

ПМ3 Разработка модулей ПО.

РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.

Тема 2. Асинхронно программирование.

Лекция 8. Основы асинхронности и стек ВЫЗОВОВ.

Цели занятия:

- Понять, как работает однопоточная модель JavaScript.
- Понять различие между синхронным и асинхронным кодом.
- Узнать реальные области применения асинхронности (сетевые запросы, работа с таймерами, событиями).
- Понять, что такое Call Stack (стек вызовов), как работает Event Loop, чем отличаются macro- и microtasks

Учебные вопросы:

- 1. Однопоточность JavaScript: особенности выполнения кода.**
- 2. Синхронный vs асинхронный код.**
- 3. Примеры использования асинхронности.**
- 4. Call Stack (стек вызовов).**
- 5. Event Loop.**
- 6. Macro- и Microtasks.**

1. Однопоточность JavaScript.

Что значит «однопоточный язык»?

В JavaScript есть только один поток выполнения кода.

Это значит: в один момент времени выполняется только одна операция.

Код выполняется построчно сверху вниз, пока стек вызовов (Call Stack) не опустеет.

В отличие от многопоточных языков (Java, C++), где можно запускать несколько задач параллельно, JS делает всё в одной последовательности.

◆ Почему это важно?

Упрощает разработку:

- Не нужно управлять потоками, мьютексами, блокировками.
- Код проще предсказуем в большинстве случаев.

Но есть ограничения:

- Если одна задача «тяжёлая» (например, бесконечный цикл или огромный цикл без пауз) → она блокирует весь поток.
- В это время браузер не сможет обработать события (клики, ввод текста, скролл).

 Поэтому асинхронность в JS используется для разгрузки потока и реакции на внешние события.

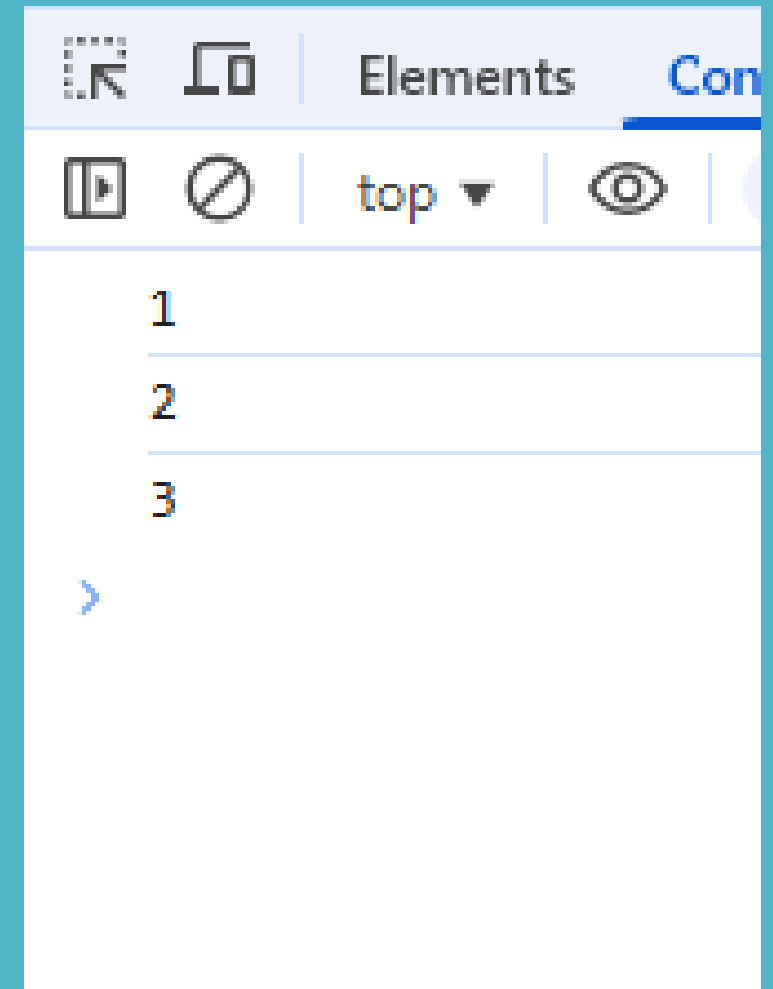
Пример: однопоточная модель

```
1  console.log("1");  
2  
3  function task() {  
4    |  console.log("2");  
5  }  
6  
7  task();  
8  
9  console.log("3");  
10
```

Порядок выполнения:

- Вызов `console.log("1")` → в стек → выполняется → удаляется.
- Вызов `task()` → помещается в стек → выполняется → вызывает `console.log("2")` → выполняется → удаляется.
- Вызов `console.log("3")`.

 Вывод всегда один и тот же:



Проблема блокирующего кода

```
1  console.log("Начало");
2
3  function heavy() {
4      let start = Date.now();
5      while (Date.now() - start < 5000) {
6          // 5 секунд процессора!
7      }
8      console.log("Тяжёлая задача завершена");
9  }
10
11  heavy();
12  console.log("Конец");
```

Что произойдёт?

- Скрипт «зависает» на 5 секунд.
 - Пользователь не может кликнуть, печатать, скроллить
→ **браузер заморожен.**
 - Только после завершения `heavy()` выполнение продолжится.
- 👉 Это классическая проблема однопоточности.

Как решается проблема?

Вместо синхронного «тяжёлого» кода — использование асинхронных операций:

- Таймеры (setTimeout, setInterval).
- Асинхронные API (fetch, XMLHttpRequest).
- События (addEventListener).

Таким образом «долгие» задачи выносятся из основного потока, и UI остаётся отзывчивым.

✓ Выводы:

- JS — однопоточный язык: выполняет только одну задачу за раз.
- Все вызовы идут через Call Stack.
- Если задача тяжёлая → весь поток блокируется.
- Чтобы избежать зависаний, нужны асинхронные механизмы (это и есть основа дальнейших лекций).

2. Синхронный vs асинхронный код.

Синхронный код:

- Выполняется строго построчно сверху вниз.
- Каждая операция ждёт завершения предыдущей.
- Подходит для быстрых вычислений (арифметика, работа с массивами).

Минус: если операция долгая или блокирующая (например, чтение файла, запрос в сеть), то весь поток «замораживается».

Асинхронный код

- Операции могут быть отложены во времени.
 - Основной поток не ждёт их завершения, а продолжает выполнять другие инструкции.
 - Асинхронный код нужен, когда задача занимает время:
 - HTTP-запросы,
 - работа с базами данных,
 - таймеры,
 - реакция на пользовательские события.
- 👉 Асинхронность позволяет не блокировать UI и делает приложения отзывчивыми.

