



ФУНКЦИИ И ОБЪЕКТЫ



ВЛАДИМИР ЧЕБУКИН



ВЛАДИМИР ЧЕБУКИН

Веб-разработчик



vovachebr@mail.ru



fb.me/vovachebr



[@User123423](https://t.me/User123423)

ПЛАН ЗАНЯТИЯ

1. Функции
2. Аргументы и `rest` оператор
3. Функциональные выражения
4. Стрелочные функции
5. Объекты и их свойства

МЫ УЖЕ ЗНАКОМЫ С ФУНКЦИЯМИ

Наш давний друг, *console.log*

```
console.log('log – это функция!');
```

Вспомним и способы работы с массивами. Там функции также используются:

```
1 [ 2, 3, 4 ].push(7); // добавит 7 в конец массива
2 [ 6, 7, 11 ].pop(); // извлечёт 11 из конца массива и удалит его оттуда
3
4 [ 67, 8, 2 ].unshift(189); // добавит 189 в начало массива
5 [ 712, 8 ].shift(); // извлечёт 712 из начала массива и удалит его оттуда
```



ЧТО ТАКОЕ ФУНКЦИЯ?

Простыми словами, функция — написанный код (нами или кем-то ещё), который мы можем многократно повторять всего лишь одной командой!

ПОЧЕМУ БЕЗ ФУНКЦИЙ ПЛОХО?

1. **Повторы.** Много дублирования кода
2. **Избыточность.** Много лишнего для понимания программы
3. **Рискованно.** Вероятность допустить ошибку в коде на 10 строчек выше, чем в коде на 1 строчку
4. **Рутина.** Нужно постоянно копировать удачный участок кода и вставлять в другое место

Код с *unshift*:

```
1 | let data = [ 7, 4, 2 ];  
2 |  
3 | data.unshift(10); // 10, 7, 4, 2
```

ПОЧЕМУ БЕЗ ФУНКЦИЙ ПЛОХО?

Код без *unshift*:

```
1  let data = [ 7, 4, 2 ];
2
3  // смещаем индексы на 1 вправо
4  for (let i = data.length; i--;) {
5      data[ i + 1 ] = data[ i ];
6  }
7
8  data[ 0 ] = 10; // добавляем 10 в начало
```



МИНИМАЛИЗМ ФУНКЦИЙ

Нам не нужно знать, как внутри устроен *console.log* для того, чтобы вывести сообщение в консоль. Всё, что необходимо знать:

1. Что нужно поместить в круглые скобочки
2. Что делает функция

ФУНКЦИИ ПРОСТО ПОНИМАТЬ!

Вопрос:

Хотите вывести в консоль сумму 4 и 5?

Ответ:

Поместите в круглые скобки `console.log()`* выражение `4 + 5`:

```
console.log(4 + 5);
```

ОБЪЯВЛЕНИЕ ФУНКЦИИ

Мы можем создавать свои функции! Для этого мы должны её *объявить*:

```
1 function имяФункции() {  
2     // тело функции  
3 }
```

Внутри обязательных **фигурных** скобок мы пишем *тело функции*, то есть тот код, к которому мы будем часто обращаться в наших программах.

ПРИМЕР НАШЕЙ ФУНКЦИИ

Например, мы могли бы терроризировать посетителей такой функцией:


```
1 function showVacancy() {  
2     console.log('Здравствуйте! Вы умеете пользоваться консолью!');  
3     console.log('Вы, наверное, разработчик?');  
4     console.log('Перейдите, пожалуйста, по ссылке: http://you.vacancy/1');  
5 }
```

ВЫЗОВ ФУНКЦИИ

Для того чтобы запустить функцию *showVacancy*, по аналогии с *console.log*, нам всего лишь нужно дописать открывающую и закрывающую скобки:

```
showVacancy();
```

Запуск кода выше выведет в консоль:



```
📄 Здравствуйте! Вы умеете пользоваться консолью! showVacancy —  
📄 Вы, наверное, разработчик? showVacancy —  
📄 Перейдите, пожалуйста, по ссылке: http://you.vacancy/1 showVacancy —
```

Такая процедура называется вызовом *функции*.

