Вопросы к коллоквиуму для студентов II-го курса

1. Переменные. Имена переменных. Константы.

2. Типы данных. Оператор typeof.

3. Преобразование типов.

4. Операторы. Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов.

5. Операторы сравнения. Сравнение с null и undefined.

6. Условные операторы: if, '?'. Конструкция switch.

7. Логические операторы.

8. Циклы while, for.

9. Функции (Function Declaration). Параметры по умолчанию.

10. Функциональные выражения (Function Expression) и функции-стрелки.

11. Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений.

12. Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк.

13. Массивы.

14. Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.

15. Методы перебора и преобразование массива.

16. Объекты. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта.

17. Копирование, клонирование, сравнение, объединение объектов. Объекты-константы.

18. Тип данных Symbol.

19. Коллекции Set, WeakSet.

20. Коллекции Map, WeakMap.

21. Деструктурирующее присваивание. Вложенная деструктуризация.

22. Дата и Время. Объект Date. Получение и установка компонентов даты. Автоисправление даты. Разность дат.

23. Глобальны объект. Создание функции с помощью конструктора (new Function).

24. Лексическое окружение (LexicalEnvironment). Замыкание.

**25.** **Объект функции. Именованное функциональное выражение (Named Function Expression).**

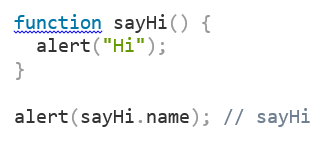
**1.Объект функции**

Как известно, в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. В JavaScript, функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

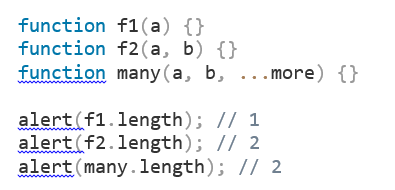
Объект функции содержит несколько полезных свойств:

Например, имя функции доступно как **свойство name**:



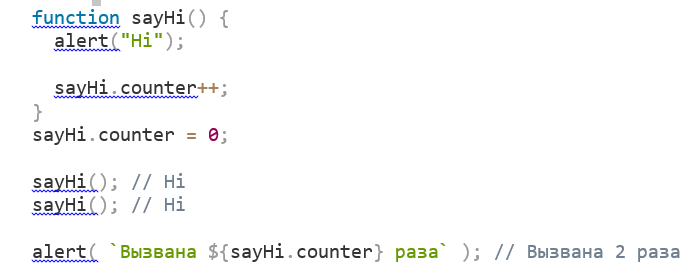
[**Свойство length**](https://learn.javascript.ru/function-object#svoystvo-length)

Ещё одно встроенное свойство length содержит количество параметров функции в её объявлении.



Свойство length иногда используется для [интроспекций](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D1%80%D0%BE%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) в функциях, которые работают с другими функциями.

Также можно добавить **свои собственные свойства**. Например, свойство counter для отслеживания общего количества вызовов:



**Свойство не есть переменная**

Свойство функции, назначенное как sayHi.counter = 0, не объявляет локальную переменную counter внутри неё. Другими словами, свойство counter и переменная let counter – это две независимые вещи. Можно использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот.

Иногда свойства функции могут использоваться вместо замыканий. Например, можно переписать функцию-счетчик из вопроса про [замыкание](https://learn.javascript.ru/closure), используя её свойство:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

Свойство count теперь хранится прямо в функции, а не в её внешнем лексическом окружении. Основное отличие такого подхода от замыкания в том, что если значение count живет во внешней переменной, то она не доступна для внешнего кода. Изменить её могут только вложенные функции. А если оно присвоено как свойство функции, то можно его получить:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

counter.count = 10;

alert( counter() ); // 10

Поэтому выбор реализации зависит от целей разработчика.

2.[**Named Function Expression**](https://learn.javascript.ru/function-object#named-function-expression)

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя:

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

Заметьте, что функция всё ещё задана как Function Expression. Добавление "func" после function не превращает объявление в Function Declaration, потому что оно все еще является частью выражения присваивания. Добавление такого имени ничего не ломает. Функция все еще доступна как sayHi():

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

sayHi("John"); // Hello, John

Есть две важные особенности имени func, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция sayHi вызывает себя с "Guest", если не передан параметр who:

let sayHi = function func(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

func("Guest");

}

};

sayHi(); // Hello, Guest

func(); // Ошибка

Не следует использовать имя sayHi для вложенного вызова, так как значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнет выдавать ошибки:

let sayHi = function(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

sayHi("Guest"); // Ошибка

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка

Так происходит, потому что функция берет sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null. Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы. Все работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все еще содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Задать «внутреннее» имя можно только для Function Expression, и не нельзя для Function Declaration. Если нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

26 **Остаточные параметры и оператор расширения**

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др

1.[**Остаточные параметры (...)**](https://learn.javascript.ru/rest-parameters-spread-operator#ostatochnye-parametry)

Вызывать функцию можно с любым количеством аргументов, независимо от того, как она была определена. Например:

function sum(a, b) {

return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

Лишние аргументы не вызовут ошибку, но приняты будут только первые два.

Остаточные параметры могут быть обозначены через три точки «...». Суть его в том, что оставшиеся параметры помещаются в массив. Например, соберём все аргументы в массив args:

function sumAll(...args) {

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

Можно положить первые несколько параметров в переменные и собрать в массив остальные. В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Юлий Цезарь

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них.

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами.

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью arguments. Этот способ всё ещё работает, его можно найти в старом коде. Но у него есть один недостаток. Хотя arguments похож на массив, и он тоже перебираемый, это всё же не массив. Он не поддерживает методы массивов, поэтому нельзя, например, вызвать arguments.map(...). К тому же, arguments всегда содержит все аргументы функции – нельзя получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

У стрелочных функций нет своего объекта arguments. Если обратиться к arguments из стрелочной функции, то получим аргументы внешней обычной функции. Пример:

function f() {

let showArg = () => alert(arguments[0]);

showArg(2);

}

f(1); // 1

**2.**[**Оператор расширения**](https://learn.javascript.ru/rest-parameters-spread-operator#spread-operator)

Иногда нужно массив преобразовать в список параметров. Например, есть встроенная функция [Math.max](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max). Она возвращает наибольшее число из списка:

alert( Math.max(3, 5, 1) ); // 5

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Оператор расширения можно использовать и для слияния массивов:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.

27. **Каррирование**

[*Каррирование*](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) или карринг (currying) в функциональном программирование – это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции – до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Говоря о [каррировании](https://medium.com/@kbrainwave/currying-in-javascript-ce6da2d324fe), можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

[*Арность*](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) – это количество аргументов функции.