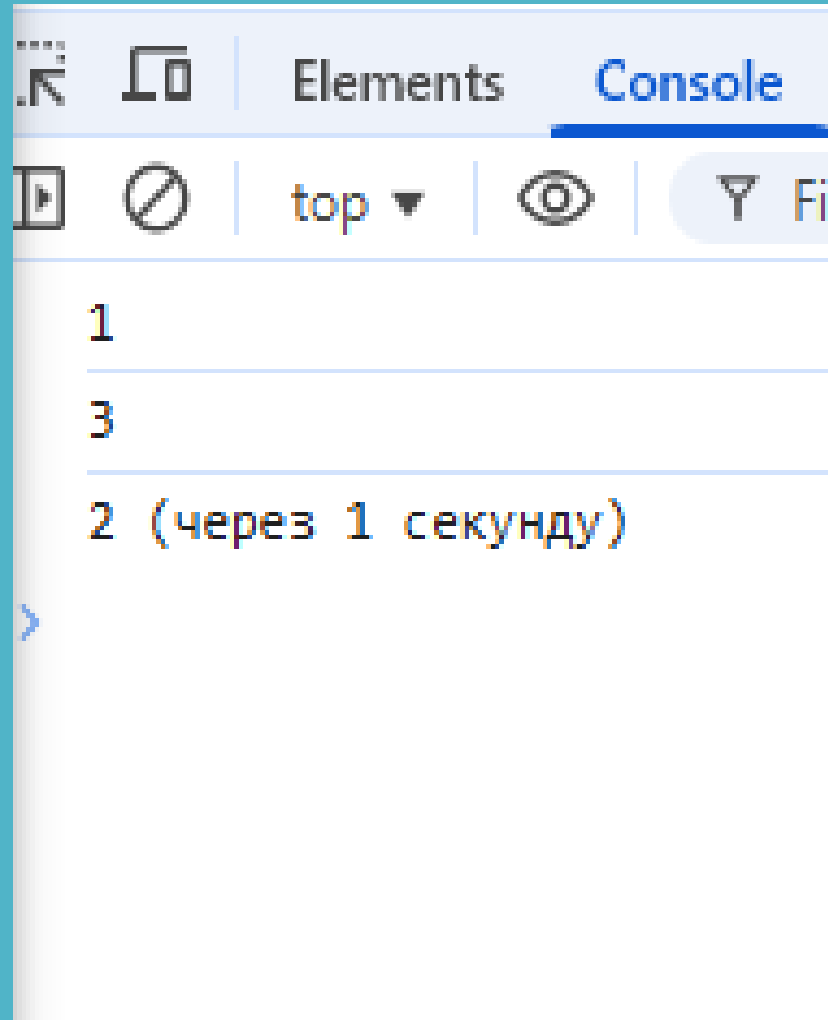
Пример (асинхронность через setTimeout):

```
1  console.log("1");
2
3  setTimeout(() => {
4    | console.log("2 (через 1 секунду)");
5  }, 1000);
6
7  console.log("3");
8
```

Что здесь происходит?

- JavaScript умеет делать только одно дело за раз.
- Строка `console.log("1")` — просто печать. Сразу видим **1**.
- `setTimeout(..., 1000)` — это как поручение «позвони мне через секунду». JS не ждёт секунду, а идёт дальше по коду.
- `console.log("3")` печатается сразу — видим **3**.
- Примерно через секунду «звонок» по поручению срабатывает, и печатается **2** (через 1 секунду).

Итоговый порядок в консоли:



Бытовая аналогия.

Представьте кафе:

- Вы говорите бариста: «Налейте кофе» → он ставит готовиться (это `setTimeout`/запрос в будущее).
- Пока кофе готовится, бариста обслуживает следующего клиента (идёт дальше по коду).
- Как только кофе готов — бариста зовёт вас (выполняет вашу функцию).

Асинхронность = «не жди — продолжай работать; когда будет готово, вернись к задаче».

Даже если поставить `setTimeout(fn, 0)`, «перезвон» случится после текущих дел:

```
setTimeout(() => console.log('A'), 0);  
console.log('B'); // сначала B, потом A
```

Если в момент «перезвона» JS занят (например, у вас тяжёлый цикл), сообщение придёт чуть позже, когда освободится.

Этого понимания сейчас достаточно. На следующей лекции разберём «как это устроено под капотом» и уже введём формальные названия и очереди.

Основные отличие		
	Синхронный код	Асинхронный код
Выполнение	Последовательно, блокирующее	Параллельно по времени, неблокирующее
Использование	Быстрые операции, вычисления	Долгие операции: сеть, ввод/вывод, таймеры
Минус	Может «заморозить» интерфейс	Сложнее для понимания (колбэки, промисы)
Пример	for, console.log	setTimeout, fetch, события



Вывод по вопросу:

- Синхронный код — проще, но блокирует поток.
- Асинхронный код — сложнее в организации, но позволяет JS обрабатывать длительные операции, не мешая работе пользователя.
- В реальных приложениях оба подхода комбинируются, но критически важные действия (сеть, работа с файловой системой) почти всегда выполняются асинхронно.

Где используется асинхронность.

JavaScript однопоточный → выполняет только одно действие за раз.

Если бы он всегда работал синхронно, то:

- сайт зависал бы при загрузке больших картинок;
- кнопки не реагировали бы, пока идёт сетевой запрос;
- анимации тормозили бы во время тяжёлых вычислений.

👉 Асинхронный код решает проблему:

- даёт возможность не блокировать интерфейс;
- позволяет браузеру/серверу делать другие дела, пока задача «ждёт ответа»;
- делает программы отзывчивыми для пользователя.

Сеть (HTTP-запросы, работа с API)

Когда мы загружаем данные с сервера, это может занять секунды.

Если делать синхронно → сайт «замёрзнет».

Поэтому в JS запросы всегда асинхронные.

Пример:

```
1  console.log("Запрос отправлен...");
2
3  fetch("https://jsonplaceholder.typicode.com/posts/10")
4    .then(response => {
5      // Ответ приходит в формате JSON, нужно преобразовать
6      return response.json();
7    })
8    .then(data => {
9      console.log("Ответ получен:", data);
10    })
11   .catch(error => {
12     console.error("Ошибка при запросе:", error);
13   });
14
15 console.log("Код продолжает выполняться...");
```


JSONPlaceholder — это бесплатный сервис, который имитирует работу настоящего API.

◆ Что делает этот код:

- `fetch` отправляет HTTP-запрос на `https://jsonplaceholder.typicode.com/posts/1`.
- Пока ответ не пришёл, JS идёт дальше → выводит "Код продолжает выполняться...".
- Когда ответ готов → срабатывает `.then(response => response.json())`.
- В следующем `.then` мы получаем готовый объект и выводим его.

```
PS C:\Users\user\Documents\JSCode\test1> node test.js
```

Запрос отправлен...

