

JS: Timers, Promises

НАШИ ПРАВИЛА



Включенная камера



Вопросы по поднятой руке



Не перебиваем друг друга



Все вопросы, не связанные с тематикой курса (орг-вопросы и т. д.), должны быть направлены куратору



Подготовьте свое рабочее окружение для возможной демонстрации экрана (закройте лишние соцсети и прочие приложения)

Повторим ;)



Какие методы массивов вы знаете?



Какие методы массивов изменяют исходный массив?

ЦЕЛЬ

Изучить встроенные в JS таймеры. Познакомиться с понятием асинхронности

ПЛАН ЗАНЯТИЯ

- **setTimeout**
- **setInterval**
- **Ассинхронность**
- **Callbacks**
- **Promises**

Reduce

Метод `reduce()` применяет функцию к аккумулятору и каждому значению массива (слева направо), сводя массив к одному значению.

```
const numbers = [1, 2, 3, 4, 5];  
const sum = numbers.reduce((acc, num) => acc + num, 0);  
// sum: 15
```

Reduce

```
[0, 1, 2, 3, 4].reduce(function (previousValue, currentValue, index, array) {  
    return previousValue + currentValue;  
});
```

	previousValue	currentValue	index	array	возвращаемое значение
первый вызов	0	1	1	[0, 1, 2, 3, 4]	1
второй вызов	1	2	2	[0, 1, 2, 3, 4]	3
третий вызов	3	3	3	[0, 1, 2, 3, 4]	6
четвёртый вызов	6	4	4	[0, 1, 2, 3, 4]	10

Значение, возвращённое методом `reduce()` будет равным последнему результату выполнения колбэк-функции — **10**.

setTimeout



Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Первый метод, который для этого существует - **setTimeout**

setTimeout позволяет вызвать функцию один раз через определённый интервал времени



Пример вызова функции sayHi() спустя одну секунду:

```
1 function sayHi() {  
2   alert('Привет');  
3 }  
4  
5 setTimeout(sayHi, 1000);
```

функция

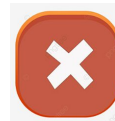
задержка в ms (delay)

1000ms=1sek

Важно!

Функцию в `setTimeout` нужно передавать, но не запускать её


`setTimeout(sayHi(), 1000);` - неправильно



`setTimeout(sayHi, 1000);` - правильно



Если функции, вызываемой через `setTimeout` нужно передать аргументы, это можно сделать после `delay`



```
1 function sayHi(phrase, who) {  
2   alert( phrase + ', ' + who );  
3 }  
4  
5 setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон
```

The diagram consists of four red arrows. Two arrows point from the top towards the parameters `phrase` and `who` in the `sayHi` function definition on line 1. The other two arrows point from the bottom towards the string arguments `"Привет"` and `"Джон"` in the `setTimeout` call on line 5. This illustrates how arguments are passed from the delayed function call back to the function's parameters.

setInterval



Второй функцией, которую используют для “планирования вызова” является **setInterval**

setInterval позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.



Метод `setInterval` имеет такой же синтаксис как `setTimeout`.

```
setInterval(() => alert('tick'), 2000)
```

code

интервал времени через
который будет повторяться
вызов функции



Отмена через clearTimeout

Вызов **setTimeout** и **setInterval** возвращает «идентификатор таймера» **timerId**, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

clearTimeout(timerId) — для остановки таймера, установленного с помощью **setTimeout**.

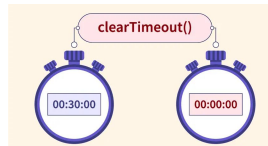
clearInterval(intervalId) — для остановки интервала, установленного с помощью **setInterval**.



Пример отмены через clearTimeout

В коде ниже планируем вызов функции и затем отменяем его (просто передумали). В результате ничего не происходит:

```
1 let timerId = setTimeout(() => alert("ничего не происходит"), 1000);
2 alert(timerId); // идентификатор таймера
3
4 clearTimeout(timerId);
5 alert(timerId); // тот же идентификатор (не принимает значение null после отмены)
```



Рассмотрим пример, который выводит сообщение каждые 2 секунды.
Через 5 секунд вывод прекращается:

```
1 // повторить с интервалом 2 секунды
2 let timerId = setInterval(() => alert('tick'), 2000);
3
4 // остановить вывод через 5 секунд
5 setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Ассинхронность

Синхронный код и его проблемы

Синхронный код понятный, его удобно читать, потому что он выполняется ровно так, как написан:

```
console.log('A')  
console.log('B')  
console.log('C')
```

Выведется:

A
B
C

Синхронный код и его проблемы

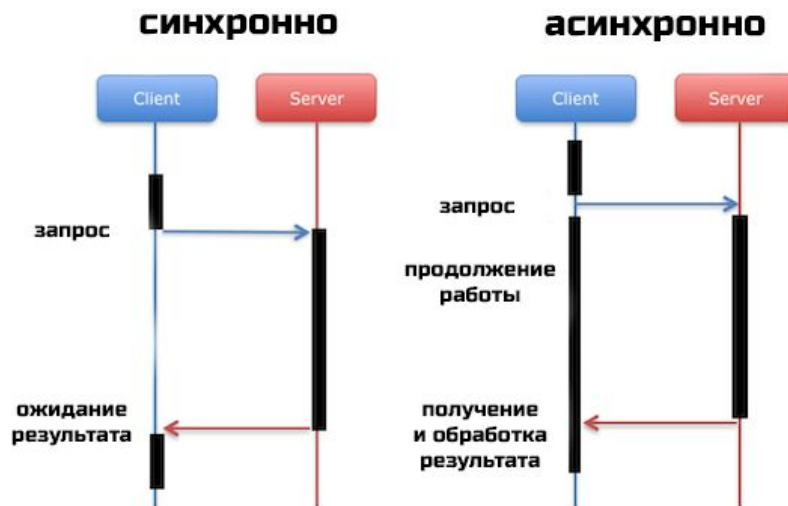
Представим, что нам нужно выполнить какую-то операцию, требующую некоторого времени — например, напечатать в консоли приветствие, но не сразу, а через 5 секунд.

```
const = greet()=> {  
  console.log('Hello!')  
}
```

`delay(5000)` – выдуманная функция
`greet()`

Асинхронность - возможности выполнять операции независимо друг от друга, не ожидая завершения предыдущих операций.

Асинхронный код выполняется в несколько этапов, позволяя другим частям программы работать в промежутках между этими этапами.

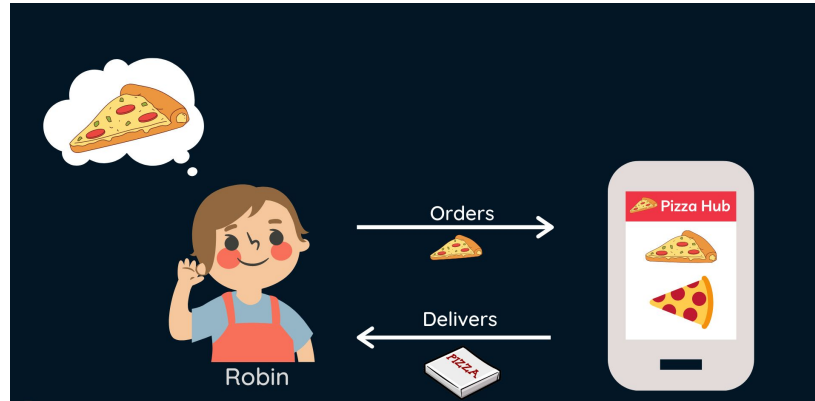


Попробуем решить предыдущую задачу, но так, чтобы наш код не блокировал выполнение. Для этого мы воспользуемся функцией **setTimeout()**:

```
setTimeout (greet () => {  
    console.log ('Hello!')  
}, 5000)
```

```
console.log ("I'm being called before greet function.")
```

Callbacks



Callback (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события.

Событием может быть что угодно:

- ответ от сервера;
- завершение какой-то длительной вычислительной задачи;
- получение доступа к каким-то API устройства, на котором выполняется код.

Колбэк — это первый способ обработать какое-либо асинхронное действие.

Изначально колбэки были единственным способом работать с асинхронным кодом в JavaScript:

«выполни эту функцию, когда случится это событие».

```
function fetchData(url, cb) {  
    // 1. Выполняет запрос к API по URL  
    // 2. Если ответ успешный, выполнить  
    обратный вызов  
  
    cb(res);  
}  
  
function callback(res) {  
    // Сделать что-то с результатом  
}  
  
// Сделать что-то  
  
fetchData('https://sitepoint.com',  
callback);  
  
// Сделать что-то ещё
```

Однако у колбэков есть неприятный минус, так называемый **ад колбэков** (callback hell).

