# Занятие 5. Функции

## null, undefined и NaN

Рассмотрим еще несколько типов данных в JavaScript.

Как вы помните, мы можем объявлять переменные без значения:

let stuff;

console.log(stuff);

// undefined

undefined - это специальный тип в JavaScript, который означает "неопределенность".

Если мы хотим особым образом отразить, что в переменной пока что пусто, мы можем написать:

let stuff = null;

console.log(stuff);

// null

// ...

// где-то дальше по коду

stuff = 'Chips, fish and beer';

null - специальное значение в JavaScript, которое означает "ничего". В основном он используется программистами, чтобы показать, что переменная ничего не содержит. Это не ноль, не пустая строка, не пустой массив - просто ничто. Обычно используется, чтобы в дальнейшем назначить этой переменной значение типа "объект" или "массив".

Таким образом, мы ожидаем в дальнейшем, что переменная получит какое-то значение. Но пока этого не произошло, она равна null.

null и undefined можно использовать в логических выражениях. Иногда это необходимо для того, чтобы понять, получила ли переменная значение или нет. Одним из примеров, когда используется null - при вызове функции prompt.

const name = prompt('Please, enter your username');

Если мы нажмем Cancel, функция вернет null, и мы сможем проверить на этот конкретный случай:

if (name === null) {

alert('Cancel was pressed');

}

Также, если нам не важно, нажал ли пользователь отмену или нажал ОК при пустой строке, мы можем просто сравнить полученное значение с true или false, иначе говоря:

if (!name) {

alert("You haven't entered your name!");

}

Здесь мы воспользовались логическим оператором NOT, и указали переменную name. Посмотрим, что будет в результате данной операции:

* name = '';  
  Пустая строка является "ложным" значением. НЕ ложь = правда, поэтому !name будет равен true.
* name = null;  
  Null также является ложным значением. НЕ ложь = правда, поэтому !name будет равен true.
* name = 'John';  
  Любая непустая строка - это правдивое значение. НЕ правда = ложь, так что !name в этом случае будет равен false.

Из этих примеров можно понять, что alert выведется в двух случаях из трех - когда пользователь ничего не ввел, или нажал Cancel. Если же нужно проверять на все случаи по-очереди, мы воспользуемся конструкцией if-else..if:

const name = prompt('Please, enter your username');

if (name === null) {

alert('You pressed cancel');

} else if (name === '') {

alert('You haven't entered your name!');

} else {

alert('Your name is ' + name);

}

NaN - еще одно специальное значение, являющееся ложным. Означает "не число" (Not a number). Его можно получить в случае, например, когда функция parseInt не может найти целое число в переданных ей данных.

Особо отметить хотелось бы, что все три значения, хоть и являются "ложными", на самом деле не равны false. Заметьте, что мы используем "двойное равно", чтобы в JS попыталось произойти приведение типов, и даже в таком случае, мы не сможем получить равенство с false:

> NaN == false

< false

> undefined == false

< false

> null == false

< false

Естественно, с тройным равно тем более, работать это не будет.

С NaN ситуация еще сложнее - он не равен даже себе самому:

> NaN == NaN

< false

> NaN === NaN

< false

Поэтому, вы не можете сравнивать значения с NaN, и такая конструкция, например, работать не будет:

const number = parseInt(prompt('Enter a number:'));

if (number === NaN) {

alert('This is not a number!'); // Алерт никогда не сработает

}

Для проверки на NaN есть специальная функция: isNaN(number). Поэтому предыдущий пример, чтобы он сработал, перепишем так:

const number = parseInt(prompt('Enter a number:'));

if (isNaN(number)) {

alert('This is not a number!');

}

Как можно заметить, если переменная number будет равна NaN, то условие выполнится, и наоборот, не выполнится, если она будет числом. То есть:

* Функция isNaN(value)
  + возвращает true, если value является NaN
  + возвращает false, если value не является NaN

## Понятие функции

Функцией можно назвать некоторую последовательность действий, которую мы хотим повторять много раз в разных местах нашего алгоритма.