Код продолжает выполняться...

```
Ответ получен: {  
  userId: 1,  
  id: 10,  
  title: 'optio molestias id quia eum',  
  body: 'quo et expedita modi cum officia vel magni\n' +  
    'doloribus qui repudiandae\n' +  
    'vero nisi sit\n' +  
    'quos veniam quod sed accusamus veritatis error'  
}
```

```
PS C:\Users\user\Documents\JSCode\test1> node test.js
```

Таймеры (setTimeout, setInterval)

Таймеры позволяют запланировать действия на будущее.

`setTimeout` — выполнить через N миллисекунд.

`setInterval` — выполнять регулярно каждые N миллисекунд.

```
1 console.log("Начало");  
2  
3 setTimeout(() => console.log("Прошло 2 секунды"), 2000);  
4  
5 console.log("Конец");
```

```
<body>
  <p id="msg">Hello</p>
  <script src="test.js"></script>
</body>
```

```
let visible = true;

setInterval(() => {
  const p = document.getElementById("msg");
  p.style.visibility = visible ? "hidden" : "visible";
  visible = !visible;
}, 500);
```

◆ Ещё примеры асинхронных задач

- Работа с базами данных.
- Чтение/запись файлов (на сервере).
- WebSockets (постоянное соединение с сервером).
- Анимации и рендеринг в браузере.



Выводы.

Асинхронный код нужен для того, чтобы:

- не блокировать поток выполнения;
- делать интерфейсы отзывчивыми;
- обрабатывать долгие операции (сеть, таймеры, события, ввод/вывод).

Асинхронность — это «я позвоню, когда будет готово», вместо «жди, пока я сделаю».

4. Call Stack (стек вызовов).

Call Stack — это структура данных (стек), в которой JavaScript хранит информацию о том, какая функция сейчас выполняется и кто её вызвал.

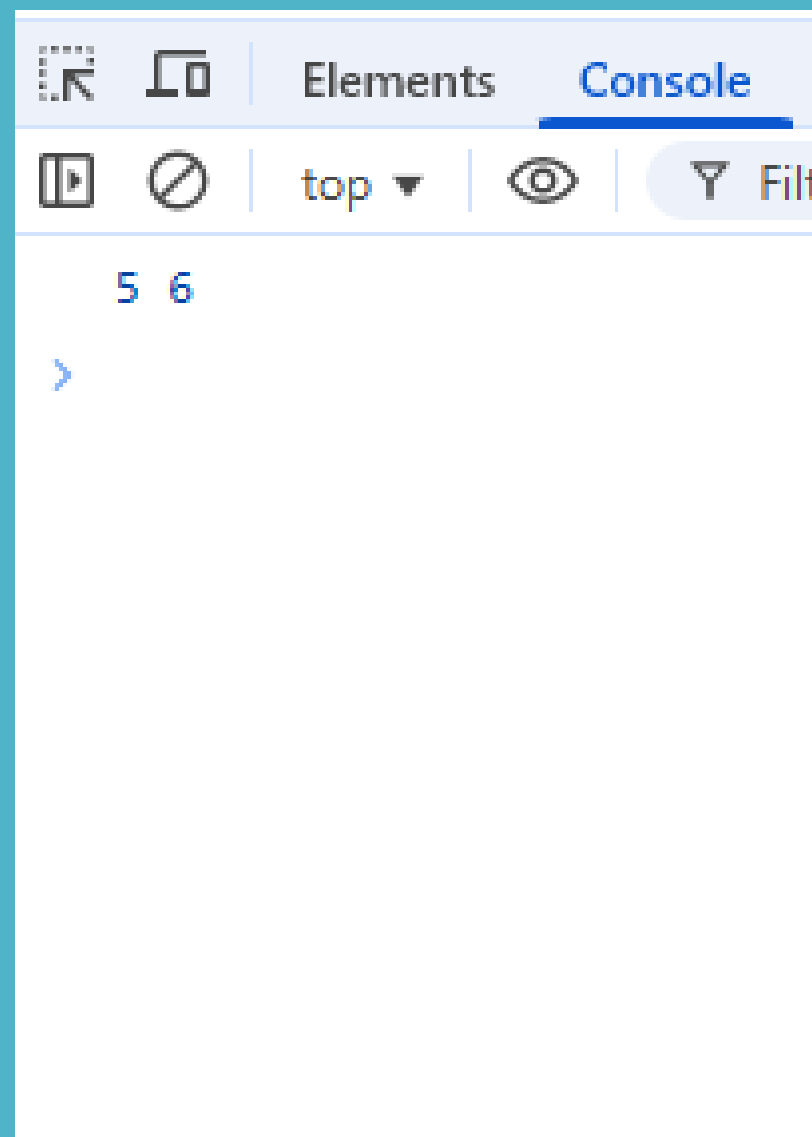
Стек работает по принципу LIFO (Last In — First Out): последняя вызванная функция выполняется первой.

Как туда попадают функции?

- Когда вызывается функция — она помещается в стек.
- Если внутри этой функции вызывается другая функция — она тоже помещается в стек (поверх предыдущей).
- Когда функция заканчивает выполнение — она убирается из стека.

Пример:

```
1  function sum(a,b) {  
2    |   return a + b  
3  }  
4  
5  function mul(a, b) {  
6    |   return a * b  
7  }  
8  
9  const x = sum(2, 3)  
10 const y = mul(2, 3)  
11 console.log(x, y)  
12
```



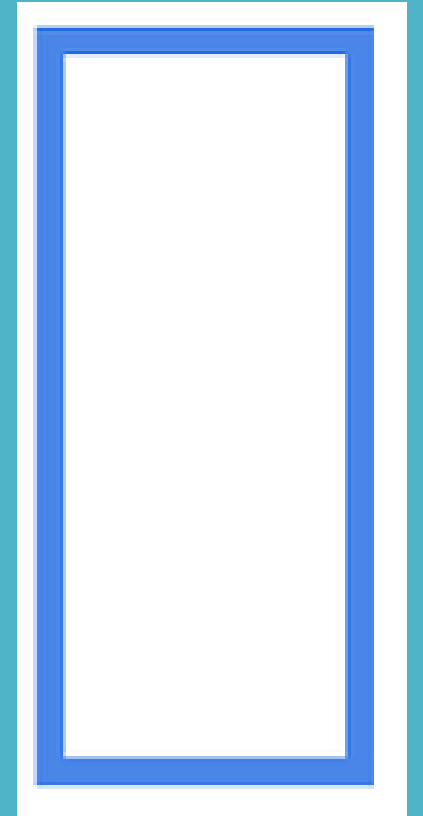
1. Объявление функций

```
function sum(...) {}
```

```
function mul(...) {}
```

На этом этапе стек пустой.

Функции просто загружаются в память, но в стек не попадают, потому что они ещё не вызываются.



2. Вызов `sum(2,3)`

В стек помещается вызов `sum`.

