

# СТРОКИ. АЛГОРИТМЫ. КАК ОТЛАЖИВАТЬ КОД



КОНСТАНТИН ПОЛЯНСКИЙ



# КОНСТАНТИН ПОЛЯНСКИЙ



[kv.polyanskiy@gmail.com](mailto:kv.polyanskiy@gmail.com)



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



# ПЛАН ЗАНЯТИЯ

1. [Повторение – мать учения](#)
2. [Что такое строки для компьютера? Кодировки](#)
3. [Создание строк и спецсимволы](#)
4. [Методы для работы со строками](#)
5. [Алгоритмы](#)
6. [Работа с ошибками. Отладка](#)



# **ПОВТОРЕНИЕ – МАТЬ УЧЕНИЯ**

# ВСПОМНИМ ПРО ВЕТВЛЕНИЯ

Что мы получим в результате исполнения такого кода?

```
1  let hour = 19;  
2  let minute = 30;  
3  
4  if (18 < 0 || (hour === 19 && minute === 30)) {  
5      console.log('Время строк в JS!');  
6  } else {  
7      console.log('It\'s party time!');  
8  }
```

## ЧТО МЫ УВИДИМ В РЕЗУЛЬТАТЕ?

Мы увидим на экране окно с сообщением "Время строк в JS!", потому что условие верно. Оно состоит из двух частей:

- `18 < 0 → false`;
- `(hour === 19 && minute === 30) → true`, потому что используется логическое И и оба сравнения истинны.

Две части соединены оператором ИЛИ `||` и нам достаточно, чтобы хотя бы одна из частей выражения была верна.

## БОНУС: SWITCH-CASE

Когда у нас возникает необходимость выполнять много разных блоков кода в зависимости от значения одной и той же переменной — вместо большой цепочки `else-if` удобнее использовать конструкцию `switch-case`. Она позволяет повысить читаемость и поддерживаемость кода.

Давайте сначала посмотрим как это будет выглядеть с уже знакомым нам подходом:

```
1  let age = 18;
2
3  if (age === 0) {
4    console.log('Вы еще не родились');
5  } else if (age === 7) {
6    console.log('Первый класс!');
7  } else if (age === 18) {
8    console.log('Доступ разрешен');
9  } else {
10   console.log('Доступ запрещен');
11 }
```

# ДРУГОЙ ПОДХОД

А теперь тоже самое, но с другим синтаксисом:

```
1  let age = 18;
2
3  switch (age) {
4      case 0:
5          console.log('Вы еще не родились');
6          break;
7      case 7:
8          console.log('Первый класс!');
9          break;
10     case 18:
11         console.log('Доступ разрешен');
12         break;
13
14     default:
15         console.log('Доступ запрещен');
16         break;
17 }
```



---

# **ЧТО ТАКОЕ СТРОКИ ДЛЯ КОМПЬЮТЕРА? КОДИРОВКИ**



## А ТЕПЕРЬ К ОСНОВНОЙ ТЕМЕ!

В JavaScript любые текстовые данные являются строками. Строки же в свою очередь состоят из символов. Даже пробел — это символ.

Но так как компьютеры умеют работать только с числами (всюду одни нули и единицы), нам нужно уметь числа сопоставлять с буквами.

Для этого используются таблицы кодировок. В вебе стандартно используют utf8, потому что в нее умещается куча символов.

# А КАК ЭТО ВЫГЛЯДИТ?

Вот отрывок ASCII таблицы, чтобы мы понимали о чем речь.

Символы с кодами 128–255 (Кодовая таблица 1251 – MS Windows).

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	Ђ	160		192	А	224	а
129	Ѓ	161	Ў	193	Б	225	б
130	„	162	ў	194	В	226	в
131	ѓ	163	Ј	195	Г	227	г
132	”	164	Ѱ	196	Д	228	д
133	...	165	Ѓ	197	Е	229	е

Таких таблиц много, но сейчас используется большая универсальная таблица **UTF-8**.

