

Функции



Алексей
Судничников



Алексей Судничников

Веб-разработчик



[Алексей Судничников](#)



План занятия

1. [Вспомним то, что уже знаем](#)
2. [Функциональные выражения](#)
3. [Стрелочные функции](#)
4. [Контекст выполнения](#)
5. [Функции-конструкторы](#)
6. [Функции высшего порядка](#)
7. [Анонимные и самовызывающиеся функции](#)



Вспомним то, что уже знаем

Что мы уже знаем о функциях?



К этому моменту мы **умеем объявлять функции** с параметрами и без, с возвращаемым значением и без.

- Приведите примеры **функций** с параметрами и без, с возвращаемым значением и без
- Сколько **переменных** может возвращать функция?
- Что мы обычно делаем с **возвращаемым значением**?
- Сколько раз может быть использован `return` внутри функции?
- Если у функции есть **аргумент**, но он не передан, какое значение у него будет?
- Как **задавать значения** аргументов по умолчанию?



Чистые функции

Чистые функции — строительные блоки в функциональном программировании. Их преимущества — простота и тестируемость:

- каждый раз функция возвращает **одинаковый результат**, когда она вызывается с тем же набором аргументов
- **нет побочных эффектов**

Чистые функции

Чистые функции — строительные блоки в функциональном программировании. Их преимущества — простота и тестируемость:

- каждый раз функция возвращает **одинаковый результат**, когда она вызывается с тем же набором аргументов
- **нет побочных эффектов**

Чистая функция:

```
const add = (x, y) => x + y  
add(2, 4) // 6
```

Примеры побочных эффектов

Примеры побочных эффектов:

- видоизменение входных параметров
- `console.log`
- HTTP-вызовы: AJAX, fetch
- изменение в файловой системе
- запросы DOM

```
let x = 2;  
const add = (y) => {  
  x += y;  
  return x;  
}  
console.log(add(4)); // 6  
console.log(add(4)); // 10
```


...rest параметры

Если мы не знаем, сколько будет аргументов, то на помощь приходит параметр **rest**:

```
function getArgs(...data) {  
  console.log(data);  
}  
  
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**:

```
function sum(...args) {  
  let total = 0;  
  for (let i = 0; i < args.length; i++) {  
    total += args[i];  
  }  
  return total;  
}  
  
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. При этом **`...rest` должен быть в конце списка аргументов**



...rest и остальные аргументы

Создадим функцию, которая создаёт тариф оплаты вместе со списком преимуществ:

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

...rest и остальные аргументы

Создадим функцию, которая создаёт тариф оплаты вместе со списком преимуществ:

Тариф «Базовый»

Преимущества:

- кровать на чердаке
- беседы с дядей Витей

Тариф «Оптимум»

Преимущества:

- кофе в постель без чашки
- отдельный 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

Функциональные выражения

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

Объявления функций vs функциональные выражения

Между **функциональными выражениями** и **объявлениями функций** есть одна принципиальная разница:

функциональные выражения

```
// Выдаст ошибку
console.log(takeFive());

let takeFive = function () {
  return 5;
}

console.log(takeFive());
```

объявления функций

```
console.log(takeFive()); // 5

function takeFive() {
  return 5;
}

console.log(takeFive());
```

Объявления функций vs функциональные выражения

Между **функциональными выражениями** и **объявлениями функций** есть одна принципиальная разница:

объявления функций

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

```
console.log(takeFive()); // 5

function takeFive() {
  return 5;
}

console.log(takeFive());
```



Стрелочные функции

Полный и краткий синтаксис

Проблема: обычные функции достаточно длинные, поэтому необходимо писать объявление функции и её тело.

В переменных `sum`, `sumArrow` и `sumArrowBlock` будут содержаться идентичные функции:

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

Скобки в стрелочных функциях

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

```
let multiply = a => a * 2;  
console.log(multiply(4)); // 8
```

аргумент один

Если аргументы отсутствуют или их больше одного, то скобки **обязательны**:

```
let multiply = () => 12 * 2;  
console.log(multiply()); //24
```

аргументы отсутствуют

```
let multiply = (a, b) => a * b;  
console.log(multiply(4, 3)); //12
```

два аргумента

Функции как методы объектов

Мы уже знакомы с **объектами** из курса рбј. Функции могут быть свойствами объектов. В этом случае они называются **методами**:

```
let human = {
  name: 'Alex',
  age: '27',
  sex: 'male',
  dojob: function () {
    console.log("I'am working");
  }
  eat() {
    console.log("I'am eating");
  }
  sayName: function () {
    console.log(`Имя: ${this.firstName}`);
  },
  sayNameArrow: () => console.log(this.name);
}
```

Чтобы говорить о **this** в стрелочных функциях, надо сначала поговорить о **this** вообще и контексте выполнения



Контекст выполнения

Глобальный объект window

Любая `var`-переменная или функция, определённая в глобальной области видимости, хранятся в рамках **глобального объекта window**:

```
var seven = 7;
function takeFive() {
    return 5;
}

console.log(seven); // 5
console.log(window.seven); // тоже 5

console.log(takeFive()); // 5
console.log(window.takeFive()); // тоже 5
```

Контекст. this

У любой функции есть ключевое слово **this**. Оно указывает на тот **объект**, к которому эта функция прикреплена.

