**Типы данных. Преобразование типов. Сравнение**

**Типы**

**В JavaScript существует 8 типов данных:**

**number** - целые и дробные числа, включает значения Infinity и NaN

**Bigint** - целые числа состоящие более чем из 16 цифр

**string** - строки

**boolean** - логические значения (true и false)

**null** - специальное значение, имеющее смысл неизвестного значения, "ничего"

**undefined** - специальное значение, имеющее смысл "значение еще не определено"

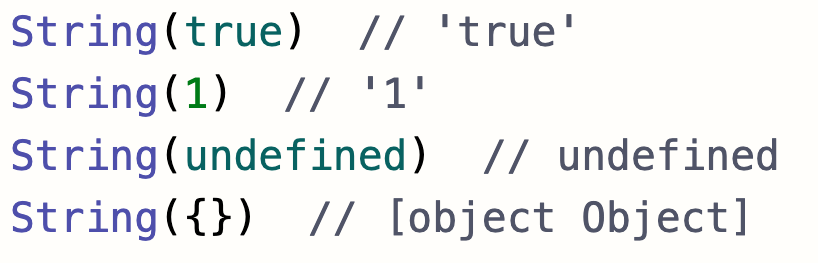
**Symbol** - уникальные значения

**object** -  сущности "ключ-значение"

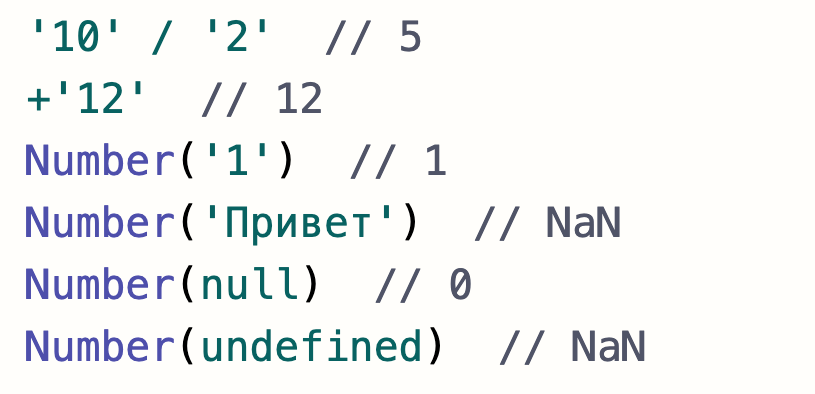
Типы с первого по седьмой являются *примитивами*, поскольку всегда содержат только одно значение. Тип "object" используется для составных сущностей.

**Преобразование типов**

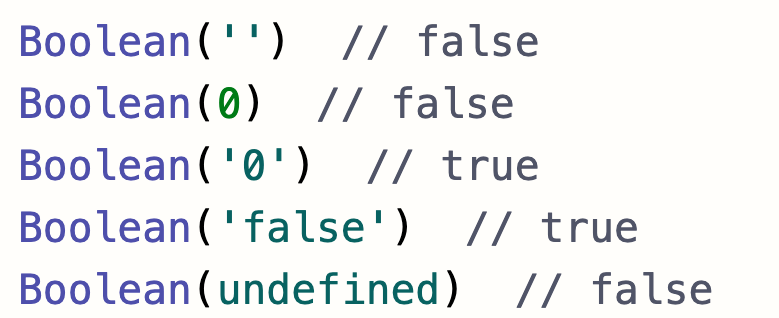
**Строковое**: String()



**Численное**: Number() или математические операции



**Логическое**: Boolean() или логические операции (например, двойное отрицание - **!!**)



👆 *Обратите внимание, что для приведения к логическому типу не нужно использовать ключевое слово* ***new****, т.к. в этом случае создается объект, который конвертируется в true в логическом выражении, даже если он был проинициализирован falsy-значением*

const isTruthy = new Boolean(false);if (isTruthy) {

// код в этом блоке выполнится}

**Сравнение**

Виды сравнения в JavaScript:

**Нестрогое сравнение (==, !=)** приводит к одному из типов при сравнении, игнорируя исходные типы операндов.

**Строгое сравнение (===, !==)** осуществляет сравнение без приведения типов, тем самым учитывая при сравнении, что операнды имеют разные типы.

