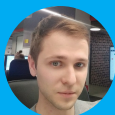




АСИНХРОННОСТЬ



АЛЕКСАНДР РУСАКОВ



АЛЕКСАНДР РУСАКОВ

co-founder PCNP Studio



a_s_rusakov@mail.ru



[@a_s_rusakov](https://t.me/as_rusakov)



ПЛАН ЗАНЯТИЯ

1. [Понятие асинхронности](#)
2. [Асинхронные функции](#)



ПОНЯТИЕ АСИНХРОННОСТИ

СИНХРОННОЕ ВЫПОЛНЕНИЕ

До настоящего момента весь код, что вы писали и что мы проходили на лекциях, был синхронный. Принцип синхронного выполнения прост: код выполняется сразу. Иногда для выполнения работы коду требуется значительное время, но суть та же: программа начинает выполняться незамедлительно.

```
1  const factorial = n => {  
2    for (var i = 1; n > 1; i *= n--);  
3    return i;  
4  }  
5  // 1 * 2 * 3 * ... * 50  
6  factorial(50);
```

АСИНХРОННОЕ ВЫПОЛНЕНИЕ

При разработке ПО бывают и специфические задачи:

1. Функция выполняется очень долго и хочется параллельно выполнять другой код. Например, пока ждем новых данных с сервера, можно работать со старыми.
2. Функция должна выполняться в определённое время. Например, будильник или рассылка Email-уведомлений.
3. Неизвестно точно, когда функция выполнится. Например, экран телефона необходимо включить только тогда, когда он нажмёт на кнопку «Power».

АСИНХРОННОЕ ВЫПОЛНЕНИЕ

Типичный случай из непрограммирования. В данном примере цвет кнопки изменится только при наведении на неё. Когда это может произойти и произойдёт ли вообще - зависит только от посетителя.

```
<button>Нажми на меня</button>
```

```
button:hover {  
  background: red;  
}
```

setTimeout

Для отложенного вызова функций в браузере есть функция `setTimeout(fn, ms)`, где *fn* - функция, которую нужно запустить через *ms* миллисекунд (1000 миллисекунд = 1 секунде).

Например, этот код через 1 секунду добавит в консоль приветствие:

```
const showGreeting = () => console.log('Поздравляем, вы стали обладателем $1000!');  
  
setTimeout(showGreeting, 1000);
```


НЕТОЧНОСТЬ ВЫЗОВА

Тем не менее, `showGreeting` из прошлого примера выполнится не обязательно через 1 секунду.

Для наглядности, 10 раз вызовем `setTimeout` с одним и тем же параметром:

```
1  const checkDelay = (index, delay) => {  
2    const start = new Date();  
3    setTimeout(() => {  
4      const end = new Date();  
5      const realDelay = end - start;  
6  
7      console.log(`${index}: Задержка между вызовом : ${realDelay} мс`);  
8    }, delay);  
9  }  
10  
11  for (let i = 0; i < 10; i++) {  
12    checkDelay(i, 1000);  
13  }
```

НЕТОЧНОСТЬ ВЫЗОВА

Каждый раз вывод в консоль может немного отличаться, но вот пример вызова:

```
l̲ 0: Задержка между вызовом : 1001 мс  
l̲ 1: Задержка между вызовом : 1001 мс  
l̲ 2: Задержка между вызовом : 1002 мс  
l̲ 3: Задержка между вызовом : 1002 мс  
l̲ 4: Задержка между вызовом : 1001 мс  
l̲ 5: Задержка между вызовом : 1001 мс  
l̲ 6: Задержка между вызовом : 1001 мс  
l̲ 7: Задержка между вызовом : 1002 мс  
l̲ 8: Задержка между вызовом : 1002 мс  
l̲ 9: Задержка между вызовом : 1002 мс
```

ОДИН ПОТОК

Данная особенность связана с тем, что код JavaScript выполняется в одном потоке. Проще говоря, в нашем случае, `showGreeting` выполнится, только если в настоящий момент не выполняется вообще ничего.

Если же параллельно с `showGreeting` выполняется какой-либо другой код, интерпретатор JavaScript сначала закончит выполнять его и только потом возьмётся за `showGreeting`.