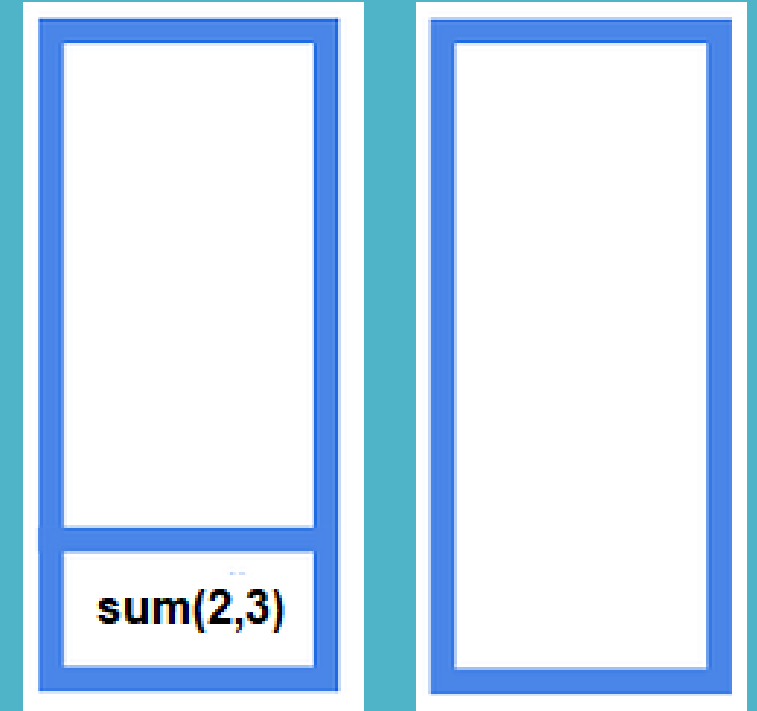
Стек: [`sum`]

Выполняется тело функции →
возвращается $2 + 3 = 5$.

После `return` функция убирается из
стека.

Стек: []

Результат (5) сохраняется в `x`.



3. Вызов `mul(2,3)`

В стек помещается вызов `mul`.

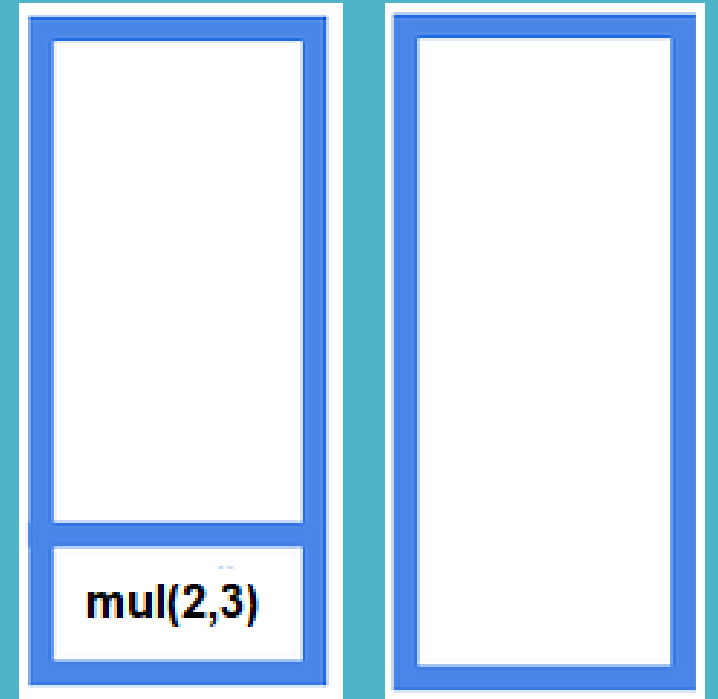
Стек: [`mul`]

Выполняется тело функции →
возвращается $2 * 3 = 6$.

После `return` функция убирается из
стека.

Стек: []

Результат (6) сохраняется в `у`.



4. Вызов `console.log(x, y)`

В стек помещается вызов `console.log`.

Стек: [`console.log`]

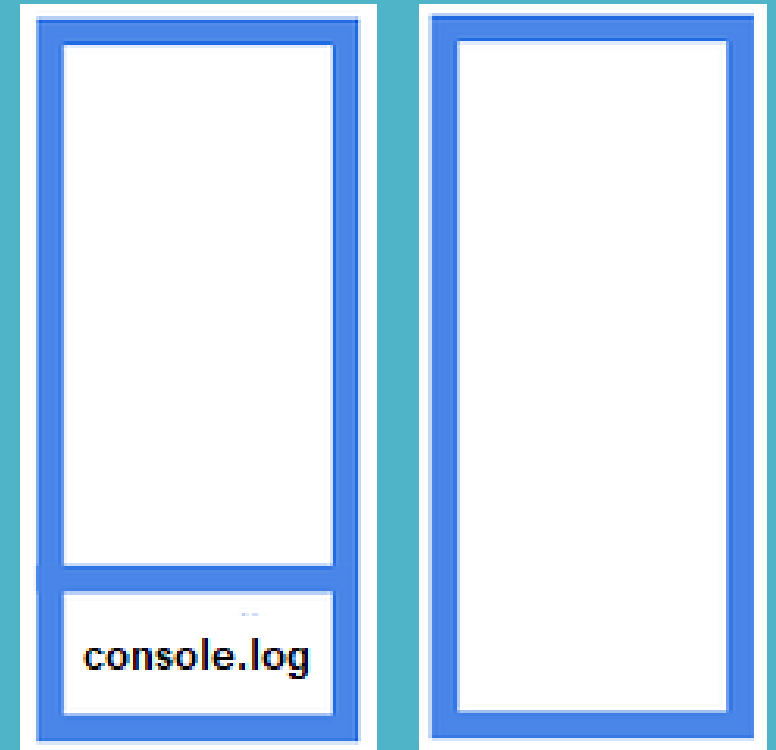
Внутри `console.log` движок вызывает нативный код (C++ внутри браузера/Node.js).

Выводит:

5 6

`console.log` завершается → убирается из стека.

