

Функции

Давайте начнем с небольшого примера, например мы хотим написать алгоритм для вычисления стипендии студента. В качестве исходных данных у нас есть три переменных, названных по фамилии студента, которые содержат среднюю успеваемость за семестр, а сама стипендия будет рассчитываться по формуле. Допустим при успеваемости меньше 3.4, стипендия не начисляется, при успеваемости от 3.4 до 4 будем начислять 5000 рублей, а при успеваемости выше 4х будем начислять 7500 рублей:

```
// Данные об успеваемости трёх студентов.  
const ivanov = 4.5;  
const petrov = 3.7;  
const sidorov = 3.2;  
  
// Расчет стипендии.  
let scholarshipIvanov = 0;  
if (ivanov > 3.4 && ivanov < 4) {  
  scholarshipIvanov = 5000;  
} else if (ivanov > 4) {  
  scholarshipIvanov = 7500;  
}  
  
let scholarshipPetrov = 0;  
if (petrov > 3.4 && petrov < 4) {  
  scholarshipPetrov = 5000;  
} else if (petrov > 4) {  
  scholarshipPetrov = 7500;  
}  
  
let scholarshipSidorov = 0;  
if (sidorov > 3.4 && sidorov < 4) {  
  scholarshipSidorov = 5000;  
} else if (sidorov > 4) {  
  scholarshipSidorov = 7500;  
}  
  
console.log(scholarshipIvanov); // 7500  
console.log(scholarshipPetrov); // 5000  
console.log(scholarshipSidorov); // 0
```

Обратите внимание, что код для расчета стипендии нам пришлось написать три раза, и он получился практически одинаковый, все различие в нем только в именах переменных. При этом если у нас вдруг изменятся правила расчета, допустим минимальный проходной балл для начисления стипендии станет не 3.4, а 3.5, то нам придется поправить наш код в трёх местах,

что не удобно. Но мы можем организовать наш код для расчета стипендии в отдельную функцию, которую можно будет вызывать, и она будет возвращать нам результат расчёта, который можно будет использовать. Давайте для начала разберем что такое функции в JavaScript, как они создаются и потом перепишем наш пример.

Функции в JavaScript — это объекты, хранящие в себе кусочек программы, собранный в отдельный блок, который можно выполнять регулярно. Это позволяет выполнить небольшой кусочек кода несколько раз. Когда часть кода вынесена в функцию, мы можем вызвать её в разных местах программы и избежать дублирования.

Функции могут быть **именованными** и **безымянными (анонимными)**. Всё зависит от того, указываем мы имя функции при её объявлении или нет. Также функцию можно объявлять как **функциональное выражение (Function Expression)** или **декларативное объявление (Function Declaration)**. Первый вариант создаёт функцию в момент выполнения выражения, например присваивает новую функцию переменной, а второй компилирует её заранее, до компиляции остального кода, её использующего.

Функция в JS объявляется с помощью ключевого слова `function`. За ним следует название функции, которое мы придумываем сами. Затем в круглых скобках через запятую указываются параметры, которые данная функция принимает. По сути, параметры – это входные данные для функции, над которыми она будет выполнять какую-то работу. После указания параметров в фигурных скобках следует тело функции. После объявления функции, мы можем её вызвать и посмотреть, как она работает. Описание функции может находиться и до, и после её вызова.

```
function имя_функции(параметр1, параметр2, ...) {  
    // Действия  
}
```

При вызове функции в неё нужно передавать такое количество параметров, которое заявили при её создании. Их может быть 0 и более. Если параметры не переданы, то при вызове функции нужно просто указать пустые скобки.

Оператор `return` позволяет завершить выполнение функции, вернув конкретное значение. Если в функции не указано, что она возвращает, то, по сути, результатом её работы может являться только вывод какого-то текста на экран (см. предыдущую функцию). Однако, в большинстве случаев, мы хотим использовать результат работы функции в остальной программе. Тогда необходимо использовать оператор `return`. Например, напомним функцию, возвращающую среднее арифметическое двух чисел.

```
function average(x, y)
{
    return (x + y)/2;
}
avg = average(42, 100500);
alert(avg);
```

Таким образом, мы не только учим наш скрипт определённым навыкам, но и можем хранить результат выполнения каждой функции для дальнейшего использования.