В результате каррирования, функция с несколькими аргументами преобразуется в набор функций, каждая из которых принимает один аргумент.

Рассмотрим пример:

function multiply(a, b, c) {

   return a \* b \* c;

}

Функция multiply принимает три аргумента и возвращает их произведение:  
  
multiply(1,2,3); *// 6*

Преобразовать её к набору функций, каждая из которых принимает один аргумент. Создадим каррированный вариант этой функции и посмотрим на то, как получить тот же результат в ходе вызова нескольких функций:

function multiply(a) {

   return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

}

log(multiply(1)(2)(3)) *// 6*

В примере вызов единственной функции с тремя аргументами multiply(1,2,3) преобразован к вызову трёх функций – multiply(1)(2)(3). При использовании новой конструкции каждая функция, кроме последней, возвращающей результат вычислений, принимает аргумент и возвращает другую функцию, также способную принять аргумент и возвратить другую функцию. Распишем конструкцию вида multiply(1)(2)(3), чтобы она была более понятной:

const mul1 = multiply(1)*;*

const mul2 = mul1(2)*;*

const result = mul2(3)*;*

log(result)*; // 6*

Подробнее разберём, что здесь происходит. Сначала передается аргумент 1 функции multiply:

const mul1 = multiply(1);

При работе этой функции срабатывает такая конструкция:

return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

Теперь в mul1 имеется ссылка на функцию, принимающую аргумент b. Вызовем функцию mul1, передав ей 2:

const mul2 = mul1(2);

В результате этого вызова выполнится следующий код:

return (c) => {

           return a \* b \* c

       }

Константа mul2 будет содержать ссылку на функцию, которая могла бы оказаться в ней, например, в результате выполнения следующей операции:

mul2 = (c) => {

           return a \* b \* c

       }

Если теперь вызвать функцию mul2, передав ей 3, то функция выполнит необходимые вычисления, воспользовавшись аргументами a и b:

const result = mul2(3);

Результатом этих вычислений будет 6:

log(result)*; // 6*

Функция mul2, обладающая самым большим уровнем вложенности, имеет доступ к областям видимости, к замыканиям, формируемым функциями multiply и mul1. Именно поэтому в функции mul2можно производить вычисления с переменными, объявленными в функциях, выполнение которых уже завершено, которые уже возвратили некие значения и обработаны сборщиком мусора.

**Каррирование и частичное применение функций**

Сейчас, возникает ощущение, что количество вложенных функций, при представлении функции в виде набора вложенных функций, зависит от количества аргументов функции. И, если речь идёт о каррировании, то это так. Особый вариант функции для вычисления объёма, можно сделать таким:

function volume(l) {

   return (w, h) => {

       return l \* w \* h

   }

}

Здесь применены идеи, очень похожие на рассмотренные выше. Пользоваться этой функцией можно так:

const hV = volume(70)*;*

hV(203,142)*;*

hV(220,122)*;*

hV(120,123)*;*

А можно и так:

volume(70)(90,30)*;*

volume(70)(390,320)*;*

volume(70)(940,340)*;*

Фактически, здесь командой volume(70), создана специализированная функция для вычисления объёма тел, одно из измерений которых (а именно – длина, l), зафиксировано. Функция volume ожидает 3 аргумента и содержит 2 вложенных функции, в отличие от предыдущей версии подобной функции, каррированный вариант которой содержал 3 вложенных функции.

Та функция, которая получилась после вызова volume(70) реализует концепцию частичного применения функции (partial function application). Каррирование и частичное применение функций очень похожи друг на друга, но концепции это разные.

При частичном применении функцию преобразуют в другую функцию, обладающую меньшим числом аргументов (меньшей арностью). Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию).

Каррирование и частичное применение функций может оказаться полезным в различных ситуациях. Например – при разработке небольших модулей, подходящих для повторного использования.

Частичное применение функций позволяет облегчить использование универсальных модулей. Например, есть интернет-магазин, в коде которого имеется функция, которая используется для вычисления суммы к оплате с учётом скидки.

function discount(price, discount) {

   return price \* discount

}

Есть определённая категория клиентов, которые получают скидку в 10%. Например, если такой клиент покупает что-то на $500, то он получает скидку размером $50:

const price = discount(500,0.10); // $50

// $500 - $50 = $450

Несложно заметить, что при таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:

const price = discount(1500,0.10); // $150

// $1,500 - $150 = $1,350

const price = discount(2000,0.10); // $200

// $2,000 - $200 = $1,800

const price = discount(50,0.10); // $5

// $50 - $5 = $45

const price = discount(5000,0.10); // $500

// $5,000 - $500 = $4,500

const price = discount(300,0.10); // $30

// $300 - $30 = $270

Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки. Функция discount() в примере имеет два аргумента. Преобразуем ее следующим образом:

function discount(discount) {

   return (price) => {

       return price \* discount;

   }

}

const tenPercentDiscount = discount(0.1);

Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть – аргумент discount, уже будет задана:

tenPercentDiscount(500); // $50

// $500 - $50 = $450

Если в магазине имеются покупатели, которым решено дать скидку размером в 20%, то получить соответствующую функцию для работы с ними можно так:

const twentyPercentDiscount = discount(0.2);

Теперь функцию twentyPercentDiscount() можно вызывать для расчёта стоимости товаров с учётом скидки в 20%:

twentyPercentDiscount(500); // 100

// $500 - $100 = $400

twentyPercentDiscount(5000); // 1000

// $5,000 - $1,000 = $4,000

twentyPercentDiscount(1000000); // 200000

// $1,000,000 - $200,000 = $600,000

28. Генераторы. Функции-генераторы. Перебор объектов-генераторов.

1. **Генераторы**

Обычные функции возвращают только одно-единственное значение (или ничего). Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отличн о работают с перебираемыми объектами и позволяют легко создавать потоки данных.

**2.**[**Функция-генератор**](https://learn.javascript.ru/generators#funktsiya-generator)

Для объявления генератора используется специальная синтаксическая конструкция: function\* f(…) (или function \*f(…)), которая называется «функция-генератор». Выглядит она так:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор» для управления её выполнением:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

alert(generator); // [object Generator]

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно полагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

* value: значение из yield.
* done: true, если выполнение функции завершено, иначе false.