---

# НАЗВАНИЕ

А почему, собственно, такое странное название у этого типа данных?

Все просто. Строки называются string, потому что они похожи на нитки (один из буквальных переводов данного слова с английского), на которые нанизываются бусины-символы.





# СОЗДАНИЕ СТРОК И СПЕЦСИМВОЛЫ

# СОЗДАНИЕ СТРОК

Строки создаются при помощи двойных или одинарных кавычек (также есть `ES6 template string`, но об этом мы будем говорить уже на продвинутом курсе по языку):

```
1 | let text = "Обычная строка";  
2 | let anotherText = 'Тоже строка';  
3 | let str = "012345";
```

# СПЕЦИАЛЬНЫЕ СИМВОЛЫ

А еще строки могут содержать специальные символы. Они позволяют нам легко добавлять в вывод новые строки или, например, смайлики/иконки.

Самый часто используемый из таких символов – это «перевод строки».

Он обозначается как `\n`:

```
1 // выведет "string" на новой строке
2 console.log( 'Привет\nstring!' );
```

## И ЧТО МНЕ ДЕЛАТЬ С ЭТИМИ СТРОКАМИ?

Вариантов достаточно много!

Одна из самых важных и наиболее часто используемых операций для строк — это операция сложения. Как думаете, что она означает?

Что мы получим в результате исполнения такой строчки кода?

```
"САША" + "МАША" = ?
```



## ПОСМОТРИМ НА ОТВЕТ?

В жизни мы, возможно, и получили бы в результате такого сложения любовь, но бессердечный JS выдаст нам просто САШАМАША, причем без пробела, так как мы его явно не задали.

# КОНКАТЕНАЦИЯ В ДЕЙСТВИИ

Конкатенацией (она же «склейка»/объединение) строк называется операция, в результате которой мы из нескольких строк получаем одну итоговую.

Мы можем как объединить все за один раз, так и динамически обновлять нашу переменную со строкой.

Вот как это можно сделать:

```
1 let myString = 'Я' + ' умею ' + 'объединять строки!';
2
3 let methodsCount = 2;
4 let dynamicString = '';
5 dynamicString += 'А теперь';
6 dynamicString += ' я умею работать со строками ';
7 dynamicString += methodsCount + ' способами!';
```



# МЕТОДЫ ДЛЯ РАБОТЫ СО СТРОКАМИ

## А ЧТО ЕЩЕ МОЖНО ДЕЛАТЬ СО СТРОКАМИ?

У строк (как и большинства других типов данных в JS) есть набор методов-помощников, которые позволяют удобно с ними работать.

С помощью этих методов мы можем приводить строку к нижнему (или верхнему) регистру, узнать из скольки символов она состоит, обрезать лишние пробелы (или просто какую-либо часть строки), превратить строку в массив или заменять те или иные символы в исходной строке.

Давайте посмотрим на несколько примеров.

## ДЛИНА LENGTH

Одно из самых частых действий со строкой – это получение ее длины. Считается количество символов:

```
1 // 3 символа. Третий – перевод строки
2 let js = "JS\n";
3 console.log(js.length); // 3
```

# СМЕНА РЕГИСТРА

Методы `toLowerCase()` и `toUpperCase()` меняют регистр строки на нижний/верхний:

```
1 | console.log("Интерфейс".toUpperCase());
```

Пример ниже получает первый символ и приводит его к нижнему регистру:

```
1 | console.log("ИНТЕРФЕЙС".toLowerCase());
```

# ПОДСТРОКИ: SUBSTRING

```
substring(start, end)
```

Метод `substring(start, end)` возвращает подстроку с позиции `start` до, но не включая `end`:

```
1 | let str = "stringify";  
2 | // "s", символы с позиции 0 по 1 не включая 1.  
3 | console.log(str.substring(0,1));
```