**[Object.is](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)** - фактически, самый строгий и самый верный способ определить, являются ли значения одним и тем же (даже оператор === допускает особые случаи как -0 === +0 -> true, NaN === NaN -> false)

**Полезные ссылки:**

[Отличия null и undefined](https://frontender.info/exploring-the-abyss-of-null-and-undefined-in-javascript/" \l ":~:text=%D0%A7%D1%82%D0%BE%D0%B1%D1%8B %D0%B8%D1%85 %D1%80%D0%B0%D0%B7%D0%BB%D0%B8%D1%87%D0%B0%D1%82%D1%8C, %D0%B2%D1%8B %D0%BC%D0%BE%D0%B6%D0%B5%D1%82%D0%B5,%D0%B0 null %E2%80%94 %D1%83%D0%BC%D1%8B%D1%88%D0%BB%D0%B5%D0%BD%D0%BD%D1%8B%D0%BC %D0%BE%D1%82%D1%81%D1%83%D1%82%D1%81%D1%82%D0%B2%D0%B8%D0%B5%D0%BC %D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F." \o "https://frontender.info/exploring-the-abyss-of-null-and-undefined-in-javascript/#:~:text=%D0%A7%D1%82%D0%BE%D0%B1%D1%8B%20%D0%B8%D1%85%20%D1%80%D0%B0%D0%B7%D0%BB%D0%B8%D1%87%D0%B0%D1%82%D1%8C%2C%20%D0%B2%D1%8B%20%D0%BC%D0%BE%D0%B6%D0%B5%D1%82%D0%B5,%D0%B0%20null%20%E2%80%94%20%D1%83%D0%BC%D1%8B%D1%88%D0%BB%D0%B5%D0%BD%D0%BD%D1%8B%D0%BC%20%D0%BE%D1%82%D1%81%D1%83%D1%82%D1%81%D1%82%D0%B2%D0%B8%D0%B5%D0%BC%20%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F." \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Почему typeof null это 'object'](https://2ality.com/2013/10/typeof-null.html" \o "https://2ality.com/2013/10/typeof-null.html" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Тип данных Bigint](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/BigInt" \o "https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/BigInt" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Тип данных Symbol](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Symbol" \o "https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Symbol" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Больше информации о сравнении](https://learn.javascript.ru/comparison" \o "https://learn.javascript.ru/comparison" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

**Переменные**

Существует три способа создания переменных: **var**, **let** и **const**.

Создание переменных через **var** является устаревшим вариантом, поскольку переменные var:

Имеют функциональную или глобальную область видимости, видимость не может быть ограничена блоком { }

Могут быть пере-объявлены

Объявляются в начале своей области видимости и доступны до своего определения ([hoisting](https://developer.mozilla.org/ru/docs/Glossary/Hoisting" \o "https://developer.mozilla.org/ru/docs/Glossary/Hoisting" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank))

Хранятся как свойства [глобального объекта Window](https://learn.javascript.ru/global-object" \o "https://learn.javascript.ru/global-object" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

В свою очередь, переменные **let** и **const**:

Имеют блочную область видимости

Не могут быть пере-объявлены

Недоступны до определения ([temporal dead zone](https://jsinthebits.com/what-is-the-temporal-dead-zone-e41369d4/" \o "https://jsinthebits.com/what-is-the-temporal-dead-zone-e41369d4/" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank))

Не хранятся как свойства глобального объекта

Отличие *const* от *let* лишь в том, что оно задает константное значение. Как для примитивных типов данных, так и для объектов, оно не может быть перезаписано, хотя свойства и методы константных объектов могут быть изменены.

**Полезные ссылки:**

[Глава про переменные на learn.javascript](https://learn.javascript.ru/variables" \o "https://learn.javascript.ru/variables" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

**Объекты**

**Объект** - это набор свойств, которые представляют из себя пару - ключ и значение. По ключу можно обратиться к значению свойства объекта.

