

## **ПМЗ Разработка модулей ПО.**

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

# **Тема 2. Асинхронно программирование.**

## **Лекция 9. Таймеры и упорядочивание асинхронных операций.**

# Цель занятия:

Ознакомиться с работой встроенных таймеров JavaScript, научиться управлять их выполнением, понимать ограничения и использовать таймеры в практических задачах.

# **Учебные вопросы:**

- 1. Механизм таймеров.**
- 2. `setTimeout` и `setInterval`: назначение, синтаксис, примеры.**
- 3. Рекурсивный `setTimeout` как альтернатива `setInterval`.**
- 4. Практические примеры.**
- 5. Упорядочивание асинхронных операций.**

# 1. Механизм таймеров

JavaScript работает в однопоточном режиме, то есть в каждый момент времени выполняется только один кусок кода в Call Stack.

Чтобы не блокировать выполнение, браузер (или Node.js) предоставляет механизмы таймеров.

## ◆ Что происходит «под капотом»:

1. Мы вызываем `setTimeout` или `setInterval` → они не выполняются сразу.
2. Эти функции передают задачу во встроенное Web API браузера (или Node.js API), которое «следит за временем».
3. Когда задержка истекла, колбэк помещается в Task Queue (очередь макрозадач).
4. Event Loop проверяет:
  - если Call Stack пуст,
  - и выполнены все микрозадачи (Promise, `async/await`),  
тогда из Task Queue берётся наш колбэк и выполняется.

## ◆ Важное следствие

- `setTimeout(..., 0)` никогда не выполнится мгновенно — его колбэк попадёт в очередь задач и выполнится только после текущего кода и всех микрозадач.
- Таймеры не гарантируют точного времени выполнения, они лишь ставят задачу «не раньше, чем через X миллисекунд».

## 2. `setTimeout` и `setInterval`: назначение, синтаксис, примеры.

### ◆ `setTimeout`

**Назначение:** выполнить функцию **один раз** через заданный промежуток времени.



## Синтаксис `setTimeout()` :

**`setTimeout(функция, задержка, [аргумент1, аргумент2, ...])`**

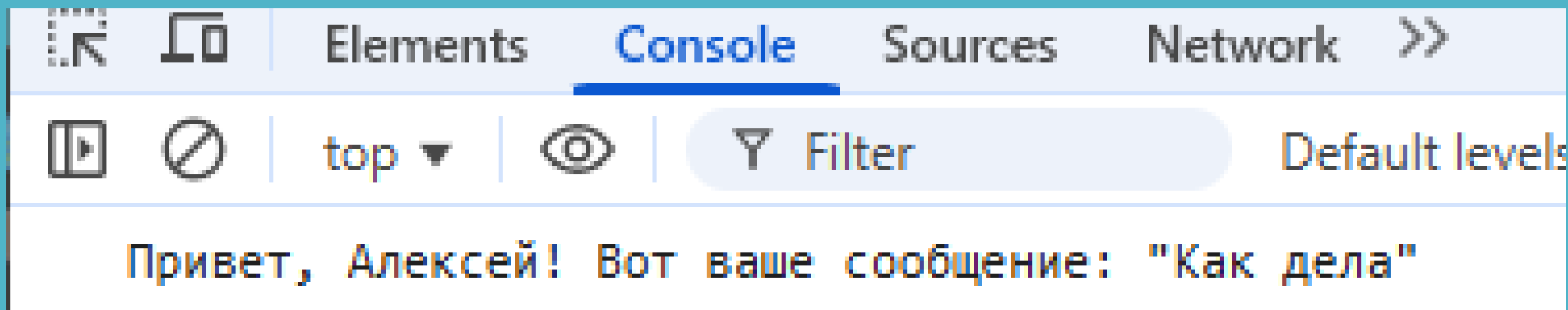
- **функция или строка:** Функция, которую нужно выполнить. Чаще всего используют анонимную функцию `(( ) => { ... })`. Также можно передать строку кода, но это не рекомендуется из соображений безопасности и производительности.
- **задержка:** Время в миллисекундах (`1000 мс = 1 секунда`), по истечении которого функция будет выполнена. Это минимальное время задержки, так как браузер может отложить выполнение, если основной поток занят.
- **[аргумент1, аргумент2, ...]:** Необязательные аргументы, которые будут переданы в вашу функцию, когда она будет вызвана.
- **`setTimeout()`** возвращает идентификатор (число), который можно использовать для отмены вызова с помощью `clearTimeout()`

