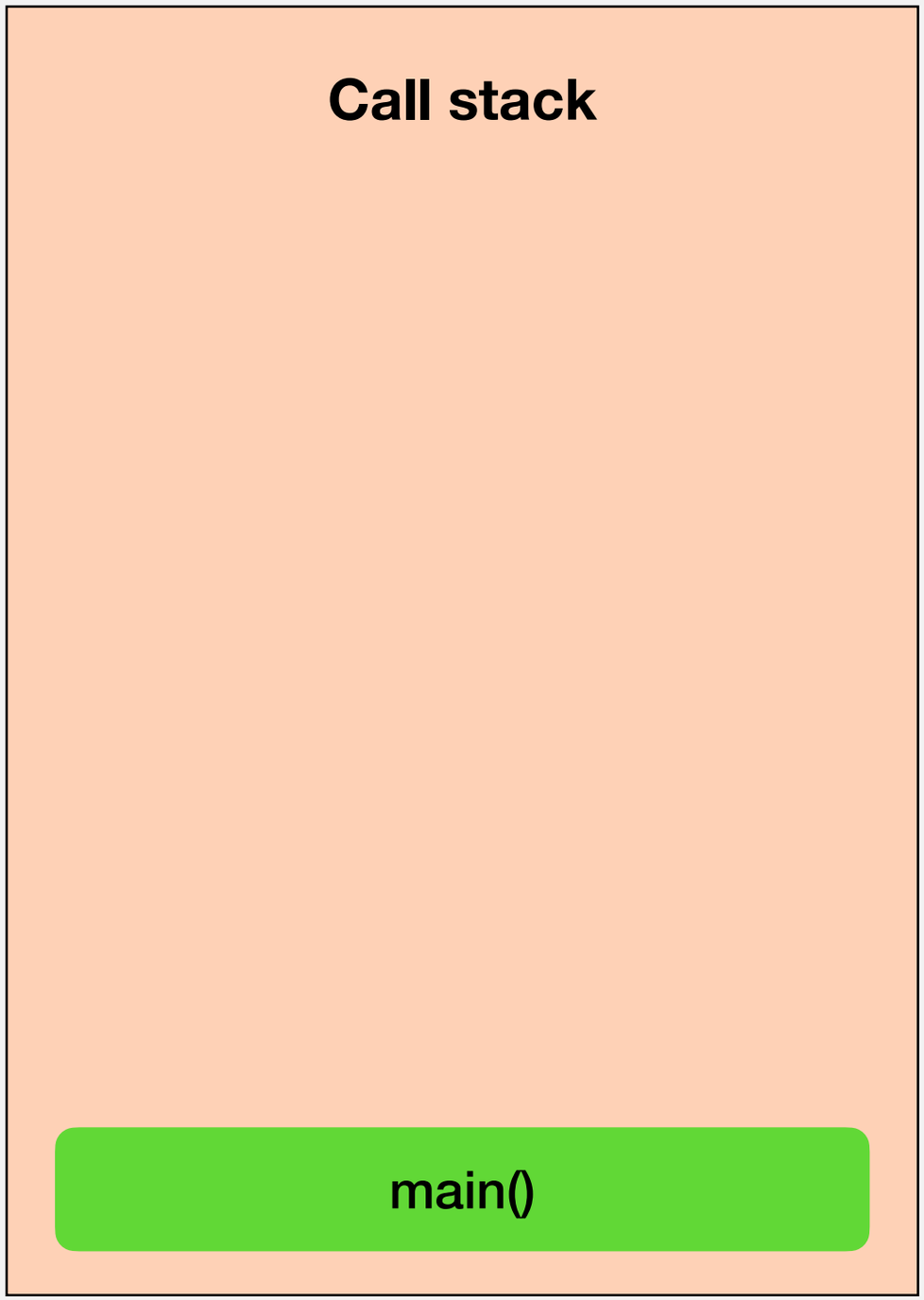
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



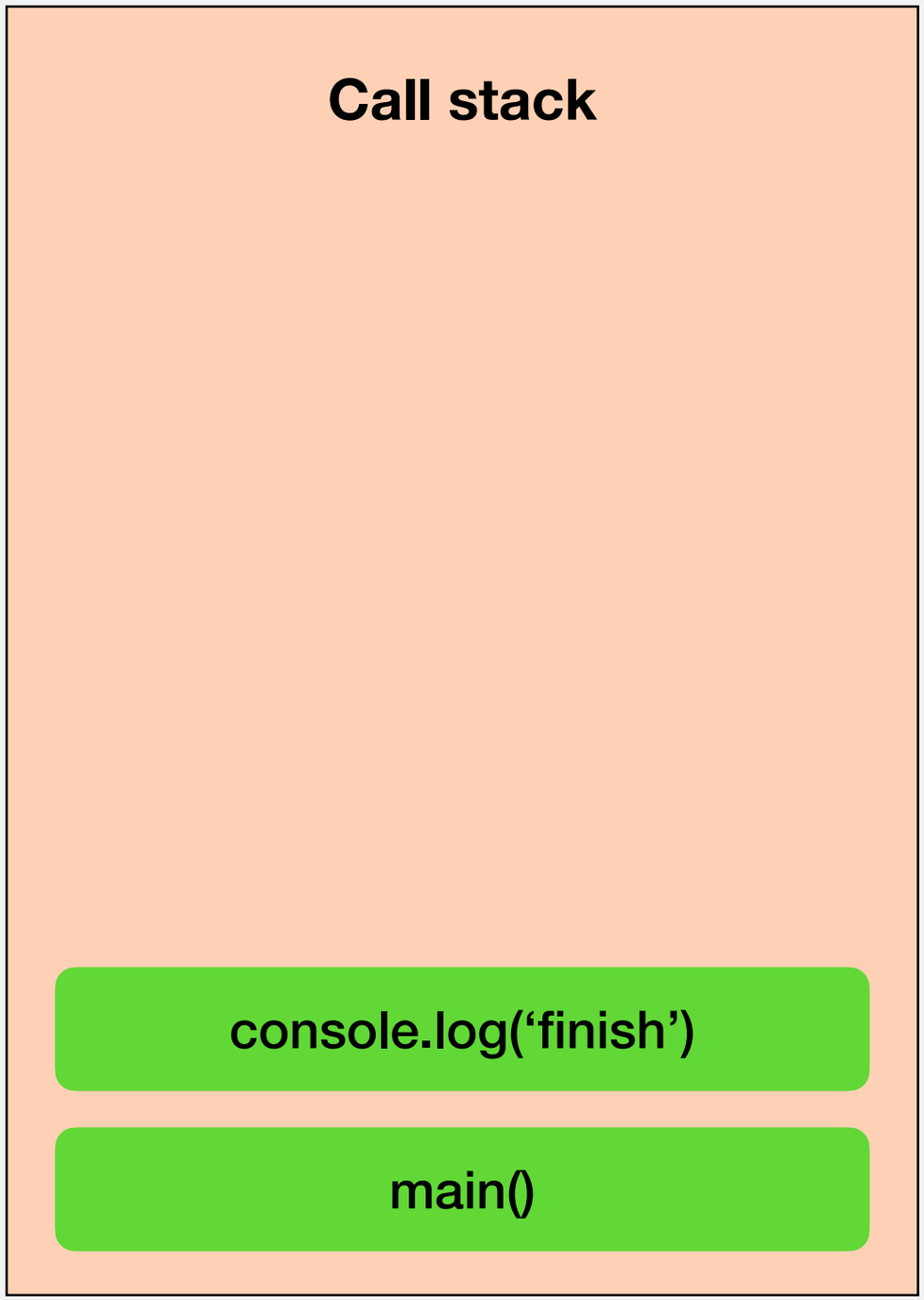
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



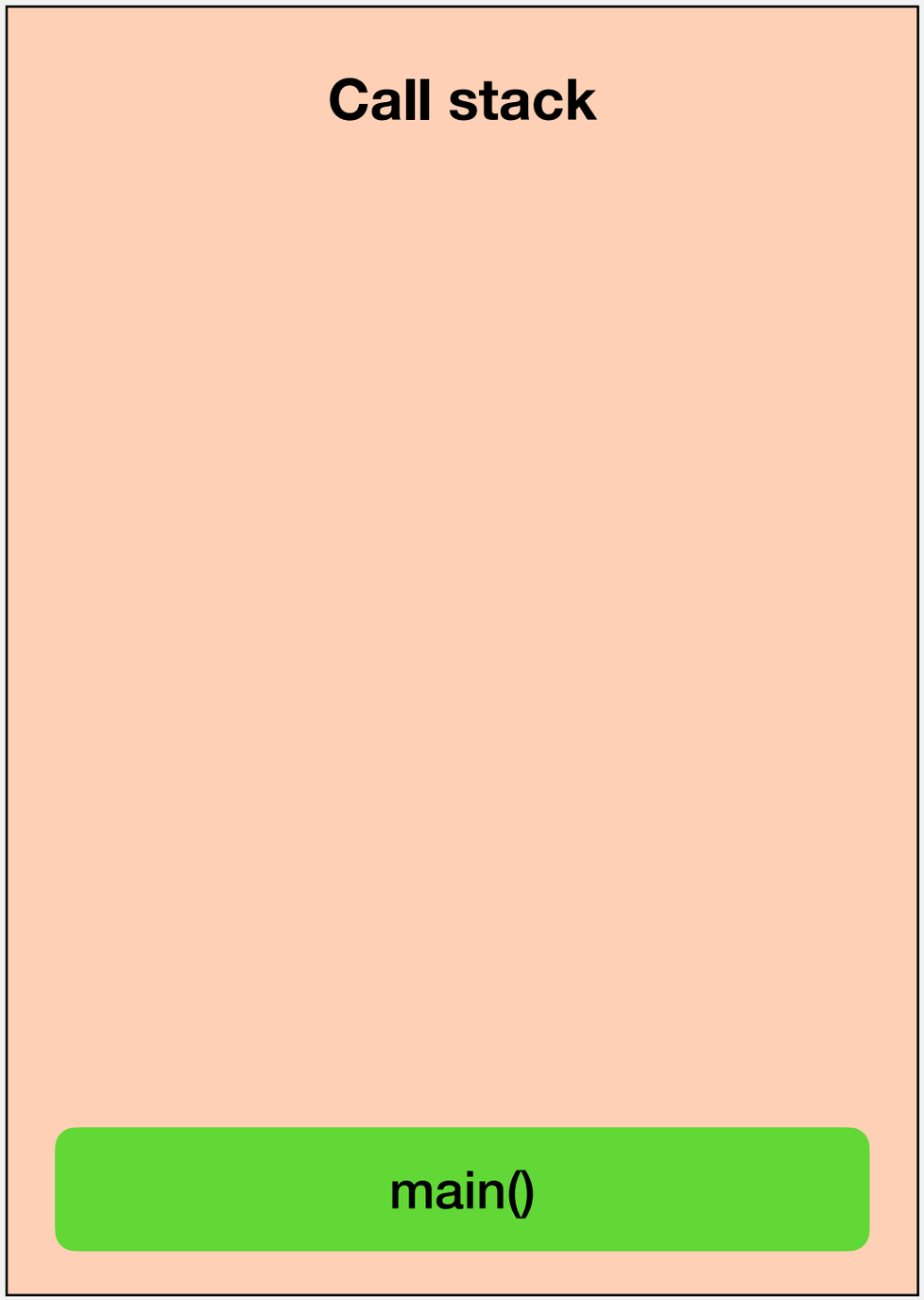
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