```
1 function request(url, onSuccess) {  
2   /*...*/  
3 }  
4  
5 request('/api/users/1', function (user) {  
6   request(`/api/photos/${user.id}/`, function (photo) {  
7     request(`/api/crop/${photo.id}/`, function (response) {  
8       console.log(response)  
9     })  
10  })  
11 })
```

Promise



Определение promise



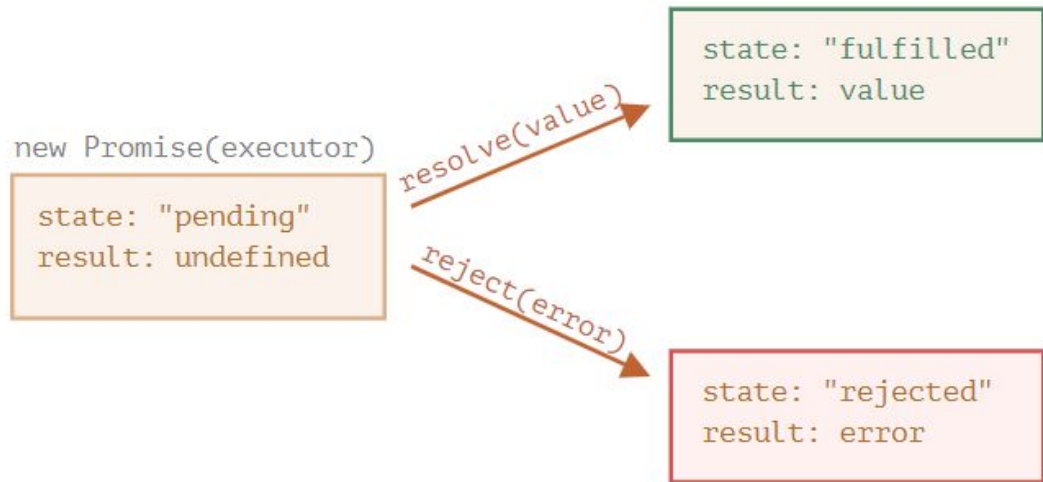
Промис (promise) - это объект, представляющий результат успешного или неудачного завершения асинхронной операции. Асинхронная операция, упрощенно говоря, это некоторое действие, которое выполняется независимо от окружающего ее кода, в котором она вызывается, не блокируя выполнение вызываемого кода.

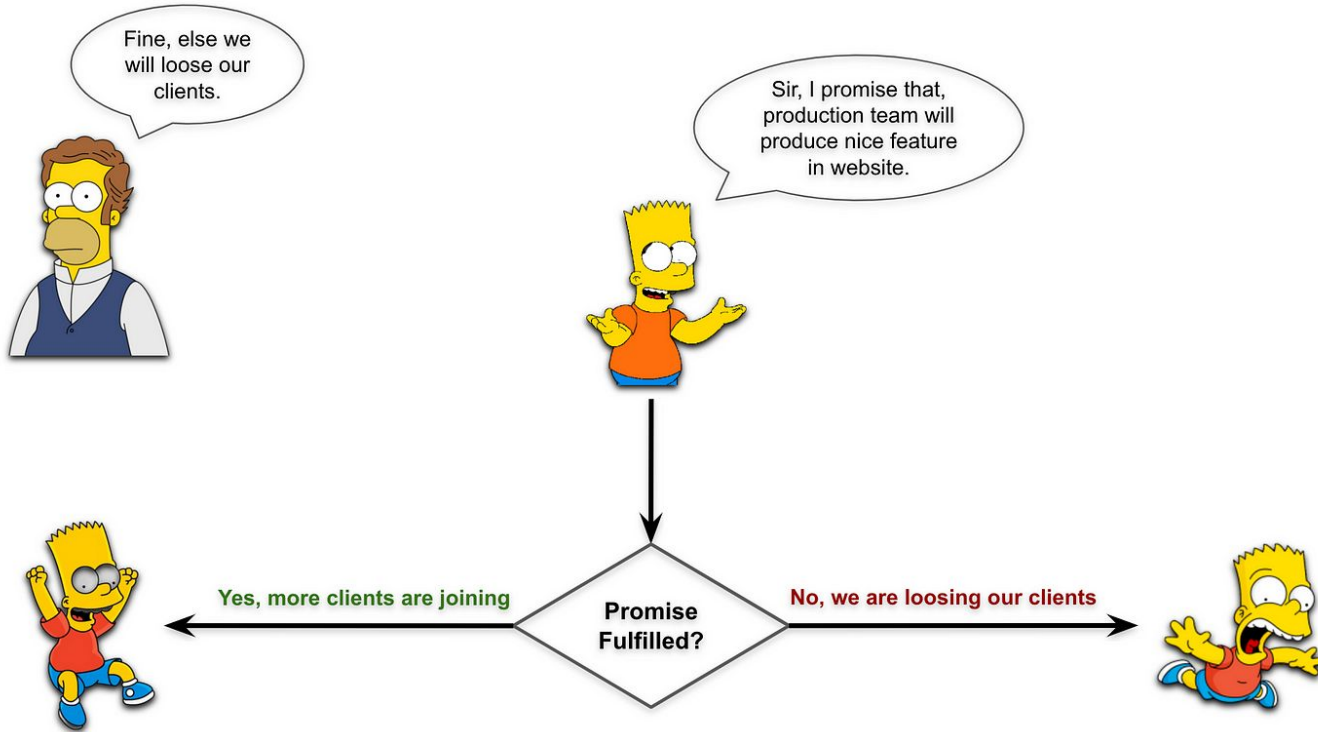
Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети.

Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции.

Промис может находиться в одном из трёх состояний:

- **pending** — стартовое состояние, операция стартовала;
- **fulfilled** — получен результат;
- **rejected** — получена ошибка;





Ниже пример конструктора Promise и простого исполнителя с кодом, дающим успешный результат с задержкой (через setTimeout):