## ◆ Простой пример с задержкой:

```
1  // Выведет "Hello, world!" в консоль через 2 секунды.  
2  setTimeout(() => {  
3    |   console.log('Hello, world!');  
4    | }, 2000);
```

## ◆ Использование с аргументами:

```
1 function showMessage(name, message) {  
2   | console.log(`Привет, ${name}! Вот ваше сообщение: "${message}"`);  
3 }  
4  
5 // Вызовет showMessage через 3 секунды, передав аргументы.  
6 setTimeout(showMessage, 3000, 'Алексей', 'Как дела');  
7
```



## ◆ Отмена с помощью clearTimeout():

```
1  const timerId = setTimeout(() => {  
2    | console.log('Этот текст никогда не появится.');
```

```
3  }, 5000);  
4  
5  console.log('Таймер запущен.');
```

```
6  
7  // Отменяем таймер до того как он сработает.  
8  clearTimeout(timerId);  
9  
10 console.log('Таймер отменен.');
```



top ▼



Filter

Default levels ▼

Таймер запущен.

Таймер отменен.

## **setInterval()**

**Назначение:** выполнять функцию **периодически**, через заданные промежутки времени.

Она, как и `setTimeout()`, является частью Web API.

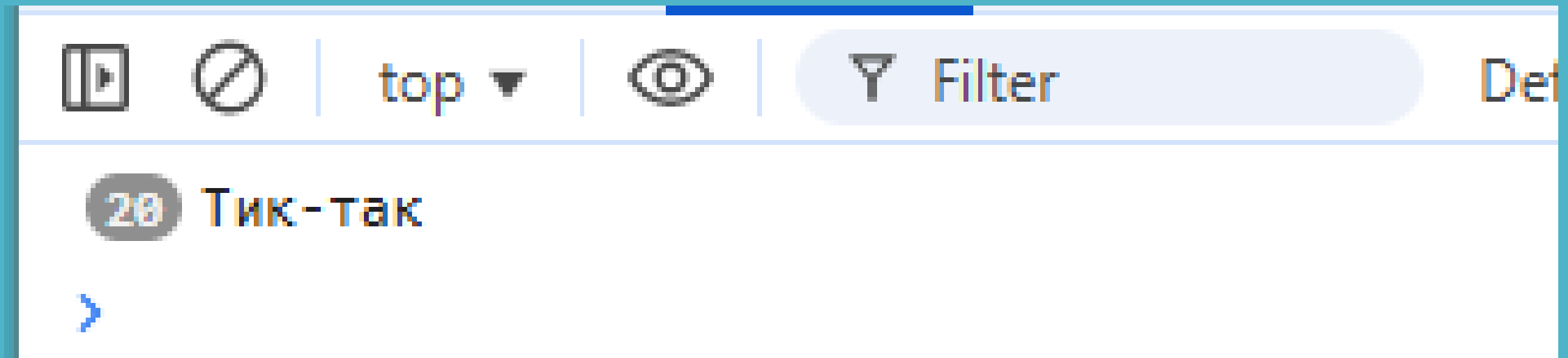
## Синтаксис setInterval():

**setInterval(функция, задержка, [аргумент1, аргумент2, ...])**

- **функция или строка:** Функция, которую нужно вызывать на каждом интервале. Как и в `setTimeout()`, рекомендуется использовать функцию, а не строку кода.
- **задержка в мс:** Интервал в миллисекундах между последовательными вызовами.
- **[аргумент1, аргумент2, ...]:** Необязательные аргументы, которые будут переданы в функцию при каждом вызове.
- **setInterval()** возвращает идентификатор, который можно использовать для остановки повторных вызовов с помощью `clearInterval()`.