Если аргумент `end` отсутствует, то идет до конца строки:

```
1 | let str = "stringify";  
2 | // ringify, символы с позиции 2 до конца  
3 | console.log(str.substring(2));
```

# ЕСТЬ ЛИ ПОДСТРОКА В СТРОКЕ?

## `includes`

`includes(substr)`

Если нужно проверить, что в строке есть какая-либо буква или подстрока, то нужно воспользоваться методом `includes`. Нужно передать в метод искомую в строку. Метод вернет `true` если такая подстрока есть в исходной строке и `false`, если ее нет.

```
1 let phrase = "Мама мыла раму";
2
3 console.log(phrase.includes("мама")); // true → слово «мама» есть во фразе
4 console.log(phrase.includes("папа")); // false → слова «папа» во фразе нет
5
6 console.log(phrase.includes("p")); // true → буква «p» есть
```





# АЛГОРИТМЫ

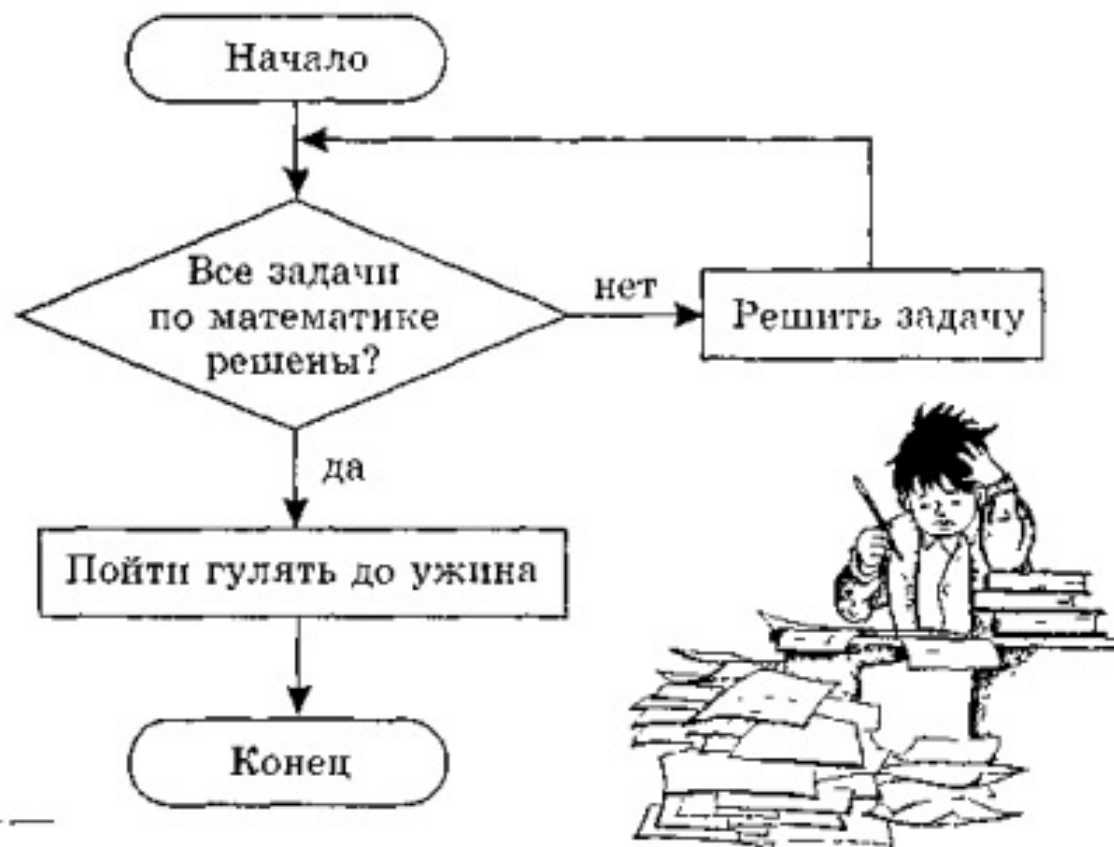


# АЛГОРИТМЫ

Алгоритм – это пошаговая инструкция, где результат прошлого шага строго определен и используется в качестве входных данных для следующего шага.

