

1. Введение
2. Инструкции и выражения
3. Подключение скрипта к странице
4. Строгий режим
5. Вкладка console в Chrome devtools
6. Переменные
7. Поднятие переменных
8. Типы примитивов
9. Взаимодействие с пользователем
10. Основные операторы
11. Числа
12. Строки
13. Преобразование типов примитивов
14. Правила хорошего кода

# 1 - Введение

**Компьютерная программа** – набор строк специального текста, написанного таким образом, чтобы машине было понятно, какие действия должны быть выполнены.

Когда мы говорим о программировании, первое что приходит в голову это набор инструкций в файле. Содержимое этого файла называется **исходный код**.

**Исходный код (source code)** - набор фраз, слов, специальных символов и т.д., специфичных для данного языка программирования, описывающих набор пошаговых инструкций для компьютера.

Но слова, фразы, символы и т.д., которые составляют программу, на самом деле непонятны для машины. Поэтому есть шаг, выполняющийся после того как вы написали программу, который конвертирует исходный код в файле, в набор инструкций понятных

компьютеру. Этим занимается специальная программа: **компилятор** или **интерпретатор**, которые преобразуют исходный код в набор нулей и единиц, так как машина не знает ничего другого.

Отсюда можно сделать вывод - код пишется не для машины, да компилятор использует его для составления машинного кода, но для разработчика, для человека. Код пишется для вас самих, чтобы спустя какое-то время вы, и другие разработчики, могли посмотреть в код и понять что там написано, как исполняется программа, какая у нее логика и т.д. Сам по себе исходный код это побочный артефакт того, что необходимо машине для выполнения инструкций.

Именно поэтому важно писать код так, чтобы он не только верно решал задачу, но и имел смысл когда на него смотришь. Код должен быть понятным, легко читаемым. Это одна из самых сложных задач начинающих (и не только), на ее решение уйдет много времени и сил.

## 1.1 - Логическое мышление

В мире существует очень много (сотни) языков программирования. На самом деле, они не такие сложные, как человеческие языки, потому что состоят из довольно маленького набора синтаксических конструкций.

Одна из самых интересных и увлекательных частей программирования – решение различных задач и проблем, которые представляют собой что-то вроде пазла или головоломки. Но это покажется интересным только тогда, когда вы разберётесь в коде и начнёте понимать его.

Опытные разработчики рассматривают проблемы с точки зрения алгоритмов – набора шагов, которые нужно выполнить для достижения определённой цели, даже если детали меняются.

Когда вы научитесь выстраивать своё мышление в виде алгоритмов, то язык программирования будет всего лишь вашим инструментом.

## 1.2 - JavaScript

JavaScript – высокоуровневый язык программирования, поддерживаемый всеми современными веб-браузерами, и изначально предназначенный для взаимодействия с элементами веб-страниц и добавления интерактивности.

При Front-end разработке, JavaScript используется с HTML и CSS для обеспечения функциональности веб-страницы, такой как создание интерактивных карт, отображение анимированных графиков и т. д. Для реализации всех этих возможностей необходимо знать синтаксис и уметь строить алгоритмы выполнения программы.

## 2 - Инструкции и выражения

При написании кода, важно не просто знать какой символ или конструкцию можно использовать, но в первую очередь необходимо понимать терминологию и составляющие исходного кода.

### 2.1 - Инструкция

**Инструкция (statement)** – это связанный набор слов и символов из синтаксиса языка, которые объединяются, чтобы выразить одну идею, одну инструкцию для машины.

```
a = b * 2;
```

Это пример инструкции. В JavaScript можно (обычно) различить инструкцию по точке с запятой в конце. Точку с запятой в данном

случае можно сравнить с точкой в конце предложения вашего родного языка, она символизирует о конце предложения.

- **a** и **b** называются переменными (как в алгебраическом уравнении), это такие коробки в которых можно хранить информацию. Переменные содержат значения, которые использует программа.
- **2** – просто значение. Это называется значением литерала (literal value), так как оно не хранится в переменной.
- **=** и **\*** – операторы, они производят действия над значениями и переменными.

Представим что переменная **b** уже содержит значение, число **10**. Тогда наша инструкция говорит машине:

- Пойди найди переменную **b** и спроси какое у нее сейчас значение.
- Подставь значение переменной **b** в утверждение на место **b**.
- Умножь **10** на **2**, получив **20**.
- После чего, запиши результат вычисления выражения правой части (то что справа от оператора **=**) в переменную **a**.

## 2.2 - Выражения

Инструкции состоят из частей, как в любом языке предложения состоят из фраз, и эти фразы называются выражениями.