Например, представим себе алгоритм похода за покупками, когда у нас есть список покупок. Он может выглядеть как-то так:

* Для каждого товара в списке покупок:
  + Найти нужную категорию товаров в магазине
  + Найти нужный товар
  + Положить нужное количество товара в корзину
* Позвонить жене и спросить, нужно ли ей что-нибудь еще
* Если что-то нужно:
  + Дописать необходимые товары в список покупок
  + Для всех новых товаров в списке покупок:
    - Найти нужную категорию товаров в магазине
    - Найти нужный товар
    - Положить нужное количество товара в корзину
* Если нет:
  + Вздохнуть с облегчением
* Пройти на кассу
* Оплатить покупки

Здесь мы можем увидеть, что после того как мы прошлись по списку товаров, нам еще надо купить некоторые товары, которые потребовались после этого. И у нас появляется **дублирование** действий, которые нужно произвести для покупки одного товара из списка.

Дублирование действий приводит к негативным эффектам. Если нам захочется изменить или добавить действия в каком-то из этих блоков, нам нужно будет не забыть сделать это уже в двух местах. Представьте, что в данный блок кода нам надо добавить пункт:

* Проверить срок годности товара

Если вы добавите это действие только в первый блок, затем выполните алгоритм, то будете иметь бледный вид, когда узнаете, что любимые хлопья сына, о которых вам сообщила жена в последний момент, на самом деле просрочены.

Это очень важный момент - **человеческий фактор**. Чем больше дублирования в коде, тем выше шанс забыть изменить код в том или ином продублированном месте и получать неверные результаты в программе.

Что мы можем сделать в этом случае? Выделим блок кода в специальную **подпрограмму**:

* Определим подпрограмму "Купить товар"
  + Найти нужную категорию товаров в магазине
  + Найти нужный товар
  + Проверить срок годности товара
  + Положить нужное количество товара в корзину
* Для каждого товара в списке покупок:
  + Выполнить подпрограмму "Купить товар"
* Позвонить жене и спросить, нужно ли ей что-нибудь еще
* Если что-то нужно:
  + Дописать необходимые товары в список покупок
  + Для всех новых товаров в списке покупок:
    - Выполнить подпрограмму "Купить товар"
* Если нет:
  + Вздохнуть с облегчением
* Пройти на кассу
* Оплатить покупки

Итак, мы сначала определили для себя порядок действий, который нужно выполнить для покупки одного товара. Теперь всякий раз, когда в алгоритме будет встречаться строка "Выполнить подпрограмму "Купить товар", мы будем обращаться к этому алгоритму. Теперь, если нам понадобится добавить туда еще одно действие, например, "Выбрать самый дешевый товар из аналогичных", оно автоматически будет выполняться в двух местах основного алгоритма.

## Создание новой функции

Вы уже знакомы со многими встроенными функциями и методами в JavaScript. Вспомним и посмотрим на некоторые из них:

alert('Message');

Эта функция вызывает на экран окошко с информацией, которую "передаем" в функцию.

const name = prompt('Enter your name');

Эта функция вызывает окошко с информацией, которую "передаем" в функцию. После того, как пользователь что-то вводит, или не вводит, или вообще нажимает "Отмена", переменная name приобретает значение, "возвращаемое" функцией.

Мы не зря сейчас, и ранее, указывали данные термины в кавычках. Сейчас мы подробно разберем, что они означают.

Функция может **принимать** один или несколько **аргументов**, указываемых через запятую. Например, мы выводим несколько переменных с помощью console.log:

console.log('The number of the beast is', 666);

Вспомним, что мы говорили, что функция, принадлежащая какому-то объекту, называется **методом**. В данном случае это встроенный в браузер объект console, который позволяет взаимодействовать с консолью разработчика, в частности, выводить туда сообщения. Этот метод принимает сколько угодно аргументов, и выводит их все в консоль:

console.log('Let's count to three', 1, 2, 3);

// В этом примере четыре аргумента.

Функция **возвращает** значение, если результат, достигнутый в результате работы функции можно присвоить какой-то переменной или использовать далее в работе. Рассмотрим уже знакомый нам пример:

const age = prompt('Enter your age');

В данном случае отработает функция prompt, а возвращаемое ей значение будет присвоено переменной age.