Создадим генератор и получим первое из возвращаемых им значений:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

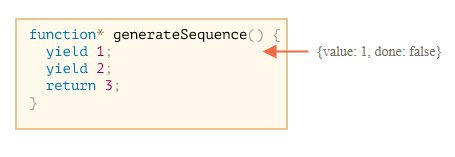
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}

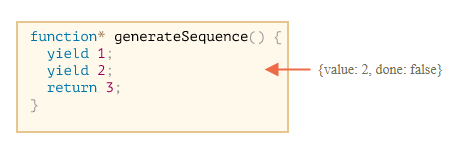
На данный момент получено только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов generator.next() возобновит выполнение кода и вернёт результат следующего yield:

let two = generator.next();

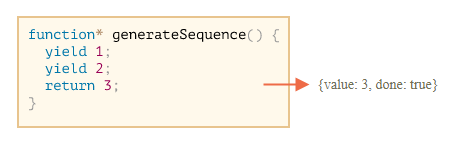
alert(JSON.stringify(two)); // {value: 2, done: false}



И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}



Сейчас генератор полностью выполнен. Можно увидеть это по свойству done:true и обработать value:3 как окончательный результат. Новые вызовы generator.next() больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: {done: true}.

**3.**[**Перебор генераторов**](https://learn.javascript.ru/generators#perebor-generatorov)

Генераторы являются [перебираемыми](https://learn.javascript.ru/iterable) объектами. Возвращаемые ими значения можно перебирать через for..of:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2

}

Обратите внимание: пример выше выводит значение 1, затем 2. Значение 3 выведено не будет. Это потому что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если необходимо, чтобы были все значения при переборе через for..of, то надо возвращать их через yield:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2, затем 3

}

Так как генераторы являются перебираемыми объектами, можно использовать всю связанную с ними функциональность, например, оператор расширения ...:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

В коде выше ...generateSequence() превращает перебираемый объект-генератор в массив элементов.

В примерах выше генерируются конечные последовательности, но также можно сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел. Тут потребуется break (или return) в цикле for..of по такому генератору, иначе цикл будет продолжаться бесконечно и скрипт «зависнет».

29. **Методы объектов, this. Оператор опциональной последовательности.**

1. **Методы объектов, this.**

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

let user = {

name: "Джон",

age: 30

};

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п. Такие действия в JavaScript представлены свойствами-функциями объекта. Для начала добавим в объект user функцию приветствия:

let user = {

name: "Джон",

age: 30

};

user.sayHi = function() {

alert("Привет!");

};

user.sayHi(); // Привет!

Здесь просто использовано Function Expression, чтобы создать функцию для приветствия, и присвоить её свойству user.sayHi объекта user. Затем она вызывается.

Функцию, которая является свойством объекта, называют *методом* этого объекта. Таким образом создан метод sayHi объекта user. Конечно, можно было бы заранее объявить функцию и использовать её в качестве метода, например так:

let user = {

// ...

};

function sayHi() {

alert("Привет!");

};

user.sayHi = sayHi;

user.sayHi(); // Привет!

Существует более короткий синтаксис для методов в литерале объекта:

user = {

sayHi() {

alert("Привет");

}

};

Т.е., можно пропустить ключевое слово "function" и просто написать sayHi(). Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов, но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

В JavaScript иногда необходимо сначала проверить, существует ли объект, а затем попытаться получить одно из его свойств, например, так:

const car = null;

const color = car && car.color;

Даже если car имеет значение null, у нас нет ошибок, а color присваивается значение null. Используя оператор &&, можно пройти несколько уровней вложенностей:

const car = {}

const colorName = car && car.color && car.color.name;

**2.Оператор *опциональной последовательности*** позволяет сделать код короче:

const car = {}

const color = car?.color;

const colorName = car?.color?.name;

Если car имеет значение null или undefined, результат будет undefined.

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода). Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user. Для доступа к информации внутри объекта метод может использовать ключевое слово this. Значение this – это объект «перед точкой», который использовался для вызова метода. Например:

let user = {

name: "Джон",

age: 30,

sayHi() {

// this - это "текущий объект"

alert(this.name);

}

};

user.sayHi(); // Джон

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user). Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

let user = {

name: "Джон",

age: 30,

sayHi() {

alert(user.name);

}

};

Но такой код будет ненадёжным. Если скопировать ссылку на объект user в другую переменную, например, admin = user, и перезаписать переменную user чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из admin. Это показано ниже:

let user = {

name: "Джон",

age: 30,

sayHi() {

alert( user.name );

}

};

let admin = user;

user = null;

admin.sayHi(); // Ошибка!

Если использовать this.name вместо user.name внутри alert, тогда этот код будет работать.

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции. В этом коде нет синтаксической ошибки:

function sayHi() {

alert( this.name );

}

Значение this вычисляется во время выполнения кода и зависит от контекста. Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

let user = { name: "Джон" };

let admin = { name: "Админ" };

function sayHi() {

alert( this.name );

}

user.f = sayHi;

admin.f = sayHi;

user.f(); // Джон (this == user)

admin.f(); // Админ (this == admin)

admin['f'](); // Админ

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

Можно вызвать функцию вовсе без использования объекта:

function sayHi() {

alert(this);

}

sayHi(); // undefined

В строгом режиме ("use strict") в таком коде значением this будет являться undefined. Если попытаться получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет глобальный объект. Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

В других языках программирования this фиксировано – методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод). В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Такая особенность вычисления this в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Некоторые хитрые способы вызова метода приводят к потере значения this, например:

let user = {

name: "Джон",

hi() { alert(this.name); },

bye() { alert("Пока"); }

};

user.hi();

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

В последней строчке кода используется условный оператор ?, который определяет, какой будет вызван метод (user.hi или user.bye) в зависимости от выполнения условия. В данном случае будет выбран user.hi. Затем метод тут же вызывается с помощью скобок (). Но вызов не работает как положено: при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим). Так работает (доступ к методу объекта через точку):

user.hi();

Так уже не работает (вызываемый метод вычисляется):

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

Чтобы понять, почему так происходит, разберёмся, как работает вызов методов (obj.method()). В выражении obj.method() сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method). Затем скобки () вызывают этот метод (исполняется код метода). Если поместить эти операции в отдельные строки, то значение this, естественно, будет потеряно:

let user = {

name: "Джон",

hi() { alert(this.name); }

}

let hi = user.hi;

hi(); // Ошибка

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого [Reference Type](https://tc39.github.io/ecma262/#sec-reference-specification-type). Этот ссылочный тип является внутренним типом. Нельзя явно использовать его, но он используется внутри языка. Значение ссылочного типа – это «триплет»: комбинация из трех значений (base, name, strict), где:

* base – это объект.
* name – это имя свойства объекта.
* strict – это режим исполнения. Является true, если действует строгий режим (use strict).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

// значение ссылочного типа (Reference Type)

(user, "hi", true)

Когда скобки () применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки .до вызывающих скобок (). При любой другой операции, например, присваивании hi = user.hi, ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без this. Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое).