Стек: []



Таким образом:

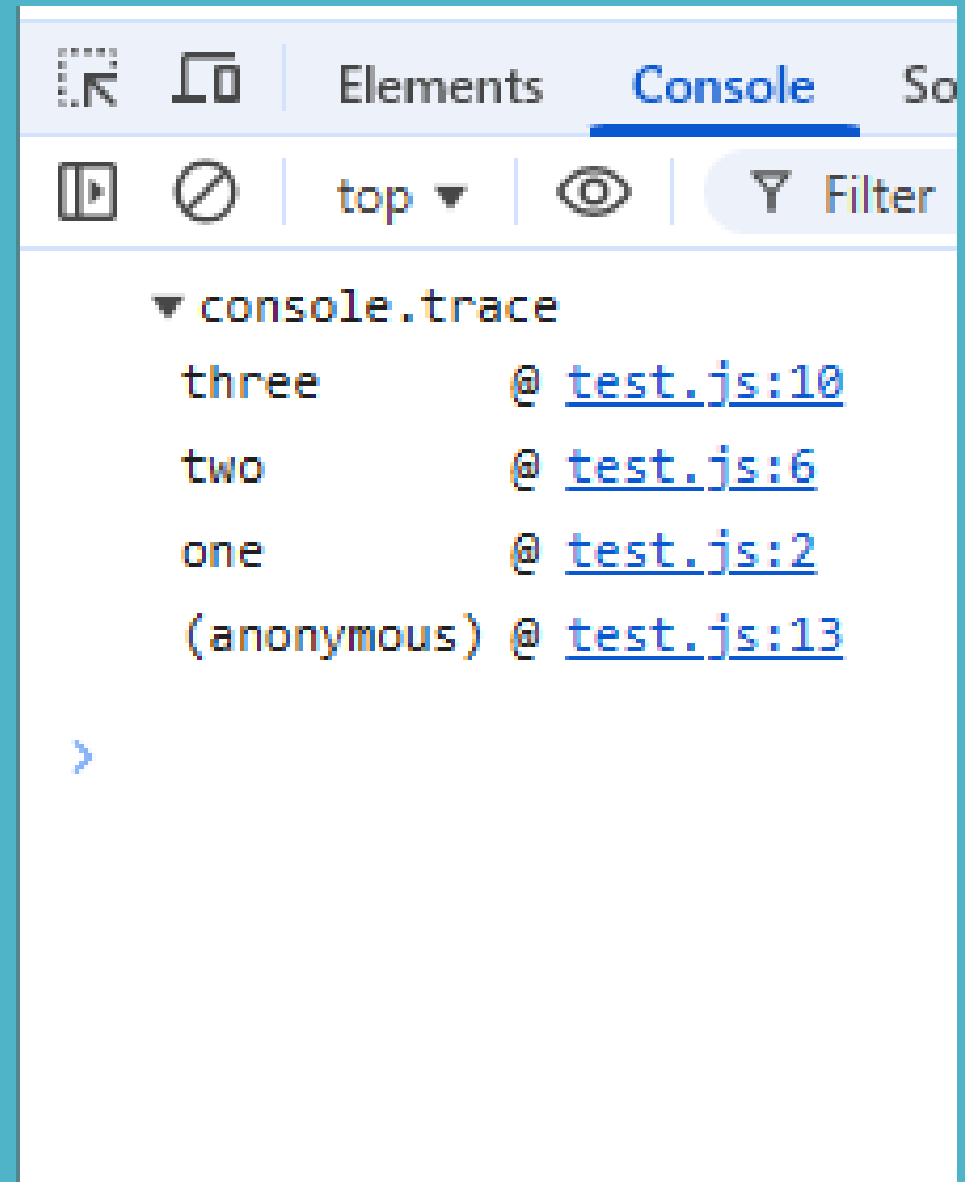
Стек всегда «растёт» при вызове функции.

Как только функция доходит до `return` или завершается — она убирается.

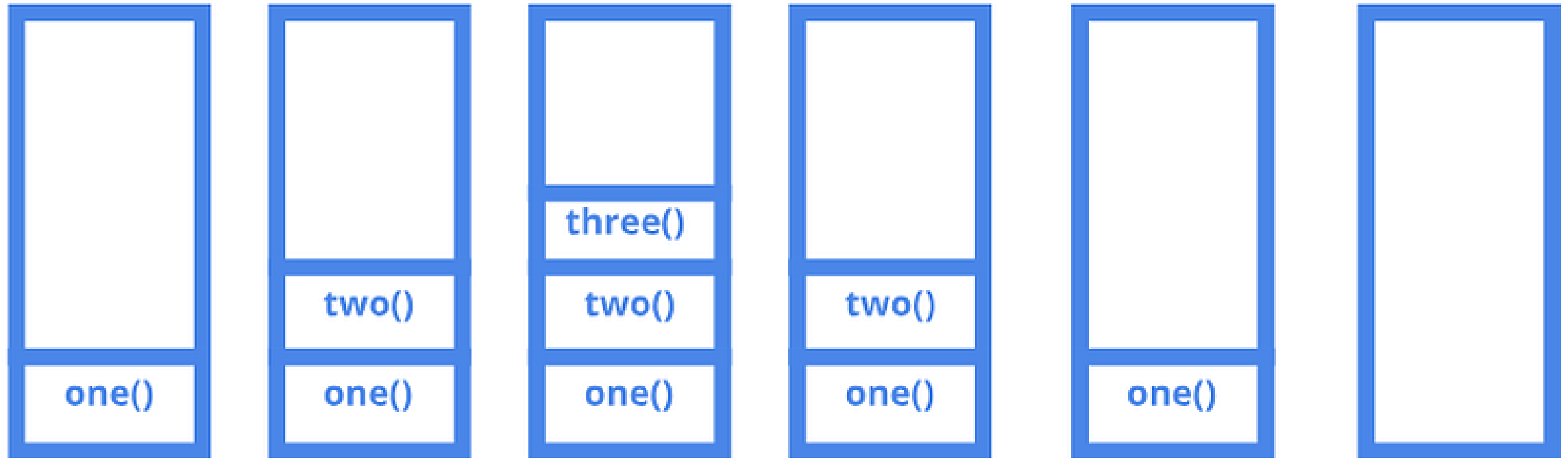
В конце программа завершается с **пустым стеком**.

Пример 2:

```
1  function one() {  
2    two()  
3  }  
4  
5  function two() {  
6    three()  
7  }  
8  
9  function three() {  
10   console.trace()  
11 }  
12  
13 one()  
14
```



Call Stack



Переполнение стека

```
1  function factorial(n) {  
2      if (n === 1) {  
3          return 1  
4      }  
5      console.trace(`Вызов factorial(${n})`)  
6      return n * factorial(n - 1)  
7  }  
8  
9  console.log(factorial(5))  
10  
11  
12  
13
```