Итак, создать собственную функцию можно одним из двуx способов. Рассмотрим их по очереди. Создадим для демонстрации простую функцию, которая будет выводить сообщение на экран. Настоящего толку от такой функции мало, только если мы постоянно вызываем это сообщение, и нам надоело постоянно писать один и тот же текст.

const sayHello = function() {

alert('Hello, world!');

};

sayHello();

Обратите внимание на синтаксис. Мы объявляем функцию как обычную переменную, за исключением того, что как ее значение мы создаем новую функцию с помощью ключевого слова function. Пустые скобки означают, что функция не принимает никаких аргументов. Затем начинаются привычные нам уже фигурные скобки, обозначающие начало и конец блока кода. Блок кода состоит из одной команды - вызова функции alert, но как вы уже поняли, можно написать в такой блок сколько угодно команд, условных конструкций, циклов и так далее.

Затем здесь, как и при объявлении любой другой переменной мы в конце (после значения) ставим точку с запятой.

**Вызов** функции заключается в указании ее названия, как названия переменной, и затем пустые круглые скобки обязательны, чтобы было понятно, что мы хотим вызвать ее.

Вызывать функцию в данном случае можно только после того, как она была определена.

Также можно объявлять функцию с помощью ключевого слова function без создания переменной:

sayHello();

function sayHello() {

alert('Hello, world!');

}

В этом случае, можно вызывать функцию в коде, до того, как она была определена. Дело в том, что при подобном объявлении функций, при выполнении кода, интерпретатор сначала "поднимает" их в самое начало, а затем начинает выполнять ваш код. Поэтому получается, что функция уже была объявлена. В остальном, данные объявления идентичны. Мы будем в основном пользоваться первым способом (через объявление переменной).

Посмотрим на пример функции, которая **принимает** аргументы и **возвращает** значение.

const getRandomNumberFromOneTo = function(limit) {

return Math.floor(Math.random() \* limit + 1);

};

console.log(getRandomNumberFromOneTo(20));

console.log(getRandomNumberFromOneTo(1000));

Функция для создания случайного числа от 1 до указанного предела.

Итак, для определения аргумента, мы придумываем его название (в данном случае, мы называем наш аргумент limit), записываем его в скобках, и он становится доступен в теле функции как **локальная** переменная, которая будет содержать переданное в функцию значение при ее вызове. Затем выполняется единственная строка в теле функции, которая **возвращает** значение с помощью ключевого слова return. Значение рассчитывается по известной формуле из прошлых занятий, с использованием переменной limit.

Ключевое слово return завершает работу функции. Никакой другой код после возврата не будет выполнен.

**Локальной** переменная limit называется потому, что после завершения тела функции она не будет более доступна никакому коду. Она существует только в пределах функции.

const number = getRandomNumberFromOneTo(100);

console.log(limit);

****

Посмотрим на пример функции с несколькими аргументами. Создадим функцию, которая позволит нам получать от пользователя число в заданных пределах:

const askForNumber = function(message, min, max) {

let number;

do {

number = parseInt(prompt(message));

} while (!number || number < min || number > max);

return number;

}

Функция принимает три аргумента - сообщение, которое будет отображаться пользователю (message), минимальный предел, в котором должно быть число (min) и максимальный предел, в котором должно быть число (max).

Затем, используя цикл do..while мы получаем у пользователя значение и не даем циклу завершиться, пока число не станет числом, будет больше или равно меньшему пределу и будет меньше или равно большему пределу.

Вызывать функцию мы будем привычным нам способом:

const grade = askForNumber('What is your grade?', 0, 100);

Аргументы в функцию передаются по очереди, через запятую, и именно в таком порядке они и будут назначены локальным переменным message, min и max внутри функции.

## Стрелочные функции

Есть еще один, относительно новый способ задания функции. Выглядит он так:

const sum = (a, b) => {

return a + b;

}

Как вы можете заметить, вместо слово function мы используем "стрелку", состоящую из двух символов - равно и знака "больше", аргументы функции располагаются перед стрелкой. Поэтому, функции объявленные таким образом, называются "стрелочными". Стрелочные функции имеют ряд преимуществ, которые возвышают их над "обычными" функциями. Рассмотрим часть из этих преимуществ.