МНОГОКРАТНЫЙ ВЫЗОВ

Мы можем обратиться к ранее написанной функции сколько угодно раз:

```
1 console.log('Добрый день!');  
2 showVacancy();  
3  
4 console.log('Пишем какой-то код');  
5 console.log('Закончили писать');  
6  
7 showVacancy();  
8 showVacancy();
```

МНОГОКРАТНЫЙ ВЫЗОВ

```
└─ Добрый день!
└─ Здравствуйте! Вы умеете пользоваться консолью!
└─ Вы, наверное, разработчик?
└─ Перейдите, пожалуйста, по ссылке: http://you.vacancy/1
└─ Пишем какой-то код
└─ Закончили писать
└─ Здравствуйте! Вы умеете пользоваться консолью!
└─ Вы, наверное, разработчик?
└─ Перейдите, пожалуйста, по ссылке: http://you.vacancy/1
└─ Здравствуйте! Вы умеете пользоваться консолью!
└─ Вы, наверное, разработчик?
└─ Перейдите, пожалуйста, по ссылке: http://you.vacancy/1
```

БАЗОВЫЕ ПЛЮСЫ ФУНКЦИЙ

Функции — это ПСП:

1. **Порядок.** Вы можете не дублировать многократно один и тот же участок кода. Вместо этого вызовите несколько раз функцию!
2. **Свобода.** В сравнение с массивом, вы можете вызвать код не строго несколько раз подряд, а в произвольных местах.
3. **Простота.** Вам не нужно изучать содержимое функции для того, чтобы знать, что она делает.

ВОЗВРАТ ЗНАЧЕНИЙ

Вычислим расход топлива по дороге от Ростова до Краснодара:

```
1 function getFuelExpense() {  
2   // расход топлива, литров на 100 км.  
3   let fuelPer100km = 10;  
4  
5   // расстояние между Ростовом и Краснодаром в км.  
6   let distance = 284;  
7  
8   // получаем расход топлива  
9   let expense = distance / fuelPer100km;  
10  
11   console.log(`Расход топлива на ${distance} км. составит ${expense} л.`);  
12 }
```

Как быть, если нам необходимо использовать информацию о расходе вне *getFuelExpense*?

ВОЗВРАТ ЗНАЧЕНИЙ

Для этого есть конструкция `return` выражение :

```
1 function getFuelExpense() {  
2   let fuelPer100km = 10;  
3   let distance = 284;  
4   let expense = distance / fuelPer100km;  
5  
6   return expense;  
7 }
```

ПОЛУЧЕНИЕ ЗНАЧЕНИЙ

Как же мы можем получить *возвращаемое значение*?

Присвойте *значение* вызванной функции в переменную!

```
let fuelData = getFuelExpense();  
console.log(fuelData); // выведет 28.4
```

или выведите выражение сразу!

```
console.log(getFuelExpense()); // выведет 28.4
```

КОД ПОСЛЕ `return` НИКОГДА НЕ БУДЕТ ВЫПОЛНЕН!

```
1 function getFuelExpense() {
2   let fuelPer100km = 10;
3   let distance = 284;
4   let expense = distance / fuelPer100km;
5
6   return expense;
7   // код ниже никогда не будет выполнен
8   console.log('Нужно больше топлива!');
9   return distance;
10 }
11
12 console.log(getFuelExpense()); // выведет 28.4
13 console.log(getFuelExpense()); // выведет 28.4
```

ФУНКЦИЯ БЕЗ РЕЗУЛЬТАТА

Вопрос, что будет выведено на консоль?

```
function nothing() {  
}
```

```
console.log(nothing())
```

ВОЗВРАЩАТЬ МОЖНО ВСЁ, ЧТО УГОДНО!

```
1  function getSomePoetry() {
2      return `
3          Мне не надо ни солнца, ни туч,
4          Ты одна мне нужна в этом мире.
5          Подарю тебе гаечный ключ,
6          Двадцать два на двадцать четыре.
7      `;
8  }
9
10 function getFamilyLength() {
11     return [ 'Феоклид', 'Ефросинья', 'Ратибор' ].length;
12 }
13
14 function getStupidFive() {
15     return 1 + 2 + 2;
16 }
```

ВОЗВРАЩАТЬ МОЖНО ДАЖЕ ФУНКЦИИ! (ДЕМО)

```
1 function getConsoleLog() {  
2     return console.log;  
3 }  
4 let log = getConsoleLog();  
5 log(2+2); // эквивалентно console.log(2+2)
```

!Важно

Возвращается **функция**, а не **вызов функции**:

```
1 // из функции getConsoleLog вернётся не функция console.log, а её результат.  
2 return console.log();
```

Перед возвращением результата будет вызвана `console.log()` и результат вызова (который является `undefined`) будет результатом функции `getConsoleLog`. *Подробнее об этом будет в теме функций высшего порядка.*



АРГУМЕНТЫ И REST ОПЕРАТОР

АРГУМЕНТЫ

Мы можем задавать разные начальные условия для функций с помощью переменных функции, которые называются *аргументами*.