**Выражение (expression)** – ссылка на переменную или значение, или на набор переменных и значений в сочетании с операторами.

[ [a] = [ [b] \* [2] ] ];

Инструкция из примера выше содержит 5 выражений (для визуализации выделены квадратными скобками):

- **[2]** - выражение значения литерала.
- **[b]** и **[a]** - выражения переменной, означают необходимость подставить значение переменной, но только в том случае, если переменная стоит в правой части операции присваивания.
- **[b \* 2]** - арифметическое выражение. В данном случае умножение.
- **[a = b \* 2]** - выражение присваивания. В данном случае указывает на необходимость вычисления правой части выражения и присваивания результата переменной **a** в левой части выражения.

Так же есть выражения вызова, выражения сравнения и т. д. Мы не будем сейчас рассматривать их все, нам важно понимать из каких частей состоит исходный код и как правильно его читать.

## 2.3 - Точка с запятой

Хотя JavaScript не требует обязательного завершения инструкции точкой с запятой, рекомендуется всегда ее ставить. Это простое правило сделает ваш код более понятным и поможет избежать многих ошибок, с которыми вы можете столкнуться.

## 3 - Подключение скрипта к странице

Чтобы добавить JavaScript к странице, в HTML-файле используется тег **script**, в атрибуте **src** которого указываем ссылку на внешний JavaScript-файл.

Чтобы подключить JavaScript из внешнего файла:

1. Создайте файл с расширением `.js` и поместите его в подпапку `js`.

Размещение JavaScript файла в папке `js` не требуется, однако это хорошая практика.

2. Затем укажите путь к файлу скрипта в атрибуте `src` элемента `script`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Module-1</title>
  </head>

  <body>
    <!-- контент -->
    <script src="js/scripts.js"></script>
  </body>
</html>
```

При отображении HTML-документа, некоторое время может отображаться пустая страница, дело в том что фаза рендеринга страницы преостаналивается до тех пор, пока не будет полностью загружен скрипт (если он указан в `head`). Чтобы этого избежать, включать файл JavaScript стоит перед закрывающим тегом `body` после все содержимого, как показано в примере.

## 3.1 - Множественные файлы скриптов

Когда вы включаете несколько JavaScript-файлов на страницу, интерпретатор обрабатывает файлы в том порядке, в котором они указаны в HTML-файле.

Браузер загружает и отображает HTML постепенно. Если браузер видит тег `script`, то по стандарту он обязан сначала выполнить его, а потом показать оставшуюся часть страницы.

Интерпретатор будет последовательно обрабатывать файлы `script-1.js` и `script-2.js`.

```
<script src="/path/to/script-1.js"></script>
<script src="/path/to/script-2.js"></script>
```

## 4 - Строгий режим

Изначально язык JavaScript развивался без потери совместимости. Новые возможности добавлялись в язык, но старые – никогда не менялись, чтобы не «сломать» уже существующие веб-страницы с их использованием.

Строгий режим – новая возможность в спецификации ECMAScript 5, которая позволяет переводить скрипт в режим полного соответствия современному стандарту. Это предотвращает определенные действия, такие как использование небезопасных конструкций.

Для того чтобы перевести скрипт в строгий режим, достаточно просто указать директиву `use strict` в начале js-файла. Всегда пишите код в строгом режиме.