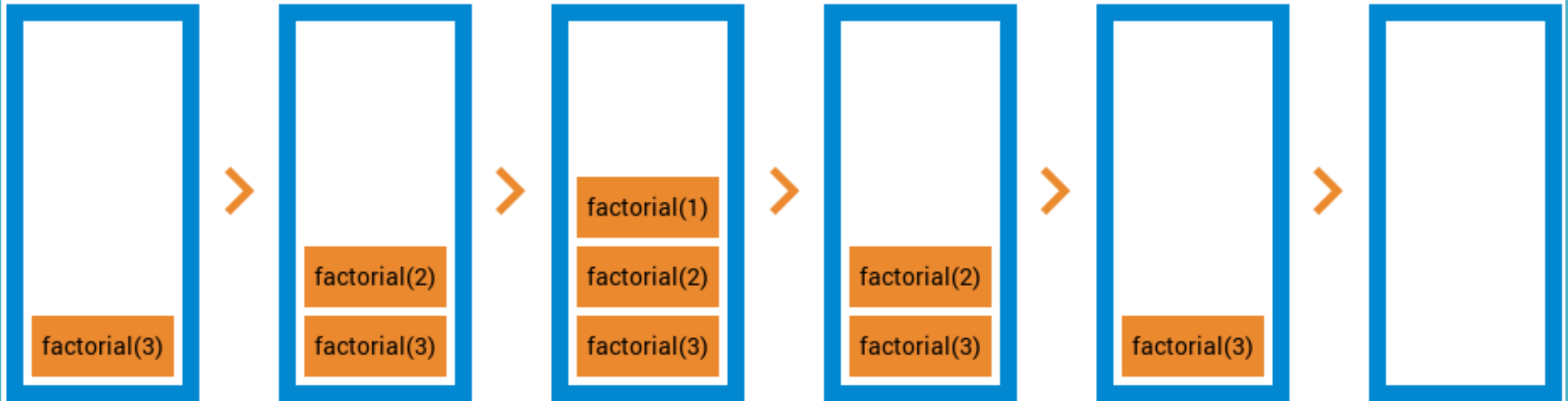
▼ Вызов factorial(5)
factorial @ test.js:5
(anonymous) @ test.js:9

▼ Вызов factorial(4)
factorial @ test.js:5
factorial @ test.js:6
(anonymous) @ test.js:9

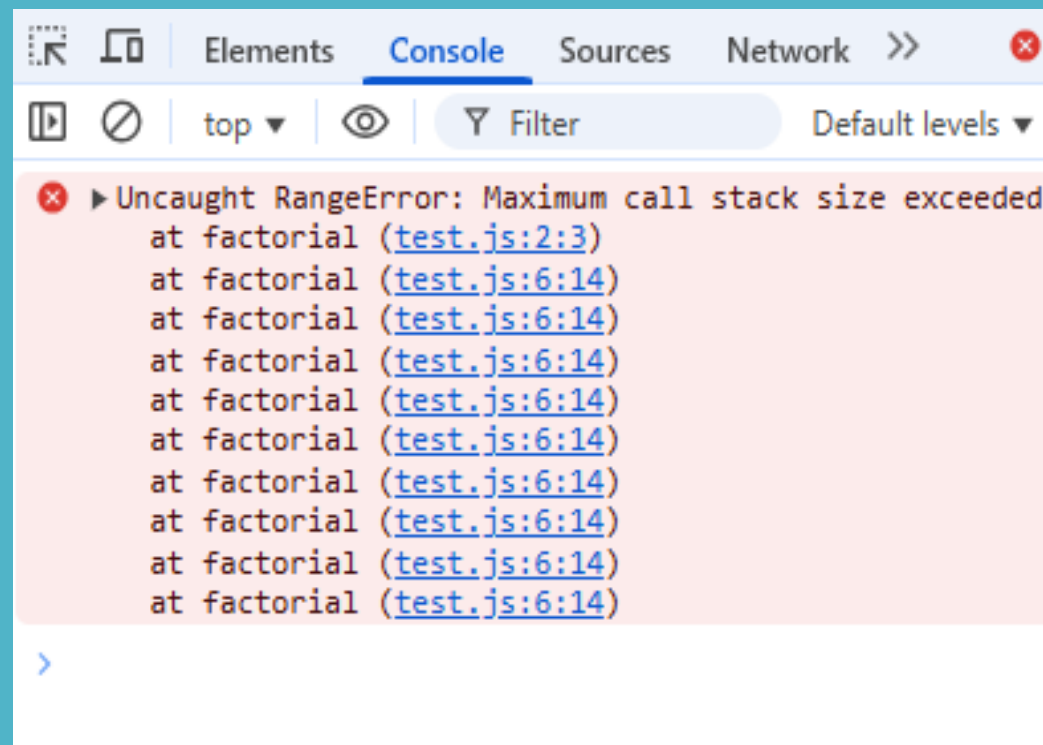
▼ Вызов factorial(3)
factorial @ test.js:5
factorial @ test.js:6
factorial @ test.js:6
(anonymous) @ test.js:9

▼ Вызов factorial(2)
factorial @ test.js:5
factorial @ test.js:6
factorial @ test.js:6
factorial @ test.js:6
(anonymous) @ test.js:9

Call stack



```
1 function factorial(n) {  
2   if (n === 1n) {  
3     return 1n  
4   }  
5   //console.trace(`Вызов factorial(${n})`)  
6   return n * factorial(n - 1n)  
7 }  
8  
9 console.log(factorial(10000n))  
10  
11
```



Что происходит при возврате из функции?

Когда выполнение функции доходит до конца (или встречается `return`), интерпретатор удаляет эту функцию из стека и продолжает выполнение с того места, где вызвали эту функцию.

Что будет при переполнении стека?

Если функция вызывает сама себя бесконечно (или слишком глубоко вызывает другие), стек никогда не освободится, и интерпретатор выдаст ошибку:

Uncaught RangeError: Maximum call stack size exceeded

5. Event Loop.

Event Loop — это механизм в JavaScript, который управляет выполнением кода, обработкой событий и асинхронных операций.

Он следит за тем, чтобы синхронный код выполнялся сразу, а асинхронные задачи обрабатывались позже — в правильном порядке.

Почему JavaScript однопоточный?

Движок JavaScript (например, V8 в Chrome/Node.js) работает в одном потоке.

Это значит, что в каждый момент времени выполняется только одна инструкция.

Однопоточность выбрана намеренно:

- исключает гонки данных и конфликты при работе с DOM;
- упрощает модель программирования;
- гарантирует, что код выполняется последовательно.

Но в браузере вокруг JS-движка есть WebAPI (таймеры, DOM-события, сетевые запросы), которые работают параллельно и «подкладывают» задачи обратно в JavaScript через очереди.

Что такое webAPI?

Web API (веб-API) — это встроенные в браузер интерфейсы для доступа к системным возможностям: DOM, таймеры, сеть, геолокация и т.д.

Web API — это набор встроенных в браузер (или Node.js) функций и объектов, которые расширяют возможности JavaScript.

Сам язык JS очень «маленький» (он умеет только работать с числами, строками, массивами, функциями и т.п.).

Все «магические» возможности — таймеры, работа с DOM, запросы по сети, события — приходят извне, из Web API.

Как Event Loop управляет синхронным и асинхронным кодом?

Синхронный код

- Выполняется немедленно.
- Функции попадают в Call Stack (стек вызовов) и выполняются сверху вниз.

Асинхронный код

- Когда JS встречает `setTimeout`, `fetch`, обработчик события — он не ждёт выполнения.
- Задача отправляется во внешнюю среду (например, WebAPI).
- После выполнения внешняя среда возвращает колбэк в одну из очередей задач.
- Event Loop проверяет, когда стек вызовов пуст → берёт задачу из очереди и помещает её в Call Stack.