Стрелочные функции особенные: у них нет своего «собственного» this. Если использовать this внутри стрелочной функции, то его значение берётся из внешней обычной функции. Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

let user = {

firstName: "Вася",

sayHi() {

let arrow = () => alert(this.firstName);

arrow();

}

};

user.sayHi(); // Вася

Это является особенностью стрелочных функций. Они полезны, когда нет необходимости иметь отдельное значение this, а надо брать его из внешнего контекста.

30 **Итерируемые объекты. Псевдомассивы.**

*Итерируемые* объекты – это концепция, которая позволяет использовать любой объект в цикле for..of.

Конечно же, сами массивы являются перебираемыми объектами. Но есть и много других встроенных перебираемыми объектов, например, строки.

Легко понять принцип устройства перебираемых объектов, создав один из них. Например, есть объект range, который представляет собой диапазон чисел. Это не массив, но он выглядит подходящим для for..of:

let range = {

from: 1,

to: 5

};

Чтобы сделать range итерируемым (и позволить с ним работать for..of), нужно добавить в объект метод с именем Symbol.iterator (специальный встроенный Symbol, созданный как раз для этого).

1. Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор – объект с методом next.
2. Дальше, for..of работает только с этим возвращённым объектом.
3. Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.
4. Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что итерация закончена, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором, запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

* У самого range нет метода next().
* Вместо этого другой объект, так называемый «итератор», создаётся вызовом range[Symbol.iterator](), и именно его next() генерирует значения.

Таким образом, итератор отделен от самого итерируемого объекта. Технически, можно объединить их и использовать сам range как итератор, чтобы упростить код. Например, вот так:

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

this.current = this.from;

return this;

},

next() {

if (this.current <= this.to) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next() и он запоминает текущее состояние итерации в this.current. Недостаток такого подхода в том, что теперь нельзя использовать этот объект в двух параллельных циклах for..of: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах for..of, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

Можно сделать бесконечный итератор. Например, range будет бесконечным при range.to = Infinity. Или можно создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод next не имеет ограничений, он может возвращать все новые и новые значения, это нормально. Конечно же, цикл for..of с таким итерируемым объектом будет бесконечным. Но всегда можно прервать его, используя break.

[**Итерируемые объекты и псевдомассивы**](https://learn.javascript.ru/iterable#array-like)

Есть два термина, которые очень похожи, но в то же время сильно различаются.

*Итерируемые объекты*  – это объекты, которые реализуют метод Symbol.iterator, как было описано выше.

*Псевдомассивы* – это объекты, у которых есть индексы и свойство length, то есть они выглядят как массивы.

При использовании JavaScript в браузере или других окружениях, можно встретить объекты, которые являются итерируемыми или псевдомассивами или и тем и другим. Например, строки итерируемы (для них работает for..of) и являются псевдомасссивами (они индексированы и есть length).

Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым. Например, объект range из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и length.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

let arrayLike = {

0: "Hello",

1: "World",

length: 2

};

// Ошибка (отсутствует Symbol.iterator)

for (let item of arrayLike) {}

И итерируемые объекты, и псевдомассивы – это обычно не массивы, у них нет методов push, pop и т.д. Довольно неудобно, если есть такой объект и надо работать с ним как с массивом.

Есть универсальный метод [Array.from](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/from), который принимает итерируемый объект или псевдомассив и делает из него «настоящий» Array. После этого уже можно использовать методы массивов. Например:

let arrayLike = {

0: "Hello",

1: "World",

length: 2

};

let arr = Array.from(arrayLike); // (\*)

alert(arr.pop()); // World (метод работает)

Array.from в строке (\*) принимает объект, проверяет, является ли он итерируемым объектом или псевдомассивом, затем создаёт новый массив и копирует туда все элементы. То же самое происходит с итерируемым объектом:

let arr = Array.from(range);

alert(arr); // 1,2,3,4,5

Полный синтаксис Array.from позволяет указать необязательную «трансформирующую» функцию:

Array.from(obj[, mapFn, thisArg])

Необязательный второй аргумент может быть функцией, которая будет применена к каждому элементу перед добавлением в массив, а thisArg позволяет установить this для этой функции. Например:

// range взят из примера выше

let arr = Array.from(range, num => num \* num);

alert(arr); // 1,4,9,16,25

Здесь используется Array.from, чтобы превратить строку в массив её элементов:

let str = '𝒳😂';

let chars = Array.from(str);

alert(chars[0]); // 𝒳

alert(chars[1]); // 😂

alert(chars.length); // 2

В отличие от str.split, этот метод в работе опирается на итеративный характер строки и поэтому, совсем как for..of, он корректно работает с суррогатными парами. Технически, это то же самое, что и код ниже, но гораздо короче:

let str = '𝒳😂';

let chars = [];

for (let char of str) {

chars.push(char);

}

alert(chars);

Мы можем даже создать slice, который поддерживает суррогатные пары:

function slice(str, start, end) {

return Array.from(str).slice(start, end).join('');

}

let str = '𝒳😂𩷶';

alert( slice(str, 1, 3) ); // 😂𩷶

// встроенный метод не поддерживает суррогатные пары

alert( str.slice(1, 3) ); // мусор

31. Преобразование объектов.

Если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj), то объекты сначала автоматически преобразуются в примитивы, а затем выполнится операция.

Правила преобразования объектов:

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда осуществляется вычитание объектов или выполняются другие математические операции. Например, объекты Date могут вычитаться и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, при выводе объекта по типу alert(obj) и в подобных случаях.

[**Преобразование к примитивам**](https://learn.javascript.ru/object-toprimitive#preobrazovanie-k-primitivam)

Существуют три варианта преобразований:

* "string" – для преобразования объекта к строке, когда операция ожидает получить строку, например, alert:

alert(obj);

// объект в качества имени свойства

anotherObj[obj] = 123;

* "number" – для преобразования объекта к числу, в случае математических операций:

// явное преобразование

let num = Number(obj);

// математическое (исключая бинарный "+")

let n = +obj; // унарный плюс

let delta = date1 - date2;

// больше/меньше сравнения

let greater = user1 > user2;

* "default" – происходит редко, когда оператор «не уверен», какой тип ожидать.

В процессе преобразования, движок JavaScript пытается найти и вызвать три следующих метода объекта:

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен "string", пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он существует.
3. В случае, если хинт равен "number" или "default", пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он существует.

