**Комментарии**

Комментарии — это строки кода, которые JavaScript намеренно игнорирует. Комментарии — отличный способ оставлять заметки себе и другим людям, которым позже понадобится выяснить, что делает этот код.

Есть два способа написания комментариев в JavaScript:

Использование // заставит JavaScript игнорировать остальную часть текста в текущей строке. Это встроенный комментарий:

// This is an in-line comment.

Вы можете создать многострочный комментарий, начинающийся с /\* и заканчивающийся \*/. Это многострочный комментарий:

/\* This is a

multi-line comment \*/

**ПРИМЕЧАНИЕ.** При написании кода вам следует регулярно добавлять комментарии, чтобы разъяснить функции частей вашего кода. Хорошие комментарии могут помочь донести суть вашего кода как до других *, так и* до вас самих в будущем.

# Объявление переменных JavaScript

В информатике данные — это все, что имеет значение для компьютера. JavaScript предоставляет восемь различных типов данных: undefined, null, boolean, string, symbol, bigint, number и object.

Например, компьютеры различают числа, например число 12, и strings, например "12", "dog" или "123 cats", которые представляют собой коллекции символов. Компьютеры могут выполнять математические операции с числами, но не со строками.

Переменные позволяют компьютерам динамически хранить данные и манипулировать ими. Они делают это с помощью «метки». указывать на данные, а не использовать сами данные. Любой из восьми типов данных может храниться в переменной.

Переменные похожи на переменные x и y, которые вы используете в математике, а это означает, что это простое имя для представления данных, на которые мы хотим ссылаться. Компьютерные переменные отличаются от математических переменных тем, что они могут хранить разные значения в разное время.

Мы указываем JavaScript создать или объявить переменную, поставив перед ней ключевое слово var, вот так:

var ourName;

создает переменную с именем ourName. В JavaScript мы заканчиваем операторы точкой с запятой. Имена переменных могут состоять из цифр, букв и $ или \_, но не могут содержать пробелов или начинаться с цифры.

# Сохранение значений с помощью оператора присваивания

В JavaScript вы можете сохранить значение в переменной с помощью оператора присваивания (=).

myVariable = 5;

При этом значение Number 5 присваивается myVariable.

Если справа от оператора = есть какие-либо вычисления, они выполняются до того, как значение будет присвоено переменной слева от оператора.

var myVar;

myVar = 5;

Сначала этот код создает переменную с именем myVar. Затем код присваивает 5 myVar. Теперь, если myVar снова появится в коде, программа будет трактовать его так, как будто это 5.

# Присвоение значения одной переменной другой

После того как значение присвоено переменной с помощью оператора присваивания, вы можете присвоить значение этой переменной другой переменной с помощью оператора /span>.оператор присваивания

var myVar;

myVar = 5;

var myNum;

myNum = myVar;

Приведенный выше пример объявляет переменную myVar без значения, а затем присваивает ей значение 5. Далее объявляется переменная с именем myNum без значения. Затем содержимое myVar (то есть 5) присваивается переменной myNum. Теперь myNum также имеет значение 5.

# Инициализация переменных с помощью оператора присваивания

Обычно инициализирует переменную начальным значением в той же строке, в которой она объявлена.

var myVar = 0;

Создает новую переменную с именем myVar и присваивает ей начальное значение 0.

# Объявление строковых переменных

Ранее для объявления переменной вы использовали следующий код:

var myName;

Но вы также можете объявить строковую переменную следующим образом:

var myName = "your name";

"your name" называется строкой литералом. Строковый литерал или строка – это серия из нуля или более символов, заключенная в одинарные или двойные кавычки.

# Понимание неинициализированных переменных

Когда объявлены переменные JavaScript, они имеют начальное значение undefined. Если вы выполните математическую операцию над переменной undefined, результатом будет NaN, что означает "Не число"undefined, вы получите строку undefined.

# Понимание учета регистра в переменных

В JavaScript все имена переменных и функций чувствительны к регистру. Это означает, что капитализация имеет значение.

MYVAR — это не то же самое, что MyVar и myvar. Возможно иметь несколько различных переменных с одинаковым именем, но в разном регистре. Для ясности настоятельно рекомендуется не использовать эту языковую функцию.

# Изучите различия между ключевыми словами var и let

Одна из самых больших проблем при объявлении переменных с помощью ключевого слова var заключается в том, что вы можете легко перезаписать объявления переменных:

var camper = "James";

var camper = "David";

console.log(camper);

В приведенном выше коде переменная camper изначально объявляется как James, а затем переопределяется на David . Затем консоль отображает строку David.

В небольшом приложении вы можете не столкнуться с проблемой такого типа. Но по мере того, как ваша кодовая база становится больше, вы можете случайно перезаписать переменную, чего не собирались делать. Поскольку такое поведение не приводит к ошибке, поиск и исправление ошибок становится сложнее.

Ключевое слово let было введено в ES6, крупном обновлении JavaScript, для решения этой потенциальной проблемы с ключевым словом var. О других функциях ES6 вы узнаете в последующих заданиях.

Если вы замените var на let в приведенном выше коде, это приведет к ошибке:

let camper = "James";

let camper = "David";

Ошибку можно увидеть в консоли браузера.

Поэтому, в отличие от var, при использовании let переменную с тем же именем можно объявить только один раз.

# Объявите переменную, доступную только для чтения, с ключевым словом const

Ключевое слово let — не единственный новый способ объявления переменных. В ES6 вы также можете объявлять переменные, используя ключевое слово const.

const обладает всеми замечательными функциями, которые есть у let, с дополнительным бонусом, заключающимся в том, что переменные, объявленные с помощью const, доступны только для чтения. Они представляют собой постоянное значение, а это означает, что после присвоения переменной const ее нельзя переназначить:

const FAV\_PET = "Cats";

FAV\_PET = "Dogs";

В консоли отобразится ошибка из-за переназначения значения FAV\_PET.

Вы всегда должны называть переменные, которые не хотите переназначать, используя ключевое слово const. Это помогает, когда вы случайно попытаетесь переназначить переменную, которая должна оставаться постоянной.

# Добавьте два числа с помощью JavaScript

Number— это тип данных в JavaScript, который представляет числовые данные.

Теперь давайте попробуем сложить два числа с помощью JavaScript.

JavaScript использует символ + в качестве оператора сложения при размещении между двумя числами.

**Пример:**

const myVar = 5 + 10;

myVar теперь имеет значение 15.

# Вычесть одно число из другого с помощью JavaScript

Мы также можем вычесть одно число из другого.

JavaScript использует символ - для вычитания.

**Пример**

const myVar = 12 - 6;

myVar будет иметь значение 6.

# Умножение двух чисел с помощью JavaScript

Мы также можем умножить одно число на другое.

JavaScript использует символ \* для умножения двух чисел.

**Пример**

const myVar = 13 \* 13;

myVar будет иметь значение 169.

# Разделите одно число на другое с помощью JavaScript

Мы также можем разделить одно число на другое.

JavaScript использует символ / для разделения.

**Пример**

const myVar = 16 / 2;

myVar теперь имеет значение 8.

# Увеличение числа с помощью JavaScript

Вы можете легко увеличить или добавить единицу к переменной с помощью оператора ++.

i++;

является эквивалентом

i = i + 1;

**Примечание:** Вся строка становится i++;, что устраняет необходимость в знаке равенства.

# Уменьшить число с помощью JavaScript

Вы можете легко уменьшить или уменьшить переменную на единицу с помощью оператора --.

i--;

является эквивалентом

i = i - 1;

**Примечание:** Вся строка становится i--;, что устраняет необходимость в знаке равенства.

# Создание десятичных чисел с помощью JavaScript

Мы также можем хранить десятичные числа в переменных. Десятичные числа иногда называют числами с плавающей запятой или числами с плавающей запятой.< /а>

**Примечание.** числа вычисляются с конечной точностью. Операции с плавающей запятой могут привести к результатам, отличным от желаемого. Если вы получили один из этих результатов, откройте тему на [форуме freeCodeCamp](https://forum.freecodecamp.org/).

# Умножение двух десятичных чисел с помощью JavaScript

В JavaScript вы также можете выполнять вычисления с десятичными числами, как и с целыми числами.

Давайте перемножим две десятичные дроби, чтобы получить их произведение.

# Поиск остатка в JavaScript

Оператор остаток % дает остаток от деления двух чисел.

**Пример**

5 % 2 = 1

5/2 = 2 остаток 1

2 \* 2 = 4

5 - 4 = 1

**Использование**  
В математике четность или нечетность числа можно проверить, проверив остаток от деления числа на 2. Четные числа имеют остаток 0, а нечетные числа — остаток 1.

17 % 2 = 1

48 % 2 = 0

**Примечание.** Оператор остатка иногда ошибочно называют оператором модуля. Он очень похож на модуль, но не работает должным образом с отрицательными числами.

# Сложное присвоение с дополненным сложением

В программировании часто используются присваивания для изменения содержимого переменной. Помните, что все, что находится справа от знака равенства, оценивается в первую очередь, поэтому мы можем сказать:

myVar = myVar + 5;

чтобы добавить 5 к myVar. Поскольку это очень распространенный шаблон, существуют операторы, которые выполняют как математическую операцию, так и присваивание за один шаг.

Одним из таких операторов является оператор +=.

let myVar = 1;

myVar += 5;

console.log(myVar);

6будет отображаться в консоли.

# Составное присваивание с дополненным вычитанием

Как и оператор +=, -= вычитает число из переменной.

myVar = myVar - 5;

вычтет 5 из myVar. Это можно переписать так:

myVar -= 5;

# Составное присваивание с расширенным умножением

Оператор \*= умножает переменную на число.

myVar = myVar \* 5;

умножит myVar на 5. Это можно переписать так:

myVar \*= 5;

# Составное присваивание с дополненным делением

Оператор /= делит переменную на другое число.

myVar = myVar / 5;

Будет делиться myVar на 5. Это можно переписать так:

myVar /= 5;

# Экранирование буквальных кавычек в строках

Когда вы определяете строку, вы должны начинать и заканчивать ее одинарной или двойной кавычкой. Что происходит, когда вам нужна буквальная кавычка: " или ' внутри строки?

В JavaScript вы можете избежать кавычки, не рассматривая ее как кавычку конца строки, поместив обратная косая черта (\) перед кавычкой.

const sampleStr = "Alan said, \"Peter is learning JavaScript\".";

Это сигнализирует JavaScript, что следующая кавычка не является концом строки, а должна появиться внутри строки. Итак, если бы вы напечатали это на консоли, вы бы получили:

Alan said, "Peter is learning JavaScript".

# Заключение строк в одинарные кавычки

Значения Строковые в JavaScript могут быть записаны с одинарными или двойными кавычками, при условии, что вы начинаете и заканчиваете кавычками одного и того же типа. В отличие от некоторых других языков программирования, в JavaScript одинарные и двойные кавычки работают одинаково.

const doubleQuoteStr = "This is a string";

const singleQuoteStr = 'This is also a string';

Причина, по которой вам может потребоваться использовать один тип кавычек вместо другого, заключается в том, что вы хотите использовать оба типа в строке. Это может произойти, если вы хотите сохранить разговор в строке и заключить его в кавычки. Другое применение — сохранение тега <a> с различными атрибутами в кавычках внутри строки.

const conversation = 'Finn exclaims to Jake, "Algebraic!"';

Однако это становится проблемой, если вам нужно использовать в нем самые внешние кавычки. Помните, что строка имеет одинаковые кавычки в начале и в конце. Но если та же самая кавычка находится где-то посередине, строка остановится раньше и выдаст ошибку.

const goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';

const badStr = 'Finn responds, "Let's go!"';

Здесь badStr выдаст ошибку.

В приведенном выше goodStr вы можете безопасно использовать обе кавычки, используя обратную косую черту \ в качестве escape-последовательности. персонаж.

