

Тема занятия:

- `setTimeout`
- `setInterval`
- Promises
- EventLoop



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

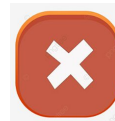
задержка в ms (delay)

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
```

A diagram consisting of four red arrows. Two arrows originate from the parameter names 'phrase' and 'who' in the function definition on line 1 and point to the first and second arguments, 'Привет' and 'Джон', in the setTimeout call on line 5. The other two arrows originate from the same two arguments in the setTimeout call and point back to the parameter names 'phrase' and 'who' in the function definition, illustrating the flow of data from the arguments to the function 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:

```
1 let timerId = setTimeout(...);  
2 clearTimeout(timerId);
```



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)
```

Примечание: обычно идентификатором таймера является число

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);
```

Promise



Определение promise



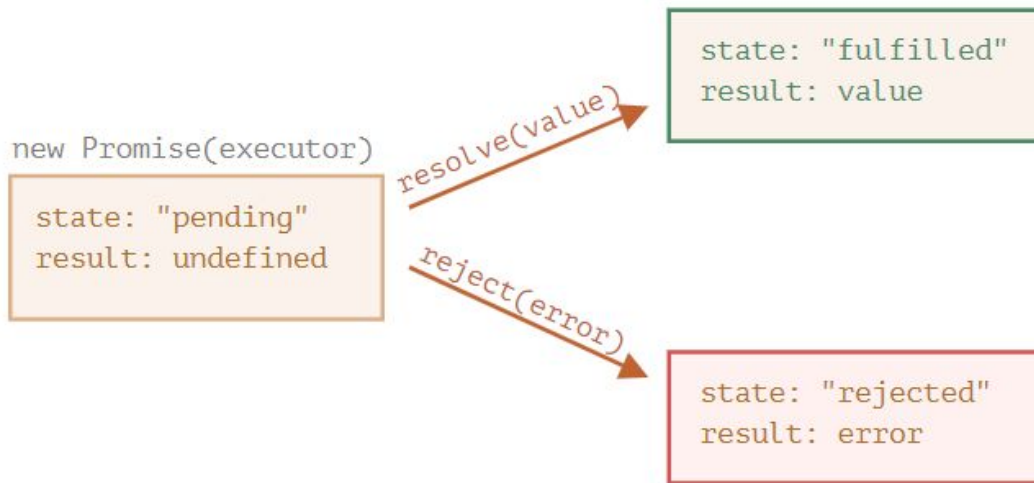
Промис (promise) - это объект, представляющий результат успешного или неудачного завершения асинхронной операции. Асинхронная операция, упрощенно говоря, это некоторое действие, которое выполняется независимо от окружающего ее кода, в котором она вызывается, не блокируя выполнение вызываемого кода.

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.





Yes, more clients are joining

Promise
Fulfilled?

No, we are loosing our clients



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 });
```

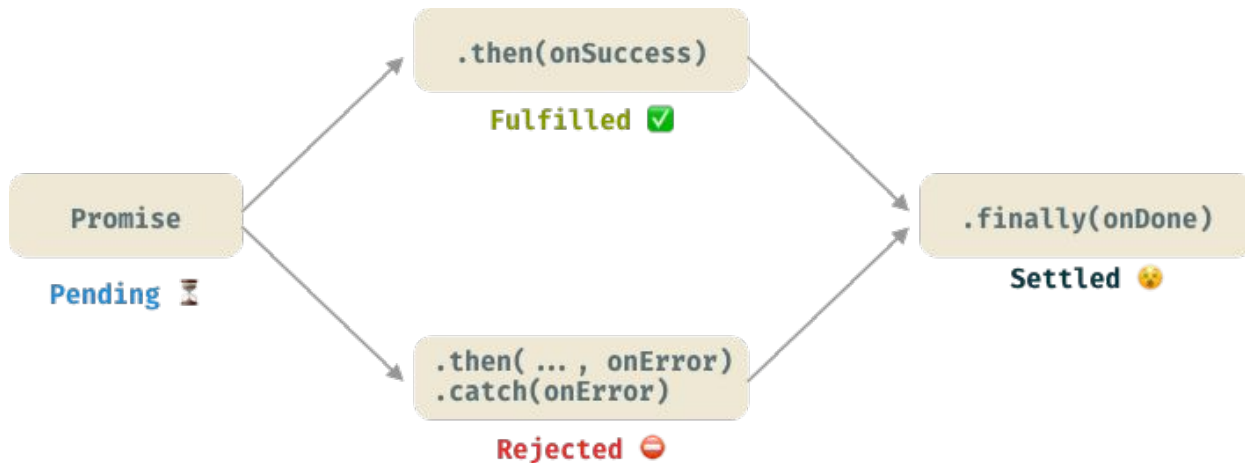
Состояние промиса может быть изменено только один раз.