Ключ свойства должен иметь тип string, symbol или другой тип, который можно преобразовать к string. Например, если задать свойство с ключом true типа boolean, то ключ будет преобразован в 'true' типа string. Значение свойства может иметь любой тип.

Чтение свойств

Обратиться к свойству объекта можно:

через точку (obj.property);

через квадратные скобки и имя свейства (obj['property']);

через квадратные скобки и переменную, содержащую имя свойства (obj[prop], где prop = 'property').

Ещё один способ обратиться к свойству, который часто удобен на практике, — с помощью **[деструктурирующего присваивания](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)**. Оно позволяет "распаковать" объект в несколько переменных по названиям свойств объекта. Давайте рассмотрим пример:

const size = {

width: 100,

height: 200,};

const { width, height } = size;

console.log(width); // 100console.log(height); // 200

В примере создались переменные width и height со значениями size.width и size.height.

Если вы хотите присвоить свойство переменной с другим названием, то это можно сделать с помощью двоеточия:

const size = {

width: 100,

height: 200,};

const { width: w, height: h } = size;

console.log(w); // 100console.log(h); // 200

Теперь свойства size.width и size.height присваиваются переменным w и h.

**Значения по умолчанию**

Также при деструктуризации можно устанавливать значения по умолчанию. Они будут использоваться, если в объекте не определено соответствующее свойство.

const size = {

height: 300,};

const { width = 100, height = 200 } = size;

console.log(width); // 100console.log(height); // 300

В примере выше в объекте не было определено свойство width, поэтому применилось значение по умолчанию.

**Параметры функции**

Деструктуризацию можно использовать в аргументах функции:

const size = {

width: 100,

height: 200,};

const calcArea = ({ width, height }) => {

return width \* height;};

calcArea(size); // 20 000

На практике такое очень удобно, когда в функцию нужно передать много параметров, часть из которых опциональны. Вместо того чтобы запоминать порядок следования аргументов, можно передать все параметры в функцию как объект, а внутри функции произвести деструктуризацию этих параметров.

Другие операции с объектами

**Удалить свойство** можно с помощью оператора **delete** (delete obj.property).

**Проверить существование свойства** можно с помощью:

Обращения к этому свойству и проверки, является ли значение undefined (например, obj.property === undefined)

Оператора **in** ('property' in obj)

Метода **hasOwnProperty** (obj.hasOwnProperty)

Статического метода **[Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)** (рекомендуется использовать вместо hasOwnProperty)

**Перебор свойств** объекта можно осуществить с помощью цикла: for (property in obj). В этом цикле также появятся унаследованные перечислимые свойства из цепочки прототипов объекта!

Также для итерации по объекту можно использовать:

**Object.keys()** - вернет массив ключей свойств объекта (перечислимые свойства)

**Object.values()** - вернет массив значений свойств объекта

**Object.entries()** - вернет массив массивов вида [ключ, значение]

**Object.getOwnPropertyNames()** - вернет массив всех собственных свойств объекта (*включая неперичислимые*)

**Object.getOwnPropertySymbols()** - аналогичен предыдущему, только возвращает символьные ключи объекта

**Полезные ссылки:**

[Общие сведения об объектах](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Working_with_Objects" \o "https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Working_with_Objects" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Глава о прототипах на learn js](https://learn.javascript.ru/prototypes" \o "https://learn.javascript.ru/prototypes" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Скрытые свойства Symbol](https://learn.javascript.ru/symbol" \l "skrytye-svoystva" \o "https://learn.javascript.ru/symbol#skrytye-svoystva" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

**Массивы**

**Создать массив** можно несколькими способами:

**const arr = []** (используя литерал массива - квадратные скобки)

**const arr = Array(3)**, где 3 - длина созданного массива. При этом массив инициализируется с пустыми значениями. Чтобы заполнить созданный массив, можно использовать метод arr.fill()

**const arr = Array.from(arrayLikeEntity)**, создание массива из итерируемой сущности. Примерами итерируемых сущностей являются встроенные объекты Map, Set, строки, объект со свойством length.