В глобальной области видимости **this** указывает на **window**:

```
function getThis() {  
  console.log(this);  
}  
  
let person = {  
  getThis,  
}  
  
person.getThis(); // объект person  
window.getThis(); // объект window  
getThis(); // объект window
```

Другое понимание

this нужен в том случае, когда функция вызывается как **метод объекта**:

```
sayHi(); // просто вызов  
human.sayHi(); // вызов как метод объекта
```

Другое понимание

this нужен в том случае, когда функция вызывается как **метод объекта**:

```
sayHi(); // просто вызов  
human.sayHi(); // вызов как метод объекта
```

Это позволяет инкапсулировать всю логику в **объект**:

```
human.work(); // вызов как метод объекта  
human.sleep();  
human.growChildren();  
human.die();
```

Все эти функции внутри обращаются к свойствам **своего** объекта, используя *this*

Контекст. Строгий режим

В **строгом режиме** *this* для функций глобальной области видимости, вызванных без **window**, имеют значение *undefined*:

```
"use strict"
function getThis() {
  console.log(this);
}

let person = {
  getThis,
}

person.getThis(); // объект person
window.getThis(); // объект window
getThis(); // undefined
```

Как *this* позволяет избавиться от дублирования кода?

До *this*:

```
let ivan = {
  firstName: "Иван",
  showName() {
    console.log(`Имя:
${ivan.firstName}`);
  },
}

let oleg = {
  firstName: "Олег",
  showName() {
    console.log(`Имя:
${oleg.firstName}`);
  },
}

ivan.showName();
```

После *this*:



Как *this* позволяет избавиться от дублирования кода?

До *this*:

```
let ivan = {
  firstName: "Иван",
  showName() {
    console.log(`Имя:
${ivan.firstName}`);
  },
}

let oleg = {
  firstName: "Олег",
  showName() {
    console.log(`Имя:
${oleg.firstName}`);
  },
}

ivan.showName();
```

После *this*:

```
function showName() {
  console.log(`Имя:
${this.firstName}`);
}

let ivan = {
  firstName: "Иван",
  showName,
}

let oleg = {
  firstName: "Олег",
  showName,
}
```

This и стрелочные функции

Подробнее о `this` и **стрелочных функциях** будет в лекции «Объекты»





Функции-конструкторы

Контекст. Строгий режим

Давайте реализуем **CRM**. Для этого создаём **объект** клиента:

```
const person = {};  
person.name = 'Vasya';  
person.gender = 'M';
```

Переиспользовать такой код не получится, так как нарушается принцип **DRY**

— *логика создания и наполнения объекта будет повторяться.*

Решение: использовать **функцию-конструктор**

Конструктор объекта

Конструктор — специальный блок инструкций, вызываемый при создании объекта:

```
function Person(name, gender) {  
  this.name = name  
  this.gender = gender  
}
```

Оператор `new`

Оператор `new` позволяет создавать объекты **через вызов функций**.

Особенности работы функций, вызванных через оператор `new`:

- создаётся новый пустой **объект**
- ключевое слово `this` получает ссылку на этот объект
- **функция** выполняется
- возвращается `this` без явного указания

Создание новых объектов при помощи оператора new

Создание новых объектов может быть реализовано путем вызова обычной функции с оператором `new`. Подробнее рассказано в лекции 7

```
function Car(engine) {  
  this.engine = engine  
}  
  
const car = new Car("v8");
```



Функции высшего порядка

Функции высшего порядка

Функция высшего порядка — это функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата:

```
function Car(engine) {  
  this.engine = engine  
}  
  
const car = new Car("v8");
```

High ordered function или **HOF** — функция высшего порядка в переводе с английского

Принимаем функцию в качестве аргумента

Для начала разберём простой пример, в котором функция `execute` принимает на вход функцию `hello` и вызывает её:

```
function execute(func) {  
    func();  
}  
  
function hello() {  
    console.log("Привет, я функция!");  
}  
  
execute(hello);  
execute(function () {  
    console.log("Я функциональное выражение!");  
})
```


Функции передаются без скобок

Обратите внимание, что функцию `hello` мы передаём просто по имени:

```
function execute(func) {  
  func();  
}  
  
function hello() {  
  console.log("Привет, я функция!");  
}  
  
execute(hello());
```