Мы используем алгоритмы каждый день и даже не замечаем этого. Например, рабочий (или учебный) день – это последовательность действий, в которой мы шаг за шагом что-то делаем.

# ДАВАЙТЕ ВИЗУАЛИЗИРУЕМ



# А ЧТО ЖЕ В ПРОГРАММИРОВАНИИ?

В программировании все точно также, только здесь мы решаем задачи и они зачастую являются более абстрактными и сложными для повседневного восприятия.

Например, нам нужно посчитать наибольший общий делитель двух чисел или вывести приветствие пользователю.

Любые задачи можно решить разными способами. Как понять, какой лучше?

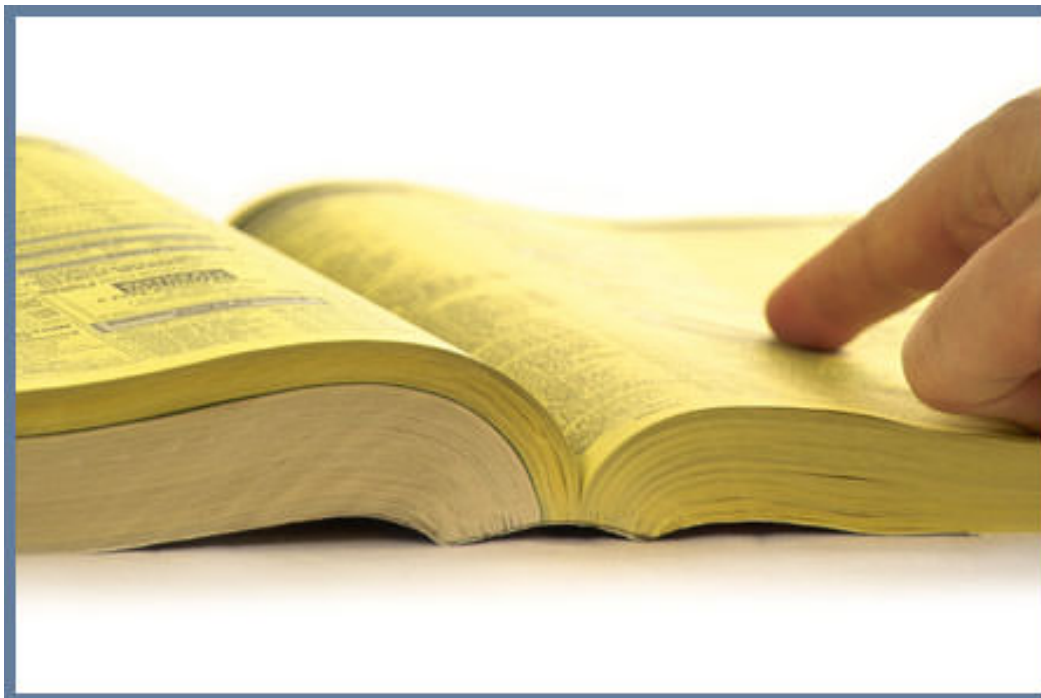
Как правило, чем меньше количество затраченных ресурсов и времени на исполнение, тем лучше код.

---

# ТЕЛЕФОННЫЙ СПРАВОЧНИК

Давайте представим, что телефонный справочник все еще актуален (да, тот бумажный, если вы их застали). Допустим, мы хотим набрать Ивана Иванова. Иван точно есть в телефонном справочнике.

**Как найти его номер? Как бы вы действовали?**



# МЕТОД ПОЛНОГО ПЕРЕБОРА

Самый простой способ – идти по номерам по порядку и сравнивать имя с искомым значением. То есть:

1. Марина Алексеева → не подходит.
2. Григорий Антипин → не подходит.