**Примечание:** Обратная косая черта \ не следует путать с прямой косой чертой /. Они не делают то же самое.

**Escape-последовательности в строках**

Кавычки — не единственные символы, которые можно *экранировать* внутри строки. Escape-последовательности позволяют использовать символы, которые иначе нельзя было бы использовать в строке.

| **Код** | **Выход** |
| --- | --- |
| \' | одинарная кавычка |
| \" | двойная кавычка |
| \\ | обратная косая черта |
| \n | новая линия |
| \t | вкладка |
| \r | возврат каретки |
| \b | возврат назад |
| \f | подача формы |

*Обратите внимание, что саму обратную косую черту необходимо экранировать, чтобы она отображалась как обратная косая черта.*

# Объединение строк с помощью оператора Plus

В JavaScript, когда оператор + используется со значением String, это называется конкатенацией< /span> их вместе.объединив оператор. Вы можете создать новую строку из других строк,

**Пример**

'My name is Alan,' + ' I concatenate.'

**Примечание.** Следите за пробелами. Конкатенация не добавляет пробелов между объединенными строками, поэтому вам придется добавлять их самостоятельно.

Пример:

const ourStr = "I come first. " + "I come second.";

Строка I come first. I come second. будет отображена в консоли.

# Объединение строк с помощью оператора Plus Equals

Мы также можем использовать оператор += для объединения строки в конец существующей строковой переменной. . Это может быть очень полезно, если разбить длинную строку на несколько строк.

**Примечание.** Следите за пробелами. Конкатенация не добавляет пробелов между объединенными строками, поэтому вам придется добавлять их самостоятельно.

Пример:

let ourStr = "I come first. ";

ourStr += "I come second.";

ourStr теперь имеет значение строки I come first. I come second..

# Создание строк с переменными

Иногда вам нужно будет построить строку. С помощью оператора конкатенации (+) вы можете вставить одну или несколько переменных в создаваемую строку.

Пример:

const ourName = "freeCodeCamp";

const ourStr = "Hello, our name is " + ourName + ", how are you?";

ourStr будет иметь значение строки Hello, our name is freeCodeCamp, how are you?.

# Добавление переменных к строкам

Точно так же, как мы можем построить строку из нескольких строк из строковых литералов, мы также можем добавлять переменные в строку, используя плюс, равный (+=) оператор.

Пример:

const anAdjective = "awesome!";

let ourStr = "freeCodeCamp is ";

ourStr += anAdjective;

ourStr будет иметь значение freeCodeCamp is awesome!.

# Найти длину строки

Длину значения String можно узнать, написав .length после строковой переменной или строкового литерала.

console.log("Alan Peter".length);

Значение 10 будет отображаться в консоли. Обратите внимание, что пробел между словами "Алан" и «Питер»; тоже учитывается.

Например, если мы создали переменную const firstName = "Ada", мы могли бы узнать длину строки Ada с помощью . =3> свойство.firstName.length

# Используйте скобочные обозначения, чтобы найти первый символ в строке

Обозначение скобок – это способ получить символ по определенному индексу в строке.

Большинство современных языков программирования, таких как JavaScript, не начинают счет с 1, как это делают люди. Они начинаются с 0. Это называется индексированием с отсчетом от нуля.

Например, символ с индексом 0 в слове Charles — это C. Итак, если const firstName = "Charles", вы можете получить значение первой буквы строки, используя firstName[0].

Пример:

const firstName = "Charles";

const firstLetter = firstName[0];

firstLetter будет иметь значение строки C.

# Понимание неизменяемости строк

В JavaScript String значения неизменяемы, что означает, что их нельзя изменить после создания.< /а>

Например, следующий код выдаст ошибку, поскольку букву B в строке Bob нельзя заменить на букву < а я=3>:J

let myStr = "Bob";

myStr[0] = "J";

Обратите внимание, что это не означает, что myStr нельзя переназначить. Единственный способ изменить myStr — присвоить ему новое значение, например:

let myStr = "Bob";

myStr = "Job";

# Используйте скобочные обозначения, чтобы найти N-й символ в строке

Вы также можете использовать обозначение скобок, чтобы получить символ в других позициях строки.

Помните, что компьютеры начинают отсчет с 0, поэтому первый символ на самом деле является нулевым символом.

Пример:

const firstName = "Ada";

const secondLetterOfFirstName = firstName[1];

secondLetterOfFirstName будет иметь значение строки d.

# Используйте скобочные обозначения, чтобы найти последний символ в строке

Чтобы получить последнюю букву строки, вы можете вычесть единицу из длины строки.

Например, если const firstName = "Ada", вы можете получить значение последней буквы строки, используя firstName[firstName.length - 1].

Пример:

const firstName = "Ada";

const lastLetter = firstName[firstName.length - 1];

lastLetter будет иметь значение строки a.

# Используйте обозначение скобок, чтобы найти символ от N-го до последнего в строке

Вы можете использовать тот же принцип, который мы только что использовали для получения последнего символа в строке, чтобы получить N-й до последнего символа.

Например, вы можете получить значение предпоследней буквы строки const firstName = "Augusta", используя firstName[firstName.length - 3]

Пример:

const firstName = "Augusta";

const thirdToLastLetter = firstName[firstName.length - 3];

thirdToLastLetter будет иметь значение строки s.

# Заготовки слов

Вам предлагаются предложения с некоторыми пропущенными словами, такими как существительные, глаголы, прилагательные и наречия. Затем вы заполняете недостающие фрагменты словами по вашему выбору так, чтобы завершенное предложение имело смысл.

Рассмотрим это предложение:

It was really \_\_\_\_, and we \_\_\_\_ ourselves \_\_\_\_.

В этом предложении три недостающих части: прилагательное, глагол и наречие, и мы можем добавить слова по нашему выбору, чтобы дополнить его. Затем мы можем присвоить завершенное предложение переменной следующим образом:

const sentence = "It was really " + "hot" + ", and we " + "laughed" + " ourselves " + "silly" + ".";

# Сохраняйте несколько значений в одной переменной, используя массивы JavaScript

С помощью переменных JavaScript array мы можем хранить несколько фрагментов данных в одном месте.

Вы начинаете объявление массива с открывающей квадратной скобки, заканчиваете его закрывающей квадратной скобкой и ставите запятую между каждой записью, например:

const sandwich = ["peanut butter", "jelly", "bread"];

# Вложение одного массива в другой массив

Вы также можете вкладывать массивы в другие массивы, как показано ниже:

const teams = [["Bulls", 23], ["White Sox", 45]];

Это также называется многомерным массивом.

**Доступ к данным массива с помощью индексов**

Мы можем получить доступ к данным внутри массивов, используя *индексы*.

Индексы массива записываются в той же скобочной записи, что и строки, за исключением того, что вместо указания символа они указывают запись в массиве. Как и строки, массивы используют *индексацию с отсчетом от нуля*, поэтому первый элемент массива имеет индекс 0.< /а>

**Пример**

const array = [50, 60, 70];

console.log(array[0]);

const data = array[1];

Компонент console.log(array[0]) печатает 50, а data имеет значение 60.< /а>

# Измените данные массива с помощью индексов

В отличие от строк, элементы массивов изменяемы и могут быть свободно изменены, даже если массив был объявлен с помощью < а я=3>.const

**Пример**

const ourArray = [50, 40, 30];

ourArray[0] = 15;

ourArray теперь имеет значение [15, 40, 30].

**Примечание.** Между именем массива и квадратными скобками не должно быть пробелов, например array [0]. Хотя JavaScript способен правильно это обработать, это может сбить с толку других программистов, читающих ваш код.

# Доступ к многомерным массивам с помощью индексов

Один из способов представить многомерный массив — это массив массивов< /а><а я=4>. Когда вы используете скобки для доступа к массиву, первый набор скобок относится к записям самого внешнего массива (первого уровня), а каждая дополнительная пара скобок относится к следующему уровню записей внутри.

**Пример**

const arr = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

[[10, 11, 12], 13, 14]

];

const subarray = arr[3];

const nestedSubarray = arr[3][0];

const element = arr[3][0][1];

В этом примере subarray имеет значение [[10, 11, 12], 13, 14], nestedSubarray имеет значение [10, 11, 12], а element имеет значение 11 .

**Примечание:** Между именем массива и квадратными скобками не должно быть пробелов, например array [0][0] и даже этой array [0] [0] не разрешено. Хотя JavaScript способен правильно это обработать, это может сбить с толку других программистов, читающих ваш код.

# Манипулирование массивами с помощью метода push

Простой способ добавить данные в конец массива — использовать метод push().

Метод push() принимает один или несколько аргументов и добавляет их в конец массива в порядок, в котором они появляются. Он возвращает новую длину массива.

Примеры:

const arr1 = [1, 2, 3];

arr1.push(4, 5);

const arr2 = ["Stimpson", "J", "cat"];

arr2.push(["happy", "joy"]);

arr1 теперь имеет значение [1, 2, 3, 4, 5], а arr2 имеет значение ["Stimpson", "J", "cat", ["happy", "joy"]].

# Манипулирование массивами с помощью метода pop

Другой способ изменить данные в массиве — использовать функцию .pop().

.pop() используется для извлечения значения из конца массива. Мы можем сохранить это всплывающее значение, присвоив его переменной. Другими словами, .pop() удаляет последний элемент из массива и возвращает этот элемент.

Из массива можно извлечь любой тип записи — числа, строки и даже вложенные массивы.

const threeArr = [1, 4, 6];

const oneDown = threeArr.pop();

console.log(oneDown);

console.log(threeArr);

Первый console.log будет отображать значение 6, а второй — значение [1, 4].

# Манипулирование массивами с помощью метода сдвига

pop()всегда удаляет последний элемент массива. Что делать, если вы хотите удалить первый?

Именно здесь на помощь приходит .shift(). Он работает так же, как .pop(), за исключением того, что он удаляет первый элемент вместо последнего. а>

Пример:

const ourArray = ["Stimpson", "J", ["cat"]];

const removedFromOurArray = ourArray.shift();

removedFromOurArray будет иметь значение строки Stimpson, а ourArray будет иметь ["J", ["cat"]].

# Манипулирование массивами с помощью метода unshift

Вы можете не только shiftдобавлять элементы из начала массива, но и unshiftдобавлять элементы в начало массива, т. е. добавлять элементы перед массивом.

.unshift()работает точно так же .push(), но вместо добавления элемента в конец массива unshift()добавляет элемент в начало массива.

Пример:

const ourArray = ["Stimpson", "J", "cat"];

ourArray.shift();

ourArray.unshift("Happy");

После shift, ourArrayбудет иметь значение ["J", "cat"]. После unshift, ourArrayбудет иметь значение ["Happy", "J", "cat"].

1. [Алгоритмы JavaScript и структуры данных](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures)
2. [Базовый JavaScript](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/#basic-javascript)

**Список покупок**

Создайте список покупок в переменной myList. Список должен представлять собой многомерный массив, содержащий несколько подмассивов.

Первый элемент каждого подмассива должен содержать строку с названием элемента. Вторым элементом должно быть число, обозначающее количество, т.е.

["Chocolate Bar", 15]

В списке должно быть не менее 5 подмассивов.

const myList = [["Chocolate Bar", 15],["Milk", 5],["Tea", 2],["Bread", 15],["Water", 1]];

console.log(myList);

**Напишите многоразовый JavaScript с функциями**

В JavaScript мы можем разделить наш код на повторно используемые части, называемые *функциями* .

Вот пример функции:

function functionName() {

console.log("Hello World");

}

Вы можете вызвать *эту* функцию, используя ее имя, за которым следуют круглые скобки, например: functionName();Каждый раз, когда вызывается функция, она выводит сообщение Hello Worldна консоль разработчика. Весь код между фигурными скобками будет выполняться каждый раз при вызове функции.