Укажите аргумент в круглых скобках при *объявлении* функции:

```
1 function showGreeting(name) {  
2   console.log(`Поздравляем, ${name}, вы выиграли АВТОМОБИЛЬ! Ваш Леонид Аркадьевич!`);  
3 }
```

Для того, чтобы передать значение в аргумент, необходимо его указать в круглых скобках при вызове функции:

```
showGreeting('Иван');
```

```
📄 Поздравляем, Иван, вы выиграли АВТОМОБИЛЬ! Ваш Леонид Аркадьевич!
```

НЕСКОЛЬКО АРГУМЕНТОВ

__Внимание!__ Эта информация будет полезна вам при решении ДЗ!

С помощью запятой «,» мы можем использовать несколько аргументов:

```
1 // Найдём значение y в выражении  $y = kx + b$ 
2 function getResult(x, k, b) {
3     return k * x + b;
4 }
5
6 // Для передачи нескольких значений мы также используем запятую
7
8 let y = getResult(3, 7, 9); // 30
```

НЕ ПЕРЕДАННЫЕ АРГУМЕНТЫ ИМЕЮТ ЗНАЧЕНИЕ *UNDEFINED*

Для наглядного примера создадим функцию, которая возвращает то значение, которое мы ей передаём.

```
1 function identity(value) {  
2   console.log(value);  
3 }  
4  
5 identity(4); // 4  
6 identity('Анна'); // Анна  
7  
8 identity(); // undefined!
```

АРГУМЕНТЫ ПО УМОЛЧАНИЮ

Значения, которые вы забыли передать в функцию, могут давать неожиданные результаты:

```
1 function sum(a, b) {  
2   return a + b;  
3 }  
4  
5 console.log(sum(3, 4)); // 7  
6  
7 console.log(sum(3)); // NaN!
```

В последнем примере аргумент *b* имеет значение *undefined* и сумма *3 + undefined* даст нам *NaN*.

АРГУМЕНТЫ ПО УМОЛЧАНИЮ (ДЕМО)

Для того, чтобы избежать таких проблем, мы можем дать аргументу значение по умолчанию:

```
1 function sum(a, b = 0) {  
2   return a + b;  
3 }  
4  
5 console.log(sum(3, 4)); // 7  
6  
7 console.log(sum(3)); // 3!
```

Значение по умолчанию будет подставлено только в случае, если значение аргумента не будет передано или будет равно *undefined*.

...REST

В случае, если функция имеет неопределённое число аргументов, на помощь приходит оператор взятия остатка ...:

```
1 function getArgs(...data) {  
2   console.log(data);  
3 }  
4  
5 getArgs(2, 4, 5, 6, 7, 10, 45, 11);
```

В данной ситуации в консоль попадёт массив всех переданных в функцию аргументов:

```
📄 [2, 4, 5, 6, 7, 10, 45, 11] (8)
```

ПЕРЕМЕННОЕ ЧИСЛО АРГУМЕНТОВ

Переменное число аргументов чаще всего используют при однородных значениях:

```
1 function sum(...args) {  
2   let total = 0;  
3   for (let i = 0; i < args.length; i++) {  
4     total += args[ i ];  
5   }  
6   return total;  
7 }  
8  
9 console.log(sum(2, 4, 5, 16, 7, 10, 11)); // 55
```

...REST И ОСТАЛЬНЫЕ АРГУМЕНТЫ

В случае, когда функция содержит не всегда однородные значения, их можно вынести в начало списка аргументов.

```
function duHast(a, b, c, ...rest) {  
}
```

При этом ***...rest* должен быть в конце списка аргументов!** Например, создадим функцию, которая создаёт тариф оплаты вместе со списком преимуществ