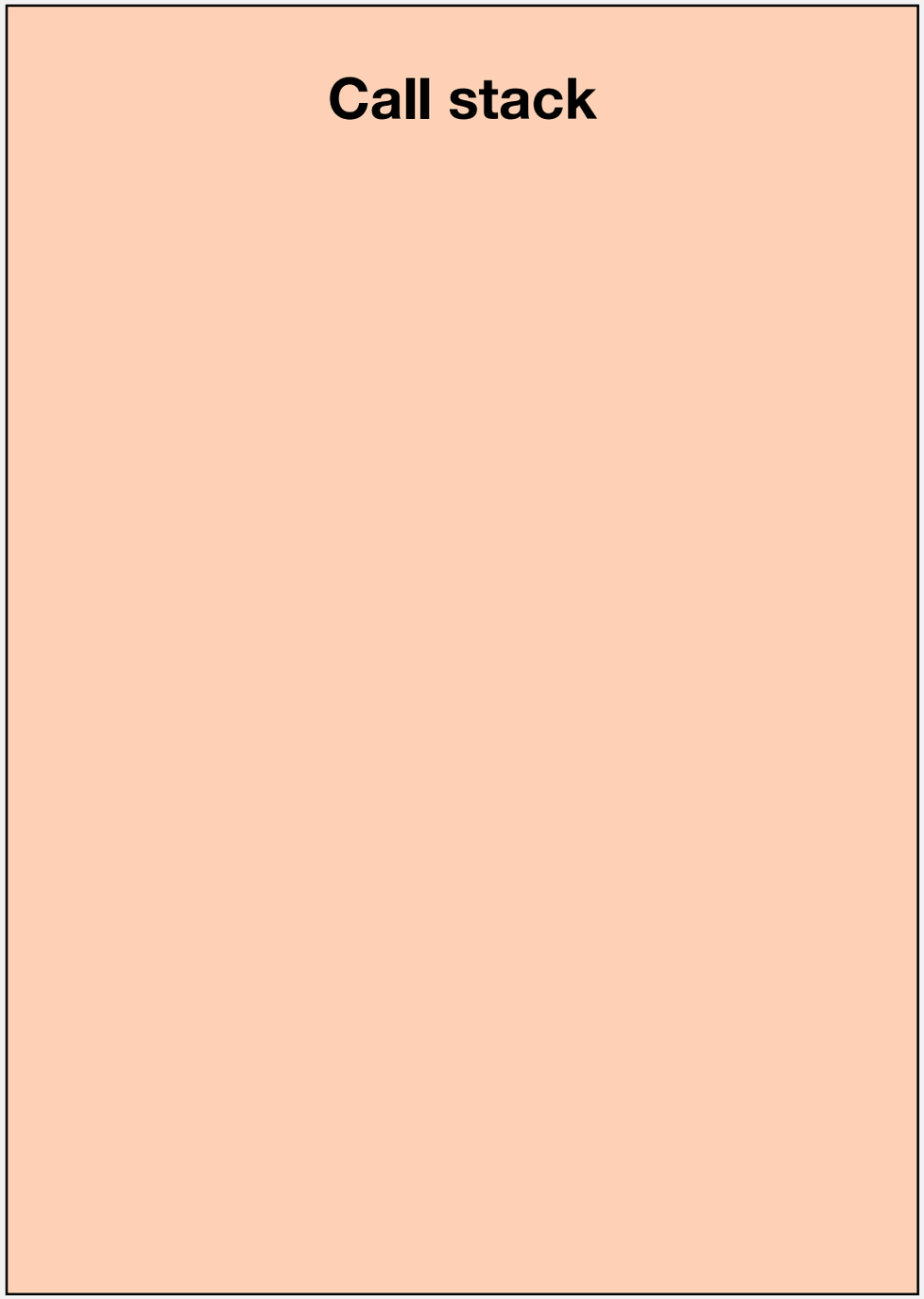
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



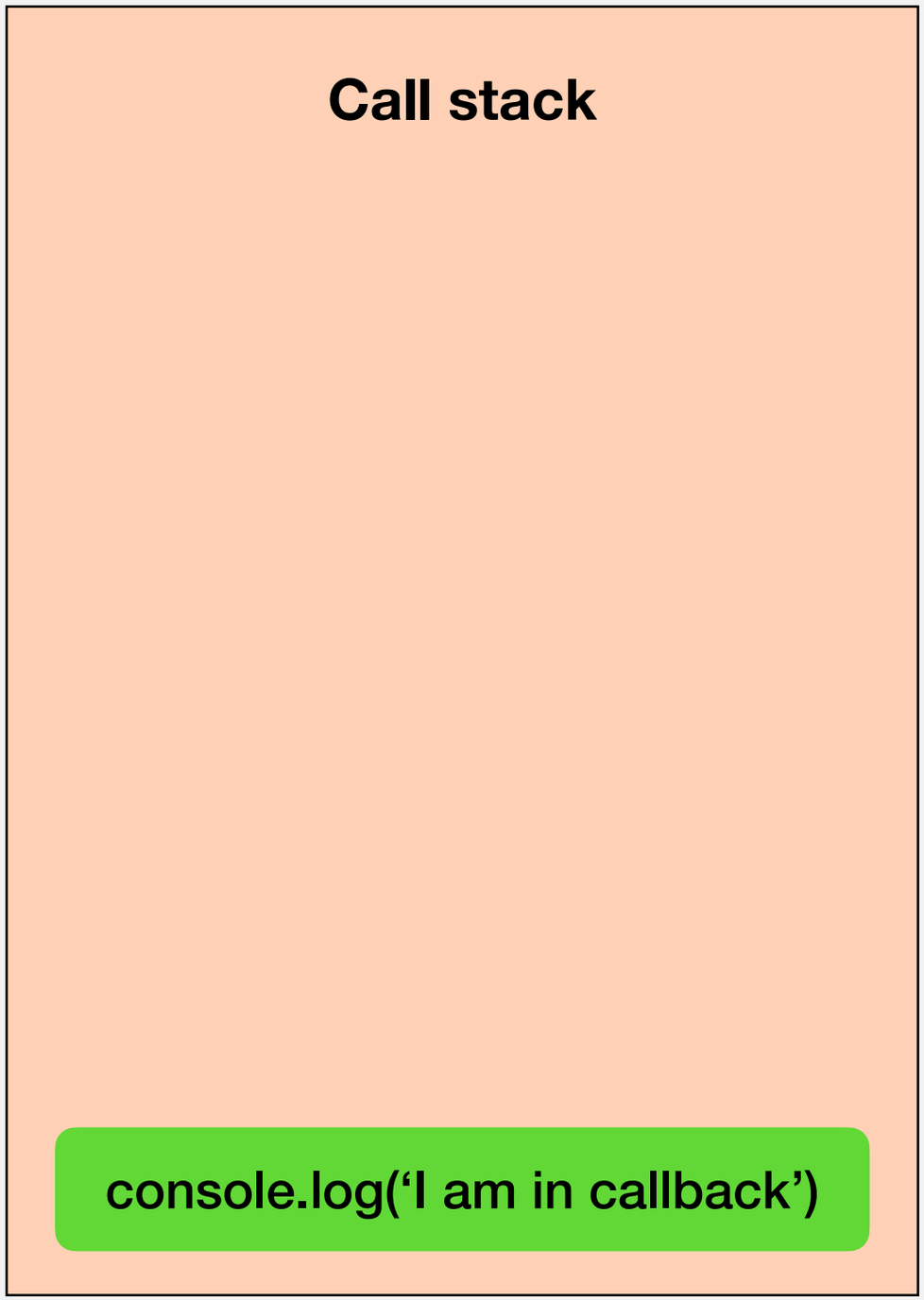
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