Взаимодействие Call Stack и очередей задач

- Call Stack (Стек вызовов). Хранит список функций, которые выполняются в данный момент.
- WebAPI (браузер). Запускает асинхронные задачи (таймеры, обработка событий, сетевые запросы).
- Очереди задач
 - Task Queue (**Macrotasks**). Сюда попадают:
 - *setTimeout*,
 - *setInterval*,
 - *обработчики событий (click, load)*,
 - *setImmediate (Node.js)*.

- Microtask Queue (Микрозадачи). Более приоритетная очередь. Сюда попадают:
 - Promise.then,
 - Promise.catch,
 - Promise.finally,
 - queueMicrotask(),
 - MutationObserver.
- Render Queue:
 - Очередь перерисовки интерфейса (браузер обновляет DOM).
 - Она выполняется после microtasks и перед следующим macrotask.

Алгоритм работы Event Loop

1. Выполнить весь синхронный код в Call Stack.
2. Выполнить все microtasks (до конца).
3. Обновить интерфейс (Render Queue).
4. Взять одну задачу из Task Queue (macrotask).
5. Вернуться к шагу 2.

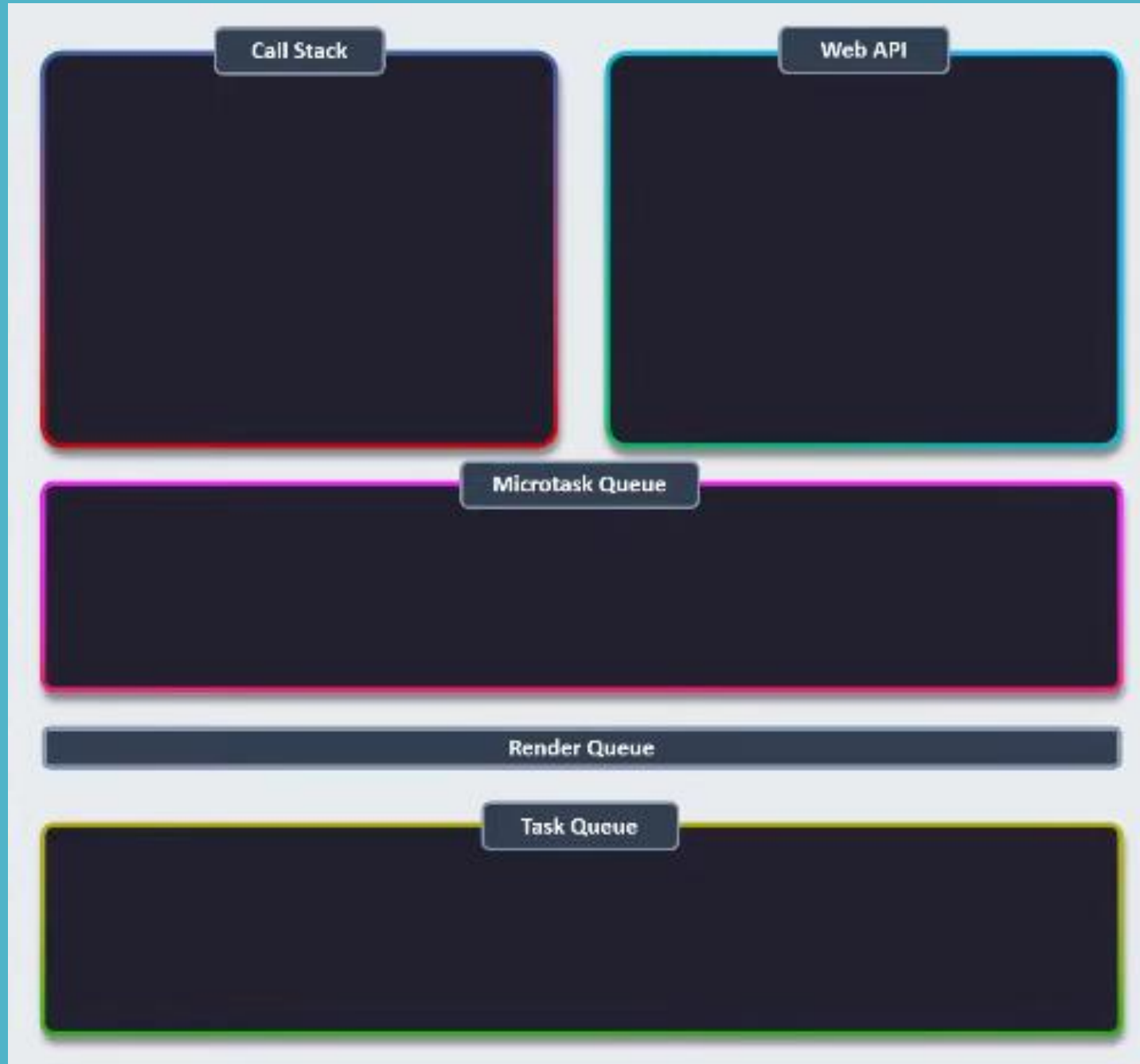
Call Stack

Web API

Microtask Queue

Render Queue

Task Queue



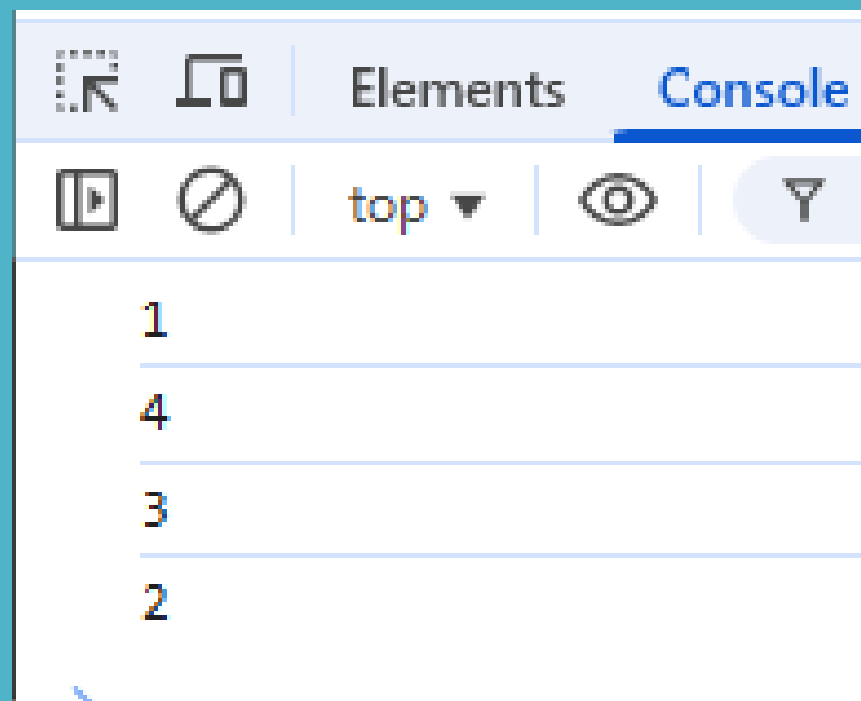
Пример:

```
console.log("1");

setTimeout(() => console.log("2"), 0);

Promise.resolve().then(() => console.log("3"));

console.log("4");
```



1. `console.log("1")` → сразу (Call Stack).
2. `setTimeout` → уходит в WebAPI, его колбэк попадёт в Task Queue.
3. `Promise.then` → добавляется в Microtask Queue.
4. `console.log("4")` → сразу (Call Stack).
5. После завершения стека: выполняем Microtask Queue (3).
6. Затем Task Queue (2).