...REST И ОСТАЛЬНЫЕ АРГУМЕНТЫ

```
1 function showTariff(name, ...advantages) {  
2   let text = `Тариф ${name}\nПреимущества:\n`;  
3   for (let i = 0; i < advantages.length; i++) {  
4     text += `-${advantages[ i ]}\n`;  
5   }  
6  
7   console.log(text);  
8 }  
9  
10 showTariff('Базовый', 'Кровать на чердаке', 'Беседы с дядей Витей');  
11 showTariff('Оптимум', 'Кофе в постель без чашки', 'Раздельный санузел', 'Гарантия на возврат 5%');
```

```
❏ Тариф Базовый  
Преимущества:  
-Кровать на чердаке  
-Беседы с дядей Витей
```

```
❏ Тариф Оптимаум  
Преимущества:  
-Кофе в постель без чашки  
-Раздельный санузел  
-Гарантия на возврат 5%
```



ФУНКЦИОНАЛЬНЫЕ ВЫРАЖЕНИЯ

ФУНКЦИОНАЛЬНЫЕ ВЫРАЖЕНИЯ

В переменную можно поместить всё что угодно, даже функцию! (вспомним пример, где возвращается `console.log`)

```
let sum = function (a, b) {  
  return a + b;  
}
```

Такая конструкция называется *функциональным выражением*. **Это просто ещё один способ объявить функцию.** Мы можем также обратиться к переменной, в которой находится функция, как и к обычному объявлению функции:

```
console.log(sum(3, 4)); // 7
```

Иными словами, *функциональным* выражением называется всё, что позволяет использовать функцию как значение.

ОБЪЯВЛЕНИЯ ФУНКЦИЙ VS ФУНКЦИОНАЛЬНЫЕ ВЫРАЖЕНИЯ

Между *функциональными выражениями* и *объявлениями функций* есть одна принципиальная разница: *функциональные выражения* можно использовать только **после** присвоения функции в переменную, *объявления функций* доступны **независимо от места** объявления:

```
1 // Выдаст ошибку
2 console.log(takeFive()); // TypeError: takeFive is not a function.
3                          //(In 'takeFive()', 'takeFive' is undefined)
4
5 let takeFive = function() {
6     return 5;
7 }
8
9 // Из-за ошибки с первым console.log, эта строка вообще не выполнится
10 console.log(takeFive());
```

ОБЪЯВЛЕНИЯ ФУНКЦИЙ VS ФУНКЦИОНАЛЬНЫЕ ВЫРАЖЕНИЯ

```
1 console.log(takeFive()); // 5
2
3 function takeFive() {
4     return 5;
5 }
6
7 console.log(takeFive());
```

Такой принцип объявления функции называется *поднятием* (hoisting). Грубо говоря, интерпретатор **дважды** обрабатывает наш код перед тем, как мы увидим конечный результат отработки кода.



СТРЕЛОЧНЫЕ ФУНКЦИИ

ПОЛНЫЙ И КРАТКИЙ СИНТАКСИС

Проблема: обычные функции достаточно длинные, необходимо писать объявление функции и её тело. В переменных `sum`, `sumArrow` и `sumArrowBlock` будут содержаться идентичные функции.

```
1  let sum = function (a, b){  
2      return a + b;  
3  }  
4  let sumArrow = (a, b) => a + b;  
5  // краткий синтаксис, используется если в функции одно действие  
6  let sumArrowBlock = (a, b) => { return a + b };  
7  // блочный синтаксис
```

СКОБКИ В СТРЕЛОЧНЫХ ФУНКЦИЯХ

Если в функции 1 аргумент, то скобки **не обязательны**:

```
1 let multiply = a => a * 2;  
2 // аргумент "a" не обернут в скобки  
3 console.log(multiply(4));  
4 // 8
```

Если аргументы отсутствуют или их больше одно, то **скобки обязательны**.

ОБЪЕКТЫ И ИХ СВОЙСТВА

ОБЪЕКТЫ. ЗАЧЕМ НУЖНЫ ОБЪЕКТЫ?

Объекты позволяют удобно организовать хранение информации.
Для создания нового объекта, мы пользуемся конструктором объекта *new*.

```
1 let person = new Object();  
2  
3 // мы можем задавать произвольные параметры объекту  
4 person.firstName = 'Иван';  
5 person.lastName = 'Орлов';  
6 person.age = 45;
```

Такой подход нагляднее, чем:

```
let person = [ 'Иван', 'Орлов', 45 ]; // что такое 45?
```

НА САМОМ ДЕЛЕ МАССИВЫ И ФУНКЦИИ – ТОЖЕ ОБЪЕКТЫ!