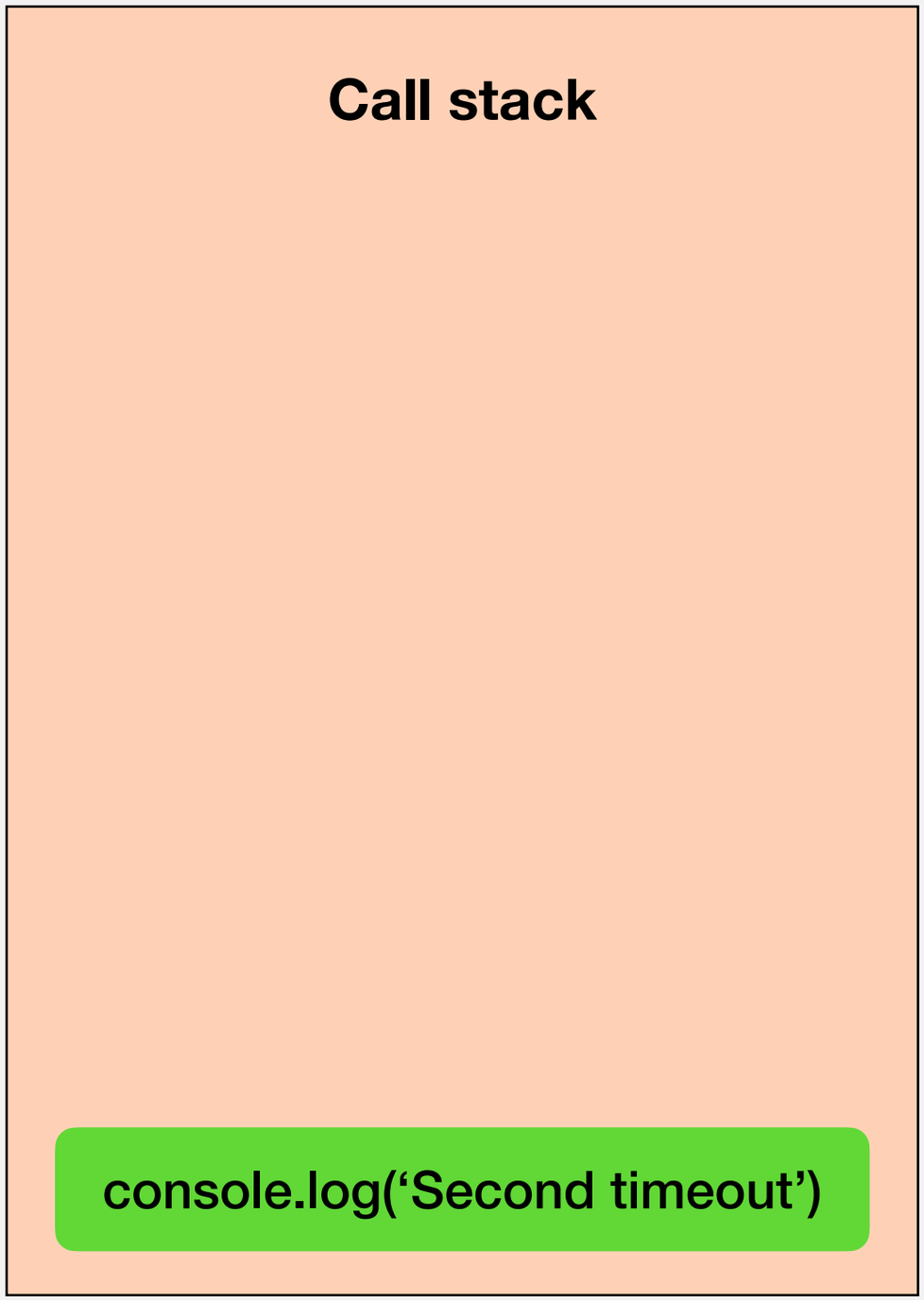
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



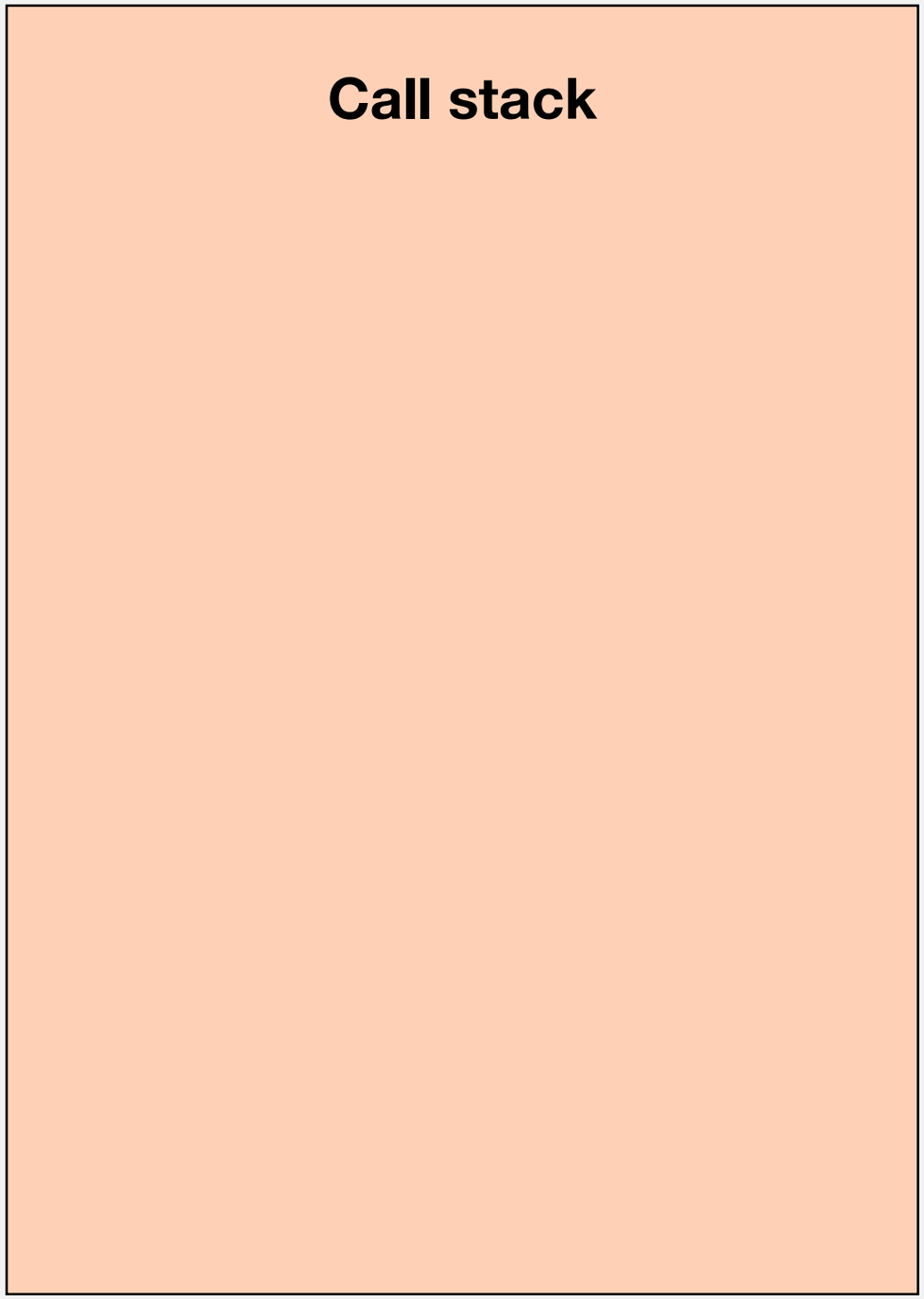
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