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) { /* обрабатывает успешное выполнение */ },  
3   function(error) { /* обрабатывает 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 делает 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.

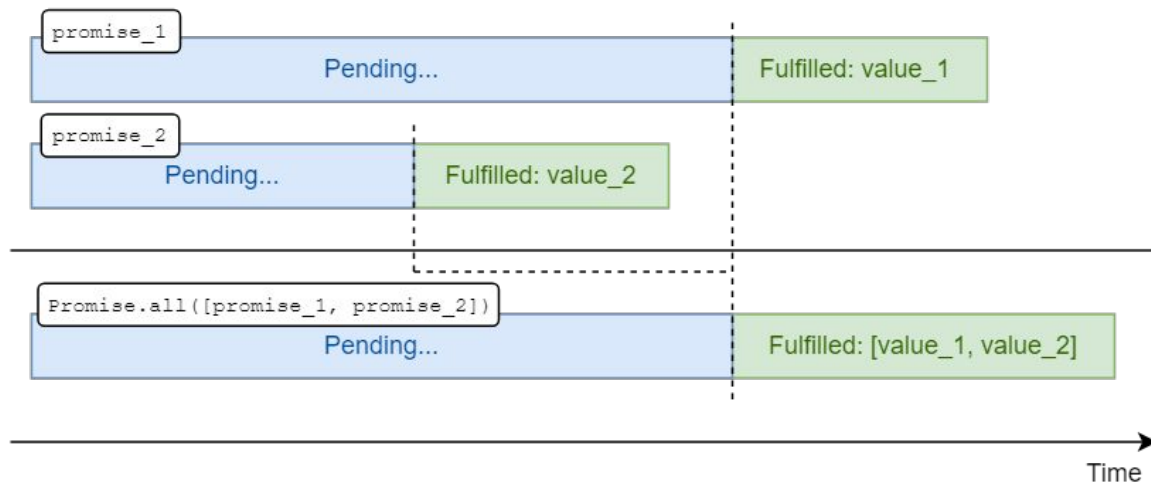
```
new Promise((resolve, reject) => {  
  /* сделать что-то, что займёт время, и после вызвать resolve или может reject */  
})  
// выполнится, когда промис завершится, независимо от того, успешно или нет  
.finally(() => остановить индикатор загрузки)  
// таким образом, индикатор загрузки всегда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

Promise.all

Допустим, нам нужно запустить множество промисов параллельно и дождаться, пока все они выполнятся.

Например, параллельно загрузить несколько файлов и обработать результат, когда он готов.

Для этого как раз и пригодится метод **Promise.all**.



Promise.all

Метод `Promise.all` принимает массив промисов и возвращает новый промис.

Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов.

```
1 Promise.all([
2   new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
3   new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
4   new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
5 ]).then(alert); // когда все промисы выполнятся, результат будет 1,2,3
6 // каждый промис даёт элемент массива
```

Promise.all

Если любой из промисов завершится с ошибкой, то промис, возвращённый Promise.all, немедленно завершается с этой ошибкой.