Виды функций

```
// Декларативный подход к объявлению функций. Мы объявляем ключевое слово
function, за которым идёт имя функции (его может и не быть) и её тело.
// Именованная функция.
function getMaximum(numbers) {
    // Реализация алгоритма поиска максимального значения.
}

// Анонимная функция. В данном случае эта функция будет бесполезной, так как
без имени мы не сможем её вызвать. Анонимные функции встречаются в местах, где
они передаются в качестве функции обратного вызова или возвращаются из другой
функции. Об этом поговорим позже.
function (result) {
    // Какой-то код обработки результата.
}

// Функциональное выражение.
const getMaximum = function(numbers) {
    // Реализация алгоритма поиска максимального значения.
}
// или в стиле ES6 с использованием толстой стрелки (fat arrow)
const getMaximum = (numbers) => {
    // Реализация алгоритма поиска максимального значения.
}
```

Теперь, когда мы знаем как объявить функцию, перепишем наш пример для расчета стипендии:

```
// Функция для расчета стипендии.
function getScholarship(academicPerformance) {
    if (academicPerformance < 3.4) {
        return 0;
    }
    if (academicPerformance < 4) {
```

```

        return 5000;
    } else {
        return 7500;
    }
}

// Данные об успеваемости трёх студентов.
const ivanov = 4.5;
const petrov = 3.7;
const sidorov = 3.2;

// Расчет стипендии.
let scholarshipIvanov = getScholarship(ivanov);
let scholarshipPetrov = getScholarship(petrov);
let scholarshipSidorov = getScholarship(sidorov);

console.log(scholarshipIvanov); // 7500
console.log(scholarshipPetrov); // 5000
console.log(scholarshipSidorov); // 0

```

Обратите внимание насколько короче стал наш код, и теперь для исправления коэффициентов, нужно будет поменять код только в одном месте, непосредственно в функции.

Стрелочные функции

В стандарте ES2015 появилась возможность использовать стрелочные функции. До появления стандарта мы объявляли функции так:

```

var f = function(number) {
    return number + 1;
}

```

Теперь мы можем использовать такую запись:

```

let f = (number) => {
    return number + 1;
}

```

Если параметр всего один, как в нашем примере, скобки можно убрать:

```

let f = number => {
    return number + 1;
}

```

Однако если параметров нет совсем, пустые скобки ставить нужно:

```
let f = () => {
  doSomething;
}
```

Иногда функция только возвращает значение:

```
let f = number => {
  return number + 1;
}
```

В таком случае можно сократить запись, убрав фигурные скобки и команду **return**:

```
let f = number => number + 1;
```

Также стандарт ES2015 позволяет устанавливать значения по умолчанию для параметров функции:

```
const f = (param = 5) => {
  console.log(param);
}
f(); // 5
f(10); // 10
```

Что мы можем заметить, что стрелочные функции будут обладать отличной наглядностью и читабельностью

```
const sum1 = function(a, b) {
  return a + b;
}

// Код с использованием стрелочной функции
const f = (a,b) => a + b;
```

Область видимости

При работе с функциями в JS нужно также помнить о т.н. областях видимости. Они бывают глобальные и локальные. Глобальными называют переменные и функции, которые не находятся внутри какой-то функции.

В JS все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object). В браузере этот объект явно доступен под именем window. Объект window одновременно является глобальным объектом и содержит

ряд свойств и методов для работы с окном браузера, но нас здесь интересует только его роль как глобального объекта.