1. Создайте функцию reusableFunction, которая выводит строку Hi Worldна консоль разработчика.
2. Вызовите функцию.

function reusableFunction() {

console.log("Hi World")

};

reusableFunction();

**Передача значений функциям с аргументами**

*Параметры* — это переменные, которые служат заполнителями для значений, которые должны вводиться в функцию при ее вызове. Когда функция определена, она обычно определяется вместе с одним или несколькими параметрами. Фактические значения, которые вводятся (или *«передаются»* ) в функцию при ее вызове, называются *аргументами* .

Вот функция с двумя параметрами param1и param2:

function testFun(param1, param2) {

console.log(param1, param2);

}

Тогда мы можем позвонить testFunтак: testFun("Hello", "World");. Мы передали два строковых аргумента Helloи World. Внутри функции param1будет равно строке Helloи param2будет равно строке World. Обратите внимание, что вы можете вызвать testFunеще раз с другими аргументами, и параметры примут значения новых аргументов.

1. Создайте функцию с именем functionWithArgs, которая принимает два аргумента и выводит их сумму на консоль разработчика.
2. Вызовите функцию с двумя числами в качестве аргументов.

function functionWithArgs(a,b){

  console.log(a+b);

}

functionWithArgs(8,16);

**Возврат значения из функции с помощью Return**

Мы можем передавать значения в функцию с *аргументами* . Вы можете использовать returnоператор для отправки значения обратно из функции.

**Пример**

function plusThree(num) {

return num + 3;

}

const answer = plusThree(5);

answerимеет значение 8.

plusThreeпринимает *аргумент* и возвращает значение , numравное num + 3.

Создайте функцию timesFive, которая принимает один аргумент, умножает его на 5и возвращает новое значение.

function timesFive(a){

return a\*5

}

timesFive(6);

**Глобальная область действия и функции**

В JavaScript *область* видимости означает видимость переменных. Переменные, определенные вне функционального блока, имеют *глобальную* область действия. Это означает, что их можно увидеть повсюду в вашем коде JavaScript.

Переменные, объявленные без ключевых слов letили, constавтоматически создаются в globalобласти видимости. Это может привести к непредвиденным последствиям в другом месте вашего кода или при повторном запуске функции. Вы всегда должны объявлять свои переменные с помощью letили const.

Используя letили const, объявите глобальную переменную, названную myGlobalвне какой-либо функции. Инициализируйте его со значением 10.

Внутри функции fun1назначьте 5ее oopsGlobal ***без*** использования ключевых слов varили .letconst

// Declare the myGlobal variable below this line

const myGlobal = 10;

function fun1() {

  oopsGlobal = 5;

  // Assign 5 to oopsGlobal here

}

// Only change code above this line

function fun2() {

  let output = "";

  if (typeof myGlobal != "undefined") {

    output += "myGlobal: " + myGlobal;

  }

  if (typeof oopsGlobal != "undefined") {

    output += " oopsGlobal: " + oopsGlobal;

  }

  console.log(output);

}

**Локальная область действия и функции**

Переменные, объявленные внутри функции, а также параметры функции имеют *локальную* область видимости. Это означает, что они видны только внутри этой функции.

Вот функция myTestс локальной переменной под названием loc.

function myTest() {

const loc = "foo";

console.log(loc);

}

myTest();

console.log(loc);

Вызов функции myTest()отобразит строку fooв консоли. Строка console.log(loc)(вне myTestфункции) выдаст ошибку, поскольку locона не определена вне функции.

В редакторе есть два console.logs, которые помогут вам увидеть, что происходит. Проверяйте консоль во время написания кода, чтобы увидеть, как она меняется. myVarОбъявите внутри локальную переменную myLocalScopeи запустите тесты.

**Примечание.** На консоли по-прежнему будет отображаться ReferenceError: myVar is not defined, но это не приведет к сбою тестов.

function myLocalScope() {

  // Only change code below this line

const myVar = 10

  console.log('inside myLocalScope', myVar);

}

myLocalScope();

// Run and check the console

// myVar is not defined outside of myLocalScope

console.log('outside myLocalScope', myVar);

**Глобальная и локальная область действия в функциях**

Можно иметь как *локальные* , так и *глобальные* переменные с одинаковым именем. При этом локальная переменная имеет приоритет над глобальной переменной.

В этом примере:

const someVar = "Hat";

function myFun() {

const someVar = "Head";

return someVar;

}

Функция myFunвернет строку Head, поскольку присутствует локальная версия переменной.

Добавьте локальную переменную в myOutfitфункцию, чтобы переопределить значение outerWearстроки sweater.

// Setup

const outerWear = "T-Shirt";

function myOutfit() {

  // Only change code below this line

const outerWear = "sweater"

  // Only change code above this line

  return outerWear;

}

myOutfit();

**Понимание неопределенного значения, возвращаемого функцией**

Функция может включать этот returnоператор, но это не обязательно. В случае, если у функции нет returnоператора, при ее вызове функция обрабатывает внутренний код, но возвращаемое значение равно undefined.

**Пример**

let sum = 0;

function addSum(num) {

sum = sum + num;

}

addSum(3);

addSumэто функция без returnоператора. Функция изменит глобальную sumпеременную, но возвращаемое значение функции равно undefined.

Создайте функцию addFiveбез аргументов. Эта функция добавляет 5 к sumпеременной, но ее возвращаемое значение равно undefined.

// Setup

let sum = 0;

function addThree() {

  sum = sum + 3;

}

// Only change code below this line

function addFive(){

  sum = sum + 5;

}

// Only change code above this line

addThree();

addFive();

**Присвоение с возвращаемым значением**

Если вы помните из нашего обсуждения [сохранения значений с помощью оператора присваивания](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/storing-values-with-the-assignment-operator) , все, что находится справа от знака равенства, разрешается до присвоения значения. Это означает, что мы можем взять возвращаемое значение функции и присвоить его переменной.

Предположим, мы определили функцию sum, которая складывает два числа.

ourSum = sum(5, 12);

Вызов sumфункции с аргументами 5и 12возвращает возвращаемое значение 17. Это возвращаемое значение присваивается переменной ourSum.

Вызовите processArgфункцию с аргументом 7и присвойте возвращаемое значение переменной processed.

// Setup

let processed = 0;

function processArg(num) {

  return (num + 3) / 5;

}

// Only change code below this line

processed = processArg(7);

**Стоять в очереди**

В информатике *очередь* — это абстрактная *структура данных* , в которой элементы хранятся в порядке. Новые элементы могут быть добавлены в конец очереди, а старые элементы удалены из начала очереди.

Напишите функцию nextInLine, которая принимает массив ( arr) и число ( item) в качестве аргументов.

Добавьте число в конец массива, затем удалите первый элемент массива.

Затем функция nextInLineдолжна вернуть удаленный элемент.

function nextInLine(arr, item) {

  // Only change code below this line

  arr.push(item);

  return arr.shift(item);

  // Only change code above this line

}

// Setup

let testArr = [1, 2, 3, 4, 5];

// Display code

console.log("Before: " + JSON.stringify(testArr));

console.log(nextInLine(testArr, 6));

console.log("After: " + JSON.stringify(testArr));

**Понимание логических значений**

Другой тип данных — *Boolean* . Логические значения могут иметь только одно из двух значений: trueили false. По сути, это маленькие переключатели, где trueвключено и falseвыключено. Эти два состояния являются взаимоисключающими.

**Примечание.** Логические значения никогда не записываются в кавычках. Строки "true"и "false"не являются логическими значениями и не имеют особого значения в JavaScript.

Измените welcomeToBooleansфункцию так, чтобы она возвращала trueвместо false.

function welcomeToBooleans() {

  // Only change code below this line

  return true; // Change this line

  // Only change code above this line

}

**Используйте условную логику с операторами if**

ifоператоры используются для принятия решений в коде. Ключевое слово ifсообщает JavaScript, что код в фигурных скобках должен выполняться при определенных условиях, определенных в круглых скобках. Эти условия известны как Booleanусловия, и они могут быть только trueили false.

Когда условие оценивается как true, программа выполняет оператор внутри фигурных скобок. Если логическое условие имеет значение false, оператор внутри фигурных скобок не будет выполнен.

**Псевдокод**

if ( *условие истинно* ) {  
  *оператор выполняется*  
}

**Пример**

function test(myCondition) {

if (myCondition) {

return "It was true";

}

return "It was false";

}

test(true);

test(false);

test(true)возвращает строку It was trueи test(false)возвращает строку It was false.

Когда testвызывается со значением true, ifоператор оценивает myCondition, так оно trueили нет. Поскольку это так true, функция возвращает значение It was true. Когда мы вызываем testзначение false, myConditionэто *не так* true , и оператор в фигурных скобках не выполняется, и функция возвращает значение It was false.

Создайте ifоператор внутри функции, который будет возвращать значение Yes, that was trueпараметра, и возвращать его в противном случае.wasThatTruetrueNo, that was false

function trueOrFalse(wasThatTrue) {

  if (wasThatTrue){

    return "Yes, that was true"

  }

  return "No, that was false";

  // Only change code below this line

  // Only change code above this line

}

**Сравнение с оператором равенства**

В JavaScript существует множество *операторов сравнения* . Все эти операторы возвращают логическое значение trueили falseзначение.

Самый простой оператор — это оператор равенства ==. Оператор равенства сравнивает два значения и возвращает, trueэквивалентны они или falseнет. Обратите внимание, что равенство отличается от присваивания ( =), которое присваивает значение справа от оператора переменной слева.

function equalityTest(myVal) {

if (myVal == 10) {

return "Equal";

}

return "Not Equal";

}

Если myValравно 10, оператор равенства возвращает true, поэтому код в фигурных скобках будет выполнен, и функция вернет Equal. В противном случае функция вернет Not Equal. Чтобы JavaScript мог сравнить два разных *типа данных* (например, numbersи strings), он должен преобразовать один тип в другой. Это известно как приведение типов. Однако как только это произойдет, он сможет сравнивать термины следующим образом:

1 == 1 // true

1 == 2 // false

1 == '1' // true

"3" == 3 // true

Добавьте оператор равенства в указанную строку, чтобы функция возвращала строку Equal, valэквивалентную 12.

// Setup

function testEqual(val) {

  if (val == 12) { // Change this line

    return "Equal";

  }

  return "Not Equal";

}

testEqual(10);

**Сравнение с оператором строгого равенства**

Строгое равенство ( ===) является аналогом оператора равенства ( ==). Однако в отличие от оператора равенства, который пытается преобразовать оба сравниваемых значения в общий тип, оператор строгого равенства не выполняет преобразование типов.

Если сравниваемые значения имеют разные типы, они считаются неравными, и оператор строгого равенства вернет значение false.

**Примеры**

3 === 3 // true

3 === '3' // false

Во втором примере — 3это Numberтип и '3'есть Stringтип.

Используйте в инструкции оператор строгого равенства if, чтобы функция возвращала строку, Equalесли valона строго равна 7.

// Setup

function testStrict(val) {

  if (val === 7) { // Change this line

    return "Equal";

  }

  return "Not Equal";

}

testStrict(10);

**Практикуйтесь в сравнении разных значений**

В последних двух заданиях мы узнали об операторе равенства ( ==) и операторе строгого равенства ( ===). Давайте проведем краткий обзор и еще немного попрактикуемся в использовании этих операторов.

Если сравниваемые значения не относятся к одному и тому же типу, оператор равенства выполнит преобразование типов, а затем оценит значения. Однако оператор строгого равенства будет сравнивать и тип данных, и значение как есть, без преобразования одного типа в другой.

**Примеры**

3 == '3'возвращается true, потому что JavaScript выполняет преобразование типа строки в число. 3 === '3'возвращает false, поскольку типы разные и преобразование типов не выполняется.

**Примечание.** В JavaScript вы можете определить тип переменной или значения с помощью typeofоператора следующим образом:

typeof 3

typeof '3'

typeof 3возвращает строку numberи typeof '3'возвращает строку string.