```
1 Promise.all([
2   new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
3   new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ошибка!")), 200)),
4   new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
5 ]).catch(alert); // Error: Ошибка!
```



В случае ошибки, остальные результаты игнорируются

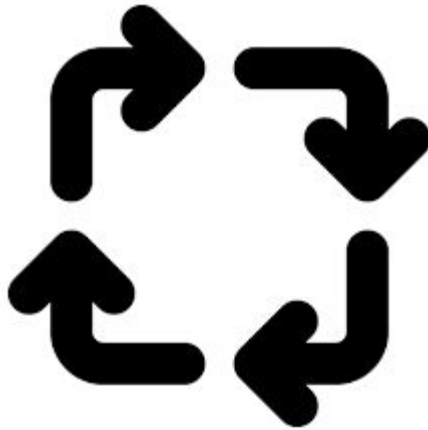
Promise.race



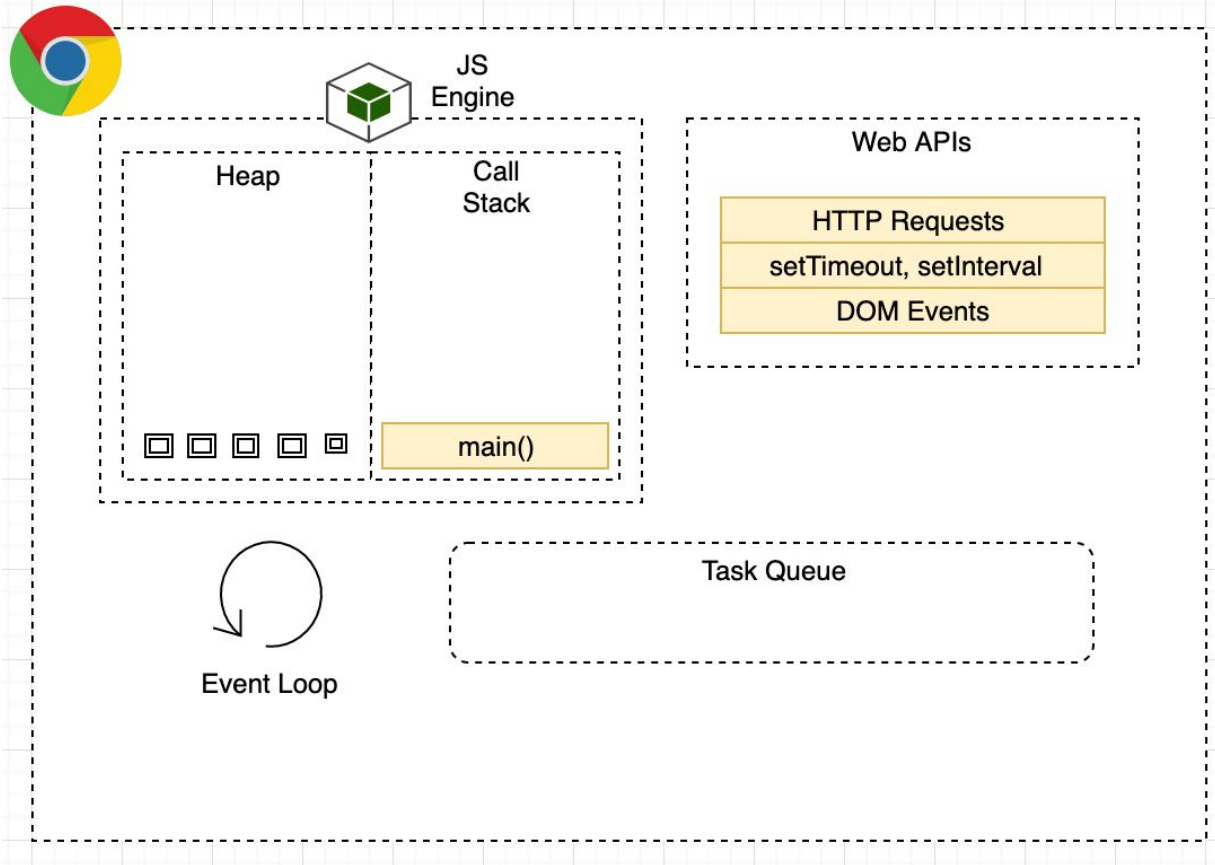
Метод очень похож на Promise.all, но ждёт только первый выполненный промис, из которого берёт результат (или ошибку).

```
1 Promise.race([
2   new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
3   new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ошибка!")), 2000)),
4   new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
5 ]).then(alert); // 1
```

Event Loop



Event Loop before Promise



Event Loop after Promise