ОДИН ПОТОК

Для демонстрации нам необходимо сделать два действия: уменьшить задержку для наглядности и загрузить интерпретатор долгими вычислениями:

```
for (let i = 0; i < 10; i++) {  
  checkDelay(i, 10);  
}
```

ОДИН ПОТОК

Вот «эталонный» результат, который тоже может немного отличаться от вызова к вызову:

```
ḷ 0: Задержка между вызовом : 11 мс
ḷ 1: Задержка между вызовом : 11 мс
ḷ 2: Задержка между вызовом : 11 мс
ḷ 3: Задержка между вызовом : 11 мс
ḷ 4: Задержка между вызовом : 11 мс
ḷ 5: Задержка между вызовом : 11 мс
ḷ 6: Задержка между вызовом : 11 мс
ḷ 7: Задержка между вызовом : 11 мс
ḷ 8: Задержка между вызовом : 12 мс
ḷ 9: Задержка между вызовом : 12 мс
```

ОДИН ПОТОК

Теперь «нагрузим» JavaScript бесполезной долгой работой:

```
1 // делает много бесполезной работы. Главное, чтобы долго!  
2 const idle = n => {  
3   let sum = 0;  
4   for(let i = 0; i < n; i++) {  
5     sum += i;  
6   }  
7 }  
8  
9 for (let i = 0; i < 10; i++) {  
10   checkDelay(i, 10);  
11   idle(1000000);  
12 }
```

ОДИН ПОТОК

И результат существенно изменится:

⌚ 0:	Задержка между вызовом : 131 мс
⌚ 1:	Задержка между вызовом : 103 мс
⌚ 2:	Задержка между вызовом : 95 мс
⌚ 3:	Задержка между вызовом : 83 мс
⌚ 4:	Задержка между вызовом : 71 мс
⌚ 5:	Задержка между вызовом : 59 мс
⌚ 6:	Задержка между вызовом : 48 мс
⌚ 7:	Задержка между вызовом : 37 мс
⌚ 8:	Задержка между вызовом : 26 мс
⌚ 9:	Задержка между вызовом : 15 мс

Это значит, что функция в `setTimeout` будет вызвана, только если у неё будет такая возможность. JavaScript не несёт никакой ответственности за то, что `setTimeout` сработает вовремя, но обещает это сделать как можно раньше в рамках заданной задержки.

ФУНКЦИИ ОБРАТНОГО ВЫЗОВА

Это всего лишь термин и ничего кроме. *Функцией обратного вызова* (callback) называется такая, которая:

1. Передаётся в другую функцию высшего порядка (HOF) как аргумент.
2. HOF должен вызвать эту функцию по достижению определённого условия.

Рассмотрим прошлый пример:

```
const showGreeting = () => console.log('Добрый день, я консольный бог!');  
  
setTimeout(showGreeting, 1000);
```

В нашем случае функцией высшего порядка является `setTimeout`, а функцией обратного вызова является `showGreeting`.

МИНИМАЛЬНАЯ ЗАДЕРЖКА

Вы можете вызывать `setTimeout` и без параметра временной задержки. В таком случае функция обратного вызова сработает как можно скорее (опять же, без каких-либо обязательств).

Пример 1:

```
const showGreeting = () => console.log('Добрый день, я консольный бог!');  
  
setTimeout(showGreeting);
```

МИНИМАЛЬНАЯ ЗАДЕРЖКА

Пример 2:

```
1  const checkDelay = ( index, delay ) => {
2      const start = new Date;
3      setTimeout(() => {
4          const end = new Date();
5          const realDelay = end - start;
6
7          console.log(`${index}: Задержка между вызовом : ${realDelay} мс`);
8      }, delay);
9  }
10
11  for (let i = 0; i < 10; i++) {
12      // второй параметр в данном случае будет равен undefined
13      checkDelay(i);
14  }
```

МИНИМАЛЬНАЯ ЗАДЕРЖКА

Пример вызова:

```
⌚ 0: Задержка между вызовом : 2 мс
⌚ 1: Задержка между вызовом : 2 мс
⌚ 2: Задержка между вызовом : 3 мс
⌚ 3: Задержка между вызовом : 3 мс
⌚ 4: Задержка между вызовом : 3 мс
⌚ 5: Задержка между вызовом : 3 мс
⌚ 6: Задержка между вызовом : 3 мс
⌚ 7: Задержка между вызовом : 3 мс
⌚ 8: Задержка между вызовом : 3 мс
⌚ 9: Задержка между вызовом : 3 мс
```

Вызов без второго аргумента равносител указанию значения 0. У каждого браузера всё равно есть минимальный порог, меньше которого задержка не будет.

```
const showGreeting = () => console.log('Добрый день, я консольный бог!');

// можно было и не указывать 2 параметр
setTimeout(showGreeting, 0);
```

clearTimeout

Запланированную задачу можно отменить с помощью функции `clearTimeout`. Для этого необходимо передать идентификатор таймаута, возвращаемый от `setTimeout`.