Функция compareEqualityв редакторе сравнивает два значения с помощью оператора равенства. Измените функцию так, чтобы она возвращала строку Equalтолько в том случае, если значения строго равны.

// Setup

function compareEquality(a, b) {

  if (a === b) { // Change this line

    return "Equal";

  }

  return "Not Equal";

}

compareEquality(10, "10");

**Сравнение с оператором неравенства**

Оператор неравенства ( !=) является противоположностью оператора равенства. Это означает «не равно» и возвращает falseтуда, где возвращалось бы равенство true, и *наоборот* . Как и оператор равенства, оператор неравенства преобразует типы данных в значения при сравнении.

**Примеры**

1 != 2 // true

1 != "1" // false

1 != '1' // false

1 != true // false

0 != false // false

!=Добавьте в оператор оператор неравенства if, чтобы функция возвращала строку, Not Equalне valэквивалентную 99.

// Setup

function testNotEqual(val) {

  if (val != 99) { // Change this line

    return "Not Equal";

  }

  return "Equal";

}

testNotEqual(10);

**Сравнение с оператором строгого неравенства**

Оператор строгого неравенства ( !==) является логической противоположностью оператора строгого равенства. Это означает «Строго не равно» и возвращает результат falseтам, где вернулось бы строгое равенство, trueи *наоборот* . Оператор строгого неравенства не преобразует типы данных.

**Примеры**

3 !== 3 // false

3 !== '3' // true

4 !== 3 // true

Добавьте в оператор оператор строгого неравенства if, чтобы функция возвращала строку, Not Equalесли valона не строго равна17

// Setup

function testStrictNotEqual(val) {

  if (val !== 17) { // Change this line

    return "Not Equal";

  }

  return "Equal";

}

testStrictNotEqual(10);

**Сравнение с оператором «Больше чем»**

Оператор «больше чем» ( >) сравнивает значения двух чисел. Если число слева больше числа справа, возвращается true. В противном случае он возвращается false.

Как и оператор равенства, оператор «больше» преобразует типы данных в значения при сравнении.

**Примеры**

5 > 3 // true

7 > '3' // true

2 > 3 // false

'1' > 9 // false

Добавьте оператор «больше» в указанные строки, чтобы операторы возврата имели смысл.

function testGreaterThan(val) {

  if (val > 100) {  // Change this line

    return "Over 100";

  }

  if (val > 10) {  // Change this line

    return "Over 10";

  }

  return "10 or Under";

}

testGreaterThan(10);

**Сравнение с оператором «больше или равно»**

Оператор «больше или равно» ( >=) сравнивает значения двух чисел. Если число слева больше или равно числу справа, возвращается true. В противном случае он возвращается false.

Как и оператор равенства, оператор больше или равно преобразует типы данных при сравнении.

**Примеры**

6 >= 6 // true

7 >= '3' // true

2 >= 3 // false

'7' >= 9 // false

Добавьте оператор «больше или равно» в указанные строки, чтобы операторы возврата имели смысл.

function testGreaterOrEqual(val) {

  if (val >= 20) {  // Change this line

    return "20 or Over";

  }

  if (val >= 10) {  // Change this line

    return "10 or Over";

  }

  return "Less than 10";

}

testGreaterOrEqual(10);

**Сравнение с оператором «Меньше чем»**

Оператор «меньше» ( <) сравнивает значения двух чисел. Если число слева меньше числа справа, возвращается true. В противном случае он возвращается false. Как и оператор равенства, оператор «меньше» преобразует типы данных при сравнении.

**Примеры**

2 < 5 // true

'3' < 7 // true

5 < 5 // false

3 < 2 // false

'8' < 4 // false

Добавьте оператор «меньше» в указанные строки, чтобы операторы возврата имели смысл.

function testLessThan(val) {

  if (val < 25) {  // Change this line

    return "Under 25";

  }

  if (val < 55) {  // Change this line

    return "Under 55";

  }

  return "55 or Over";

}

testLessThan(10);

**Сравнение с оператором «меньше или равно»**

Оператор меньше или равно ( <=) сравнивает значения двух чисел. Если число слева меньше или равно числу справа, возвращается true. Если число слева больше числа справа, возвращается false. Как и оператор равенства, оператор меньше или равно преобразует типы данных.

**Примеры**

4 <= 5 // true

'7' <= 7 // true

5 <= 5 // true

3 <= 2 // false

'8' <= 4 // false

Добавьте оператор «меньше или равно» в указанные строки, чтобы операторы возврата имели смысл.

function testLessOrEqual(val) {

  if (val <= 12) {  // Change this line

    return "Smaller Than or Equal to 12";

  }

  if (val <= 24) {  // Change this line

    return "Smaller Than or Equal to 24";

  }

  return "More Than 24";

}

testLessOrEqual(10);

**Сравнение с логическим оператором И**

Иногда вам нужно будет протестировать несколько вещей одновременно. Логический оператор *и*&& ( ) возвращает значение trueтогда и только тогда, когда *операнды* слева и справа от него истинны.

Того же эффекта можно добиться, вложив один ifоператор в другой if.

if (num > 5) {

if (num < 10) {

return "Yes";

}

}

return "No";

Этот код вернет значение Yes, если numоно больше 5или меньше 10. Ту же логику можно записать с помощью *логического оператора и* .

if (num > 5 && num < 10) {

return "Yes";

}

return "No";

Замените два оператора if одним оператором, используя &&оператор, который вернет строку, Yesесли valменьше или равно 50и больше или равно 25. В противном случае вернет строку No.

**function** **testLogicalAnd**(val) {

*// Only change code below this line*

**if** (val <= 50 && val >= 25) {

**return** "Yes";

}

*// Only change code above this line*

**return** "No";

}

*// Change this value to test*

**testLogicalAnd**(10);

**Сравнение с логическим оператором ИЛИ**

Логический оператор *или* ( ||) возвращает значение true, если любой из *операндов* равен true. В противном случае он возвращается false.

Логический оператор *или* состоит из двух символов вертикальной черты: ( ||). Обычно его можно найти между клавишами Backspace и Enter.

Схема ниже должна выглядеть знакомой по предыдущим путевым точкам.

if (num > 10) {

return "No";

}

if (num < 5) {

return "No";

}

return "Yes";

Этот код вернется Yes, если numнаходится между 5и 10( 5и 10включено). Ту же логику можно записать с помощью *логического оператора или* .

if (num > 10 || num < 5) {

return "No";

}

return "Yes";

Объедините два ifоператора в один, который возвращает строку, Outsideесли valона не находится между 10и 20включительно. В противном случае верните строку Inside.

**if** (val < 10 || val > 20) {

**return** "Outside";

}

**Знакомство с операторами Else**

Если условие оператора ifистинно, выполняется следующий за ним блок кода. А как насчет того, когда это условие ложно? Обычно ничего не происходит. С помощью elseоператора может быть выполнен альтернативный блок кода.

if (num > 10) {

return "Bigger than 10";

} else {

return "10 or Less";

}

Объедините ifутверждения в одно if/elseутверждение.

**function** **testElse**(val) {

**var** result = "";

*// Only change code below this line*

**if** (val > 5) {

result = "Bigger than 5";

} **else** {

result = "5 or Smaller";

}

*// Only change code above this line*

**return** result;

}

*// Change this value to test*

**testElse**(4);

**Знакомство с операторами Else If**

Если у вас есть несколько условий, которые необходимо устранить, вы можете объединить ifоператоры вместе с else ifоператорами.

if (num > 15) {

return "Bigger than 15";

} else if (num < 5) {

return "Smaller than 5";

} else {

return "Between 5 and 15";

}

Преобразуйте логику в else ifоператоры использования.

**function** **testElseIf**(val) {

**if** (val > 10) {

**return** "Greater than 10";

} **else** **if** (val < 5) {

**return** "Smaller than 5";

} **else** {

**return** "Between 5 and 10";

}

}

*// Change this value to test*

**testElseIf**(7);

**Логический порядок в операторах if else**

Порядок важен в ifутверждениях else if.

Функция выполняется сверху вниз, поэтому будьте осторожны с тем, какой оператор идет первым.

Возьмем эти две функции в качестве примера.

Вот первое:

function foo(x) {

if (x < 1) {

return "Less than one";

} else if (x < 2) {

return "Less than two";

} else {

return "Greater than or equal to two";

}

}

А второй просто меняет порядок операторов:

function bar(x) {

if (x < 2) {

return "Less than two";

} else if (x < 1) {

return "Less than one";

} else {

return "Greater than or equal to two";

}

}

Хотя эти две функции выглядят почти одинаково, если мы передаем им число, мы получим разные выходные данные.

foo(0)

bar(0)

foo(0)вернет строку Less than oneи bar(0)вернет строку Less than two.

Измените порядок логики в функции, чтобы она во всех случаях возвращала правильные утверждения.

**function** **orderMyLogic**(val) {

**if** (val < 5) {

**return** "Less than 5";

} **else** **if** (val < 10) {

**return** "Less than 10";

} **else** {

**return** "Greater than or equal to 10";

}

# }Объединение операторов if else

if/elseутверждения могут быть объединены в цепочку для создания сложной логики. Вот *псевдокод* нескольких связанных операторов if/ else if:

if (condition1) {

statement1

} else if (condition2) {

statement2

} else if (condition3) {

statement3

. . .

} else {

statementN

}

Напишите связанные операторы if/ else ifдля выполнения следующих условий:

num < 5- вернуться Tiny  
num < 10- вернуться Small  
num < 15- вернуться Medium  
num < 20- вернуться Large  
num >= 20- вернутьсяHuge

**function** **testSize**(num) {

*// Only change code below this line*

**if** (num < 5) {

**return** "Tiny";

} **else** **if** (num < 10) {

**return** "Small";

} **else** **if** (num < 15) {

**return** "Medium";

} **else** **if** (num < 20) {

**return** "Large";

} **else** {

**return** "Huge";

}

*// Only change code above this line*

# }Кодекс гольфа

В игре в гольф каждая лунка имеет par, то есть среднее число, которое strokesигрок в гольф должен сделать, чтобы забить мяч в лунку и завершить игру. В зависимости от того, насколько выше или ниже parвы strokesнаходитесь, существует разное прозвище.

Ваша функция будет передана parи strokesаргументы. Верните правильную строку в соответствии с этой таблицей, в которой штрихи перечислены в порядке приоритета; сверху (самый высокий) вниз (самый низкий):

| **Штрихи** | **Возвращаться** |
| --- | --- |
| 1 | "Отверстие в одном!" |
| <= номинал - 2 | «Орел» |
| пар - 1 | "Птичка" |
| номинал | "Пар" |
| номинал + 1 | "Богги" |
| номинал + 2 | "Двойной призрак" |
| >= номинал + 3 | "Иди домой!" |

parи strokesвсегда будет числовым и положительным. Для вашего удобства мы добавили массив всех имен.

**onst** names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];

**function** **golfScore**(par, strokes) {

*// Only change code below this line*

**if** (strokes == 1) {

**return** names[0];

} **else** **if** (strokes <= par - 2) {

**return** names[1];

} **else** **if** (strokes === par - 1) {

**return** names[2];

} **else** **if** (strokes === par) {

**return** names[3];

} **else** **if** (strokes === par + 1) {

**return** names[4];

} **else** **if** (strokes === par + 2) {

**return** names[5];

} **else** {

**return** names[6];

}

*// Only change code above this line*

}

*// Change these values to test*

**golfScore**(5, 4);

(Using Multiple Conditional (Ternary) Operators)

const names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];

function golfScore(par, strokes) {

return strokes === 1

? names[0]

: strokes <= par - 2

? names[1]

: strokes === par - 1

? names[2]

: strokes === par

? names[3]

: strokes === par + 1

? names[4]

: strokes === par + 2

? names[5]

: names[6];

}

**Выбор из множества вариантов с помощью операторов Switch**

Если вам нужно сопоставить одно значение со многими параметрами, вы можете использовать оператор *переключения* . Оператор switchсравнивает значение с операторами *case* , которые определяют различные возможные значения. Любые допустимые операторы JavaScript могут выполняться внутри блока *case* и будут выполняться с первого совпадающего caseзначения до тех пор, пока breakне встретится a.