JS

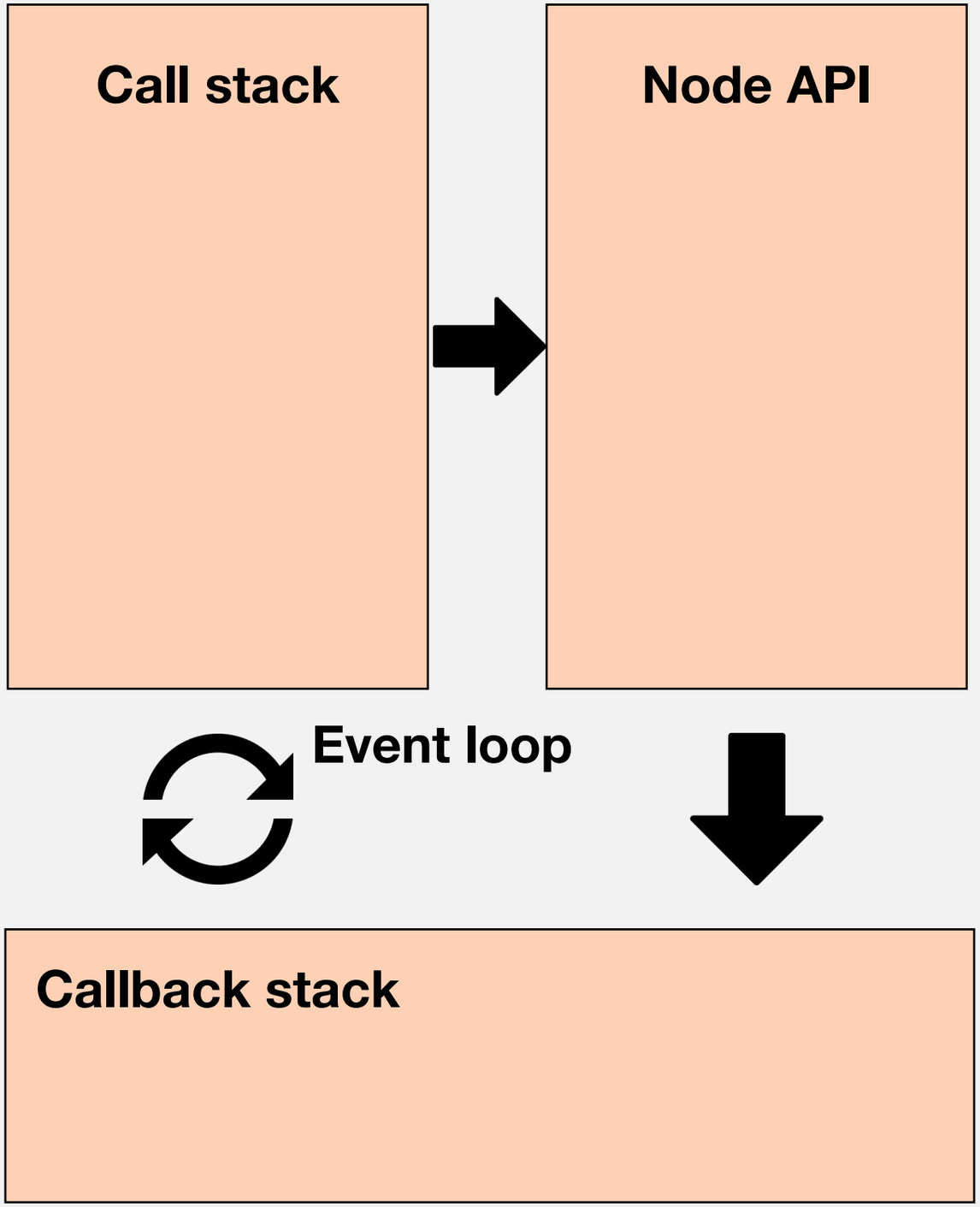
I am

- ~~Single threaded~~
- ~~Non-blocking~~
- ~~Asynchronous~~
- **Concurrent**

Concurrency and event loop

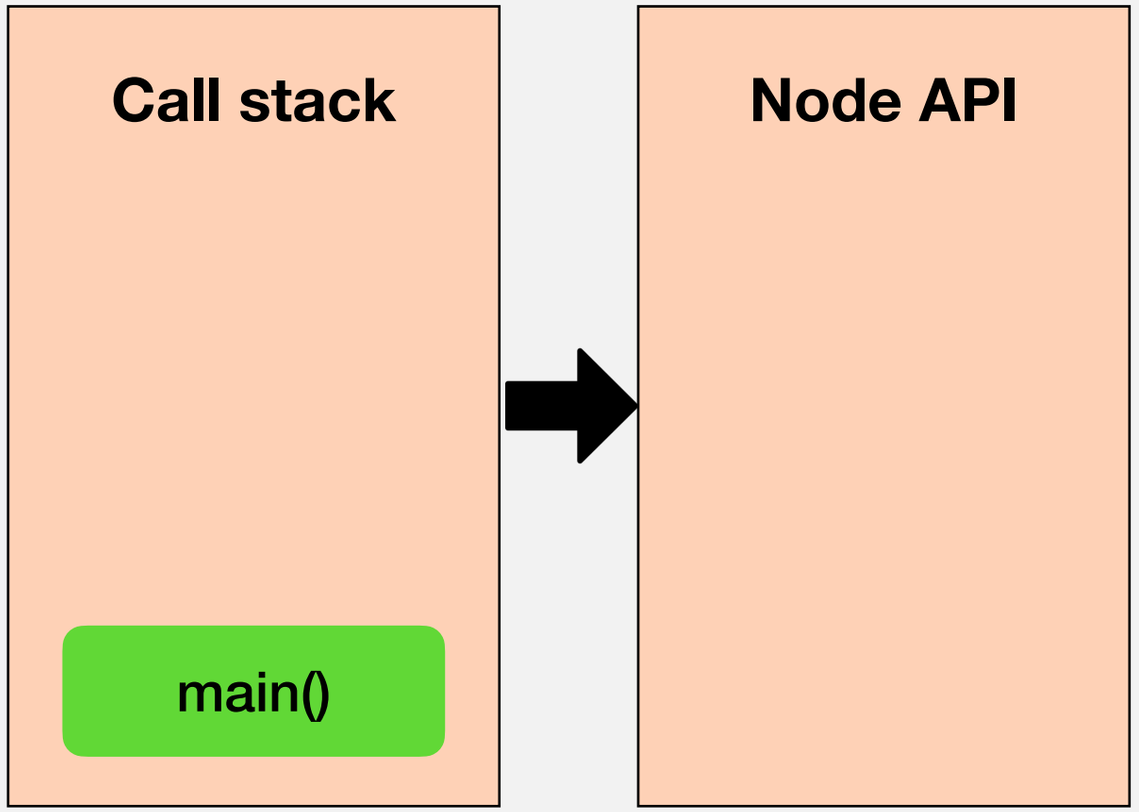
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



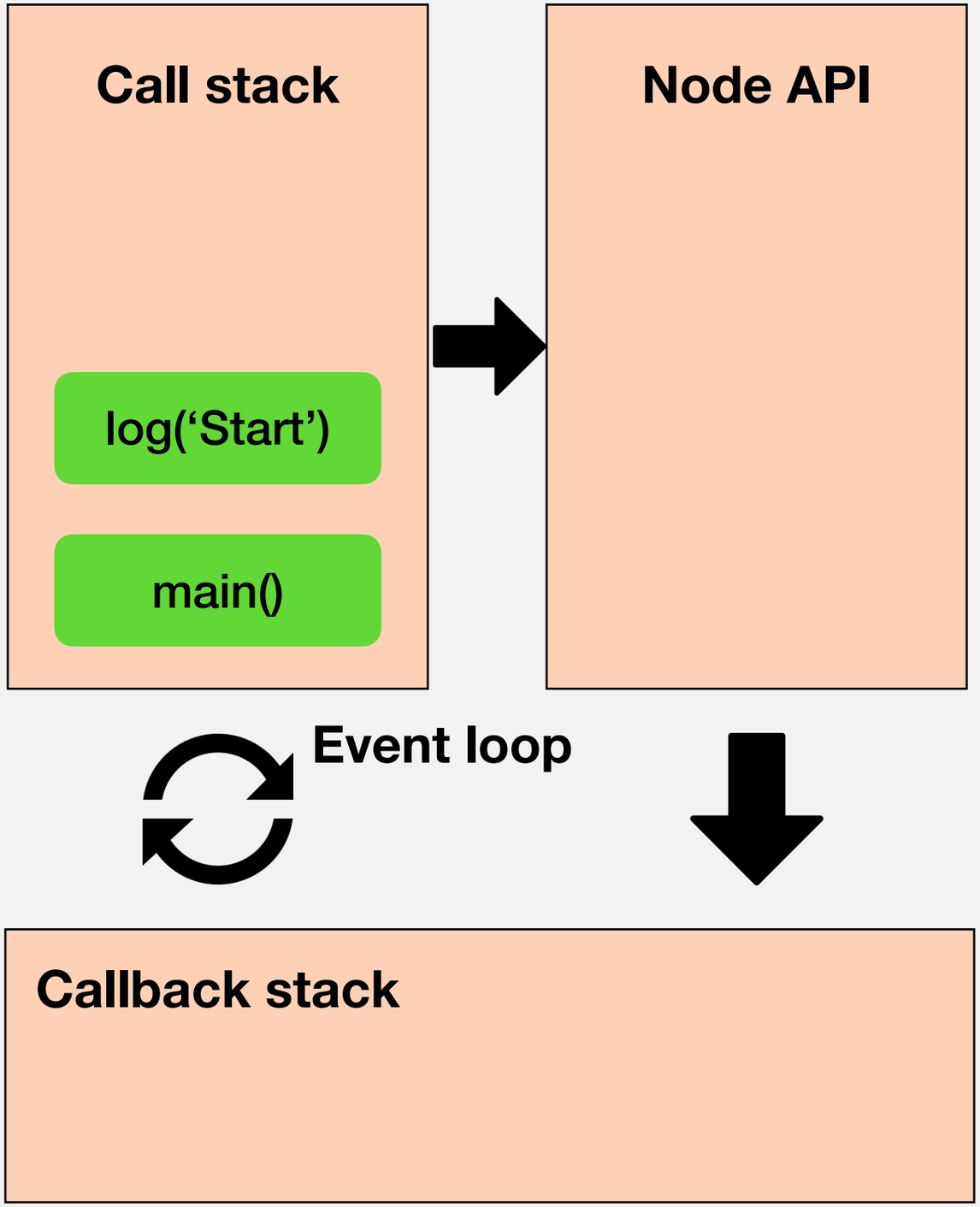
JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