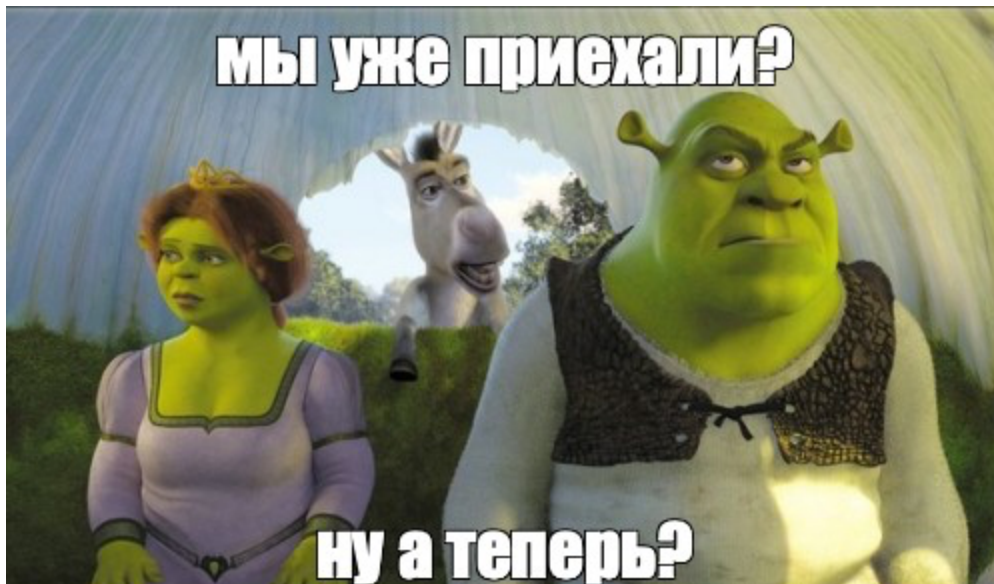
```
1 // сообщение в консоль не будет выведено
2 const id = setTimeout(() => console.log('Я хочу жить :('), 500);
3
4 clearTimeout(id);
```

```
1 // сообщение в консоль не будет выведено
2 const sendPayment = () => console.log( 'Вам начислена зарплата!' ),
3 const isCrisis = true;
4 const id = setTimeout(sendPayment, 1000);
5
6 // не выплачиваем сотрудникам деньги. У нас кризис!
7 if (isCrisis) {
8     clearTimeout(id);
9 }
```

setInterval (ДЕМО)

Аналогично с `setTimeout`, `setInterval` запускает функцию обратного вызова. Но делает это постоянно!

```
1  const areWeHome = () => console.log( 'Мы уже приехали?' );  
2  // будет запускать areWeHome до посинения  
3  setInterval( areWeHome, 1000 );
```



БОЛЕЗНЬ ЗАПУСКА

Как и у `setTimeout`, `setInterval` также «болеет» непредсказуемым временем запуска функций обратного вызова.

```
1 let start = new Date();
2 setInterval(() => {
3     const end = new Date();
4     const delay = end - start;
5     console.log(`Задержка: ${delay}`);
6     start = new Date;
7 }, 1000 );
```

БОЛЕЗНЬ ЗАПУСКА

Пример выполнения:

⌚	Задержка: 1020
⌚	Задержка: 1019
⌚	Задержка: 1020
⌚	Задержка: 1021
⌚	Задержка: 1020
⌚	Задержка: 1019
⌚	Задержка: 1021
⌚	3 Задержка: 1020
⌚	2 Задержка: 1019
⌚	2 Задержка: 1020
⌚	Задержка: 1019
⌚	Задержка: 1018

ПЕРЕДАЧА АРГУМЕНТОВ

Все аргументы, которые передаются в `setTimeout` и `setInterval` после второго, становятся аргументами callback-функции в момент вызова:

```
// выведет сообщение «Блиц, скорость без границ!» через 1с.  
setTimeout(console.log, 1000, 'Блиц, скорость без границ!');
```

```
const sum = (a, b) => a + b;  
  
// Выведет 29 через 1с.  
setTimeout(sum, 1000, 10, 19);
```




РАБОТА С HTTP

Наиболее распространенный случай работы с асинхронным кодом —
запросы по HTTP.