В JavaScript почти всё является объектом:

Массивы:

```
1 let data = new Array('Иннокентий', 'Ильдар', 'Ирина');
2
3 data.push('Ираида');
4
5 console.log(data.length); // 4
```

И даже функции:

```
1 // функция, которая складывает 2 числа
2 let sum = new Function('a', 'b', 'return a + b');
3
4 console.log(sum(3, 4)); // 7
```

ЛИТЕРАЛ ОБЪЕКТА

У объектов, как и массивов, есть сокращённая форма записи. Она называется *литералом*.

```
1 let person = new Object();  
2  
3 person.firstName = 'Иван';  
4 person.lastName = 'Орлов';  
5 person.age = 45;
```

аналогично

```
1 let person = {  
2   firstName: 'Иван',  
3   lastName: 'Орлов',  
4   age: 45  
5 };
```

СВОЙСТВА. ЗАДАНИЕ СВОЙСТВ

У объектов есть свойства. Вы можете задавать объекту абсолютно произвольные свойства с произвольными значениями.

Свойства в объекте — примерно то же самое, что и атрибуты у HTML-тегов.

```
1 let customProperty = 'isCat';
2 let person = {
3   firstName: 'Иван',
4   // свойства можно задавать и в кавычках
5   'lastName': 'Орлов',
6   // кавычки удобны для задания специфических значений
7   "font-size": '20px',
8   // ES6+, customProperty может быть любым JS-выражением
9   [customProperty]: false // создаст свойство isCat со значением false
10 };
11
12 // можно задавать свойства после создания объекта
13 person.fatherName = 'Борис';
14 person['patronym'] = 'Борисович';
15
16 let newProperty = 'gender';
17 // в квадратные скобки можно поместить любое JS-выражение
18 person[newProperty] = 'male'; // создаст свойство gender со значением male
```

ES6+. СОКРАЩЁННОЕ ЗАДАНИЕ СВОЙСТВ

Можно задавать свойства на основе имени переменной:

```
1 let firstName = 'Иван';  
2 let lastName = 'Печенькин';  
3  
4 let person = {  
5   firstName,  
6   lastName  
7 };
```

аналогично ES6-:

```
1 let firstName = 'Иван';  
2 let lastName = 'Печенькин';  
3  
4 let person = {  
5   firstName: firstName,  
6   lastName: lastName  
7 };
```

ЧТЕНИЕ СВОЙСТВ

Чтение свойств аналогично их заданию

```
1  let person = {  
2    firstName: 'Иван',  
3    lastName: 'Орлов',  
4    age: 45  
5  };  
6  
7  console.log(person.firstName); // Иван  
8  console.log(person[ 'lastName' ]); // Орлов  
9  
10 let myProperty = 'age';  
11 console.log(person[ myProperty ]); // 45
```

НЕСУЩЕСТВУЮЩИЕ СВОЙСТВА

Доступ к любому несуществующему свойству даёт *undefined*:

```
1  let person = {  
2    firstName: 'Иван',  
3    lastName: 'Орлов',  
4    age: 45  
5  };  
6  
7  console.log(person.fatherName); // undefined
```


ОБЪЕКТЫ ЯВЛЯЮТСЯ ЗНАЧЕНИЯМИ ПО ССЫЛКЕ

В отличие от примитивных типов данных, при присваивании объекта, копируется не значение, а ссылка на этот объект.

Примитивы:

```
1  let x = 6,  
2  y = x;  
3  
4  y = 9;  
5  
6  console.log(x, y); // 6, 9
```

Объекты:

```
1  let ivan = {  
2    firstName: 'Иван',  
3    lastName: 'Зайцев'  
4  }  
5  
6  let oleg = ivan;  
7  oleg.firstName = 'Олег';  
8  
9  console.log(ivan.firstName, oleg.firstName);  
10 // Что будет выведено?
```

ПОЯСНЕНИЕ

```
1  /*
2  | | | Попробуем занести информацию о 2-ух братьях
3  | | | P.S. У них должна быть одинаковая фамилия
4  */
5  var ivan = {
6  | |   firstName: 'Иван',
7  | |   lastName: 'Зайцев'
8  }
9
10 var oleg = ivan;
11 oleg.firstName = 'Олег';
12
13 // Тут нас ждёт неприятный сюрприз
14 console.log( ivan.firstName, oleg.firstName ); // Олег, Олег
```