Во-первых, если функция делает только одно действие и результат этого действия должен быть возвращен из нее, то мы можем обойтись одной строкой без фигурных скобок. Наша функция как раз может таковой стать:

const sum = (a, b) => a + b;

Итак, в этом случае мы убираем ключевое слово return и результат операции a + b будет автоматически возвращен из этой функции. Мы можем это попробовать:

sum(3, 4);

> 7

Если стрелочная функция принимает только один аргумент, то мы можем также избавиться и от круглых скобок для аргумента:

const circleArea = radius => Math.PI \* (radius \*\* 2);

Возможно неизвестный вам оператор \*\* - это возведение в степень, здесь мы должны число "пи" умножить на квадрат радиуса, чтобы вычислить площадь круга.

Итак, эта функция как вы можете заметить, принимает один аргумент, радиус круга, и возвращает площадь этого круга. Если аргумент один, то круглые скобки вокруг аргумента не обязательны. Но, если аргументов функции нет, то пустые круглые скобки должны быть:

const sayHello = () => console.log('Hello');

Также в примере выше вы увидели, что для двух, трех и более аргументов круглые скобки тоже должны присутствовать.

Стрелочные функции позволяют уменьшать количество кода, который вы пишете, и имеют еще ряд других преимуществ, с которыми мы познакомимся в дальнейших занятиях.

В данном занятии и домашнем задании мы будем пользоваться как обычными, так и стрелочными функциями, а в дальнейшем будем стараться пользоваться **только** стрелочными функциями.

## Порядок вызовов

Мы уже неоднократно пользовались конструкциями, похожими на эту:

const age = parseInt(prompt('What is your age?'));

Если рассматривать эту конструкцию слева направо, мы видим:

* Объявление переменной. Мы хотим ей присвоить некоторое значение справа от оператора присваивания.
* Однако, справа от оператора присваивания у нас вызов функции parseInt. Чтобы вернуть значение для оператора присваивания, необходимо ее выполнить. Эта функция принимает один аргумент.
* Однако, вместо аргумента у нас вызов другой функции (prompt), и чтобы получить значение для parseInt, необходимо выполнить еще и эту функцию и посмотреть, что получится.
* Вызов функции prompt у нас в конце "цепочки", но будет выполнен первым.

Итак, вызовы всегда будут происходить "из глубины" наружу, чтобы возвращаемые значения могли быть переданы в более наружные функции.

## Чистый код. Именование переменных и функций. Оформление.

Сейчас, когда мы узнали, что такое функции, и как их объявлять самостоятельно, мы можем вернуться немного к теме чистого кода.

Говоря о названиях переменных, практически все было сказано об этом на предыдущих занятиях. Однако, сделаем здесь важное сравнение:

* Переменные практически всегда - существительные
  + order
  + name
  + container
* Функции и методы практически всегда - глаголы.
  + getOrder
  + setName
  + calculate

Исключения составляют булевые переменные и функции, возвращающие булевые значения: они обычно сообщают о своем "состоянии":

* Переменные и функции:
  + isActive, isActive()
  + hasName, hasName()

Говоря об оформлении, мы уже долго работаем с различными конструкциями, также точно и вложенного типа: вложенные циклы, условия, теперь еще и функции.

Общее правило таково: Блоки кода, обозначенные фигурными скобками должны иметь некоторый отступ относительно "родительского кода". В некоторых редакторах используется один символ табуляции (кнопка Tab). В многих редакторах исходного кода при нажатии кнопки Tab вместо символа табуляции ставится 2 или 4 пробела, поскольку если открывать такой файл в разных редакторах, то он будет гарантированно выглядеть и открываться одинаково. Тогда как символ табуляции не гарантирует, что его ширина равна ровно 2 или 4 или 8 пробелов, так что рекомендуется вместо табуляции использовать пробелы.

В редакторе VS Code можно настроить по желанию сколько угодно пробелов. Однако, стандарты - это как правило 2 или 4 пробела. В данных раздатках использутся отступы в 4 пробела, но 2 может выглядеть лучше и быть удобнее в долгой перспективе.

## Переход к функциям. Рефакторинг.

Вы, возможно, задумывались, зачем нам нужны функции, если все можно просто писать в столбик, как вы успешно решали предыдущие домашние задания? Ответ на этот вопрос был частично раскрыт в первой части этого занятия, когда мы говорили о том, что вопрос с дублированием процесса покупки товара решается, когда мы определяем специальную "подпрограмму". Так же и здесь, когда в коде начинает наблюдаться достаточно много дублирования, необходимо выносить это дублирование в функции.