◆ Простой пример с повторным выводом:

```
1 // Выводит "Тик-так" в консоль каждую секунду
2 setInterval(() => {
3   | console.log( 'Тик-так' );
4   }, 1000);
```



## ◆ Создание простого таймера:

```
1  let countdown = 5;
2  // Запускаем интервал, который будет уменьшать значение каждую секунду
3  const timer = setInterval(() => {
4    console.log(countdown);
5    countdown--;
6
7    // Когда отсчёт достигнет 0, мы останавливаем интервал
8    if (countdown === 0) {
9      clearInterval(timer);
10     console.log('Поехали!');
11   }
12 }, 1000);
```

5

4

3

2

1

Поехали!

t



## ◆ Отмена с помощью clearInterval():

```
1  const myInterval = setInterval(() => {  
2    | console.log('Этот текст будет появляться...');  
3    | }, 2000);  
4  
5  // Останавливаем интервал через 6 секунд  
6  setTimeout(() => {  
7    | clearInterval(myInterval);  
8    | console.log('...но теперь он остановлен.');
```

3 Этот текст будет появляться...

...но теперь он остановлен.



## ◆ Особенности

Оба метода возвращают **ID таймера**, который можно использовать для его остановки:

- `clearTimeout(timerId)`
- `clearInterval(intervalId)`

**`setTimeout(func, 0)` → колбэк выполнится после завершения текущего кода и микрозадач, а не мгновенно.**

В реальности задержка **минимум ~4 мс** в браузере (для вложенных таймеров или если вкладка неактивна).

**Важно:** без очистки `setInterval` будет работать бесконечно.

### 3. Рекурсивный `setTimeout` как альтернатива `setInterval`.

Дрейф интервалов (Interval Drift) — это явление, при котором реальное время между последовательными вызовами `setInterval()` постепенно увеличивается. ⌚

Это происходит потому, что браузер помещает каждую задачу в очередь, но не гарантирует, что она будет выполнена точно в срок.

## Почему это происходит?

JavaScript работает в однопоточном режиме. Это значит, что в любой момент времени может выполняться только одна задача. Когда вы используете `setInterval(функция, 1000)`, вы говорите браузеру: "Добавь эту функцию в очередь задач каждую секунду".

Однако, если основной поток JavaScript занят выполнением других, более долгих задач (например, сложными вычислениями, обработкой данных или отрисовкой), то запланированная задача из `setInterval()` вынуждена ждать, пока поток освободится.

Таким образом, время между вызовами будет равно не 1000 мс, а 1000 мс + время ожидания. Со временем это ожидание накапливается, и интервал "дрейфует" от своего первоначального значения.

```
1  let count = 0;
2  const startTime = Date.now();
3
4  // Запускаем интервал
5  const myInterval = setInterval(() => {
6    count++;
7    const actualTime = Date.now() - startTime;
8    const expectedTime = count * 1000;
9    const drift = actualTime - expectedTime;
10
11    console.log(`Прошло: ${actualTime} мс. Ожидалось: ${expectedTime} мс. Дрейф: ${drift} мс.`);
12  }, 1000);
13
14  // Имитация "тяжёлой" работы, которая блокирует основной поток
15  function blockMainThread() {
16    const start = Date.now();
17    // Этот цикл будет работать примерно 3 секунды
18    while (Date.now() - start < 3000) {
19      // Просто ждём и ничего не делаем, но поток занят
20    }
21  }
22
23  // Запускаем блокирующую функцию через 2 секунды
24  setTimeout(blockMainThread, 2000);
```

Прошло: 2132 мс.	Ожидалось: 1000 мс.	Дрейф: 1132 мс.	<a href="#">test.js:11</a>
Прошло: 5133 мс.	Ожидалось: 2000 мс.	Дрейф: 3133 мс.	<a href="#">test.js:11</a>
Прошло: 6016 мс.	Ожидалось: 3000 мс.	Дрейф: 3016 мс.	<a href="#">test.js:11</a>
Прошло: 7013 мс.	Ожидалось: 4000 мс.	Дрейф: 3013 мс.	<a href="#">test.js:11</a>
Прошло: 8016 мс.	Ожидалось: 5000 мс.	Дрейф: 3016 мс.	<a href="#">test.js:11</a>
Прошло: 9013 мс.	Ожидалось: 6000 мс.	Дрейф: 3013 мс.	<a href="#">test.js:11</a>
Прошло: 10006 мс.	Ожидалось: 7000 мс.	Дрейф: 3006 мс.	<a href="#">test.js:11</a>
Прошло: 11009 мс.	Ожидалось: 8000 мс.	Дрейф: 3009 мс.	<a href="#">test.js:11</a>
Прошло: 12004 мс.	Ожидалось: 9000 мс.	Дрейф: 3004 мс.	<a href="#">test.js:11</a>
Прошло: 13004 мс.	Ожидалось: 10000 мс.	Дрейф: 3004 мс.	<a href="#">test.js:11</a>
Прошло: 14002 мс.	Ожидалось: 11000 мс.	Дрейф: 3002 мс.	<a href="#">test.js:11</a>