Вот пример заявления switch:

switch (fruit) {

case "apple":

console.log("The fruit is an apple");

break;

case "orange":

console.log("The fruit is an orange");

break;

}

caseзначения проверяются со строгим равенством ( ===). Приказывает breakJavaScript прекратить выполнение операторов. Если breakопущено, будет выполнен следующий оператор.

Напишите оператор переключения, который проверяет valи устанавливает answerследующие условия:  
1- alpha  
2- beta  
3- gamma  
4-delta

**function** **caseInSwitch**(val) {

**let** answer = "";

*// Only change code below this line*

**switch** (val) {

**case** 1:

answer = "alpha";

**break**;

**case** 2:

answer = "beta";

**break**;

**case** 3:

answer = "gamma";

**break**;

**case** 4:

answer = "delta";

**break**;

}

*// Only change code above this line*

**return** answer;

}

*// Change this value to test*

**caseInSwitch**(1);

**function** **caseInSwitch**(val) {

**switch** (val) {

**case** 1:

**return** "alpha";

**break**;

**case** 2:

**return** "beta";

**break**;

**case** 3:

**return** "gamma";

**break**;

**case** 4:

**return** "delta";

**break**;

}

}

*// Change this value to test*

**caseInSwitch**(1);

**Добавление параметра по умолчанию в операторы Switch**

В switchоператоре вы не сможете указать все возможные значения как caseоператоры. Вместо этого вы можете добавить defaultоператор, который будет выполняться, если соответствующие caseоператоры не найдены. Думайте об этом как о последнем elseутверждении в if/elseцепочке.

Заявление defaultдолжно быть в последнюю очередь.

switch (num) {

case value1:

statement1;

break;

case value2:

statement2;

break;

...

default:

defaultStatement;

break;

}

Напишите оператор переключения, который будет устанавливать answerследующие условия:  
a- apple  
b- bird  
c- cat  
default-stuff

**function** **switchOfStuff**(val) {

**let** answer = "";

**switch** (val) {

**case** "a":

answer = "apple";

**break**;

**case** "b":

answer = "bird";

**break**;

**case** "c":

answer = "cat";

**break**;

default:

answer = "stuff";

}

**return** answer;

}

**Несколько одинаковых опций в операторах Switch**

Если breakоператор опущен в switchоператоре case, следующие caseоператоры выполняются до тех пор, пока breakне встретится a. Если у вас есть несколько входов с одним и тем же выходом, вы можете представить их в switchследующем операторе:

let result = "";

switch (val) {

case 1:

case 2:

case 3:

result = "1, 2, or 3";

break;

case 4:

result = "4 alone";

}

Случаи 1, 2 и 3 дадут одинаковый результат.

Напишите оператор переключения, который будет установлен answerдля следующих диапазонов:  
1-3- Low  
4-6- Mid  
7-9-High

**Примечание.** Вам понадобится caseзаявление для каждого числа в диапазоне.

**function** **sequentialSizes**(val) {

**var** answer = "";

*// Only change code below this line*

**switch** (val) {

**case** 1:

**case** 2:

**case** 3:

**return** "Low";

**break**;

**case** 4:

**case** 5:

**case** 6:

**return** "Mid";

**break**;

**case** 7:

**case** 8:

**case** 9:

**return** "High";

**break**;

}

*// Only change code above this line*

**return** answer;

}

*// Change this value to test*

**sequentialSizes**(1);

**function** **sequentialSizes**(val) {

**var** answer = "";

*// Only change code below this line*

**switch** (val) {

**case** 1:

**case** 2:

**case** 3:

answer = "Low";

**break**;

**case** 4:

**case** 5:

**case** 6:

answer = "Mid";

**break**;

**case** 7:

**case** 8:

**case** 9:

answer = "High";

}

*// Only change code above this line*

**return** answer;

}

*// Change this value to test*

**sequentialSizes**(1);

**Замена цепочек If Else на Switch**

Если у вас есть много вариантов выбора, switchнаписать оператор может быть проще, чем множество связанных операторов if/ else if. Следующее:

if (val === 1) {

answer = "a";

} else if (val === 2) {

answer = "b";

} else {

answer = "c";

}

можно заменить на:

switch (val) {

case 1:

answer = "a";

break;

case 2:

answer = "b";

break;

default:

answer = "c";

}

Измените связанные операторы if/ else ifна switchоператоры.

**function** **chainToSwitch**(val) {

**var** answer = "";

*// Only change code below this line*

**switch** (val) {

**case** "bob":

answer = "Marley";

**break**;

**case** 42:

answer = "The Answer";

**break**;

**case** 1:

answer = "There is no #1";

**break**;

**case** 99:

answer = "Missed me by this much!";

**break**;

**case** 7:

answer = "Ate Nine";

**break**;

}

*// Only change code above this line*

**return** answer;

}

*// Change this value to test*

**chainToSwitch**(7);

# Возврат логических значений из функций

Возможно, вы помните из [раздела «Сравнение с оператором равенства»](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/comparison-with-the-equality-operator) , что все операторы сравнения возвращают логическое значение trueили falseзначение.

Иногда люди используют if/elseоператор для сравнения, например:

function isEqual(a, b) {

if (a === b) {

return true;

} else {

return false;

}

}

Но есть лучший способ сделать это. Поскольку ===возвращает trueили false, мы можем вернуть результат сравнения:

function isEqual(a, b) {

return a === b;

}

Исправьте функцию isLessудаления if/elseоператоров.

**function** **isLess**(a, b) {

*// Fix this code*

**return** a <= b;

}

*// Change these values to test*

**isLess**(10, 15);

# Возврат раннего шаблона для функций

Когда returnоператор достигнут, выполнение текущей функции прекращается, и управление возвращается в вызывающую позицию.

**Пример**

function myFun() {

console.log("Hello");

return "World";

console.log("byebye")

}

myFun();

Вышеупомянутое отобразит строку Helloв консоли и вернет строку World. Строка byebyeникогда не будет отображаться в консоли, поскольку функция завершается на этом returnоператоре.

Измените функцию abTestтак, чтобы, если aили bменьше, 0функция немедленно завершала работу со значением undefined.

**Подсказка**  
Помните, что [undefinedэто ключевое слово](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/understanding-uninitialized-variables) , а не строка.

*// Setup*

**function** **abTest**(a, b) {

*// Only change code below this line*

**if** (a < 0 || b < 0) {

**return** **undefined**;

}

*// Only change code above this line*

**return** **Math**.**round**(**Math**.**pow**(**Math**.**sqrt**(a) + **Math**.**sqrt**(b), 2));

}

*// Change values below to test your code*

**abTest**(2, 2);

**Подсчет карт**

В игре казино «Блэкджек» игрок может определить, есть ли у него преимущество в следующей раздаче над казино, отслеживая относительное количество старших и младших карт, оставшихся в колоде. Это называется подсчетом карт.

Наличие большего количества старших карт в колоде идет на пользу игроку. Каждой карте присвоено значение в соответствии с таблицей ниже. Когда счет положительный, игроку следует сделать большую ставку. Когда счет равен нулю или отрицателен, игроку следует делать низкую ставку.

| **Подсчитать изменение** | **Карты** |
| --- | --- |
| +1 | 2, 3, 4, 5, 6 |
| 0 | 7, 8, 9 |
| -1 | 10, «J», «Q», «К», «А» |

Вы напишете функцию подсчета карт. Он получит cardпараметр, который может быть числом или строкой, и увеличит или уменьшит глобальную countпеременную в соответствии со значением карты (см. таблицу). Затем функция вернет строку с текущим счетчиком и строку Bet, если счетчик положительный, Holdнулевой или отрицательный. Текущий счет и решение игрока ( Betили Hold) должны быть разделены одним пробелом.

**Пример выходных данных:** -3 Hold или5 Bet

**Подсказка**  
НЕ ​​сбрасывайте countзначение в 0, если значение равно 7, 8 или 9.  
НЕ возвращайте массив.  
НЕ включайте в вывод кавычки (одинарные или двойные).

**let** count = 0;

**function** **cc**(card) {

*// Only change code below this line*

**switch** (card) {

**case** 2:

**case** 3:

**case** 4:

**case** 5:

**case** 6:

count++;

**break**;

**case** 10:

**case** "J":

**case** "Q":

**case** "K":

**case** "A":

count--;

**break**;

}

**if** (count > 0) {

**return** count + " Bet";

} **else** {

**return** count + " Hold";

}

*// Only change code above this line*

}

**cc**(2); **cc**(3); **cc**(7); **cc**('K'); **cc**('A');

Решение 2 (УТОЧНИТЬ)

**let** count = 0;

**function** **cc**(card) {

*// Only change code below this line*

**var** regex = /[JQKA]/;

**if** (card > 1 && card < 7) {

count++;

} **else** **if** (card === 10 || regex.**test**(card)) {

count--;

}

**if** (count > 0) **return** count + " Bet";

**return** count + " Hold";

*// Only change code above this line*

}

**Создание объектов JavaScript**

Возможно, вы слышали этот термин objectраньше.

Объекты аналогичны arrays, за исключением того, что вместо использования индексов для доступа и изменения их данных вы получаете доступ к данным в объектах через так называемые properties.

Объекты полезны для структурированного хранения данных и могут представлять собой объекты реального мира, например кошку.

Вот пример объекта кошки:

const cat = {

"name": "Whiskers",

"legs": 4,

"tails": 1,

"enemies": ["Water", "Dogs"]

};

В этом примере все свойства хранятся в виде строк, таких как name, legsи tails. Однако вы также можете использовать числа в качестве свойств. Вы даже можете опустить кавычки для строковых свойств, состоящих из одного слова, как показано ниже:

const anotherObject = {

make: "Ford",

5: "five",

"model": "focus"

};

Однако, если ваш объект имеет какие-либо нестроковые свойства, JavaScript автоматически преобразует их в строки.

Создайте объект, представляющий позывную собаку и myDogсодержащий свойства name(строку) legsи tails.friends

Вы можете установить для этих свойств объекта любые значения, которые хотите, при условии, что они являются nameстрокой, legsчислами tailsи friendsмассивом.

const myDog = {

  // Only change code below this line

name: "Bombom",

legs: 5,

tails: 2,

friends: [5,1]

  // Only change code above this line

};

**Доступ к свойствам объекта с помощью точечной записи**

Существует два способа доступа к свойствам объекта: запись через точку ( .) и запись в скобках ( []), аналогичную массиву.

Точечная нотация — это то, что вы используете, когда заранее знаете имя свойства, к которому пытаетесь получить доступ.

Вот пример использования точечной записи ( .) для чтения свойства объекта:

const myObj = {

prop1: "val1",

prop2: "val2"

};

const prop1val = myObj.prop1;

const prop2val = myObj.prop2;

prop1valбудет иметь значение строки val1и prop2valбудет иметь значение строки val2.

Прочитайте значения свойств, testObjиспользуя точечную запись. Установите переменную, hatValueравную свойству объекта, hatи установите переменную, shirtValueравную свойству объекта shirt.

// Setup

const testObj = {

  hat: "ballcap",

  shirt: "jersey",

  shoes: "cleats"

};

// Only change code below this line

const hatValue = testObj.hat;      // Change this line

const shirtValue = testObj.shirt;    // Change this line

**Доступ к свойствам объекта с помощью скобок**

Второй способ доступа к свойствам объекта — это обозначение скобок ( []). Если свойство объекта, к которому вы пытаетесь получить доступ, содержит пробел в имени, вам нужно будет использовать скобки.