**Метод** [**Symbol.toPrimitive**](https://learn.javascript.ru/object-toprimitive#symbol-toprimitive)

Метод используется для всех преобразований:

obj[Symbol.toPrimitive] = function(hint) {

};

Для примера используем его в реализации объекта user:

let user = {

name: "John",

money: 1000,

[Symbol.toPrimitive](hint) {

alert(`hint: ${hint}`);

return hint == "string" ? `{name: "${this.name}"}` : this.money;

}

};

alert(user); // hint: string -> {name: "John"}

alert(+user); // hint: number -> 1000

alert(user + 500); // hint: default -> 1500

Как видно из кода, user преобразовался в информативную читаемую строку, либо в денежный счёт, в зависимости от значения хинта. Единственный метод user[Symbol.toPrimitive] смог обработать все случаи преобразований.

[**Методы toString/valueOf**](https://learn.javascript.ru/object-toprimitive#metody-tostring-valueof)

Методы toString и valueOf существуют давно. Они не символы, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода Symbol.toPrimitive, движок JavaScript пытается найти эти методы и вызвать следующим образом:

* toString – valueOf для хинта со значением «string».
* valueOf – toString – в ином случае.

Для примера, используем их в реализации всё того же объекта user. Воспроизведём его поведение комбинацией методов toString и valueOf:

let user = {

name: "John",

money: 1000,

// "string"

toString() {

return `{name: "${this.name}"}`;

},

// "number" или "default"

valueOf() {

return this.money;

}

};

alert(user); // toString -> {name: "John"}

alert(+user); // valueOf -> 1000

alert(user + 500); // valueOf -> 1500

Как видно, получилось то же поведение, что и у предыдущего примера с Symbol.toPrimitive.

Часто надо описать одно универсальное преобразование объекта к примитиву, для всех ситуаций. Для этого достаточно создать один toString:

let user = {

name: "John",

toString() {

return this.name;

}

};

alert(user); // toString -> John

alert(user + 500); // toString -> John500

В отсутствие Symbol.toPrimitive и valueOf, toString обработает все случаи преобразований к примитивам.

**Единственное обязательное требование: методы должны возвращать примитив, а не объект.**

Если toString или valueOf вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало). Метод Symbol.toPrimitive, напротив, обязан возвращать примитив, иначе будет ошибка.

[**Последующие операции**](https://learn.javascript.ru/object-toprimitive#posleduyuschie-operatsii)

Операция, инициализировавшая преобразование, получает примитив, и затем продолжает работу с ним, производя дальнейшие преобразования, если это необходимо. Например:

* Математические операции, исключая бинарный плюс, преобразуют примитив к числу:

let obj = {

toString() {

return "2";

}

};

alert(obj \* 2); // 4

32. Создание объектов через "new".

создать множество однотипных объектов, таких как пользователи, элементы меню можно сделать при помощи функции-конструктора и оператора "new".

[**Функция-конструктор**](https://learn.javascript.ru/constructor-new#funktsiya-konstruktor)

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

// this = {};

this.name = name;

this.isAdmin = false;

// return this;

}

То есть, результат вызова new User("Вася") – это тот же объект, что и:

let user = {

name: "Вася",

isAdmin: false

};

Теперь, когда необходимо будет создать других пользователей, можно использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически, любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new и выполнит алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором и её нужно вызывать через new.

**new function() { … }**

Если коде большое количество строк, создающих один сложный объект, можно обернуть их в функцию-конструктор следующим образом:

let user = new function() {

this.name = "Вася";

this.isAdmin = false;

// ...

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет инкапсулировать код, который создаёт отдельный объект, но без возможности его повторного использования. Данный метод используется очень редко.

Используя специальное свойство new.target внутри функции, можно проверить, вызвана ли функция при помощи оператора new или без него.

Обычно конструкторы ничего не возвращают. Их задача – записать все необходимое в this, который в итоге станет результатом.

Но если return всё же есть, то применяется простое правило:

* при вызове return с объектом, будет возвращён объект, а не this;
* при вызове return с примитивным значением, примитивное значение будет отброшено.

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this. В примере ниже return возвращает объект вместо this:

function BigUser() {

this.name = "Вася";

// возвращает объект

return { name: "Godzilla" };

}

alert( new BigUser().name );

Пример с пустым return (можно поставить примитив после return, не важно):

function SmallUser() {

this.name = "Вася";

return; // возвращает this

// ...

}

alert( new SmallUser().name ); // Вася

Можно не ставить скобки после new, если вызов конструктора идёт без аргументов:

let user = new User;

// то же, что и

let user = new User();

Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

[**Создание методов в конструкторе**](https://learn.javascript.ru/constructor-new#sozdanie-metodov-v-konstruktore)

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать. В this можно добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi:

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Меня зовут: " + this.name );

};

}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/\*

vasya = {

name: "Вася",

sayHi: function() { ... }

}

\*/

33. Флаги и дескрипторы свойств.

[**Флаги свойств**](https://learn.javascript.ru/property-descriptors#flagi-svoystv)

Помимо значения value, свойства объекта имеют три специальных атрибута (так называемые «флаги»):

* writable – если true, свойство можно изменить, иначе оно только для чтения.
* enumerable – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
* configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Эти атрибуты обычно скрыты. Когда создается свойство «обычным способом», все эти атрибуты имеют значение true. Но можно изменить их в любое время.

Метод [Object.getOwnPropertyDescriptor](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) позволяет получить полную информацию о свойстве. Его синтаксис:

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

* obj – объект, из которого получаем информацию.
* propertyName – имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги. Например:

let user = {

name: "John"

};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\* дескриптор свойства:

{

"value": "John",

"writable": true,

"enumerable": true,

"configurable": true

}

\*/

Чтобы изменить флаги, можно использовать метод [Object.defineProperty](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty). Его синтаксис:

Object.defineProperty(obj, propertyName, descriptor)

* obj, propertyName – объект и его свойство, для которого нужно применить дескриптор.
* descriptor – применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создает новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false. Например, здесь создаётся свойство name, все флаги которого имеют значение false:

let user = {};

Object.defineProperty(user, "name", {

value: "John"

});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\*

{

"value": "John",

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

[**Только для чтения**](https://learn.javascript.ru/property-descriptors#tolko-dlya-chteniya)

Сделаем свойство user.name доступным только для чтения. Для этого изменим флаг writable:

let user = {

name: "John"

};

Object.defineProperty(user, "name", {

writable: false

});

user.name = "Pete"; // Ошибка

[**Неперечисляемое свойство**](https://learn.javascript.ru/property-descriptors#neperechislimoe-svoystvo)

Добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечисляемый, его не видно в цикле for..in. Но если написать свой собственный метод toString, цикл for..in будет выводить его по умолчанию:

let user = {

name: "John",

toString() {

return this.name;

}

};

for (let key in user) alert(key); // name, toString

Если в этом нет необходимости, можно установить для свойства enumerable: false. Тогда оно перестанет появляться в цикле for..in, аналогично встроенному toString:

let user = {

name: "John",

toString() {

return this.name;

}

};

Object.defineProperty(user, "toString", {

enumerable: false

});

for (let key in user) alert(key); // name

Неперечисляемые свойства также не возвращаются Object.keys:

alert(Object.keys(user)); // name

[**Неконфигурируемое свойство**](https://learn.javascript.ru/property-descriptors#nekonfiguriruemoe-svoystvo)

Флаг неконфигурируемого свойства (configurable: false) иногда предустановлен для некоторых встроенных объектов и свойств. Неконфигурируемое свойство не может быть удалено или изменено с помощью defineProperty. Например, свойство Math.PI – только для чтения, неперечисляемое и неконфигурируемое:

let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );

/\*

{

"value": 3.141592653589793,

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

То есть программист не сможет изменить значение Math.PI или перезаписать его.

Math.PI = 3; // Ошибка

[**Метод Object.defineProperties**](https://learn.javascript.ru/property-descriptors#metod-object-defineproperties)

Существует метод [Object.defineProperties(obj, descriptors)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties), который позволяет определять множество свойств сразу. Его синтаксис:

Object.defineProperties(obj, {

prop1: descriptor1,

prop2: descriptor2

// ...

});

Например:

Object.defineProperties(user, {

name: { value: "John", writable: false },

surname: { value: "Smith", writable: false },

// ...

});.

**Метод** [**Object.getOwnPropertyDescriptors**](https://learn.javascript.ru/property-descriptors#object-getownpropertydescriptors)

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом [Object.getOwnPropertyDescriptors(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors). Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));

Обычно при клонировании объекта используется присваивание, чтобы скопировать его свойства:

for (let key in user) {

clone[key] = user[key]

}

Но это не копирует флаги. Поэтому если нужен клон с флагами, предпочтительнее использовать Object.defineProperties. Другое отличие в том, что for..in игнорирует символьные свойства, а Object.getOwnPropertyDescriptors возвращает дескрипторы всех свойств, включая свойства-символы.

[**Глобальное запечатывание объекта**](https://learn.javascript.ru/property-descriptors#globalnoe-zapechatyvanie-obekta)

Дескрипторы свойств работают на уровне конкретных свойств. Но еще есть методы, которые ограничивают доступ ко всему объекту:

* [Object.preventExtensions(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions) – запрещает добавлять новые свойства в объект.
* [Object.seal(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/seal) – запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.
* [Object.freeze(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) – запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть методы для их проверки:

* [Object.isExtensible(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible) – возвращает false, если добавление свойств запрещено, иначе true.
* [Object.isSealed(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed) – возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.
* [Object.isFrozen(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isFrozen) – возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

34. Геттеры и сеттеры.

Есть два типа свойств объекта. Первый тип это свойства-данные (data properties). Все свойства, которые использовались до текущего момента были свойствами-данными. Второй тип свойств это свойства-аксессоры (accessor properties). По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

let obj = {

get propName() {

// геттер, срабатывает при чтении obj.propName

},

set propName(value) {

// сеттер, срабатывает при записи obj.propName = value

}

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение назначается. Например, есть объект user со свойствами name и surname:

let user = {

name: "John",

surname: "Smith"

};

Добавим свойство объекта fullName для полного имени – "John Smith". Реализуем его при помощи аксессора:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

}

};

alert(user.fullName); // John Smith

Снаружи свойство-аксессор выглядит как обычное свойство user.fullName  не вызывается  как функция, а читается как обычное свойство: геттер сам вернет нужное значение.

Добавив сеттер для user.fullName:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

},

set fullName(value) {

[this.name, this.surname] = value.split(" ");

}

};

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге получим «виртуальное» свойство fullName. Его можно прочитать и изменить, но по факту его не существует.

При попытке удалить свойство-аксессор оператором delete будет ошибка.

Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

**То есть, дескриптор аксессора может иметь**:

* get – функция без аргументов, которая сработает при чтении свойства,
* set – функция, принимающая один аргумент, вызываемая при присвоении свойства,
* enumerable – то же самое, что и для свойств-данных,
* configurable – то же самое, что и для свойств-данных.

Например, для создания аксессора fullName при помощи defineProperty можно передать дескриптор с использованием get и set:

let user = {

name: "John",

surname: "Smith"

};

Object.defineProperty(user, 'fullName', {

get() {

return `${this.name} ${this.surname}`;

},

set(value) {

[this.name, this.surname] = value.split(" ");

}

});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname

Ещё раз заметим, что свойство объекта может быть только свойством-аксессором (с методами get/set) или свойством-данных (со значением value). При попытке указать и get и value в одном дескрипторе будет ошибка:

// Error: Invalid property descriptor.

Object.defineProperty({}, 'prop', {

get() {

return 1

},

value: 2

});

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними. Например, если надо запретить устанавливать короткое имя для user, можно использовать сеттер name для проверки, а само значение хранить в отдельном свойстве \_name:

Аксессоры позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер. Например, представим, что реализован объект user, с использованием свойств-данных имя name и возраст age:

function User(name, age) {

this.name = name;

this.age = age;

}

let john = new User("John", 25);

alert( john.age ); // 25

Но со временем взамен возраста age можно хранить дату рождения birthday, потому что так более точно и удобно:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

}

let john = new User("John", new Date(1992, 6, 1));

Чтобы не менять весь старый код, который использует свойство age можно добавить геттер для age:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

Object.defineProperty(this, "age", {

get() {

let todayYear = new Date().getFullYear();

return todayYear - this.birthday.getFullYear();

}

});

}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday );

alert( john.age );

35. Декораторы. Методы call(), apply().