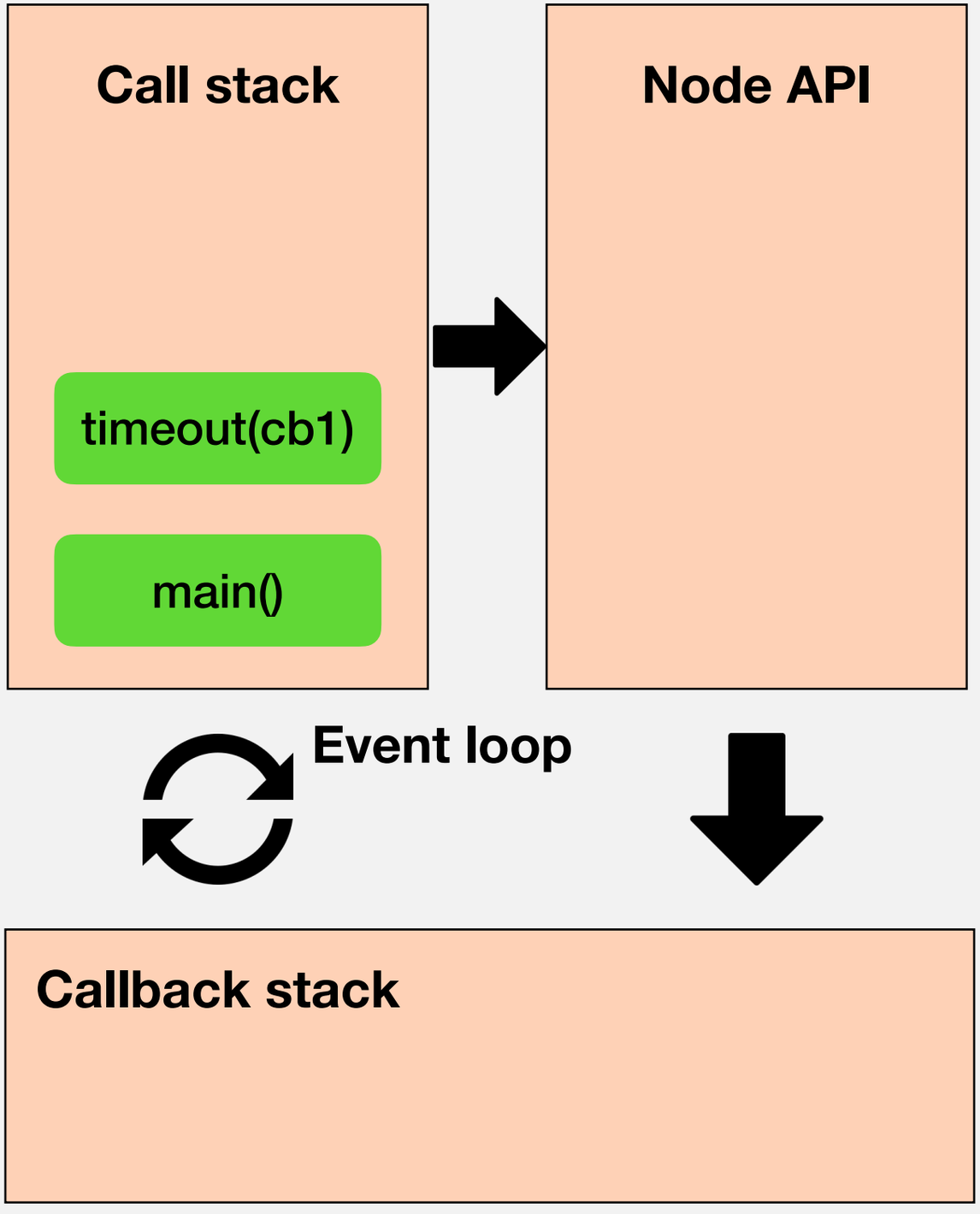


JS

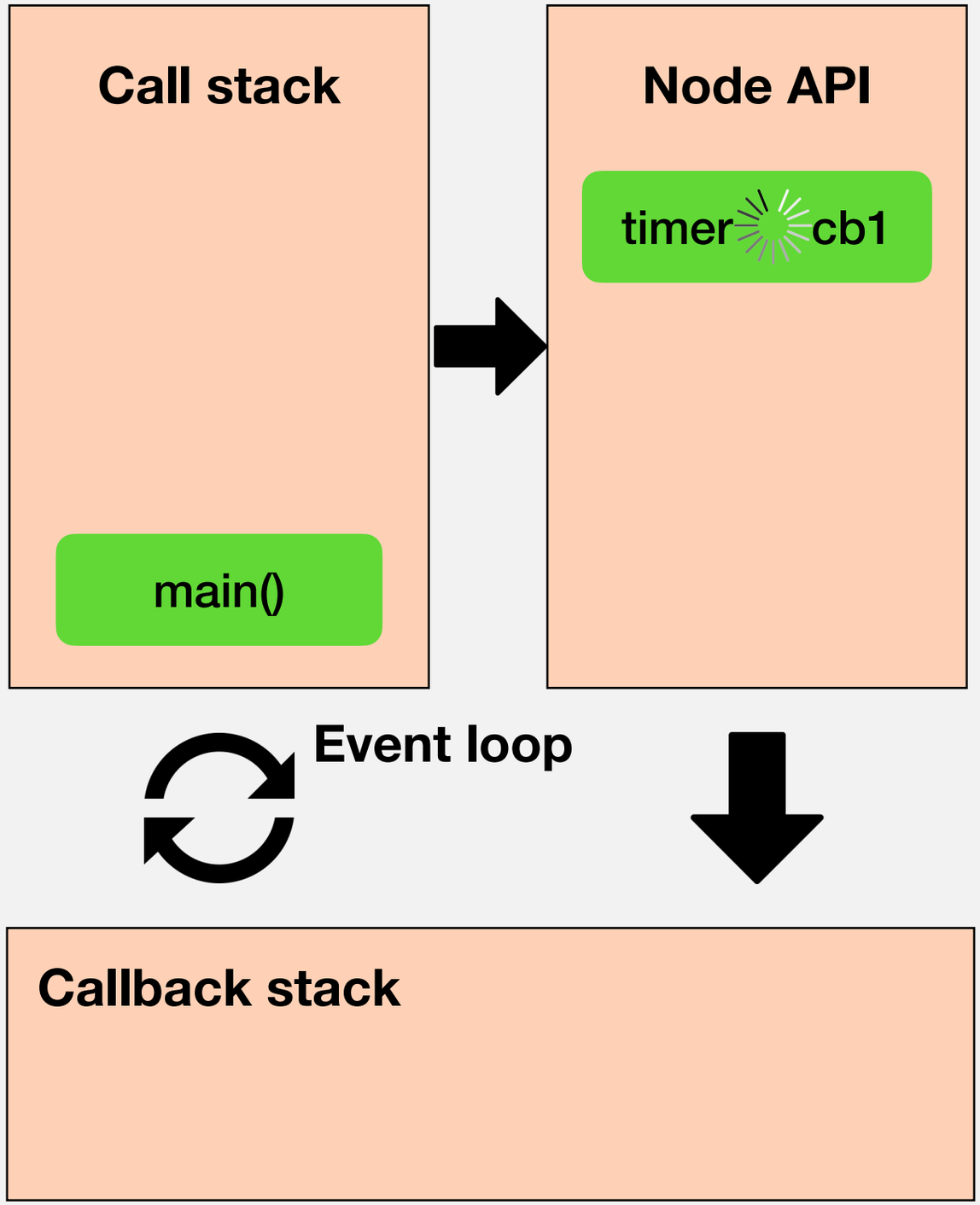
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



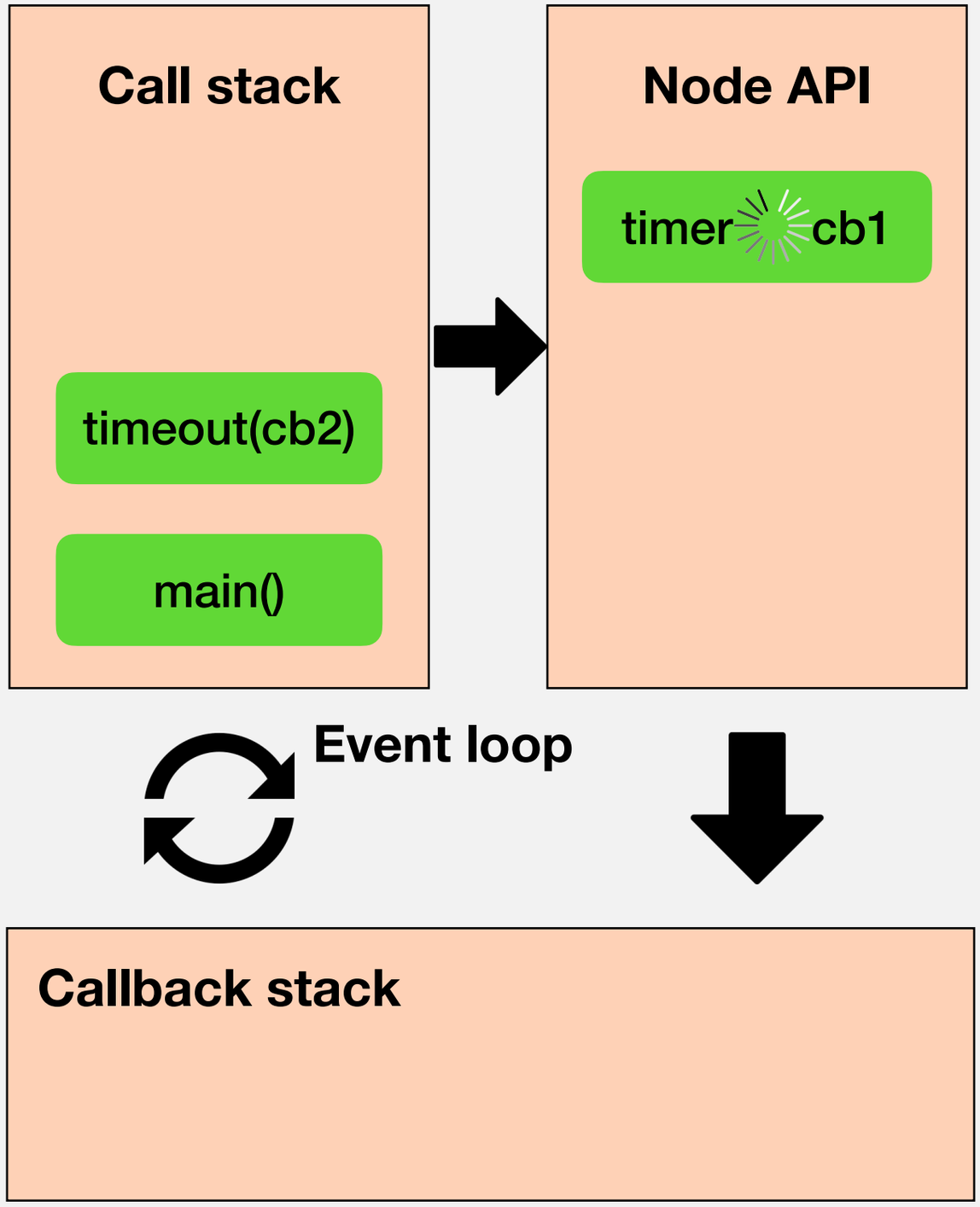
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



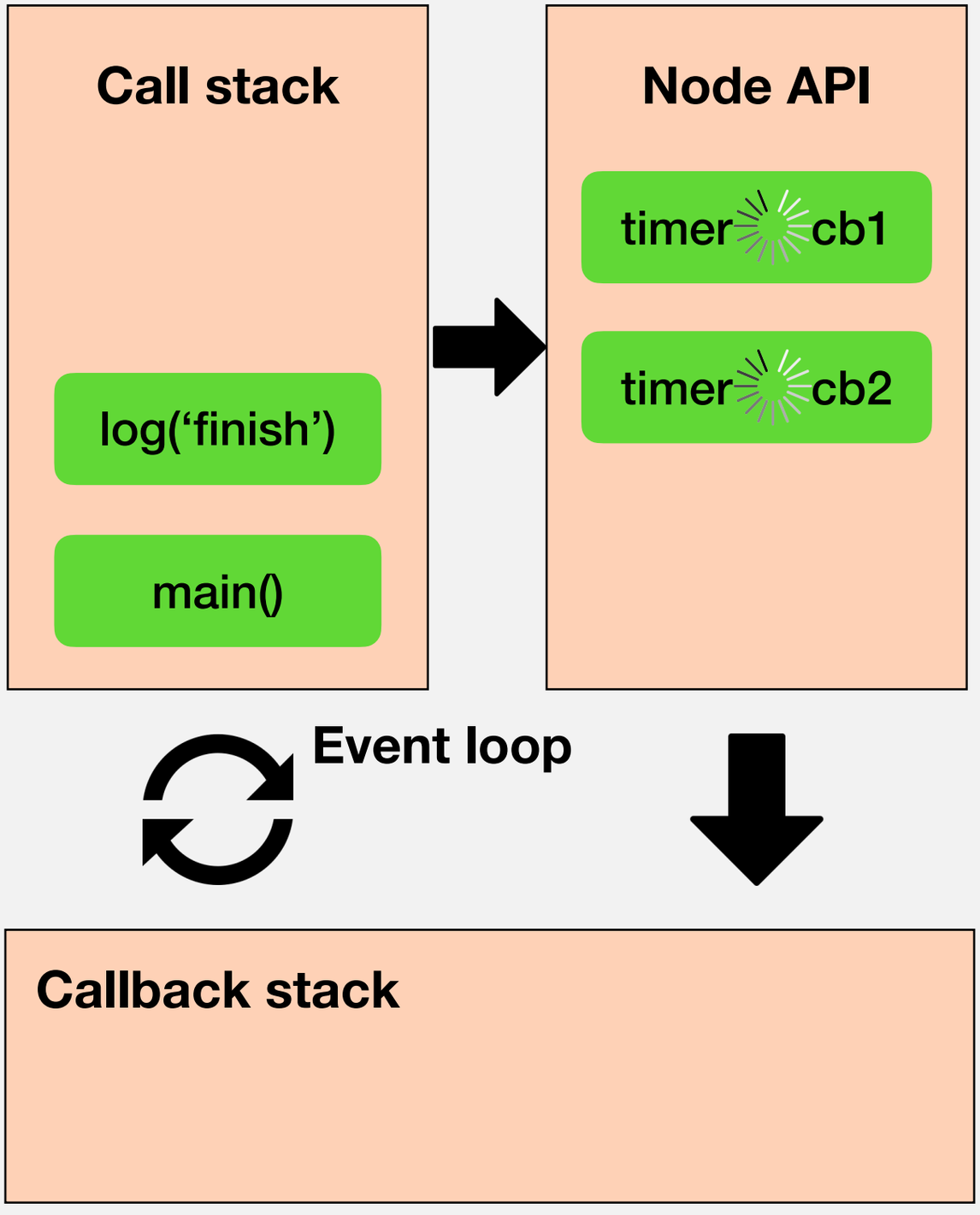
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