И так далее, пока не найдем Ивана Иванова. Вероятно, понадобятся десятки и даже сотни операций сравнения – все зависит от толщины справочника.

Такой способ называется линейным поиском. Здесь можно посмотреть на визуализацию данного подхода –

<http://www.cs.armstrong.edu/liang/animation/web/LinearSearch.html>

# БИНАРНЫЙ ПОИСК

Есть способ лучше. Воспользуемся фактом, что справочник отсортирован по фамилиям.

В нашем случае алгоритм будет примерно такой:

1. Открываем справочник посередине.
2. Смотрим, на какую букву идут фамилии.
3. Если нам повезло, то мы сразу попадем на страницу с фамилиями на И, тогда мы нашли Иванова!
4. Если фамилии идут на буквы, идущие после И, то Иванов будет в левой половине книги. Иначе — в правой половине.
5. Повторяем все с первого пункта, только делим пополам уже не весь справочник, а ту его половину, где находится фамилия Иванов.

Посмотрим демо: <http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html>



## КТО БЫСТРЕЕ?

Давайте сравним эти подходы с помощью наглядной демонстрации по ссылке:

<https://www.mathwarehouse.com/programming/images/binary-vs-linear-search/binary-and-linear-search-animations.gif>






## ВЫВОДЫ?

Есть целый раздел в Computer Science, который занимается изучением алгоритмов. На этом этапе важно понять лишь то, что код, решающий одну и ту же задачу может быть лучше или хуже.

Чем больше объем входных данных и нагрузки на ваше приложение/сайт, тем сильнее будет видна разница между алгоритмами.

Подробнее эту тему будем обсуждать на углубленных курсах.



# **РАБОТА С ОШИБКАМИ. ОТЛАДКА**

# ОТЛАДКА

А теперь давайте поговорим о том, как убедиться, что ваш код работает верно. Как его проверить?

В первую очередь мы должны проверить очевидные сценарии. Если, скажем, у нас есть программа, в которой мы делим числа, то мы можем легко проверить, что мы получаем верный результат этого деления.

```
1  let a = 30;  
2  let b = 3;  
3  
4  console.log(a / b); // 10
```

## ПОГРАНИЧНЫЕ СЛУЧАИ

А что, если в качестве знаменателя мы случайно получим ноль (представим, что мы получаем входные данные от пользователя, а не определяем самостоятельно в программе)? В JS никакие арифметические действия не приводят к ошибкам. Мы просто получим совершенно неожиданный (и нежеланный) для нас результат:

```
1  let a = 30;  
2  let b = 0;  
3  
4  console.log(a / b); // Infinity
```

## СОВЕТ

Если что-то идет не так, то можно печатать промежуточные значения; так лучше понимаешь, что происходит и как выполняется код:

```
1 // допустим пользователь ввел 9
2 let bobAge = prompt('Введите возраст Боба');
3 // допустим пользователь ввел 12
4 let yourAge = prompt('Введите ваш возраст');
5
6 if (bobAge > yourAge) {
7     console.log('Боб старше вас');
8 } else {
9     console.log('Вы старше Боба');
10 }
```

## ЧТО ЖЕ МЫ УВИДИМ?

Мы увидим сообщение в консоле о том, что `'Боб старше вас'`. Но ведь очевидно, что 12 больше, чем 9. В чем же проблема?

А проблема в том, что пользовательский ввод всегда представляет из себя строку. А строки в JS сравниваются совсем иначе, нежели числа. Если перед условной конструкцией мы выведем в консоль `bobAge` и `yourAge`, то мы сможем увидеть, что это строки и догадаться привести их к числам с помощью встроенных методов языка (например, `parseInt` или просто глобального объекта `Number`):

```
1 let bobAge = prompt('Введите возраст Боба');
2 let yourAge = prompt('Введите ваш возраст');
3
4 console.log(typeof bobAge, typeof yourAge); // string, string
```

# ОШИБКИ

А что делать, если код падает с ошибкой? Скажем вы просто опечатались и вместо хоть какого-то результата получаете ошибку. Консоль разработчика вся красная (как-будто тут кого-то убили), приложение не работает и кажется, что все плохо.

```
1  let a = 30; // let
2  let b = 0; // let
3
4  console.log(a / b); // Uncaught ReferenceError: log is not defined
```

В этом случае важно понять, что ошибки — это на самом деле наши друзья. На них мы учимся. А с помощью сообщений об ошибках легко понять, в каком месте и из-за чего это произошло.