[**Прозрачное кеширование**](https://learn.javascript.ru/call-apply-decorators#prozrachnoe-keshirovanie)

Представим, что есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же *x* она всегда возвращает один и тот же результат. Если функция вызывается часто, то стоит кешировать (запомнить) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Вот код с объяснениями:

function slow(x) {

// здесь могут быть CPU ресурсоёмкие вычисления

alert(`Called with ${x}`);

return x;

}

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) { // если кэш содержит такой x

return cache.get(x); // читаем из него результат

}

let result = func(x); // иначе, вызываем функцию

cache.set(x, result); // и кешируем (запоминаем) результат

return result;

};

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем

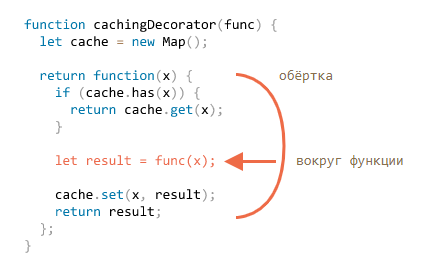
alert( "Again: " + slow(1) ); // возвращаем кеш

alert( slow(2) ); // slow(2) кешируем

alert( "Again: " + slow(2) ); // возвращаем кеш

В коде выше cachingDecorator – это декоратор, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что можно вызвать cachingDecorator с любой функцией, в результате чего получим кеширующую обёртку. Это удобно, т.к. может быть множество функций, использующих такой функционал, и все, что нужно сделать – это применить к ним cachingDecorator. Отделяя кеширующий код от основного кода, также сохраняем чистоту и простоту последнего.

Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:



С точки зрения внешнего кода обёрнутая функция slow по-прежнему делает то же самое. Обёртка всего добавляет к её поведению аспект кеширования.

**Подводя итог, можно выделить несколько преимуществ использования отдельной cachingDecorator вместо изменения кода самой slow:**

* Функцию cachingDecorator можно использовать повторно. Можно применить её к другой функции.
* Логика кеширования является отдельной, она не увеличивает сложность самой slow (если таковая была).
* При необходимости можно объединить несколько декораторов (речь о них пойдёт позже).

[**Применение «func.call» для передачи контекста**](https://learn.javascript.ru/call-apply-decorators#primenenie-func-call-dlya-peredachi-konteksta)

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов.

Существует специальный встроенный метод функции [func.call(context, …args)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call), который позволяет вызывать функцию, явно устанавливая this. Синтаксис:

func.call(context, arg1, arg2, ...)

Он запускает функцию func, предоставляя первый аргумент как this, а последующие как её аргументы. Проще говоря, эти два вызова делают почти то же самое:

func(1, 2, 3);

func.call(obj, 1, 2, 3)

Они оба вызывают func с аргументами 1, 2 и 3. Единственное отличие состоит в том, что **func.call также устанавливает this в obj**. Например, в приведённом ниже коде вызывается sayHi в контексте различных объектов: sayHi.call(user) запускает sayHi, передавая this=user, а следующая строка устанавливает this=admin:

function sayHi() {

alert(this.name);

}

let user = { name: "John" };

let admin = { name: "Admin" };

sayHi.call( user ); // this = John

sayHi.call( admin ); // this = Admin

Здесь используется call для вызова say с заданным контекстом и фразой:

function say(phrase) {

alert(this.name + ': ' + phrase);

}

let user = { name: "John" };

say.call( user, "Hello" ); // John: Hello

В рассматриваемом примере можно использовать call в обёртке для передачи контекста в исходную функцию:

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func.call(this, x); // 'this' передаётся правильно

cache.set(x, result);

return result;

};

}

worker.slow = cachingDecorator(worker.slow); // сделаем её кеширующей

alert( worker.slow(2) ); // работает

alert( worker.slow(2) ); // работает

**Рассмотрим подробнее, как передаётся this:**

1. После декорации worker.slow становится обёрткой function (x) { ... }.
2. Так что при выполнении worker.slow(2) обёртка получает 2 в качестве аргумента, и this=worker (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, func.call(this, x) передаёт текущий this (=worker) и текущий аргумент (=2) в оригинальную функцию.

[**Функция с несколькими аргументами – func.apply**](https://learn.javascript.ru/call-apply-decorators#perehodim-k-neskolkim-argumentam-s-func-apply)

Сделаем cachingDecorator ещё более универсальным. До сих пор он работал только с функциями с одним аргументом. Кешируем метод с несколькими аргументами worker.slow:

let worker = {

slow(min, max) {

return min + max; // здесь может быть тяжёлая задача

}

};

// should remember same-argument calls

worker.slow = cachingDecorator(worker.slow);

Соединить два значения в одно. В нашем конкретном случае можно просто использовать строку "min,max" как ключ к Map. Для гибкости, можно позволить передавать хеширующую функцию в декоратор, которая знает, как сделать одно значение из многих.

Нужно заменить func.call(this, x) на func.call(this, ...arguments), чтобы передавать все аргументы обёрнутой функции, а не только первый. Вот более мощный cachingDecorator:

let worker = {

slow(min, max) {

alert(`Called with ${min},${max}`);

return min + max;

}

};

function cachingDecorator(func, hash) {

let cache = new Map();

return function() {

let key = hash(arguments); // (\*)

if (cache.has(key)) {

return cache.get(key);

}

let result = func.call(this, ...arguments); // (\*\*)

cache.set(key, result);

return result;

};

}

function hash(args) {

return args[0] + ',' + args[1];

}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает

alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)

Теперь он работает с любым количеством аргументов. Есть два изменения:

* В строке (\*) вызываем hash для создания одного ключа из arguments. Здесь используем простую функцию «объединения», которая превращает аргументы (3, 5) в ключ "3,5". В более сложных случаях могут потребоваться другие функции хеширования.
* Затем (\*\*) используем func.call(this, ...arguments) для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их количества), в исходную функцию.

Вместо func.call(this, ...arguments) можно написать func.apply(this, arguments). Синтаксис встроенного метода [func.apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply):

func.apply(context, args)

Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args. Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив. Эти два вызова почти эквивалентны:

func.call(context, ...args);

func.apply(context, args);

**Есть только одна небольшая разница**: оператор расширения ... позволяет передавать перебираемый объект args в виде списка в call, а apply принимает только псевдомассив args. Так что эти вызовы дополняют друг друга. Для перебираемых объектов сработает call, а там, где ожидается псевдомассив – apply.

Если есть, например, реальный массив, то технически можно использовать любой, но apply, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше. Передача всех аргументов вместе с контекстом другой функции называется «перенаправлением вызова» (call forwarding). Простейший вид такого перенаправления:

let wrapper = function() {

return func.apply(this, arguments);

};

При вызове wrapper из внешнего кода его не отличить от вызова исходной функции.