Однако вы по-прежнему можете использовать скобочные обозначения свойств объекта без пробелов.

Вот пример использования скобочных обозначений для чтения свойства объекта:

const myObj = {

"Space Name": "Kirk",

"More Space": "Spock",

"NoSpace": "USS Enterprise"

};

myObj["Space Name"];

myObj['More Space'];

myObj["NoSpace"];

myObj["Space Name"]будет строка Kirk, myObj['More Space']будет строка Spockи myObj["NoSpace"]будет строка USS Enterprise.

Обратите внимание, что имена свойств с пробелами должны быть в кавычках (одинарных или двойных).

Прочтите значения свойств an entreeи the drinkзначений, testObjиспользуя скобочные обозначения, и присвойте их entreeValueи drinkValueсоответственно.

// Setup

const testObj = {

  "an entree": "hamburger",

  "my side": "veggies",

  "the drink": "water"

};

// Only change code below this line

const entreeValue = testObj["an entree"];   // Change this line

const drinkValue = testObj["the drink"];    // Change this line

**Доступ к свойствам объекта с помощью переменных**

Другое использование скобок для объектов — доступ к свойству, которое хранится как значение переменной. Это может быть очень полезно для перебора свойств объекта или при доступе к таблице поиска.

Вот пример использования переменной для доступа к свойству:

const dogs = {

Fido: "Mutt",

Hunter: "Doberman",

Snoopie: "Beagle"

};

const myDog = "Hunter";

const myBreed = dogs[myDog];

console.log(myBreed);

Строка Dobermanбудет отображаться в консоли.

Обратите внимание: мы *не* заключаем имя переменной в кавычки при доступе к свойству, поскольку мы используем значение *переменной* , а не *имя* .

Установите playerNumberпеременную на 16. Затем используйте переменную, чтобы найти имя игрока и присвоить его player.

// Setup

const testObj = {

  12: "Namath",

  16: "Montana",

  19: "Unitas"

};

// Only change code below this line

const playerNumber = 16;  // Change this line

const player = testObj[playerNumber];   // Change this line

**Обновление свойств объекта**

После создания объекта JavaScript вы можете обновить его свойства в любое время так же, как и любую другую переменную. Для обновления можно использовать обозначение через точку или скобку.

Например, давайте посмотрим ourDog:

const ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

Поскольку он особенно счастливый пес, давайте заменим его имя на строку Happy Camper. Вот как мы обновляем свойство name его объекта: ourDog.name = "Happy Camper";или ourDog["name"] = "Happy Camper";Теперь, когда мы оцениваем ourDog.name, вместо получения Camperмы получаем его новое имя Happy Camper.

Обновите myDogсвойство имени объекта. Давайте изменим ее имя с Coderна Happy Coder. Вы можете использовать обозначение через точку или скобку.

// Setup

const myDog = {

  "name": "Coder",

  "legs": 4,

  "tails": 1,

  "friends": ["freeCodeCamp Campers"]

};

myDog.name = "Happy Coder";

// Only change code below this line

**Добавьте новые свойства к объекту JavaScript**

Вы можете добавлять новые свойства к существующим объектам JavaScript так же, как если бы вы их изменяли.

Вот как мы добавим barkсвойство в ourDog:

ourDog.bark = "bow-wow";

или

ourDog["bark"] = "bow-wow";

Теперь, когда мы оценим ourDog.bark, мы получим его лай bow-wow.

Пример:

const ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

ourDog.bark = "bow-wow";

Добавьте barkсвойство myDogи установите его для звука собаки, например «гав». Вы можете использовать обозначение через точку или скобку.

const myDog = {

  "name": "Happy Coder",

  "legs": 4,

  "tails": 1,

  "friends": ["freeCodeCamp Campers"]

};

myDog.bark = "Gav";

console.log(myDog)

**Удаление свойств из объекта JavaScript**

Мы также можем удалять свойства объектов следующим образом:

delete ourDog.bark;

Пример:

const ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"],

"bark": "bow-wow"

};

delete ourDog.bark;

После последней строки, показанной выше, ourDogэто выглядит так:

{

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

}

Удалить tailsсвойство из myDog. Вы можете использовать обозначение через точку или скобку.

// Setup

const myDog = {

  "name": "Happy Coder",

  "legs": 4,

  "tails": 1,

  "friends": ["freeCodeCamp Campers"],

  "bark": "woof"

};

delete myDog.tails;

// Only change code below this line

console.log(myDog)

**Использование объектов для поиска**

Объекты можно рассматривать как хранилище ключей/значений, например словарь. Если у вас есть табличные данные, вы можете использовать объект для поиска значений, а не switchоператор или if/elseцепочку. Это наиболее полезно, когда вы знаете, что ваши входные данные ограничены определенным диапазоном.

Вот пример объекта статьи:

const article = {

"title": "How to create objects in JavaScript",

"link": "https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/",

"author": "Kaashan Hussain",

"language": "JavaScript",

"tags": "TECHNOLOGY",

"createdAt": "NOVEMBER 28, 2018"

};

const articleAuthor = article["author"];

const articleLink = article["link"];

const value = "title";

const valueLookup = article[value];

articleAuthorэто строка Kaashan Hussain, articleLinkэто строка https://www.freecodecamp.org/news/a-complete-guide-to-creating-objects-in-javascript-b0e2450655e8/и valueLookupэто строка How to create objects in JavaScript.

Преобразуйте оператор переключения в объект с именем lookup. Используйте его для поиска valи присвоения связанной строки переменной result.

// Setup

function phoneticLookup(val) {

  var result = "";

  // Only change code below this line

var lookup = {

"alpha": "Adams",

"bravo": "Boston",

"charlie": "Chicago",

"delta": "Denver",

"echo": "Easy",

"foxtrot": "Frank"

};

result = lookup[val];

  // Only change code above this line

 return result;

}

console.log(phoneticLookup("charlie"));

**Проверка объектов на свойства**

Чтобы проверить, существует ли свойство данного объекта или нет, вы можете использовать этот .hasOwnProperty()метод. someObject.hasOwnProperty(someProperty)возвращает trueили falseзависит от того, обнаружено ли свойство на объекте или нет.

**Пример**

function checkForProperty(object, property) {

return object.hasOwnProperty(property);

}

checkForProperty({ top: 'hat', bottom: 'pants' }, 'top'); // true

checkForProperty({ top: 'hat', bottom: 'pants' }, 'middle'); // false

Первый checkForPropertyвызов функции возвращает результат true, а второй возвращает результат false.

Измените функцию checkObj, чтобы проверить, содержит ли объект, переданный параметру функции, objконкретное свойство, переданное параметру функции checkProp. Если переданное свойство checkPropнайдено в obj, верните значение этого свойства. Если нет, вернитесь Not Found.

function checkObj(obj, checkProp) {

  // Only change code below this line

  if (obj.hasOwnProperty(checkProp) == true){

    return obj[checkProp];

  } else {

    return "Not Found";

  };

     // Only change code above this line

}

**Манипулирование сложными объектами**

Иногда вам может потребоваться хранить данные в гибкой *структуре данных* . Объект JavaScript — это один из способов обработки гибких данных. Они допускают произвольные комбинации *строк* , *чисел* , *логических значений* , *массивов* , *функций* и *объектов* .

Вот пример сложной структуры данных:

const ourMusic = [

{

"artist": "Daft Punk",

"title": "Homework",

"release\_year": 1997,

"formats": [

"CD",

"Cassette",

"LP"

],

"gold": true

}

];

Это массив, внутри которого находится один объект. Объект содержит различные метаданные *об* альбоме. Он также имеет вложенный formatsмассив. Если вы хотите добавить больше записей альбома, вы можете сделать это, добавив записи в массив верхнего уровня. Объекты содержат данные в свойстве, имеющем формат «ключ-значение». В приведенном выше примере — "artist": "Daft Punk"это свойство, имеющее ключ artistи значение Daft Punk.

**Примечание.** Вам нужно будет поставить запятую после каждого объекта в массиве, если только это не последний объект в массиве.

Добавьте в массив новый альбом myMusic. Добавьте artistи titleстроки, release\_yearчисло и formatsмассив строк.

const myMusic = [

  {

    "artist": "Billy Joel",

    "title": "Piano Man",

    "release\_year": 1973,

    "formats": [

      "CD",

      "8T",

      "LP"

    ],

    "gold": true

  },

  {

    "artist": "Billy Joel",

    "title": "Piano Man",

    "release\_year": 1973,

    "formats": [

      "CD",

      "8T",

      "LP"

    ],

    "gold": true

  }

];

**Доступ к вложенным объектам**

Доступ к подсвойствам объектов можно получить, соединив вместе обозначения точки или скобки.

Вот вложенный объект:

const ourStorage = {

"desk": {

"drawer": "stapler"

},

"cabinet": {

"top drawer": {

"folder1": "a file",

"folder2": "secrets"

},

"bottom drawer": "soda"

}

};

ourStorage.cabinet["top drawer"].folder2;

ourStorage.desk.drawer;

ourStorage.cabinet["top drawer"].folder2будет строка secretsи ourStorage.desk.drawerбудет строка stapler.

Получите доступ к myStorageобъекту и присвойте содержимое свойства glove boxпеременной gloveBoxContents. По возможности используйте точечную запись для всех свойств, в противном случае используйте запись в скобках.

const myStorage = {

  "car": {

    "inside": {

      "glove box": "maps",

      "passenger seat": "crumbs"

     },

    "outside": {

      "trunk": "jack"

    }

  }

};

const gloveBoxContents = myStorage.car.inside["glove box"];

**Доступ к вложенным массивам**

Как мы видели в предыдущих примерах, объекты могут содержать как вложенные объекты, так и вложенные массивы. Подобно доступу к вложенным объектам, нотация скобок массива может быть объединена в цепочку для доступа к вложенным массивам.

Вот пример того, как получить доступ к вложенному массиву:

const ourPets = [

{

animalType: "cat",

names: [

"Meowzer",

"Fluffy",

"Kit-Cat"

]

},

{

animalType: "dog",

names: [

"Spot",

"Bowser",

"Frankie"

]

}

];

ourPets[0].names[1];

ourPets[1].names[0];

ourPets[0].names[1]будет строка Fluffyи ourPets[1].names[0]будет строка Spot.

Используя обозначение точки и скобок, задайте для переменной secondTreeвторой элемент массива listиз второго объекта в myPlantsмассиве.

const myPlants = [

  {

    type: "flowers",

    list: [

      "rose",

      "tulip",

      "dandelion"

    ]

  },

  {

    type: "trees",

    list: [

      "fir",

      "pine",

      "birch"

    ]

  }

];

const secondTree = myPlants[1].list[1];

**Коллекция пластинок**

Вы создаете функцию, которая помогает поддерживать коллекцию музыкальных альбомов. Коллекция организована как объект, содержащий несколько альбомов, которые также являются объектами. Каждый альбом представлен в коллекции с уникальным idименем свойства. Внутри каждого объекта альбома существуют различные свойства, описывающие информацию об альбоме. Не во всех альбомах есть полная информация.

Функция updateRecordsпринимает 4 аргумента, представленных следующими параметрами функции:

* records- объект, содержащий несколько отдельных альбомов
* id- число, обозначающее конкретный альбом в recordsобъекте
* prop— строка, представляющая имя свойства альбома, которое необходимо обновить.
* value— строка, содержащая информацию, используемую для обновления свойства альбома.

Завершите функцию, используя приведенные ниже правила, чтобы изменить объект, переданный в функцию.

* Ваша функция всегда должна возвращать весь recordsобъект.
* Если valueэто пустая строка, удалите данное propсвойство из альбома.
* Если propэто не так tracksи valueне является пустой строкой, присвойте valueэтому альбому prop.
* Если propэто tracksи valueне пустая строка, вам необходимо обновить массив альбома tracks. Во-первых, если у альбома нет tracksсвойства, присвойте ему пустой массив. Затем добавьте valueпоследний элемент в массив альбома tracks.