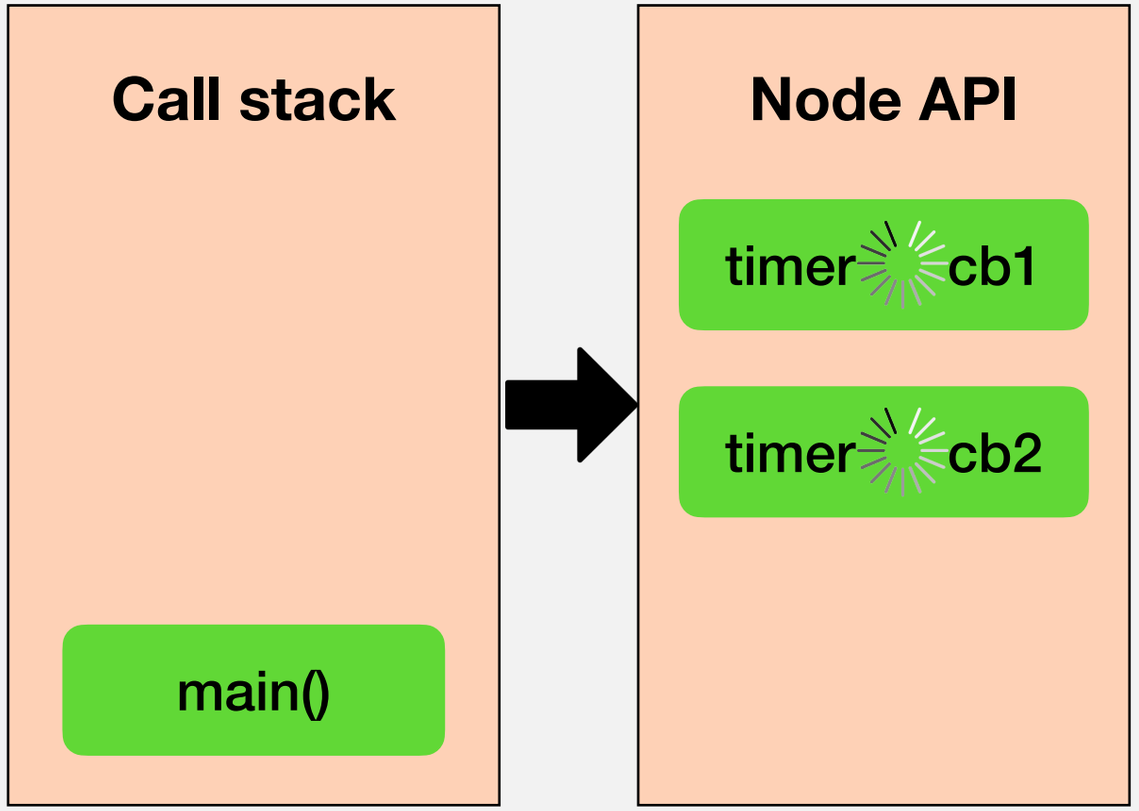
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

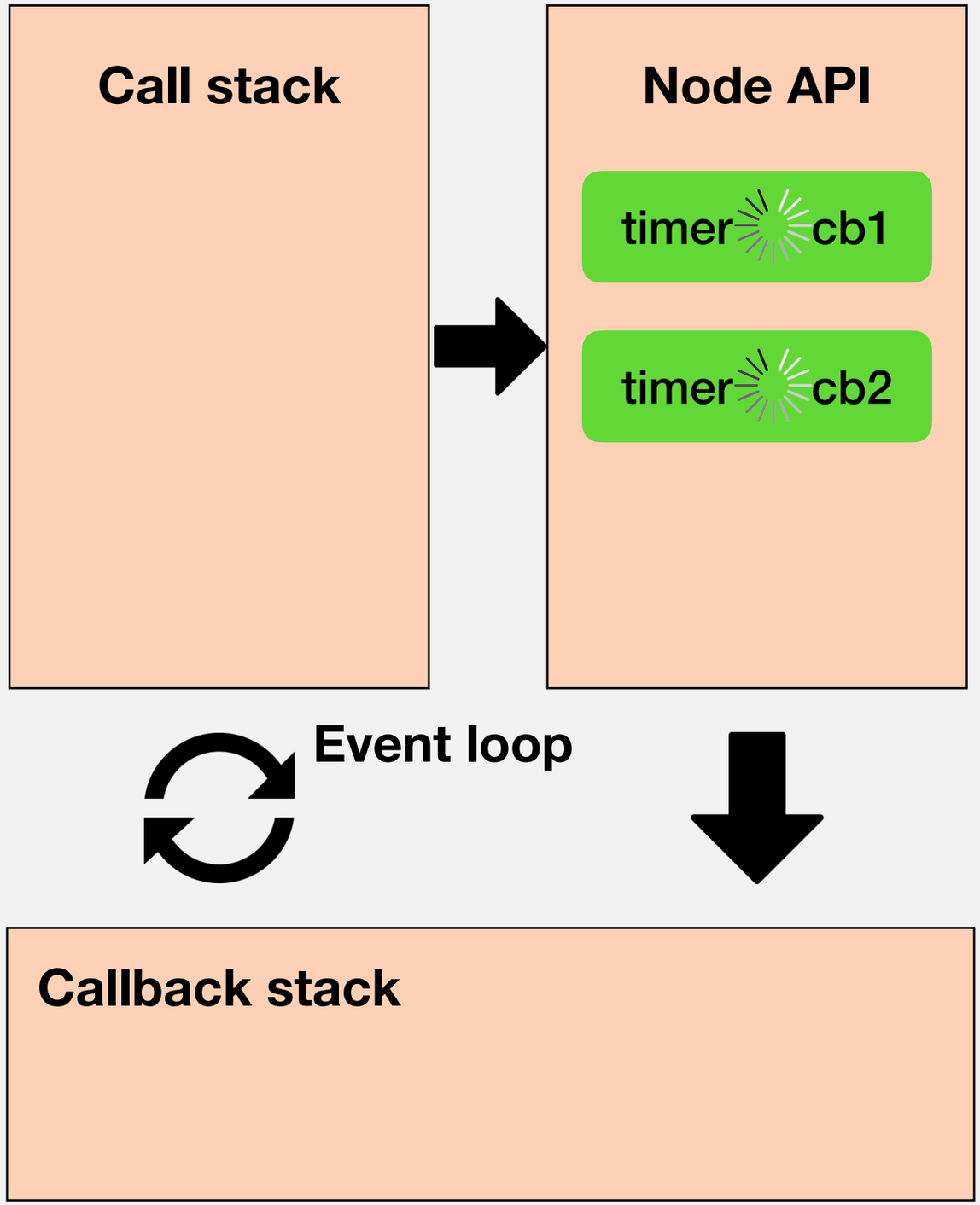


```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

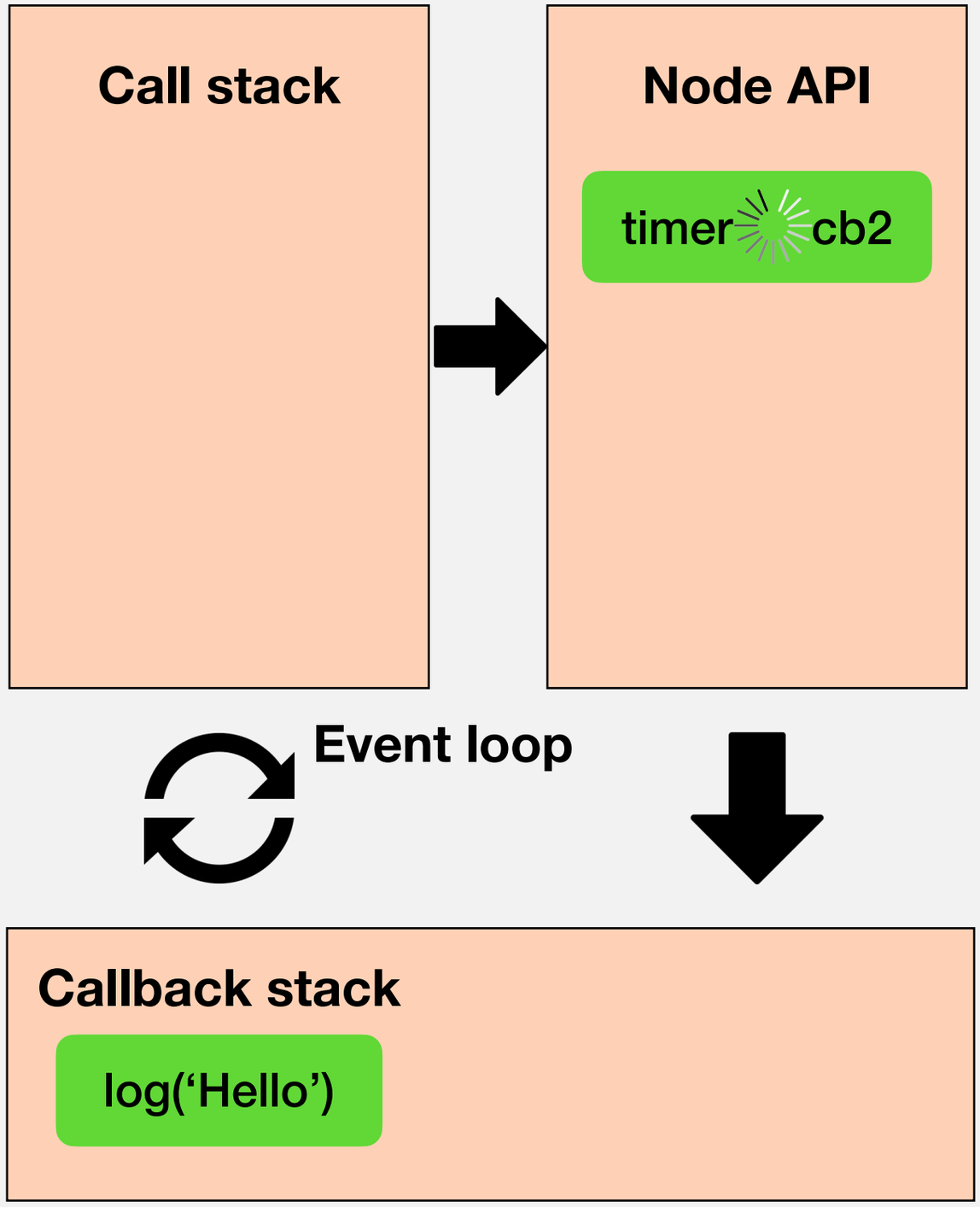


JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

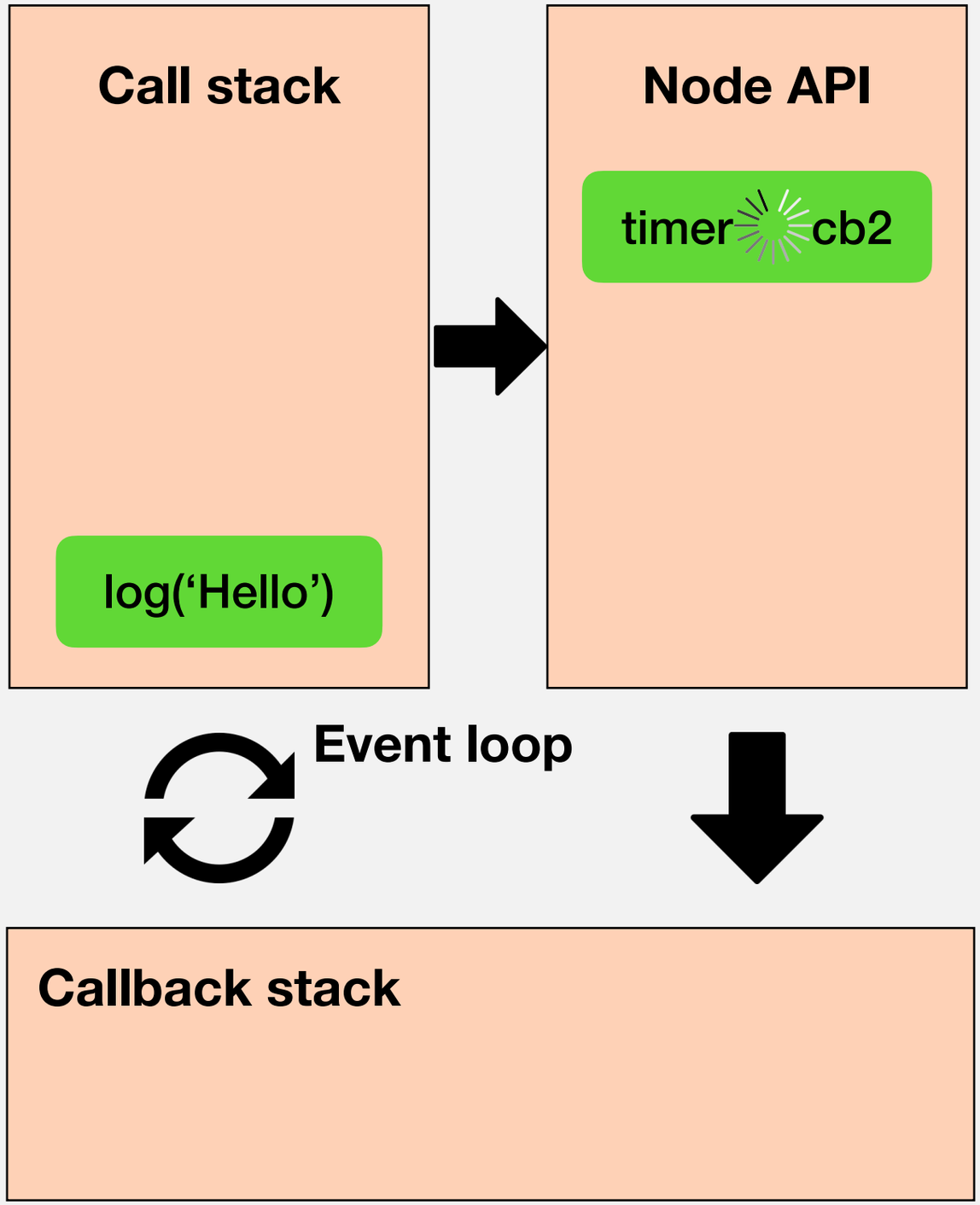