Для создания массива через объект Array можно не использовать ключевое слово new

**Массив — это объект.** Чтобы проверить, является ли объект массивом, можно использовать метод **Array.isArray(arr)**.

Узнать длину массива можно с помощью свойства **arr.length**.

**Добавление/удаление элементов** в массив можно делать с помощью методов:

arr.**push**(item) - добавляет item в конец массива

arr.**pop**() - удаляет последний элемент массива и возвращает его значение

arr.**unshift**(item) - добавляет item в начало массива

arr.**shift**() - удаляет первый элемент массива и возвращает его значение

arr.**splice**(startIndex, deleteCount, element1, element2, ...) - вставка элементов в любое место массива

Чтобы осуществить **обход массива**, можно использовать:

Цикл **for**: for (let i = 0; i < arr.length; i++)

Цикл **for-of**, перебирающий элементы: for (const item of arr)

Метод **forEach**: arr.forEach((item, index, arr) => { })

Метод **map** (возвращает копию массива, и при обходе элементов их можно как-то трансформировать)

Метод **reduce** (позволяет пройтись по элементам массива и создать одну итоговую сущность)

Для **чтения элементов массива**, как и в объектах, можно использовать **[деструктуризацию](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)**:

const users = ['John', 'Pete'];

const [firstUser, secondUser] = users; // в firstUser запишется строка 'John', а в secondUser — 'Pete'

А если массив содержит больше элементов, чем мы хотим прочитать, оставшиеся элементы можно получить через rest-оператор:

const users = ['John', 'Pete', 'Ann'];

const [firstUser, ...rest] = users;

console.log(`${firstUser} and ${rest.length} other users liked`); // John and 2 other users liked

В переменную rest записался массив, состоящий из двух элементов: "Pete" и "Ann".

**Основные методы массивов:**

arr.**map**() - создание нового массива в соответствие с возвращаемыми значениями переданной функции

arr.**reduce**() - вычисление значения в результате обхода всех элементов массива

arr.**every**() - проверка, удовлетворяют ли все элементы массива заданному в функции условию

arr.**some**() - проверка, удовлетворяет ли хотя бы один элемент массива заданному в функции условию

arr.**splice**() - удаление/добавление элементов

arr.**slice**() - создание копии целого массива или его части

arr.**join**() - склейка элементов массива в строку

arr.**reverse**() - разворачивает исходный массив, не делает его копию

arr.**sort**() - сортировка массива в соответствии с переданной функцией-компаратором. Если функция-компаратор не передана, массив будет отсортирован лексикографически (в соответствии с порядком символов в кодировке Unicode)

**Поиск в массиве**

arr.**indexOf**() / arr.**lastIndexOf**() - возвращает первый/последний индекс в массиве

arr.**includes**() - возвращает true/false

arr.**find**() / arr.**findIndex**() - возвращает значение/индекс первого найденного элемента

arr.**filter**() - возвращает новый отфильтрованный массив

**Полезные ссылки:**

[Array - MDN reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

**Функции. Контекст. Замыкания, каррирование**

**Функция - это выполняемый объект,** который позволяет вынести некоторую логику. Функции могут принимать аргументы и возвращать значения.

**Параметры в функцию передаются *по значению***, если это значения примитивных типов, и ***по ссылке*,** если это объекты. К аргументам функции можно обращаться с помощью псевдомассива **arguments** (только для функций, объявленных через ключевое слово function), либо, если подразумевается неограниченное количество аргументов, можно обратиться с помощью **[spread/rest оператора](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax" \o "https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)**.

Если у функции явно не прописано, какое значение она возвращает, то такая функция возвращает undefined.

**Объявления функций** могут быть:

***Function Declaration*** - function foo() { }. Такие функции создаются до выполнения кода и могут быть вызваны до объявления.

***Function Expression***- const foo = function() { }. Такие функции до объявления вызвать нельзя.

**Контекст**

Функции имеют контекст выполнения **this**. Не в строгом режиме (без 'use strict') для глобальных функций это Window/Global. Для методов объектов this - это сам объект.