**Примечание.**recordCollection Для тестов используется копия объекта . Не следует напрямую изменять recordCollectionобъект.

// Setup

const recordCollection = {

  2548: {

    albumTitle: 'Slippery When Wet',

    artist: 'Bon Jovi',

    tracks: ['Let It Rock', 'You Give Love a Bad Name']

  },

  2468: {

    albumTitle: '1999',

    artist: 'Prince',

    tracks: ['1999', 'Little Red Corvette']

  },

  1245: {

    artist: 'Robert Palmer',

    tracks: []

  },

  5439: {

    albumTitle: 'ABBA Gold'

  }

};

// Only change code below this line

function updateRecords(records, id, prop, value) {

 if (value == "") {

   delete records[id][prop];

 } else if (prop !== "tracks" && value !== "") {

   records[id][prop] = value;

 } else if (prop == "tracks" && value !== "") {

if (records[id].hasOwnProperty("tracks") === false ){

  records[id][prop] = [];

}

records[id][prop].push(value);

 }

  return records;

 }

updateRecords(recordCollection, 5439, 'artist', 'ABBA');

**Итерация с помощью циклов while в JavaScript**

Вы можете запустить один и тот же код несколько раз, используя цикл.

Первый тип цикла, который мы изучим, называется циклом, whileпоскольку он выполняется, пока заданное условие истинно, и останавливается, когда это условие перестает быть истинным.

const ourArray = [];

let i = 0;

while (i < 5) {

ourArray.push(i);

i++;

}

В приведенном выше примере кода whileцикл выполнится 5 раз и добавит числа от 0 до 4 к ourArray.

Давайте попробуем заставить работать цикл while, помещая значения в массив.

Сложите числа от 5 до 0 (включительно) в порядке убывания, myArrayиспользуя whileцикл.

// Setup

const myArray = [];

let i = 5;

while (i >= 0) {

 myArray.push(i);

  i--;

}

// Only change code below this line

console.log(myArray)

**Итерация с помощью циклов For в JavaScript**

Вы можете запустить один и тот же код несколько раз, используя цикл.

Самый распространенный тип цикла JavaScript называется циклом, forпоскольку он выполняется определенное количество раз.

Циклы for объявляются с тремя необязательными выражениями, разделенными точкой с запятой:

for (a; b; c), где a— оператор инициализации, b— оператор условия и c— окончательное выражение.

Оператор инициализации выполняется один раз только перед началом цикла. Обычно он используется для определения и настройки переменной цикла.

Оператор условия оценивается в начале каждой итерации цикла и будет продолжаться до тех пор, пока его значение равно true. Когда условие находится falseв начале итерации, цикл прекратит выполнение. Это означает, что если условие начинается как ложное, ваш цикл никогда не выполнится.

Последнее выражение выполняется в конце каждой итерации цикла, перед следующей проверкой условия, и обычно используется для увеличения или уменьшения счетчика цикла.

В следующем примере мы инициализируем i = 0и выполняем итерацию, пока наше условие i < 5истинно. Мы будем увеличивать значение iна 1на каждой итерации цикла, используя в i++качестве окончательного выражения.

const ourArray = [];

for (let i = 0; i < 5; i++) {

ourArray.push(i);

}

ourArrayтеперь будет иметь значение [0, 1, 2, 3, 4].

Используйте forцикл, чтобы передать значения от 1 до 5 в myArray.

// Setup

const myArray = [];

for (let i = 1; i <= 5;i++) {

  myArray.push(i);

}

// Only change code below this line

**Перебор нечетных чисел с помощью цикла for**

Циклы for не должны выполняться по одному. Изменяя наш final-expression, мы можем считать четными числами.

Мы начнем с i = 0и зациклимся while i < 10. Мы увеличим iна 2 каждый цикл с помощью i += 2.

const ourArray = [];

for (let i = 0; i < 10; i += 2) {

ourArray.push(i);

}

ourArrayтеперь будет содержать [0, 2, 4, 6, 8]. Давайте изменим наш initialization, чтобы мы могли считать нечетными числами.

Нажмите нечетные числа от 1 до 9, чтобы myArrayиспользовать forцикл.

// Setup

const myArray = [];

for (let i = 1; i <= 9; i+=2) {

  myArray.push(i);

}

// Only change code below this line

**Счет в обратном направлении с помощью цикла for**

Цикл for также может вести обратный отсчет, если мы можем определить правильные условия.

Чтобы уменьшать на два на каждой итерации, нам нужно изменить инициализацию, условие и окончательное выражение.

Мы начнем с i = 10и зациклимся while i > 0. Мы будем уменьшать iна 2 каждый цикл с помощью i -= 2.

const ourArray = [];

for (let i = 10; i > 0; i -= 2) {

ourArray.push(i);

}

ourArrayтеперь будет содержать [10, 8, 6, 4, 2]. Давайте изменим нашу инициализацию и окончательное выражение, чтобы мы могли считать по два в обратном направлении, чтобы создать массив убывающих нечетных чисел.

Нажмите нечетные числа от 9 до 1, используя myArrayцикл for.

// Setup

const myArray = [];

for ( let i = 9; i > 0 ; i -= 2) {

  myArray.push(i);

}

// Only change code below this line

**Перебор массива с помощью цикла for**

Распространенной задачей в JavaScript является перебор содержимого массива. Один из способов сделать это — использовать forцикл. Этот код выведет каждый элемент массива arrна консоль:

const arr = [10, 9, 8, 7, 6];

for (let i = 0; i < arr.length; i++) {

console.log(arr[i]);

}

Помните, что индексация массивов начинается с нуля, что означает, что последний индекс массива равен length - 1. Нашим условием для этого цикла является i < arr.length, которое останавливает цикл, когда iравно length. В этом случае последняя итерация, т.е. i === 4когда iстановится равной arr.length - 1и выводится 6на консоль. Затем iувеличивается до 5, и цикл завершается, поскольку i < arr.lengthесть false.

Объявите и инициализируйте totalпеременную 0. Используйте forцикл для добавления значения каждого элемента массива myArrв total.

// Setup

const myArr = [2, 3, 4, 5, 6];

let total = 0;

for ( let i = 0; i < myArr.length; i++) {

  total += myArr[i];

}

// Only change code below this line

#### Объяснение кода

* Инициализация: iполучает значение 0и используется в качестве счетчика.
* Условие: последующий код выполняется до тех пор, пока iего длина меньше myArr(которая равна 5; пять чисел, но массивы начинаются с нуля).
* Итоговое выражение: iувеличивается на 1.
* Утверждение: функция добавляет myArr[i]значение до totalтех пор, пока условие не будет выполнено следующим образом:

total + myArr[0] -> 0 + 2 = 2

total + myArr[1] -> 2 + 3 = 5

total + myArr[2] -> 5 + 4 = 9

total + myArr[3] -> 9 + 5 = 14

total + myArr[4] -> 14 + 6 = 20

**Вложение циклов**

Если у вас есть многомерный массив, вы можете использовать ту же логику, что и для предыдущей путевой точки, для обхода как массива, так и любых его подмассивов. Вот пример:

const arr = [

[1, 2], [3, 4], [5, 6]

];

for (let i = 0; i < arr.length; i++) {

for (let j = 0; j < arr[i].length; j++) {

console.log(arr[i][j]);

}

}

Это выводит каждый подэлемент по arrодному. Обратите внимание, что во внутреннем цикле мы проверяем of .length, arr[i]поскольку arr[i]сам является массивом.

Измените функцию multiplyAllтак, чтобы она возвращала произведение всех чисел в подмассивах arr.

function multiplyAll(arr) {

  let product = 1;

  // Only change code below this line

for (let i = 0; i < arr.length; i++){

  for (let j = 0; j < arr[i].length; j++){

  product = product \* arr[i][j];

  }

}

  // Only change code above this line

  return product;

}

multiplyAll([[1, 2], [3, 4], [5, 6, 7]]);

* Мы проверяем длину arrв iцикле for и arr[i]длину в jцикле for.
* Мы умножаем productпеременную саму на себя, поскольку она равна 1, а затем умножаем ее на подмассивы.
* Два подмассива для умножения — это arr[i]и j.
* **Итерация с помощью JavaScript Do...While циклы**
* Следующий тип цикла, который вы изучите, называется циклом do...while. Это называется do...whileциклом, потому что сначала выполняется doодин проход кода внутри цикла, несмотря ни на что, а затем продолжается выполнение цикла, whileзначение которого оценивается указанным условием true.
* const ourArray = [];
* let i = 0;
* do {
* ourArray.push(i);
* i++;
* } while (i < 5);
* Приведенный выше пример ведет себя аналогично другим типам циклов, и результирующий массив будет иметь вид [0, 1, 2, 3, 4]. Однако отличие do...whileот других циклов заключается в том, как он ведет себя, когда условие не выполняется при первой проверке. Давайте посмотрим на это в действии. Вот обычный whileцикл, который будет выполнять код до тех пор, пока i < 5:
* const ourArray = [];
* let i = 5;
* while (i < 5) {
* ourArray.push(i);
* i++;
* }
* В этом примере мы инициализируем значение ourArrayпустого массива и значение i5. Когда мы выполняем whileцикл, условие оценивается как « falseпотому iчто не меньше 5», поэтому мы не выполняем код внутри цикла. В результате ourArrayк нему не будут добавлены никакие значения, и он по-прежнему будет выглядеть так, []как будто весь код в приведенном выше примере завершил работу. Теперь взгляните на do...whileцикл:
* const ourArray = [];
* let i = 5;
* do {
* ourArray.push(i);
* i++;
* } while (i < 5);
* В этом случае мы инициализируем значение i5, как и в случае с циклом while. Когда мы доходим до следующей строки, условие для оценки отсутствует, поэтому мы переходим к коду внутри фигурных скобок и выполняем его. Мы добавим один элемент в массив, а затем увеличим его, iпрежде чем перейдем к проверке условия. Когда мы, наконец, оцениваем условие i < 5в последней строке, мы видим, что iтеперь оно равно 6, что означает неудачу при проверке условия, поэтому мы выходим из цикла и все готово. В конце приведенного выше примера значение ourArrayравно [5]. По сути, do...whileцикл гарантирует, что код внутри цикла будет выполнен хотя бы один раз. Давайте попробуем заставить do...whileцикл работать, помещая значения в массив.
* Измените whileцикл в коде на do...whileцикл, чтобы цикл передавал только число 10в myArray, и iоно было равно тому, 11когда ваш код завершил выполнение.
* // Setup
* const myArray = [];
* let i = 10;
* // Only change code below this line
* do {
* myArray.push(i);
* i++;
* }
* while (i < 5);

Do...Whileциклы гарантируют, что код будет выполнен хотя бы один раз, и после выполнения, если условие внутри **true**while() , цикл продолжается, в противном случае он останавливается.

**Замените циклы с помощью рекурсии**

Рекурсия — это концепция, согласно которой функция может быть выражена через саму себя. Чтобы это понять, начните со следующей задачи: умножьте первые nэлементы массива, чтобы получить произведение этих элементов. Используя forцикл, вы можете сделать это:

function multiply(arr, n) {

let product = 1;

for (let i = 0; i < n; i++) {

product \*= arr[i];

}

return product;

}

Однако обратите внимание на это multiply(arr, n) == multiply(arr, n - 1) \* arr[n - 1]. Это означает, что вы можете переписать код multiplyсам по себе, и вам никогда не понадобится использовать цикл.

function multiply(arr, n) {

if (n <= 0) {

return 1;

} else {

return multiply(arr, n - 1) \* arr[n - 1];

}

}

Рекурсивная версия multiplyломается следующим образом. В *базовом случае* , где n <= 0, он возвращает 1. При больших значениях nон вызывает себя, но с n - 1. Этот вызов функции оценивается таким же образом, вызывая multiplyснова до тех пор, пока n <= 0. На этом этапе все функции могут вернуться, а оригинал multiplyвозвращает ответ.