[**Заимствование метода**](https://learn.javascript.ru/call-apply-decorators#method-borrowing)

Сделаем ещё одно небольшое улучшение функции хеширования:

function hash(args) {

return args[0] + ',' + args[1];

}

способ использовать соединение массива:

function hash() {

alert( [].join.call(arguments) ); // 1,2

}

hash(1, 2);

Этот способ называется **заимствование метода**. Т.е. заимствуется метод join из обычного массива [].join и используется [].join.call, чтобы выполнить его в контексте arguments. Это связано с тем, что внутренний алгоритм встроенного метода arr.join(glue) очень прост:

1. Пускай первым аргументом будет glue или, в случае отсутствия аргументов, им будет запятая ",".
2. Пускай result будет пустой строкой "".
3. Добавить this[0] к result.
4. Добавить glue и this[1].
5. Добавить glue и this[2].
6. Выполнять до тех пор, пока this.length элементов не будет склеено.
7. Вернуть result.

Таким образом, технически он принимает this и объединяет this[0], this[1]… и т.д. вместе. Он намеренно написан так, что допускает любой псевдомассив this (не случайно, многие методы следуют этой практике). Вот почему он также работает с this=arguments.

36. Привязка контекста.

[**Потеря «this»**](https://learn.javascript.ru/bind#poterya-this)

Примеры потери this рассматривались ранее. Как только метод передается отдельно от объекта – this теряется. Вот как это может произойти с setTimeout:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(user.sayHi, 1000); // Привет, undefined!

При запуске этого кода видно, что вызов this.firstName возвращает не «Вася», а undefined. Это произошло потому, что setTimeout получил функцию sayHi, отдельно от объекта user (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

let f = user.sayHi;

setTimeout(f, 1000); // контекст user потеряли

Задача состоит в том, что надо передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван.

Есть несколько решений этой задачи. Одно из них – [сделать функцию-обёртку](https://learn.javascript.ru/bind#reshenie-1-sdelat-funktsiyu-obyortku), обернуть вызов в анонимную функцию, создав замыкание:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(function() {

user.sayHi(); // Привет, Вася!

}, 1000);

Теперь код работает корректно, так как объект user достаётся из замыкания, а затем вызывается его метод sayHi. То же самое, только короче:

setTimeout(() => user.sayHi(), 1000); // Привет, Вася!

Теперь в коде появилась небольшая уязвимость: до момента срабатывания setTimeout (задержка составляет целую секунду) в переменную user может быть записано другое значение. Тогда вызов будет совсем не тот:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(() => user.sayHi(), 1000);

user = { sayHi() { alert("Другой пользователь в 'setTimeout'!"); } };

Второе решение описанной выше задачи гарантирует, что такого не случится. Оно заключается в привязке [контекста с помощью bind](https://learn.javascript.ru/bind#reshenie-2-privyazat-kontekst-s-pomoschyu-bind). В современном JavaScript у функций есть встроенный метод [bind](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/bind), который позволяет зафиксировать this. Базовый синтаксис bind:

let boundFunc = func.bind(context);

Результатом вызова func.bind(context) является особый, который вызывается как функция и прозрачно передает вызов в func, при этом устанавливая this=context. Другими словами, вызов boundFunc подобен вызову func с фиксированным this. Например, в коде ниже funcUser передает вызов в func, фиксируя this=user:

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

Здесь func.bind(user) – это «связанный вариант» func, с фиксированным this=user. Все аргументы передаются исходному методу func «как есть», например:

let user = {

firstName: "Вася"

};

function func(phrase) {

alert(phrase + ', ' + this.firstName);

}

// привязка this к user

let funcUser = func.bind(user);

funcUser("Привет"); // Привет, Вася

Теперь давайте попробуем с методом объекта:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

let sayHi = user.sayHi.bind(user); // (\*)

sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!

В строке (\*) метод user.sayHi привязываем к user. Теперь SayHi – это «связанная» функция, которая может быть вызвана отдельно или передана в setTimeout (контекст всегда будет правильным). Здесь можно видеть, что bind исправляет только this, а аргументы передаются «как есть»:

let user = {

firstName: "Вася",

say(phrase) {

alert(`${phrase}, ${this.firstName}!`);

}

};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася

say("Пока"); // Пока, Вася

До сих пор речь шла только о привязывании this. Можно привязать не только this, но и аргументы. Это делается редко, но иногда может быть полезно.

Полный синтаксис bind:

let bound = func.bind(context, [arg1], [arg2], ...);

Это позволяет привязать контекст this и начальные аргументы функции. Например, у нас есть функция умножения mul(a, b):

function mul(a, b) {

return a \* b;

}

Воспользуемся bind, чтобы создать функцию double на её основе:

function mul(a, b) {

return a \* b;

}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

Вызов mul.bind(null, 2) создаёт новую функцию double, которая передаёт вызов mul, фиксируя null как контекст и 2 – как первый аргумент. Следующие аргументы передаются «как есть». Это называется [частичное применение](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) – создаётся новая функция, которая фиксирует некоторые из существующих параметров.

Обратите внимание, что в данном случае не используется this. Но для bind это обязательный параметр, так что надо передать туда что-нибудь вроде null. В следующем коде функция triple умножает значение на три:

function mul(a, b) {

return a \* b;

}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9

alert( triple(4) ); // = mul(3, 4) = 12

alert( triple(5) ); // = mul(3, 5) = 15

Польза от частично применённой функции в том, что можно создать независимую функцию с понятным названием (double, triple). Можно использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью bind.

В других случаях частичное применение полезно, когда есть очень общая функция и для удобства надо создать её частный вариант. Например, есть функция send(from, to, text) и позже может возникнуть необходимость внутри объекта user использовать её частный вариант: sendTo(to, text), который отправляет текст от имени текущего пользователя.

Возможна ситуация, когда надо зафиксировать некоторые аргументы, но не контекст this. Например, для метода объекта. Встроенный bind не позволяет этого. Нельзя просто опустить контекст и перейти к аргументам. Но можно создать вспомогательную функцию partial, которая привязывает только аргументы. Вот так:

function partial(func, ...argsBound) {

return function(...args) { // (\*)

return func.call(this, ...argsBound, ...args);

}

}

let user = {

firstName: "John",

say(time, phrase) {

alert(`[${time}] ${this.firstName}: ${phrase}!`);

}

};

user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello"); // [10:00] John: Hello!

Результатом вызова partial(func[, arg1, arg2...]) будет обёртка (\*), которая вызывает func с:

* Тем же this, который она получает (для вызова user.sayNow – это будет user)
* Затем передаёт ей ...argsBound – аргументы из вызова partial ("10:00")
* Затем передаёт ей ...args – аргументы, полученные обёрткой ("Hello")

Благодаря оператору расширения ... это реализовать легко.