# ВИДЫ ОШИБОК

В JS существует четыре основных типа ошибок:

- `ReferenceError`;
- `TypeError`;
- `SyntaxError`;
- `Logical error`.

Давайте рассмотрим их чуть подробнее.



# REFERENCEERROR

Неправильное имя. Либо мы забыли определить переменную, либо опечатались, либо она была удалена в ходе исполнения программы:

```
1  let x = 1;  
2  
3  console.log(x + y); // Uncaught ReferenceError: y is not defined
```

# TYPEERROR

Путать одно с другим. Если мы назначили в переменную число или строку, а потом решили работать с ней как с другим типом данных — у нас ничего не выйдет. В примере ниже мы пытаемся вызвать метод `includes` у переменной. Значение переменной `undefined`, а у этого типа данных нет такого метода.

```
1  let obj;  
2  
3  console.log(obj.includes()); // Uncaught TypeError:  
4  // Cannot read property 'includes' of undefined
```

# SYNTAXERROR

Нарушение правил языка. Например, мы решили начать название переменной с цифры:

```
1 | let 1player = 'Bob'; // Uncaught SyntaxError:  
2 | // Invalid or unexpected token
```

# LOGICAL ERROR

Ошибка в нашей с вами логике. Это самый коварный тип ошибок — программа будет функционировать, но мы будем получать неверный результат работы.

```
1 let x = "1";  
2 let y = "2";  
3  
4 console.log(x + y); // ошибки не будет, но в результате  
5 // мы получим строку '12', так как переменные у нас разных типов
```

---

## ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Строки не так просты, как кажутся;
2. Стали на шаг ближе к нинзя JS, положив в свой арсенал такой мощный инструмент, как строки и методы работы с ними;
3. Наконец-то разобрались почему в документах мы иногда встречали «кракозябры»;
4. Узнали, что такое алгоритмы и зачем они нам нужны;
5. Научились разбираться в ошибках и теперь готовы во всеоружии к любым неожиданностям.

---

## РАДУЕМСЯ!

# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в группе Facebook!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты все **3 задачи**.



# ЧТО ПОЧИТАТЬ?

- [https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String) — классическая документация от MDN;
- <https://learn.javascript.ru/string> — разбор от Ильи Кантора.

# А ПОСМОТРЕТЬ?

- <https://www.youtube.com/watch?v=VRz0nbax0uI> — 20 методов, которые помогут вам жонглировать строками в JS, как настоящий PRO;
- <https://www.youtube.com/watch?v=DljZQyNwdOg> — хороший обзор того, на что мы сегодня посмотрели (на английском);
- <https://stepik.org/lesson/99598/step/1?unit=75361> — про типы ошибок в JS и то как с ними бороться;
- <https://www.youtube.com/watch?v=U9o49qwa6hk> — увлекательнейшая лекция по алгоритмам (в рамках легендарного курса CS50 от Harvard University);
- <https://stepik.org/course/217/> — курс от Computer Science Center на stepik;
- <https://www.youtube.com/watch?v=JQhciTuD3E8> — еще немного про алгоритмы (How Binary Search Works).





Спасибо за внимание! Время задавать вопросы

**КОНСТАНТИН ПОЛЯНСКИЙ**



[kv.polyanskiy@gmail.com](mailto:kv.polyanskiy@gmail.com)



[@kvpolyanskiy](https://www.instagram.com/kvpolyanskiy)