```
1 let promise = new Promise(function(resolve, reject) {  
2   // эта функция выполнится автоматически, при вызове new Promise  
3  
4   // через 1 секунду сигнализировать, что задача выполнена с результатом "done"  
5   setTimeout(() => resolve("done"), 1000);  
6 });
```


Ниже пример конструктора Promise и простого исполнителя с кодом, дающим результат с ошибкой с задержкой (через setTimeout):

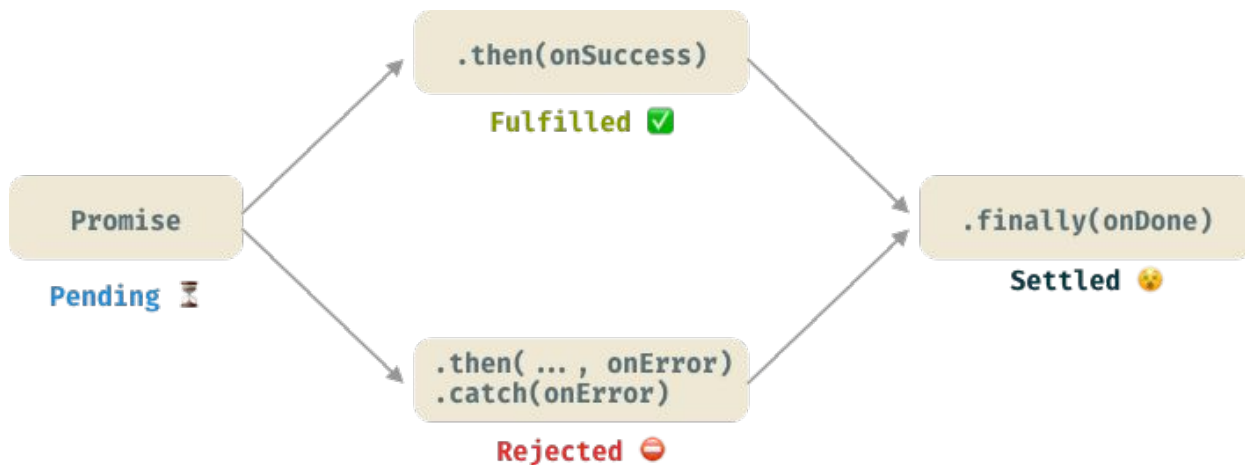
```
1 let promise = new Promise(function(resolve, reject) {
2   // спустя одну секунду будет сообщено, что задача выполнена с ошибкой
3   setTimeout(() => reject(new Error("Whoops!")), 1000);
4 });
```

Состояние промиса может быть изменено только один раз. Все последующие вызовы `resolve` и `reject` будут проигнорированы:

```
let promise = new Promise(function(resolve, reject) {  
  resolve("done");  
  
  reject(new Error("...")); // игнорируется  
  setTimeout(() => resolve("...")); // игнорируется  
});
```

Существует три метода, которые позволяют работать с результатом выполнения вычисления внутри промиса:

- `then()`
- `catch()`
- `finally()`



Метод then()

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

```
1 promise.then(  
2   function(result) { /* обработает успешное выполнение */ },  
3   function(error) { /* обработает ошибку */ }  
4 );
```

Метод catch()


Если мы хотели бы только обработать ошибку, то можно использовать `null` в качестве первого аргумента: `.then(null, errorHandlerFunction)`. Или можно воспользоваться методом `.catch(errorHandlerFunction)`, который сделает то же самое:

```
1 let promise = new Promise((resolve, reject) => {
2   setTimeout(() => reject(new Error("Ошибка!")), 1000);
3 });
4
5 // .catch(f) это то же самое, что promise.then(null, f)
6 promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

Метод finally()

Вызов `.finally(f)` похож на `.then(f, f)`, в том смысле, что `f` выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

```
new Promise((resolve, reject) => {  
  /* сделать что-то, что займёт время, и после вызвать resolve или может reject */  
})  
  // выполнится, когда промис завершится, независимо от того, успешно или нет  
  .finally(() => остановить индикатор загрузки)  
  // таким образом, индикатор загрузки всегда останавливается, прежде чем мы продолжим  
  .then(result => показать результат, err => показать ошибку)
```



Ваша новая IT-профессия – Ваш новый уровень жизни

Программирование с нуля в
немецкой школе AIT TR GmbH