одно и то же место

ivan

oleg

память

{
 firstName: 'Иван',
 lastName: 'Зайцев'
}

{
 firstName: ~~'Иван'~~ **Олег**,
 lastName: 'Зайцев'
}

РЕШЕНИЕ

Для решения этого казуса нам нужно создать отдельный объект для Олега:

```
1  let ivan = {  
2    firstName: 'Иван',  
3    lastName: 'Зайцев'  
4  }  
5  
6  let oleg = {  
7    firstName: 'Олег',  
8    lastName: 'Зайцев'  
9  }  
10  
11 // Всё супер!  
12 console.log(ivan.firstName, oleg.firstName); // Иван, Олег
```

ОБХОД СВОЙСТВ

Один из способов обхода всех свойств объекта — использование конструкции `for in`:

```
1  let ivan = {  
2    firstName: 'Иван',  
3    lastName: 'Зайцев'  
4  }  
5  
6  for (let prop in ivan) {  
7    let value = ivan[ prop ];  
8    console.log(`Свойство ${prop}, значение: ${value}`);  
9  }
```

ОБХОД СВОЙСТВ

Результат работы:

```
└─ Свойство firstName, значение: Иван
└─ Свойство lastName, значение: Зайцев
```

ВОЗВРАТ ОБЪЕКТОВ В ФУНКЦИЯХ

__Внимание!__ Эта информация будет полезна вам при решении ДЗ!

Напишем функцию, которая выведет информацию о сотруднике компании.

```
1 function getProfile(firstName, lastName, birthYear, jobYear) {
2   // Текущий год
3   let year = (new Date).getFullYear();
4   return {
5     firstName,
6     lastName,
7     // Полное имя
8     name: firstName + ' ' + lastName,
9     birthYear,
10    // возраст (± 1 год)
11    age: year - birthYear,
12    jobYear,
13    // трудовой стаж
14    seniority: year - jobYear
15  };
16 }
17
18 console.log(getProfile('Максим', 'Иванов', 1980, 2000));
```

ВОЗВРАТ ОБЪЕКТОВ В ФУНКЦИЯХ

Результат:

```
{ firstName: 'Максим',  
  lastName: 'Иванов',  
  name: 'Максим Иванов',  
  birthYear: 1980,  
  age: 38,  
  jobYear: 2000,  
  seniority: 18 }
```


МЕТОДЫ

Если в свойстве объекта значением будет функция, такое свойство называется *методом*.

```
1  let person = {  
2    firstName: 'Иван',  
3    showName: function() {  
4      console.log(`Имя: ${person.firstName}`)  
5    }  
6  }  
7  
8  // вызов метода  
9  person.showName(); // Имя: Иван
```

ОТЛИЧИЕ СТРЕЛОЧНЫХ ФУНКЦИЙ ОТ ОБЫЧНЫХ (ДЕМО)

Давайте вернёмся к стрелочным функциям и рассмотрим их различие обычных функций.

```
1 let person = {  
2   name: "Иван",  
3   printName: function () {  
4     return this.name;  
5   },  
6   printNameArrow: () => this.name,  
7 }  
8 console.log(person.printName()); // "Иван"  
9 console.log(person.printNameArrow()); // ""  
10 // Исправим ситуацию  
11 person.printNameArrow = () => person.name;  
12 // Теперь выведет  
13 console.log(person.printNameArrow()); // "Иван"
```

Вывод: Стрелочные функции особенные: у них нет своего «собственного» `this`. Если мы используем `this` внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции. Более подробно контекст будет рассмотрен на следующем занятии.



ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Изучили основы создания функций и использованию их аргументов;
2. Познакомились с функциональными выражениями и узнали про отличия их объявлений;
3. Познакомились со стрелочными функциями и поняли их отличие от обычных;
4. Узнали про основы работы с объектами: задание, чтение и обход их свойств.



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

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



Спасибо за внимание!
Время задавать вопросы

ВЛАДИМИР ЧЕБУКИН

 vovachebr@mail.ru

 fb.me/vovachebr

 [@User123423](https://t.me/User123423)