Таким образом, Event Loop — это «дирижёр», который следит, чтобы код исполнялся строго в порядке:

сначала стек → потом микрозадачи →

потом макрозадачи → потом рендер → заново.

6. Macro- и Microtasks.

Task Queue (Очередь задач) — это структура данных в JavaScript Event Loop, которая содержит callback-функции, готовые к выполнению после завершения операций Web API.

Task Queue — это FIFO очередь (First In, First Out), куда Web APIs помещают завершённые callback-функции для последующего выполнения в основном потоке JavaScript.

Что попадает в Task Queue?

Macrotasks (основные задачи):

- javascript// setTimeout callbacks
- setInterval callbacks
- DOM events
- HTTP requests (fetch, XMLHttpRequest)

Microtask Queue (Очередь микрозадач) — это специальная высокоприоритетная очередь в JavaScript Event Loop, которая содержит callback-функции с более высоким приоритетом выполнения, чем обычная Task Queue.

Microtask Queue — это FIFO очередь с наивысшим приоритетом, которая полностью очищается перед обработкой любой задачи из Task Queue.

В Microtask Queue обычно попадают:

- Промисы (`.then()`, `.catch()`, `.finally()`): Когда промис разрешается (`resolves`) или отклоняется (`rejects`), его колбэк-функции ставятся в очередь микрозадач.
- `await`: Если `await` находится внутри `async` функции, код после него будет выполняться как микрозадача, после того как промис, на который указывает `await`, будет разрешён.
- `queueMicrotask()`: Это специальная функция, которая позволяет явно добавить свой колбэк в очередь микрозадач.

Render Queue (Очередь рендеринга) — это специальная очередь в браузере, которая управляет обновлением визуального отображения страницы (перерисовкой DOM, CSS, анимаций).

Render Queue содержит задачи по **обновлению внешнего вида** веб-страницы: перерисовка элементов, применение CSS, анимации, скролл и т.д.

Основные события, которые могут инициировать процесс отрисовки:

- **Изменения в DOM:** Когда вы добавляете, удаляете или изменяете элементы на странице с помощью JavaScript, это может вызвать пересчёт стилей и перерисовку.
- **Изменения в CSS:** Изменение CSS-свойств, например, цвета или размера элемента.
- **События браузера:** Некоторые события, такие как изменение размера окна или прокрутка страницы, могут запускать процесс рендеринга.
- **Запрос на анимацию:** Функции, такие как `requestAnimationFrame()`, специально созданы для синхронизации вашего кода с циклом отрисовки браузера.

В конечном итоге, браузер собирает все эти изменения и выполняет их в одном цикле рендеринга.

Microtask Queue

```
() => { ... };
```

```
() => { ... };
```

```
() => { ... };
```

Render Queue

```
() => { ... };
```

```
() => { ... };
```

```
() => { ... };
```

Task Queue

Приоритеты в Event Loop

1. Сначала — Call Stack (синхронный код). Всё, что написано напрямую в коде, выполняется сразу. Пока стек не опустеет, никакие асинхронные задачи не начнут выполняться.
2. Microtasks (очередь микрозадач). После завершения текущего стека выполняются **все** задачи из Microtask Queue, пока очередь не станет пустой. Они имеют высший приоритет среди асинхронных задач.
3. Macrotasks (Task Queue, очередь макрозадач). Когда стек пуст и microtasks выполнены, Event Loop берёт **одну** задачу из Task Queue и выполняет её. После выполнения этой задачи снова проверяются microtasks → и только потом следующая макрозадача.
4. Render Queue (отрисовка интерфейса). После выполнения microtasks и макрозадачи, браузер может обновить UI. Обычно происходит ~60 раз в секунду (каждые 16 мс). Если microtasks будут бесконечно добавляться, отрисовка может "зависнуть" (фриз интерфейса).

Заключение

JavaScript однопоточен — выполняется только одна операция в один момент времени, через Call Stack (стек вызовов).

Для асинхронных задач среда выполнения (браузер или Node.js) подключает дополнительные механизмы:

- WebAPI (таймеры, события, сеть и т. д.).
- Встроенный в движок механизм микрозадач (Promises, async/await).

Классификация задач:

- Microtask Queue → Promise.then, async/await, queueMicrotask. Добавляются напрямую в очередь микрозадач (WebAPI не участвует).
- Task Queue (Macrotasks) → setTimeout, setInterval, обработчики DOM-событий, сетевые запросы (fetch). Сначала обрабатываются в WebAPI (ожидание времени/события), затем колбэки переходят в очередь макрозадач.

Порядок работы Event Loop:

- Выполнить весь синхронный код (Call Stack).
- Выполнить **все** микрозадачи (Microtask Queue).
- Взять **одну** макрозадачу (Task Queue).
- Обновить интерфейс (Render Queue).
- Повторить цикл.

Главная мысль:

JavaScript хоть и однопоточный, но благодаря Event Loop и разделению задач на микро- и макрозадачи он умеет эффективно управлять асинхронными операциями (сеть, события, таймеры), сохраняя отзывчивость интерфейса.

<https://vault-developer.github.io/event-loop-explorer/>

Контрольные вопросы:

- Почему JavaScript называют однопоточным языком?
- Чем отличается синхронный код от асинхронного?
- Приведите примеры задач, которые невозможно эффективно решать без асинхронности.
- Что произойдёт, если в JS написать бесконечный цикл `while(true)`?
- Для чего используются таймеры и сетевые запросы в асинхронном коде?
- Что такое стек вызовов и как он работает?
- Что произойдет при переполнении стека вызовов?
- Какую роль играет Event Loop в JavaScript?
- В чем разница между `macro-` и `microtasks`?
- Почему `Promise.then` выполняется раньше, чем `setTimeout(fn, 0)`?

Домашнее задание:

1. <https://ru.hexlet.io/courses/js-asynchronous-programming>

1	Введение	Знакомимся с курсом
2	Стек вызовов (Call Stack)	Разбираемся с тем, как работает стек вызовов
3	Асинхронный код	Знакомимся с работой асинхронного кода

2. Повторить материал лекции.

Материалы лекций:

<https://github.com/ShViktor72/Education2025>

Обратная связь:

colledge20education23@gmail.com