Рассмотрим такую программу, которая выводит на экран животных на ферме. При этом мы хотим, чтобы они отображались красиво, например так:

007 Cows

011 Chickens

Программа:

const cows = 7;

const chicken = 11;

let cowString = String(cows);

while (cowString.length < 3) {

cowString = "0" + cowString;

}

console.log(cowString + " Cows");

let chickenString = String(chickens);

while (chickenString.length < 3) {

chickenString = "0" + chickenString;

}

console.log(chickenString + " Chickens");

Неизвестная вам функция String(cows) - переводит число в строку, для того, чтобы мы могли потом узнать длину строки, которую занимает это число, и затем добавить нужное количество нулей в начало. Обратите внимание, что она начинается с заглавной буквы, в отличие от, например, parseInt.

Как мы могли бы улучшить такой код? Во-первых, если количество коров и цыплят изменится по ходу программы, нам придется заново писать этот код, чтобы их вывести. Вынесем в функцию:

const printFarmInventory = function(cows, chicken) {

let cowString = String(cows);

while (cowString.length < 3) {

cowString = "0" + cowString;

}

console.log(cowString + " Cows");

let chickenString = String(chickens);

while (chickenString.length < 3) {

chickenString = "0" + chickenString;

}

console.log(chickenString + " Chickens");

};

const currentCows = 7;

const currentChicken = 11;

printFarmInventory(currentCows, currentChicken);

Отлично, теперь, если нам надо будет в будущем показать еще раз количество животных на ферме, мы сможем просто вызвать эту функцию заново, передав в нее новое количество. Однако, посмотрим на нее - в ней два практически идентичных блока кода, который добавляет нули к числам cows и chicken. Когда вы видите подобное, это сигнал к тому, что необходимо производить **рефакторинг**.

Рефакторинг - это процесс организации кода, удаление дублирования, улучшение читаемости кода. Даже если наша функция работает полностью, как нам хочется, рефакторинг необходим для достижения этих целей.

Также, важно отметить, что функция как правило, должна делать одно дело, и делать это хорошо. Еще, для увеличения читаемости, часто принимается, что функция должна быть такой маленькой, как это только возможно. Однако не следует переусердствовать, если вы избавились от дублирования, но функции еще не очень маленькие, следует остановиться на этом этапе и продолжить писать программу. Если возникнет необходимость что-то переиспользовать, можно произвести повторный рефакторинг.

У рефакторинга однако существует один минус - вы можете что-то **сломать**, и не заметить этого. Для этого существует проверка в виде контрольных таблиц - вы знаете, что должна возвращать функция при каких входных данных. Проверьте ее, запустив с этими данными и проверьте результат. В дальнейшем, вы будете использовать автоматизированное тестирование, которое будет делать это за вас. Но на данном этапе, пока наши программы не такие сложные, и часто помещаются на один экран, вы можете проверить их без особых проблем и трудозатрат.

Итак, рефакторим нашу функцию:

const printFarmInventory = function(cows, chickens) {

console.log(zeroFill(cows, 3) + ' Cows');

console.log(zeroFill(chickens, 3) + ' Chickens');

};

const zeroFill = function(number, width) {

let string = String(number);

while (string.length < width) {

string = '0' + string;

}

return string;

};

printFarmInventory(currentCows, currentChicken);

У нас получилась функция zeroFill, с понятным названием (заполнить нулями), принимающая два аргумента - число, которое нужно заполнить, и длину, до которой нужно добавить нулей. Теперь наша функция printFarmInventory - простая, делающая то, что нам надо. Ее стало проще читать - понятно, что в ней выводится в консоль две строки, состоящие из числа животных и названия животного. Число при этом выводится с нулями вначале, до длины 3, если это необходимо.

Здесь необходимо отметить, что мы расположили функцию zeroFill сразу после функции printFarmInventory, поскольку она вызывается в ней, и так будет логично для читателя - сначала общие вещи, затем - частные.

Так мы, например, можем легко добавить в функцию новый тип животного, например:

const printFarmInventory = function(cows, chickens, pigs) {

console.log(zeroFill(cows, 3) + ' Cows');

console.log(zeroFill(chickens, 3) + ' Chickens');

console.log(zeroFill(pigs, 3) + ' Pigs');

};