Локальные переменные доступны только внутри функции. Если на момент определения функции переменная существовала, то она будет существовать и внутри функции, откуда бы ее не вызывали.

```
function changeX(x){
  x += 5;
  alert(x);
}
var x = 1;
alert(x);    // выводит 1
changeX(x); // выводит 6
alert(x);    // выводит 1
```

Браузерные события

Как мы уже знаем для html элементов есть атрибут **onclick** с помощью которого мы можем вызвать javascript код, но как мы успели заметить, записать весь код внутри атрибута будет крайне неудобно

```
<button onclick="alert('Вы нажали на кнопку купить')">Купить</button>
```

Теперь представим задучу, что нам сначала нужно вывести сообщение “Поздравляем” а затем уже “Вы нажали на кнопку купить”, конечно мы можем разместить два alert внутри значения атрибута, но очевидно что кода может быть в десятки или тысячи раз больше, тут к нам на помощь приходят функции

```
function buy() {
  alert('Поздравляем!');
  alert('Вы нажали на кнопку купить');
}
```

Как же вызвать эту функцию при клике на кнопку “Купить”?

Для этого нам потребуется просто записать название функции внутри атрибута onclick=""

Собираем всё вместе и получаем код

```
<button onclick="buy()">Купить</button>
<script>
    function buy() {
        alert('Поздравляем!');
        alert('Вы нажали на кнопку купить');
    }
</script>
```

Реализации игры Загадки

Теперь давайте разберём, как быть в ситуации, если нам нужно внутри функции использовать еще одну функцию. Для этого предлагаю разобрать игру “Загадки” которая также будет вызываться при клике на кнопку на сайте

Первым делом необходимо создать код для одной загадки

```
let userAnswer = prompt('Не лает, не кусает, а в дом не пускает');
if (userAnswer === 'замок') {
    alert('Молодец, ответил верно');
} else {
    alert('Не угадал');
}
```

Что мы можем заметить, данный код будет дублироваться с каждой загадкой, нам нужно будет только менять текст самой загадки и ответ на нее. Именно для таких ситуаций и используются функции

```
function riddles(question, answer) { // где первый параметр это
    загадка, а второй ответ на нее
    let userAnswer = prompt(question); // внутри prompt передаём
    вопрос
```

```

    if (userAnswer === answer) { // внутри условия if вместо answer
будет ответ на загадку
        alert('Молодец, ответил верно');
    } else {
        alert('Не угадал');
    }
}

riddles('Не лает, не кусает, а в дом не пускает', 'замок');

```

На первый взгляд кода стало больше, но теперь для того чтобы загадать еще одну загадку, нам необходимо просто записать

```
riddles('Сто одежек и все без застежек', 'капуста');
```

и это уже будет вызов новой загадки.

С какой проблемой мы можем столкнуться, если пользователь введёт ответ на загадку с заглавной буквы, для этого мы можем использовать метод **toLowerCase()** который поможет привести ответ пользователя к нижнему регистру, не важно хоть он одну букву написал заглавную, хоть все сразу, его ответ преобразуется таким образом, что все буквы станут строчными. Тут снова замечен серьезный плюс функций, ведь мне необходимо поменять только тело функции, а если бы мы использовали дублирование кода и у нас было 50 загадок, нам бы пришлось исправить 50 элементов.

Осталось только собрать все элементы воедино и вызвать нашу игру при клике на кнопку

```

<button onclick="goRiddle()">Играть в игру Загадки</button>
<script>
    function goRiddle() {
        function riddles(question, answer) {
            let userAnswer = prompt(question);
            userAnswer = userAnswer.toLowerCase();
            if (userAnswer === answer) {
                alert('Молодец, ответил верно');
            } else {
                alert('Не угадал');
            }
        }
    }

```



```
    riddles('Не лает, не кусает, а в дом не пускает', 'замок');  
    riddles('Сто одежек и все без застёжек', 'капуста');  
  }  
</script>
```

В данном коде мы можем увидеть функцию внутри функции и это очень частое явление, не бойтесь создавать функции и вызывать их внутри функции, так ваш код станет еще более структурированным и понятным для другого программиста

Имена функций и аргументов

Правильные имена функций помогут вам и вашим коллегам понимать код. В имени функции основную роль всегда играет глагол: она ведь что-то делает. Это имя должно, как комментарий в коде, рассказывать, что произойдет после её выполнения. Чтобы было легче давать имя функции, она должна быть максимально короткой и выполнять строго одно действие. Если ваша функция пытается выполнить два действия и более, то её лучше разбить на отдельные функции. Очень часто в начале функции используются слова:

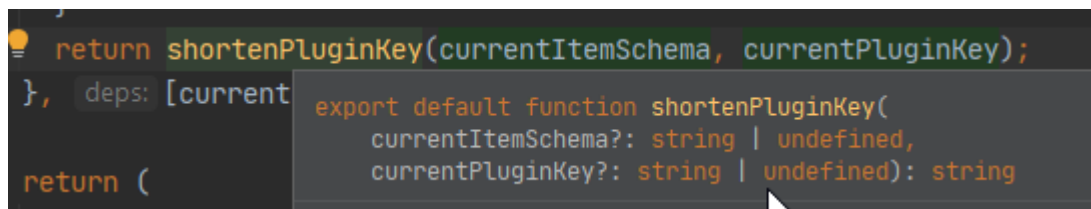
- `get (getUserName);`
- `calculate (calculateUserSalary);`
- `check (checkAccess).`

Посмотрим на примеры плохих имён и хороших:

```
// Плохие имена функций.  
const user = nameOfUser(); // Понятно, что функция что-то делает с именем  
пользователя. Можно догадаться, что получает его имя, но для полного понимания  
надо заходить в функцию.  
const hourlyRate = rateTable(user); // Тут непонятно, что делает функция.  
Может, она устанавливает какой-то рейтинг для пользователя или возвращает его  
рейтинг.  
const workedHours = tableOfHours(user); // То же самое, что и выше.  
const result = salary(hourlyRate, workedHours); // Ну и, как результат, мы  
получаем зарплату пользователя, а может быть, сравниваем внутри что-то.  
  
// Ещё один пример плохого имени, когда функция делает больше, чем положено.  
Const access = getUserByNameSaveDBAuth(user);  
  
// Хорошие имена функций — код читается как книга и всё понятно.  
const user = getUserFromDB();  
const hourlyRate = getUserHourlyRate(user);  
const workedHours = getUserWorkedHours(user);  
const result = calculateUserSalary(hourlyRate, workedHours);  
// Тут мы видим, что получаем пользователя из базы данных, потом получаем его
```

часовую ставку и количество отработанных этим пользователем часов и в итоге рассчитываем его зарплату.

С аргументами функции всё проще: используйте существительные, объясняющие, что хранится в аргументах. Правильное наименование аргументов очень полезно при использовании встроенных средств помощи в редакторах кода (IDE или среде разработки). Они дают возможность отобразить краткую справку по функции: какие аргументы она принимает и что в итоге вернёт. Правильно названные аргументы позволяют понять, что нужно передать в функцию для её работы. Вот так это выглядит:



```
return shortenPluginKey(currentItemSchema, currentPluginKey);
}, deps: [current
return (
export default function shortenPluginKey(
  currentItemSchema?: string | undefined,
  currentPluginKey?: string | undefined): string
```

Тут, благодаря TypeScript, также показаны типы аргументов.

Ещё одно замечание: старайтесь создавать функцию так, чтобы она принимала как можно меньше аргументов. В большом их количестве легко запутаться, можно указать аргументы не в том порядке. Это плохо сказывается на тестировании кода, так как функцию надо проверить на все комбинации аргументов (об этом на следующих уроках). Если очень нужно передать много аргументов, лучше объединить их в объект и передать одним аргументом:

```
const showProperties = (length, width, weight, color, material) => {
  console.log('Item length: ', length);
  console.log('Item width: ', width);
  console.log('Item weight: ', weight);
  console.log('Item color: ', color);
  console.log('Item material: ', material);
};

// Лучше все эти параметры передать одним аргументом.
const showProperties = (properties) => {
  if (!properties) return;
  console.log('Item length: ', properties.length);
  console.log('Item width: ', properties.width);
  console.log('Item weight: ', properties.weight);
  console.log('Item color: ', properties.color);
  console.log('Item material: ', properties.material);
};
```