Как избежать дрейфа интервалов?

Чтобы избежать дрейфа, вместо `setInterval()` лучше использовать рекурсивный `setTimeout()`.

Этот подход гарантирует, что следующий вызов будет запланирован только после того, как предыдущий завершился.



```
1  let count = 0;
2  const startTime = Date.now();
3  const interval = 1000; // Наш желаемый интервал в 1000 мс
4  function scheduleNextCall() {
5      count++;
6      const actualTime = Date.now() - startTime; // Вычисляем, сколько времени прошло с начала
7      const expectedTime = count * interval;    // Вычисляем, сколько времени должно было пройти
8      const drift = actualTime - expectedTime;   // Вычисляем дрейф
9      console.log(`Прошло: ${actualTime} мс. Ожидалось: ${expectedTime} мс. Дрейф: ${drift} мс.`);
10     // Вычисляем, через сколько времени нужно запланировать следующий вызов
11     // Если дрейф положительный, уменьшаем задержку
12     // Если дрейф слишком большой, то следующая задержка будет 0, и функция вызовется сразу
13     const nextCallIn = interval - drift;
14     // Запускаем следующий вызов. Используем Math.max(0, nextCallIn), чтобы избежать отр. значений
15     setTimeout(scheduleNextCall, Math.max(0, nextCallIn));
16 }
17 // Запускаем первый вызов, чтобы начать цепочку
18 setTimeout(scheduleNextCall, interval);
19 // Имитация "тяжёлой" работы, чтобы показать дрейф
20 function blockMainThread() {
21     const start = Date.now();
22     while (Date.now() - start < 3000) {
23         // Блокируем поток на 3 секунды
24     }
25 }
26 // Запускаем блокирующую функцию через 2 секунды
27 setTimeout(blockMainThread, 2000);
```



Прошло: 1486 мс.	Ожидалось: 1000 мс.	Дрейф: 486 мс.	<a href="#">test.js:17</a>
Прошло: 2003 мс.	Ожидалось: 2000 мс.	Дрейф: 3 мс.	<a href="#">test.js:17</a>
Прошло: 5005 мс.	Ожидалось: 3000 мс.	Дрейф: 2005 мс.	<a href="#">test.js:17</a>
Прошло: 5005 мс.	Ожидалось: 4000 мс.	Дрейф: 1005 мс.	<a href="#">test.js:17</a>
Прошло: 5005 мс.	Ожидалось: 5000 мс.	Дрейф: 5 мс.	<a href="#">test.js:17</a>
Прошло: 6009 мс.	Ожидалось: 6000 мс.	Дрейф: 9 мс.	<a href="#">test.js:17</a>
Прошло: 7013 мс.	Ожидалось: 7000 мс.	Дрейф: 13 мс.	<a href="#">test.js:17</a>
Прошло: 8004 мс.	Ожидалось: 8000 мс.	Дрейф: 4 мс.	<a href="#">test.js:17</a>
Прошло: 9003 мс.	Ожидалось: 9000 мс.	Дрейф: 3 мс.	<a href="#">test.js:17</a>
Прошло: 10892 мс.	Ожидалось: 10000 мс.	Дрейф: 892 мс.	<a href="#">test.js:17</a>
Прошло: 11884 мс.	Ожидалось: 11000 мс.	Дрейф: 884 мс.	<a href="#">test.js:17</a>
Прошло: 12882 мс.	Ожидалось: 12000 мс.	Дрейф: 882 мс.	<a href="#">test.js:17</a>
Прошло: 13697 мс.	Ожидалось: 13000 мс.	Дрейф: 697 мс.	<a href="#">test.js:17</a>
Прошло: 14007 мс.	Ожидалось: 14000 мс.	Дрейф: 7 мс.	<a href="#">test.js:17</a>