**Примечание.** Рекурсивные функции должны иметь базовый случай, когда они возвращаются без повторного вызова функции (в этом примере, когда n <= 0), иначе они никогда не смогут завершить выполнение.

Напишите рекурсивную функцию, sum(arr, n)которая возвращает сумму первых nэлементов массива arr.

function sum(arr, n) {

  // Only change code below this line

if (n <= 0) {

  return 0;

} else {

  return sum(arr, n - 1) + arr[n-1];

}

  // Only change code above this line

}

**Поиск профиля**

У нас есть массив объектов, представляющих разных людей в наших списках контактов.

Для вас заранее написана функция, которая принимает свойство ( ) в качестве lookUpProfileаргументов name.prop

Функция должна проверить, nameявляется ли это фактическим контактом firstNameи заданное свойство ( prop) является свойством этого контакта.

Если оба значения верны, верните «значение» этого свойства.

Если nameне соответствует ни одному контакту, верните строку No such contact.

Если propне соответствует ни одному допустимому свойству найденного контакта, nameверните строку No such property.

Запустите тесты (Ctrl + Enter)Сбросить этот урок

// Setup

const contacts = [

  {

    firstName: "Akira",

    lastName: "Laine",

    number: "0543236543",

    likes: ["Pizza", "Coding", "Brownie Points"],

  },

  {

    firstName: "Harry",

    lastName: "Potter",

    number: "0994372684",

    likes: ["Hogwarts", "Magic", "Hagrid"],

  },

  {

    firstName: "Sherlock",

    lastName: "Holmes",

    number: "0487345643",

    likes: ["Intriguing Cases", "Violin"],

  },

  {

    firstName: "Kristian",

    lastName: "Vos",

    number: "unknown",

    likes: ["JavaScript", "Gaming", "Foxes"],

  },

];

function lookUpProfile(name, prop) {

  // Only change code below this line

  for (let x = 0; x < contacts.length; x++) {

    if (contacts[x].firstName === name) {

      if (contacts[x].hasOwnProperty(prop)) {

        return contacts[x][prop];

      } else {

        return "No such property";

      }

    }

  }

  return "No such contact";

}

  // Only change code above this line

lookUpProfile("Akira", "likes");

# Генерация случайных дробей с помощью JavaScript

Случайные числа полезны для создания случайного поведения.

В JavaScript есть Math.random()функция, которая генерирует случайное десятичное число между 0(включительно) и 1(исключительно). Таким образом Math.random(), можно вернуть a, 0но никогда не вернуть a 1.

**Примечание.** Как и [при хранении значений с помощью оператора присваивания](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/storing-values-with-the-assignment-operator) , все вызовы функций будут обработаны до их returnвыполнения, поэтому мы можем получить returnзначение функции Math.random().

Измените randomFraction, чтобы возвращать случайное число вместо возврата 0.

function randomFraction() {

  // Only change code below this line

  return Math.random();

  // Only change code above this line

}

**Генерация случайных целых чисел с помощью JavaScript**

Вы можете генерировать случайные десятичные числа с помощью Math.random(), но иногда вам нужно генерировать случайные целые числа. Следующий процесс даст вам случайное целое число меньше 20:

1. Используйте Math.random()для генерации случайного десятичного числа.
2. Умножьте это случайное десятичное число на 20.
3. Используйте Math.floor()для округления этого числа до ближайшего целого числа.

Помните, что это Math.random()никогда не сможет полностью вернуть 1, поэтому на самом деле получить его невозможно 20, поскольку вы округляете с помощью Math.floor(). Этот процесс даст вам случайное целое число в диапазоне от 0до 19.

Собрав все вместе, вот как выглядит ваш код:

Math.floor(Math.random() \* 20);

Вы вызываете Math.random(), умножаете результат на 20, а затем передаете значение, чтобы Math.floor()округлить его до ближайшего целого числа.

Используйте этот метод для генерации и возврата случайного целого числа в диапазоне от 0до 9.

function randomWholeNum() {

  return Math.floor(Math.random() \* 10);

}

console.log(randomWholeNum)

**Генерация случайных целых чисел в диапазоне**

Вы можете сгенерировать случайное целое число в диапазоне от нуля до заданного числа. Вы также можете выбрать другое меньшее число для этого диапазона.

Вы позвоните на свой минимальный номер minи максимальный номер max.

Эта формула дает случайное целое число в диапазоне от minдо max. Найдите минутку, чтобы прочитать его и попытаться понять, что делает этот код:

Math.floor(Math.random() \* (max - min + 1)) + min

Создайте функцию randomRange, которая принимает диапазон myMinи myMaxвозвращает случайное целое число, которое больше или равно myMinи меньше или равно myMax.

function randomRange(myMin, myMax) {

  return Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin;

}

#### Объяснение кода

* Math.random()генерирует наше случайное число от 0 до ≈ 0,9.
* Перед умножением он разрешает часть между круглыми скобками (myMax - myMin + 1)благодаря оператору группировки ( ).
* За результатом этого умножения следует сложение, myMinа затем «округление» до наибольшего целого числа, меньшего или равного ему (например: 9,9 приведет к 9).

Если бы значения были myMin = 1, myMax= 10, одним из результатов могло бы быть следующее:

1. Math.random() = 0.8244326990411024
2. (myMax - myMin + 1) = 10 - 1 + 1 -> 10
3. a \* b = 8.244326990411024
4. c + myMin = 9.244326990411024
5. Math.floor(9.244326990411024) = 9

**Используйте функцию parseInt**

Функция parseInt()анализирует строку и возвращает целое число. Вот пример:

const a = parseInt("007");

Вышеупомянутая функция преобразует строку 007в целое число 7. Если первый символ строки не может быть преобразован в число, возвращается NaN.

Используйте parseInt()в convertToIntegerфункции, чтобы она преобразовывала входную строку strв целое число и возвращала его.

function convertToInteger(str) {

return parseInt(str)

}

convertToInteger("56");

**Используйте функцию parseInt с основанием системы счисления**

Функция parseInt()анализирует строку и возвращает целое число. Он принимает второй аргумент для системы счисления, который определяет основание числа в строке. Система счисления может быть целым числом от 2 до 36.

Вызов функции выглядит так:

parseInt(string, radix);

И вот пример:

const a = parseInt("11", 2);

Переменная системы счисления указывает, что 11она находится в двоичной системе счисления или по основанию 2. В этом примере строка преобразуется 11в целое число 3.

Используйте parseInt()в convertToIntegerфункции, чтобы она преобразовывала двоичное число в целое число и возвращала его.

function convertToInteger(str) {

return parseInt(str,2)

}

convertToInteger("10011");

**Используйте условный (тройной) оператор**

Условный *оператор* , также называемый *тернарным оператором* , может использоваться как однострочное выражение if-else.

Синтаксис a ? b : c: где a— условие, b— код, запускаемый при возврате условия true, и c— код, запускаемый при возврате условия false.

Следующая функция использует if/elseоператор для проверки условия:

function findGreater(a, b) {

if(a > b) {

return "a is greater";

}

else {

return "b is greater or equal";

}

}

Это можно переписать с помощью условного оператора:

function findGreater(a, b) {

return a > b ? "a is greater" : "b is greater or equal";

}

Используйте условный оператор в checkEqualфункции, чтобы проверить, равны ли два числа. Функция должна возвращать либо строку Equal, либо строку Not Equal.

function checkEqual(a, b) {

return a == b ? "Equal" : "Not Equal"

}

checkEqual(1, 2);

**Используйте несколько условных (тройных) операторов**

В предыдущем задании вы использовали один условный оператор. Вы также можете объединить их вместе, чтобы проверить наличие нескольких условий.

Следующая функция использует операторы if, else ifи elseдля проверки нескольких условий:

function findGreaterOrEqual(a, b) {

if (a === b) {

return "a and b are equal";

}

else if (a > b) {

return "a is greater";

}

else {

return "b is greater";

}

}

Приведенную выше функцию можно переписать с использованием нескольких условных операторов:

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal"

: (a > b) ? "a is greater"

: "b is greater";

}

Рекомендуется форматировать несколько условных операторов таким образом, чтобы каждое условие располагалось на отдельной строке, как показано выше. Использование нескольких условных операторов без правильных отступов может затруднить чтение вашего кода. Например:

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";

}

В checkSignфункции используйте несколько условных операторов (в соответствии с рекомендуемым форматом findGreaterOrEqual), чтобы проверить, является ли число положительным, отрицательным или нулевым. Функция должна вернуть positive, negativeили zero.

function checkSign(num) {

return (num == 0) ? "zero" :

(num > 0) ? "positive" : "negative";

}

checkSign(10);

# Используйте рекурсию для создания обратного отсчета

В [предыдущем задании](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/replace-loops-using-recursion) вы узнали, как использовать рекурсию для замены forцикла. Теперь давайте посмотрим на более сложную функцию, которая возвращает массив последовательных целых чисел, начиная с 1числа, переданного в функцию.

Как упоминалось в предыдущей задаче, будет базовый вариант . Базовый случай сообщает рекурсивной функции, когда ей больше не нужно вызывать саму себя. Это простой случай, когда возвращаемое значение уже известно. Также будет рекурсивный вызов , который выполняет исходную функцию с разными аргументами. Если функция написана правильно, в конечном итоге будет достигнут базовый случай.

Например, предположим, что вы хотите написать рекурсивную функцию, которая возвращает массив, содержащий числа 1через n. Эта функция должна будет принять аргумент , nпредставляющий окончательное число. Затем ему нужно будет вызывать себя со все меньшими значениями, nпока не достигнет 1. Вы можете написать функцию следующим образом:

function countup(n) {

if (n < 1) {

return [];

} else {

const countArray = countup(n - 1);

countArray.push(n);

return countArray;

}

}

console.log(countup(5));

Значение [1, 2, 3, 4, 5]будет отображаться в консоли.

На первый взгляд это кажется нелогичным, поскольку значение n уменьшается , но значения в конечном массиве увеличиваются . Это происходит потому, что нажатие происходит последним, после возврата рекурсивного вызова. В тот момент, когда nон помещается в массив, countup(n - 1)он уже вычислен и возвращен [1, 2, ..., n - 1].

Мы определили функцию, вызываемую countdownс одним параметром ( n). Функция должна использовать рекурсию для возврата массива, содержащего целые числа, nна 1основе nпараметра. Если функция вызывается с номером меньше 1, функция должна вернуть пустой массив. Например, вызов этой функции n = 5должен вернуть массив [5, 4, 3, 2, 1]. Ваша функция должна использовать рекурсию, вызывая саму себя, и не должна использовать какие-либо циклы.

// Only change code below this line

function countdown(n){

  if (n < 1) {

    return []

  } else {

    const countArray = countdown(n - 1);

    countArray.unshift(n);

    return countArray;

  }

}

// Only change code above this line

**Используйте рекурсию для создания диапазона чисел**

Продолжая предыдущую задачу, мы предоставляем вам еще одну возможность создать рекурсивную функцию для решения проблемы.

Мы определили функцию rangeOfNumbersс двумя параметрами. Функция должна возвращать массив целых чисел, который начинается с числа, представленного параметром, startNumи заканчивается числом, представленным параметром endNum. Начальное число всегда будет меньше или равно конечному числу. Ваша функция должна использовать рекурсию, вызывая саму себя, а не использовать какие-либо циклы. Это также должно работать в случаях, когда оба startNumи endNumодинаковы.

function rangeOfNumbers(startNum, endNum) {

  if (startNum > endNum) {

    return []

  } else {

    const numbers = rangeOfNumbers(startNum, endNum - 1);

    numbers.push(endNum);

    return numbers;

  }

};