Для того чтобы избежать **потери контекста**, можно использовать:

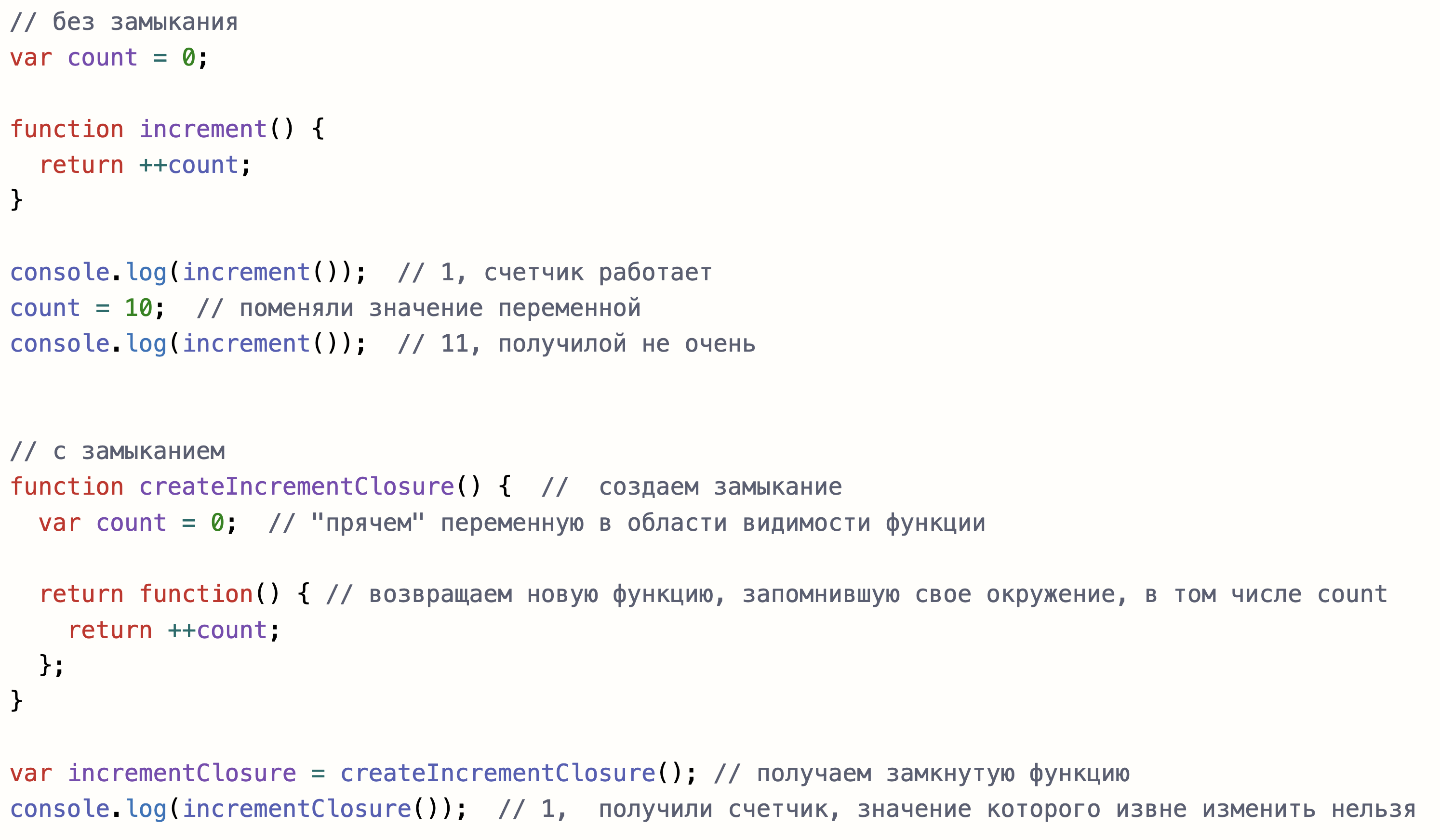
**bind()** - создание новой функции с фиксированным контекстом