# 4. Практические примеры.


Обратный отсчет:

```
1  <p id="timer"></p>
2  <script>
3      let count = 10;
4      const timerEl = document.getElementById("timer");
5
6      const intervalId = setInterval(() => {
7          timerEl.textContent = count;
8          if (count === 0) {
9              clearInterval(intervalId);
10             timerEl.textContent = "🚀 Старт!";
11         }
12         count--;
13     }, 1000);
14 </script>
```

Часы:

```
1  <p id="clock"></p>
2  <script>
3      function updateClock() {
4          document.getElementById("clock")
5              .textContent = new Date()
6              .toLocaleTimeString();
7      }
8      updateClock(); // сразу показать
9      setInterval(updateClock, 1000);
10 </script>
```

# Светофор:

```
<div id="light" style="width:100px; height:100px; border-radius:50%; background:  red;"></div>

<script>
  const light = document.getElementById("light");
  const colors = ["red", "yellow", "green"];
  let i = 0;

  function changeColor() {
    light.style.background = colors[i];
    i = (i + 1) % colors.length;
    setTimeout(changeColor, i === 1 ? 500 : 2000);
    // жёлтый — 0.5 сек, остальные — 2 сек
  }
  changeColor();
</script>
```

## Заключение

Сегодня мы разобрали, как JavaScript работает с таймерами и почему они важны для организации асинхронного кода:

- Таймеры (`setTimeout`, `setInterval`) работают не напрямую, а через Event Loop и WebAPI:
- `setTimeout` — откладывает выполнение функции на указанное время.
- `setInterval` — выполняет функцию периодически с заданным интервалом.

Очистка таймеров обязательна, если больше не нужен повторный вызов, чтобы не перегружать приложение (`clearTimeout`, `clearInterval`).

Минимальная задержка в браузере ограничена: даже `setTimeout(fn, 0)` реально выполняется не мгновенно, а после текущего стека и микрозадач (обычно не быстрее 4 мс).

Дрейф интервалов — проблема `setInterval`: если код внутри колбэка выполняется дольше задержки, то вызовы начинают «накладываться», что может привести к накоплению задач.

Рекурсивный `setTimeout` решает проблему дрейфа: он запускает новый таймер только после завершения предыдущего кода, позволяя более точно управлять временем.

## 👉 Главная мысль:

Таймеры в JavaScript — это не гарантия «точного времени», а лишь механизм планирования задач через Event Loop.

Для стабильного поведения нужно понимать их ограничения (задержка, дрейф) и выбирать правильный подход (`setInterval` или рекурсивный `setTimeout`).

# 5. Упорядочивание асинхронных операций.

## Проблемы асинхронности.

- **Гонки данных:** Возникают, когда несколько асинхронных операций пытаются изменить одно и то же состояние.
- **Неопределенный порядок выполнения:** Асинхронные операции могут завершаться в произвольном порядке, что затрудняет управление логикой приложения.

Рассмотрим простой пример **гонки данных**, где два асинхронных процесса изменяют одно и то же значение.

```
let counter = 0;
function incrementCounter() {
  const delay = Math.random() * 1000; // Случайная задержка
  setTimeout(() => {
    const currentValue = counter; // Читаем текущее значение
    console.log(`Текущая значение: ${currentValue}`);
    // Случайная логика, чтобы увеличить счетчик
    // Увеличиваем на 0, 1 или 2
    counter = currentValue + Math.floor(Math.random() * 3);
    console.log(`Увеличенное значение: ${counter}`);
  }, delay);
}
// Запускаем несколько асинхронных операций
incrementCounter();
incrementCounter();
incrementCounter();
```