Мы передали **результат выполнения** `hello`. Так как функция `hello` ничего не возвращает, то внутри функции `execute` переменная `func` будет указывать на `undefined`, а не на функцию `hello`, как ожидалось изначально

Возвращаем функцию

Когда мы говорили о **замыканиях**, у нас уже был пример функции, которая возвращает функцию:

```
let name = "Ann";
function generateFunction() {
  let name = "Mark"; // переменная в замыкании в момент создания
  const sayName = () => {
    console.log(name);
  }
  return sayName;
}

const newFunc = generateFunction();
newFunc();
```

Возвращаем функцию из функции

Разберём пару функций:

```
function example1() {  
  let innerExample1Func = () =>  
    "innerExample1Func_result";  
  return innerExample1Func();  
}  
  
function example2() {  
  let innerExample2Func = () =>  
    "innerExample2Func_result";  
  return innerExample2Func;  
}
```



Какие значения получим при вызове функций `example1` и `example2`?

Возвращаем функцию из функции

```
> function example1(){
  let innerExample1Func = () => "innerExample1Func_result";
  return innerExample1Func();
}

function example2(){
  let innerExample2Func = () => "innerExample2Func_result";
  return innerExample2Func;
}
< undefined
> example1()
< "innerExample1Func_result"
> example2()
< () => "innerExample2Func_result"
```

Возвращаем функцию из функции

Так как из функции возвращается функция, то вызывать её можно, просто дописав `()`:


```
> function example2(){  
  return () => "innerExample2Func_result";  
}  
let resultFunc = example2();  
< undefined  
  
> resultFunc();  
< "innerExample2Func_result"  
  
> // без переменной  
example2()();  
< "innerExample2Func_result"  
  
>
```

Возвращаем функцию из функции

В первом вызове результат функции `example2` сохраняется в переменную `resultFunc`. Затем `resultFunc` вызывается как функция.

Во втором вызове результат функции `example2` не сохраняется, а сразу вызывается как функция:

```
> function example2(){  
  return () => "innerExample2Func_result";  
}  
let resultFunc = example2();  
↵ undefined  
  
> resultFunc();  
↵ "innerExample2Func_result"  
  
> // без переменной  
  example2()();  
↵ "innerExample2Func_result"  
  
>
```



Анонимные и самовывзывающиеся функции

Возвращаем функцию из функции

Подробное объяснение замыканий будет в лекции «Обработка исключений и замыкания». Сейчас достаточно знать, что **функция имеет доступ к переменным**, доступным ей в момент создания:

```
function generator() {  
  let value = 'Переменная, всегда доступная функции valueUser';  
  function valueUser() {  
    console.log(value);  
  }  
  return valueUser;  
}  
const func = generator();  
func(); // Переменная, всегда доступная функции valueUser
```


Возвращаем функцию из функции

Создадим и вернём две функции, обращающиеся с переменной `count` в замыкании:

```
function counterGenerator() {  
  let count = 0; // переменная count существует только в замыкании.  
                  // Получить/изменить её иначе нельзя  
  function showCounter() {  
    console.log(count);  
  }  
  function increaseCounter() {  
    count += 1;  
  }  
  return [showCounter, increaseCounter];  
}  
const [showCounter, increaseCounter] = counterGenerator(); // деструктуризация  
increaseCounter();  
increaseCounter();  
showCounter(); // 2
```

Воспользуемся анонимной самовызывающейся функцией

```
const [showCounter, increaseCounter] = (function () {  
  let count = 0;  
  function showCounter() {  
    console.log(count);  
  }  
  function increaseCounter() {  
    count += 1;  
  }  
  return [showCounter, increaseCounter];  
})();
```

```
increaseCounter();  
showCounter(); // 1
```

Зачем?

Главной идеей является то, что **анонимная функция** вызывается **сразу после** своего **объявления**.

Вы получите преимущество от использования самовывзывающихся функций, если нужно будет выполнить код **один раз** и **сохранить** его результаты во внешней среде *без объявления глобальных переменных*



Имя функции

Функция — тоже **объект**. И поэтому у неё есть некоторые свойства и методы. Свойство `name` содержит строку названия функции:

```
function mult(a, b) {  
  return a * b;  
}  
  
console.log(mult.name); // mult
```



Итоги



Чему мы научились?

- **Вспомнили** чистые функции
- **Научились** пользоваться rest параметрами
- **Познакомились** со стрелочными функциями
- **Познакомились** с функциональными выражениями
- **Узнали** о контексте выполнения
- **Познакомились** с замыканиями
- **Научились** использовать функции-конструкторы для создания объектов
- **Освоили** анонимные и самовывзывающиеся функции

Домашнее задание

Давайте посмотрим ваше [домашнее задание](#):

- вопросы по домашней работе задавайте **в чате** мессенджера Slack
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как **приняты все задачи**

**Задавайте вопросы и
пишите отзыв о лекции!**

Алексей Судничников