**call() / apply()** - вызов функции с явным указанием контекста

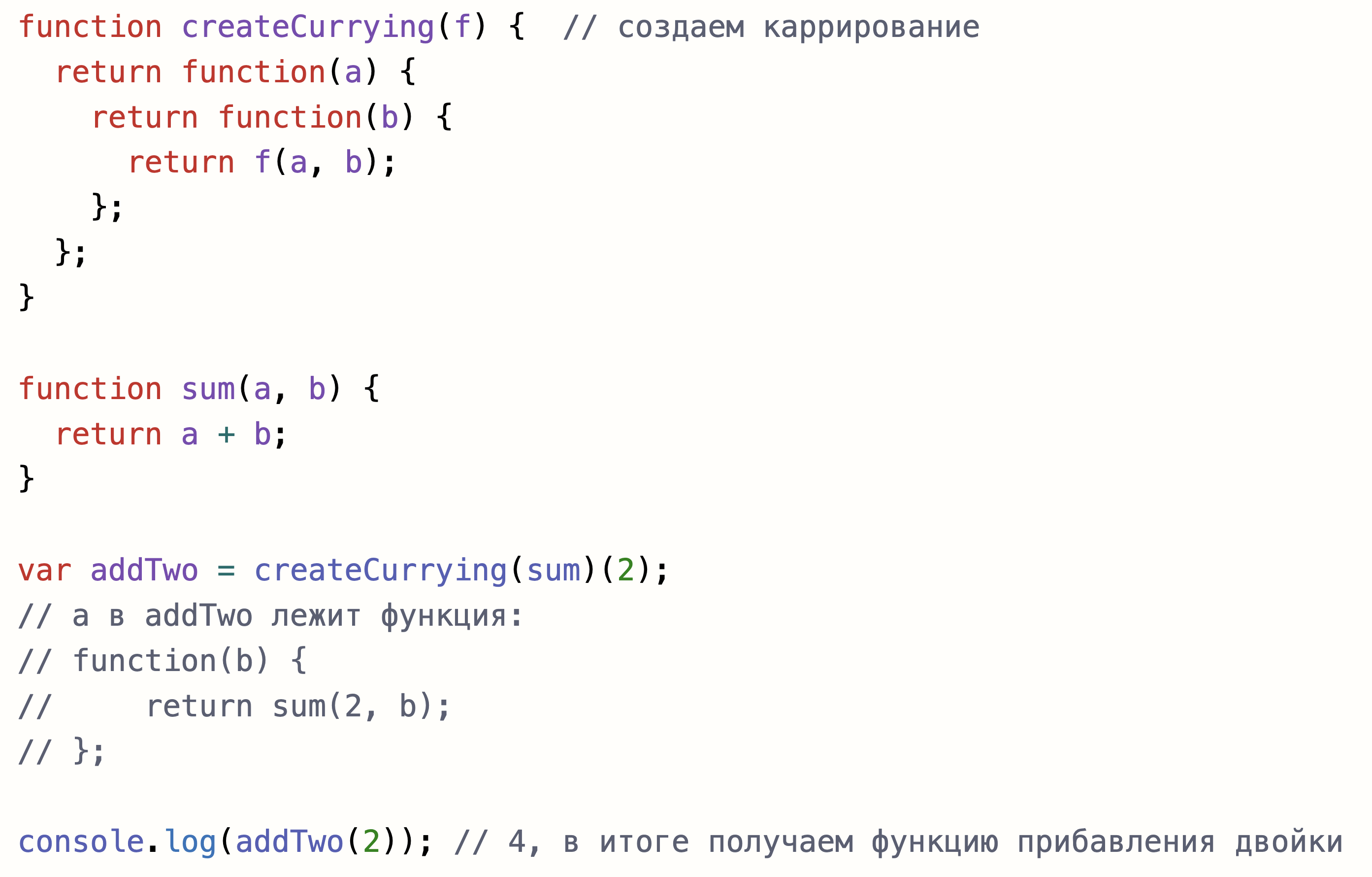
**стрелочные функции** - функции, не имеющие своего this и сохраняющие контекст выполнения своего окружения. Кроме того, такие функции не содержат псевдомассив аргументов arguments.