```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

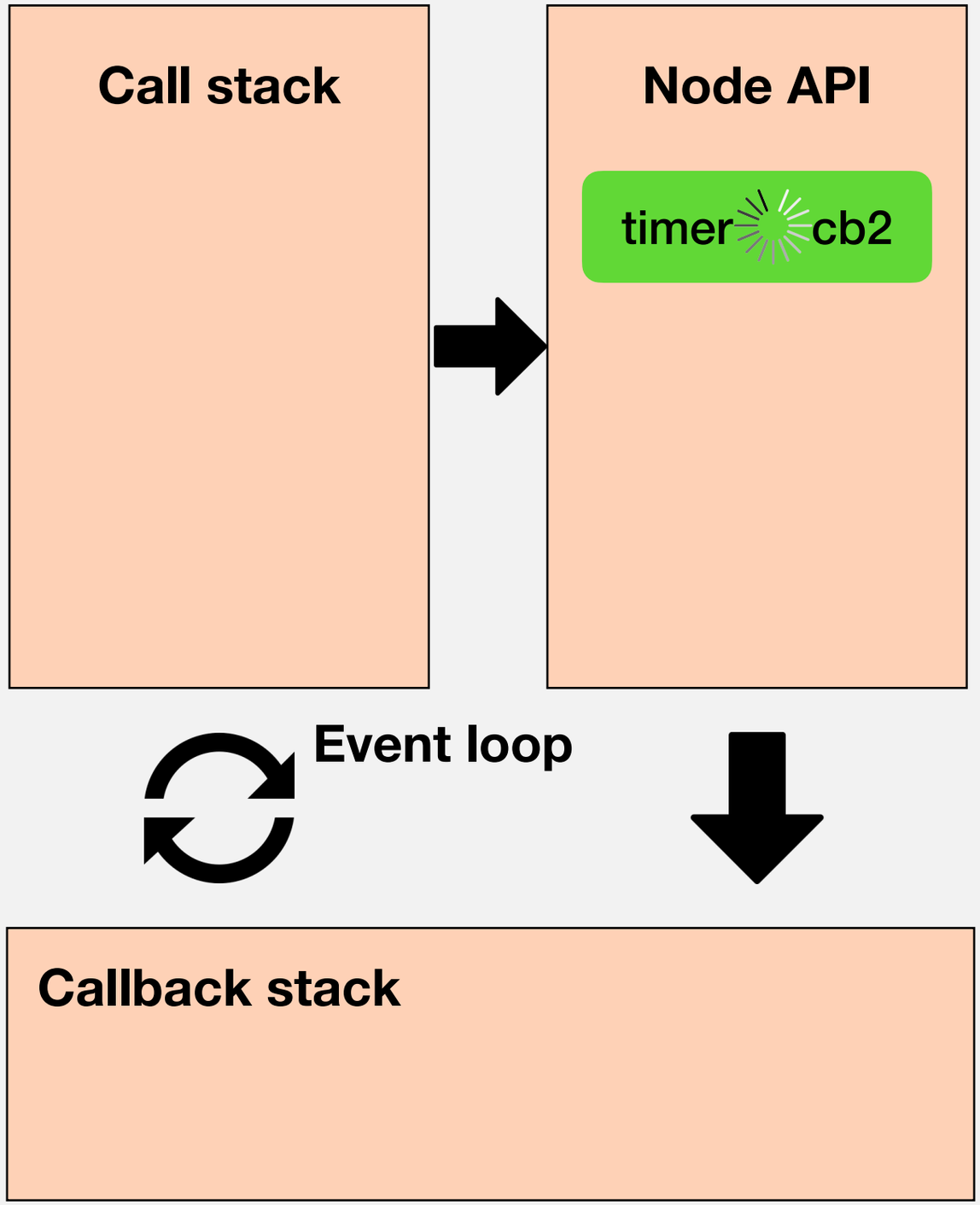


JS

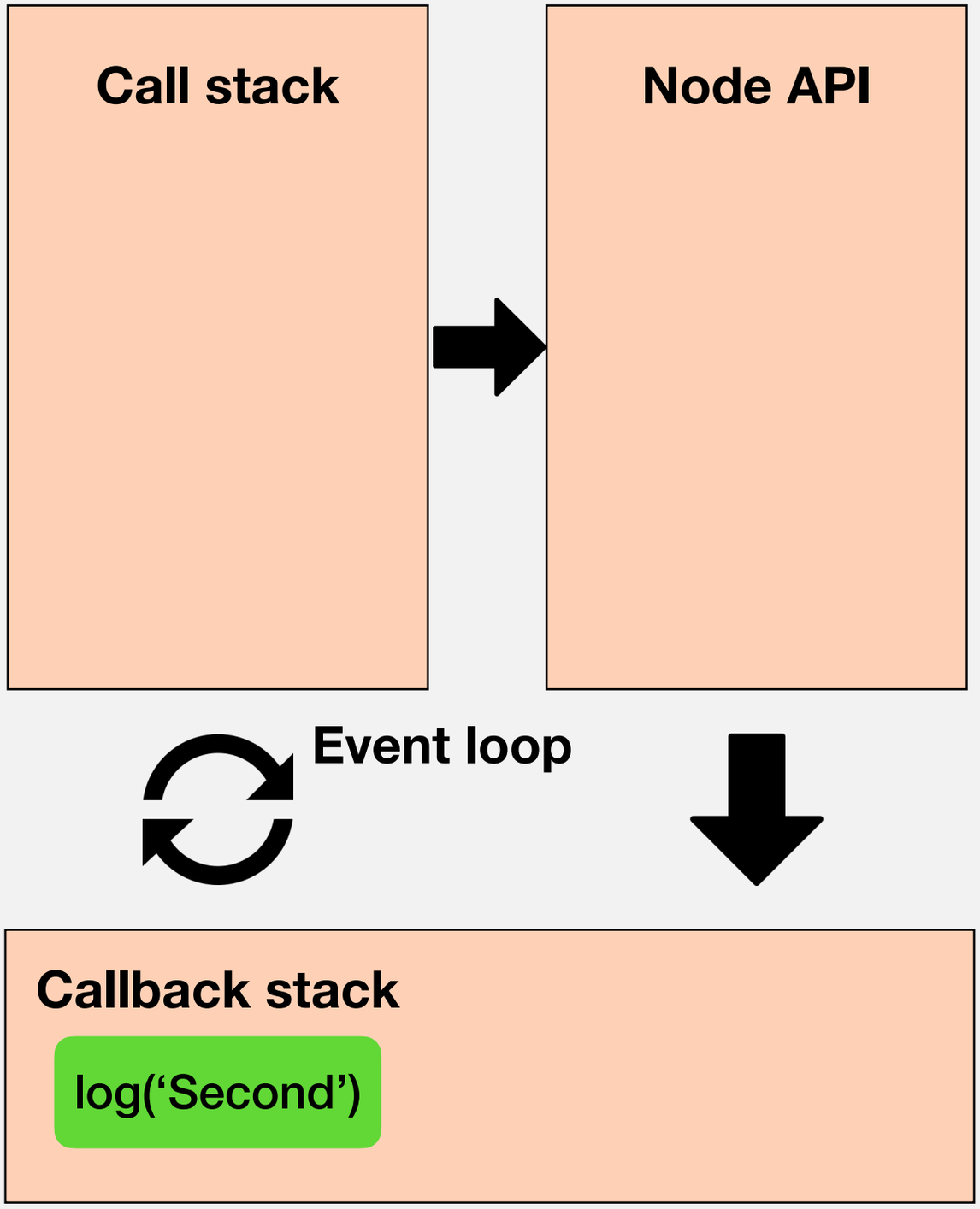
```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

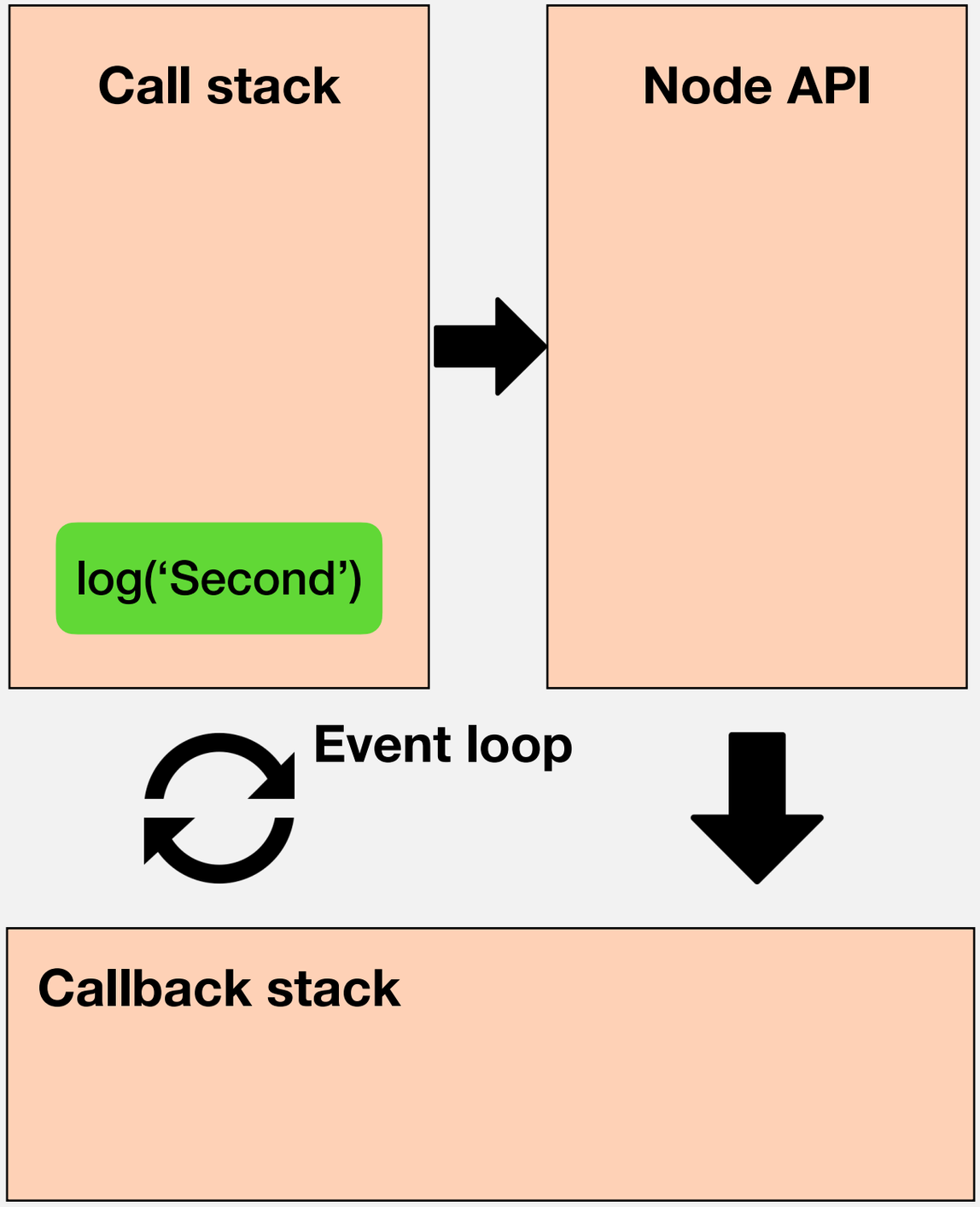


```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```

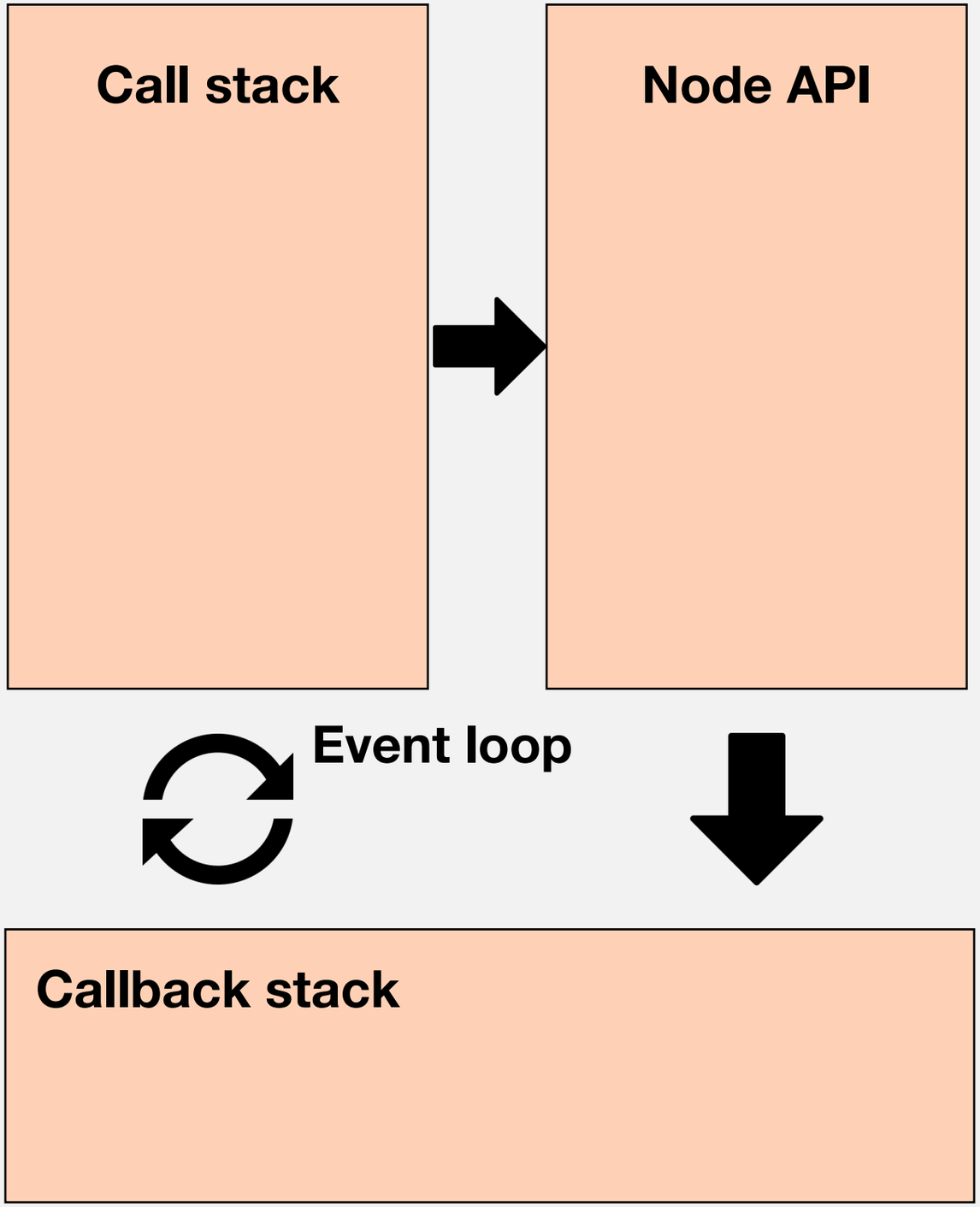


JS

```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



```
console.log('Start program');

setTimeout(function() {
  console.log('I am in callback');
}, 2000);

setTimeout(function() {
  console.log('Second timeout');
}, 5000);

console.log('finish program');
```



JS

setTimeout(callback, 0)

JS

```
setTimeout(function() {
  console.log('Zero timeout');
}, 0);

console.log('finish program');
```

JS

```
setTimeout(function() {
  console.log('Zero timeout');
}, 0);

console.log('finish program');
```

finish program

Zero timeout

JS

I am

- ~~Single threaded~~
- ~~Non-blocking~~
- ~~Asynchronous~~
- ~~Concurrent~~

JS

This

Привязка по умолчанию

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

Неявная привязка

```
var a = 10;
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

JS

Неявная привязка

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

JS

Неявно потерянный

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // ссылка/алиас на функцию!

var a = "ой, глобальная"; // `a` также и свойство глобального объекта

bar(); // "ой, глобальная"
```

Неявно потерянный

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` – просто еще одна ссылка на `foo`

    fn(); // <-- точка вызова!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "ой, глобальная"; // `a` еще и переменная в глобальном объекте

doFoo( obj.foo ); // "ой, глобальная"
```

JS

Явная привязка

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

JS

Привязка new

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```