Это ситуация, когда несколько асинхронных операций читают и изменяют одно и то же значение, и это приведет к непредсказуемым результатам.

```
Текущая значение: 0
Увеличенное значение: 1
Текущая значение: 1
Увеличенное значение: 3
Текущая значение: 3
Увеличенное значение: 4
```

```
Текущая значение: 0
Увеличенное значение: 2
Текущая значение: 2
Увеличенное значение: 2
Текущая значение: 2
Увеличенное значение: 2
```

# Неопределенный порядок выполнения.

```
let data = [];  
function fetchData(id) {  
  return new Promise((resolve) => {  
    const delay = Math.random() * 2000; // Случайная задержка до 2 секунд  
    setTimeout(() => {  
      console.log(`Данные для ID ${id} загружены`);  
      resolve(`Данные ${id}`);  
    }, delay);  
  });  
}  
async function run() {  
  // Запускаем асинхронные операции параллельно  
  fetchData(1).then(result => data.push(result));  
  fetchData(2).then(result => data.push(result));  
  fetchData(3).then(result => data.push(result));  
  // Ждем некоторое время, чтобы все данные загрузились  
  setTimeout(() => {  
    console.log("Финальные данные:", data);  
  }, 3000); // Ждем достаточно времени для завершения всех запросов  
}  
run();
```

Пример, в котором выполняется несколько асинхронных запросов, и порядок их завершения влияет на финальный результат.

Порядок загрузки данных может меняться из-за случайной задержки.

```
Данные для ID 3 загружены  
Данные для ID 2 загружены  
Данные для ID 1 загружены  
Финальные данные: [ 'Данные 3', 'Данные 2', 'Данные 1' ]
```

```
Данные для ID 2 загружены  
Данные для ID 1 загружены  
Данные для ID 3 загружены  
Финальные данные: [ 'Данные 2', 'Данные 1', 'Данные 3' ]
```

# Методы упорядочивания асинхронных операций.

## 1. Использование Promise.

Обещания (Promise): Объект, представляющий завершение или неудачу асинхронной операции и её результат.

Состояния Promise:

Ожидание (pending): Начальное состояние, ни выполнено, ни отклонено.

Исполнено (fulfilled): Операция завершена успешно.

Отклонено (rejected): Операция завершена с ошибкой.

## 2. `async/await`.

`async`: Обозначает, что функция является асинхронной и всегда возвращает `Promise`.

`await`: Ожидает завершения `Promise` и возвращает его результат.

Пример использования `async/await`

```
let counter = 0;
function incrementCounter() {
  return new Promise((resolve) => {
    const delay = Math.random() * 1000;
    setTimeout(() => {
      const currentValue = counter;
      console.log(`Текущая значение: ${currentValue}`);
      counter = currentValue + 1; // Увеличиваем на 1
      resolve(counter);
    }, delay);
  });
}
async function run() {
  // Запускаем асинхронные операции последовательно
  await incrementCounter();
  await incrementCounter();
  await incrementCounter();
  console.log(`Финальное значение счетчика: ${counter}`);
}
run();
```

В этом примере используются `async/await` для последовательного выполнения операций, что гарантирует, что каждая операция завершится перед началом следующей.

```
Текущая значение: 0
```

```
Текущая значение: 1
```

```
Текущая значение: 2
```

```
Финальное значение счетчика: 3
```

Подробнее о Promise и async/await в следующих лекциях.



# Контрольные вопросы:

- Чем отличаются `setTimeout` и `setInterval`?
- Что возвращает вызов `setTimeout/setInterval`?
- Как отменить выполнение таймера?
- Что произойдёт, если указать `setTimeout(..., 0)`?
- Почему `setInterval` может «дрейфовать» во времени?
- В чём преимущество рекурсивного `setTimeout` перед `setInterval`?
- Каково минимальное значение задержки в современных браузерах?
- Почему важно очищать таймеры в долгоживущих приложениях?

# Домашнее задание:

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

4 Возврат в асинхронном коде

Учимся писать асинхронные функции и работать с результатом их работы

5 Упорядочивание асинхронных операций

Учимся управлять потоком выполнения асинхронных операций

2. Повторить материал лекции.

# **Материалы лекций:**

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

# **Обратная связь:**

[colledge20education23@gmail.com](mailto:colledge20education23@gmail.com)