**Замыкания, каррирование**

**Замыкание** - создание функции, запоминающей свое окружение. Пример:



**Каррирование** - создание функции, трансформирующей некоторую исходную функцию и возвращающей результат этой трансформации в виде новой функции. Пример:



**Полезные ссылки:**

[Про стандарт ES6](https://habr.com/ru/post/305900/" \o "https://habr.com/ru/post/305900/" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Function Expression и Function Declaration](https://learn.javascript.ru/function-expressions" \o "https://learn.javascript.ru/function-expressions" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Отдельно про деструктуризацию](https://learn.javascript.ru/destructuring-assignment" \o "https://learn.javascript.ru/destructuring-assignment" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

**Отложенные действия. Промисы**

**setTimeout, setInterval**

Для выполнения отложенных действий в JavaScript существуют следующие способы:

**setTimeout** - вызывает функцию через определенное количество миллисекунд и возвращает числовой идентификатор таймера. Функция вызывается только один раз. Таймер можно сбросить, если передать его идентификатор в функцию **clearTimeout**.

**setInterval** - вызывает функцию бесконечно с интервалом в определенное количество миллисекунд. Так же возвращает идентификатор установленного таймера, который можно передать в функцию **clearInterval**, чтобы остановить регулярное выполнение функции.

**"Callback hell" и что такое Promise**

**Promise** это объект, который ждёт выполнения асинхронной операции (**pending**), после чего (т.е. после выполнения) промис принимает одно из двух состояний: **fulfilled** (успешное выполнение) или **rejected** (выполнено с ошибкой). При создании промис принимает функцию, в аргументах которой передаются методы resolve и reject, отвечающие за то, как должен быть обработан промис в результате переданной функции. Если в resolve/reject передать некоторое значение, то промис будет завершен успешно/неуспешно с этим значением.

На отлов определенного результата завершения промиса можно "подписаться" с помощью методов:

**then** - принимает два колбэка, первый из которых вызывается, если промис завершен успешно, а второй - если с ошибкой. Если передан один колбэк, он будет вызван как для успешного промиса, так и для завершившегося ошибкой. Возвращает Promise, чтобы цепочку можно было продолжать.

**catch** - принимает один колбэк, который будет вызван только в случае, если промис будет завершен с ошибкой. Возвращает Promise, чтобы цепочку можно было продолжать.

**finally** - принимает один колбэк, который выполнится в любом случае, независимо от результата завершения промиса. finally также возвращает Promise, так что формально цепочку .then() можно продолжить и дальше.

**Методы работы с несколькими промисами**

**Promise.all()** - возвращает промис, который исполнится после исполнения всех переданных промисов. В случае, если любой из промисов будет отклонен, финальный промис будет также отклонен (и Promise.all завершится в тот же момент).

**Promise.any()** - возвращает промис, который завершится успешно, если хотя бы один из переданных промисов завершится успешно, и возвращает результат первого успешно завершившегося промиса. Завершается с ошибкой, если все промисы завершатся с ошибкой.

**Promise.race()** - возвращает промис, который будет исполнен или отклонен с результатом исполнения первого исполненного или отклонённого промиса

**Promise.allSettled()** - возвращает промис, который исполняется когда все полученные промисы завершены (исполнены или отклонены), содержащий массив результатов исполнения полученных промисов

**Promise.resolve()**, **Promise.reject()** - методы явного вызова промиса, завершающегося с успехом или ошибкой

**Немного о промисах в цикле событий**

**async/await**

**async** - гарантирует, что функция асинхронная и возвращает промис.

**await** - заставляет JavaScript ждать выполнения промиса. Код продолжит выполняться только когда будет получен результат промиса. Не может быть использован вне асинхронной функции (в функции без async).

**Полезные ссылки:**

[Глава про промисы и async/await](https://learn.javascript.ru/async" \o "https://learn.javascript.ru/async" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank)

[Немного про цикл событий](https://learn.javascript.ru/event-loop" \o "https://learn.javascript.ru/event-loop" \t "https://lms.metaclass.kts.studio/lesson/31/unit/_blank) (event loop)