РАССМОТРИМ ЭТОТ КОД

```
1 // 1. Создаем запрос
2 var xhr = new XMLHttpRequest();
3
4 // 2. Определяем функцию обратного вызова
5 xhr.onreadystatechange = processResponse;
6
7 // Этот код выполнится когда запрос будет в пути
8 function processResponse(e) {
9     if (xhr.readyState === 4) {
10         // Запрос выполнен!
11         console.log(xhr.responseText);
12     } else {
13         // Запрос еще выполняется
14         console.log('Загружаем ...')
15     }
16 }
17
18 // 3. Определяем куда и как отправлять запрос
19 xhr.open('GET', 'employees.json', true);
20
21 // 4. Отправляем запрос
22 xhr.send();
23
24 console.log('Другая важная работа ...')
```



ЖИВОЙ ПРИМЕР

<https://repl.it/repls/CourteousPoisedBrace>

РЕЗУЛЬТАТА РАБОТЫ

```
1  Загружаем ...
2  Другая важная работа ...
3  Загружаем ...
4  Загружаем ...
5  [
6    {
7      "name": "Jim",
8      "inOffice": false
9    },
10   {
11     "name": "Joe",
12     "inOffice": true
13   },
14   {
15     "name": "John",
16     "inOffice": true
17   }
```



ОПТИМИЗАЦИЯ ВЫЧИСЛЕНИЙ

ПРОБЛЕМА ДОЛГИХ ВЫЧИСЛЕНИЙ

Второй вид функций: выполняющиеся очень долго. При больших значениях n страница будет существенно подвисать:

```
1  const sum = n => {  
2    let sum = n;  
3    for (let i = 0; i < n; i++){  
4      sum += i;  
5    }  
6    return sum;  
7  }  
8  
9  sum( 100000 );
```



РЕШЕНИЕ

Интервалы и таймеры - идеальное решение для долгих или рекурсивных вычислений. С помощью них мы можем проделывать полезную работу небольшими порциями, не нагружая страницу.

```

1 // вычисляем сумму от 0 до n, по itemsPerStep элементов за раз
2 const sumStep = (n, itemsPerStep, onload) => {
3   const size = Math.ceil(n / itemsPerStep); // количество шагов
4   let index = 0; // текущий шаг
5   let sum = 0; // сумма вычислений
6
7   // эта функция будет вызываться каждые 500 мс
8   return () => {
9     // окончание вычислений
10    if (index === size) {
11      onload(sum);
12      return;
13    }
14
15    // начальные и конечные значения шага
16    const start = index * itemsPerStep;
17    const end = Math.min((index + 1) * itemsPerStep, n + 1);
18
19    // сами вычисления
20    for (let i = start; i < end; i++){
21      sum += i;
22    }
23
24    console.log(`Шаг ${index}: ${sum}`);
25    index++;
26    // планируем новый шаг
27    scheduleStep();
28  };
29 };
30
31
32 // функция обратного вызова для вывода результата
33 const onload = result => console.log(`Результат вычислений: ${result}`);
34
35 // задаем начальные настройки
36 const step = sumStep( 1000000, 1000, onload );
37 const scheduleStep = () => setTimeout( step, 500);
38
39 scheduleStep();

```


ТАЙМАУТЫ В ИНТЕРФЕЙСАХ

Использование таймаутов и интервалов в JavaScript - довольно распространенное явление. На веб-сайтах вы наверняка встречались со следующими примерами:

1. Появление всплывающих окон в заданное время.
2. Таймеры обратного отсчёта на странице («До конца акции осталось...»).
3. Показ подсказок при поиске только по окончании ввода в текстовое поле.
4. Простая анимация.
5. Автоматическая смена слайдов в фотогалерее.
6. Проверка в простых чатах новых сообщений.

ЧЕМУ МЫ НАУЧИЛИСЬ?

- Разобрались с основами таймеров и интервалов.
- Узнали о «Болезни запуска» `setTimeout` и `setInterval`.
- Научились оптимизировать долгие и рекурсивные вычисления.
- Узнали разницу между синхронной и асинхронной работой.
- Разобрались как отправлять HTTP-запросы и обрабатывать результаты.



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты все **все задачи**.



Спасибо за внимание! Время задавать вопросы

АЛЕКСАНДР РУСАКОВ

 a_s_rusakov@mail.ru

 [@a_s_rusakov](https://t.me/as_rusakov)