```
'use strict';
// дальше идет весь код файла
```

[Strict mode на MDN](#)

# 5 - Вкладка console в Chrome Devtools

При написании кода всегда будут ошибки, это нормально. Консоль показывает информацию, связанную с веб-страницей: сетевые запросы, JavaScript, CSS, ошибки безопасности, а также сообщения об ошибках, предупреждениях и сообщениях JavaScript, запущенным в контексте страницы.

- [Introduction to Google Chrome Developer tool for JavaScript](#)
- [Статья о работе с devtools](#)
- [How you can improve your workflow using the JavaScript console](#)

# 6 - Переменные

Переменные используются для хранения данных. Думайте о них как о коробках с названием, в которые мы можем положить какое-то значение. Переменная является только именованным заполнителем для значения, она состоит из имени и выделенной области памяти, которая ему соответствует.

## 6.1 - Имена переменных

Идентификатор - это имя переменной, функции, параметра или класса. Состоит из одного или нескольких символов в следующем формате:

- Первым символом должна быть буква `a-z` или `A-Z`, символ подчеркивания `_` или знак доллара `$`.

- Другие символы могут быть буквами `a-z`, `A-Z`, цифрами `0-9`, подчеркиваниями `_` и знаками доллара `$`.

Все в JavaScript, включая переменные, имена функций, имена классов и операторы, чувствительны к регистру. Это означает, что переменные `user`, `usEr` и `User` различны.

Имя переменной должно быть понятным.

Плохо	Хорошо
<code>yabloko</code>	<code>apple</code>
<code>chislo</code>	<code>number</code>
<code>massivProductov</code>	<code>goods</code>
<code>spisok_iz_4</code>	<code>list</code>

Стандартной практикой является использование camelCase-нотации для идентификаторов, при котором первое слово полностью пишется строчными буквами, а каждое дополнительное слово начинается с прописной буквы. К примеру: `user`, `greetUser`, `getUserData`, `isActive`.

## 6.2 - Ключевые слова

JavaScript определяет список зарезервированных ключевых слов, которые имеют специальное значение и используются для определенных конструкций. Нельзя использовать ключевые слова как идентификаторы.

[Таблица зарезервированных слов](#)

## 6.3 - Объявление переменных

В современном JavaScript есть 2 ключевых слова для объявления переменных.

- **let** – в переменную объявленную как **let** можно присвоить новое значение.
- **const** – "константа", используется как ключевое слово **let**, но переменная, которую вы объявляете, должна быть немедленно инициализирована значением, и этой переменной не может быть присвоено другое значение после инициализации.

Создание переменной без ключевого слова **let** или **const**, в строгом режиме, приведет к ошибке выполнения скрипта.

```
// Для объявления переменной используются
// ключевые слова let и const, за которыми следует имя переменной
let firstNumber;
// Переменные объявленные используя const обязательно должны быть
// инициализированы значением во время объявления, иначе будет ошибка
const secondNumber = 15;

// Значение переменной можно получить обратившись к ней по имени
// console.log используется для вывода данных в консоль разработчика
// с этой конструкцией более детально мы познакомимся позже.
// Если мы изначально не присваиваем переменной никакого значения
// в нее помещается специальное значение undefined (не определено)
console.log('Переменная до присваивания ей значения: ', firstNumber);

// После объявления, если переменная объявлена через let,
// можно записать значение.
firstNumber = 5;
// Помним о том что нельзя перезаписать значение переменной
// объявленной используя const
secondNumber = 666; // будет ошибка
```

```
// После присвоения значений  
console.log('Первое число: ', firstNumber);
```

### 6.3.1 - const vs. let

Единственное отличие `const` от `let` состоит в том, что `const` запрещает повторное присваивание какого-либо значения.

`const` делает код более читабельным. В своей области видимости `const`-переменная всегда ссылается на один и тот же объект. В случае с `let`-переменными такой уверенности быть уже не может.

Будет разумно использовать `let` и `const` так:

- Используйте `const` по умолчанию
- Используйте `let`, если потребуется присвоить переменной другое значение
- Не используйте `var`

### 6.4 - undefined vs. undeclared

Важно различать неопределенные и необъявленные переменные.

- **Неопределенная переменная (`undefined`)** - это переменная, которая была объявлена, но поскольку мы не присвоили ей значение, переменной присвоилось значение `undefined` как ее начальное значение.
- **Необъявленная переменная (`undeclared`)** - та которая не была объявлена в доступной области видимости.

```
let declaredVariable;  
// Переменная declaredVariable объявлена, но не инициализирована  
// поэтому ее значение undefined (не определено),  
// тогда как переменная undeclaredVariable не была объявлена,  
// поэтому доступ к ней вызывает ошибку ReferenceError  
  
console.log(  
    declaredVariable  
) // undefined  
  
console.log(  
    undeclaredVariable  
) // ReferenceError: undeclaredVariable is not defined
```

## 7 - Поднятие переменных

Hoisting (поднятие переменных) – механизм, при котором компилятор перемещает все объявления переменных в верхнюю часть области видимости.

Если вы объявили переменную с ключевым словом `let` или `const`, в отличии от многих ошибочных заявлений, переменная будет поднята до вершины охватывающего ее лексического окружения.

Но во время исполнения кода, при входе в область видимости где она была объявлена, она не будет инициализирована значением `undefined` до той строки кода, где она изначально была объявлена ключевым словом `let` или `const`.

В результате, если вы обращаетесь к такой переменной перед ее объявлением, результатом будет ошибка `ReferenceError`. Это тот

механизм, что в спецификации описывается как TDZ (temporal dead zone).

```
// Попытка вызвать переменную по имени до  
// ее определения в коде вызовет ошибку  
  
// console.log(a); // ReferenceError: b is not defined  
let a = 10;  
console.log(a); // 10  
  
// console.log(b); // ReferenceError: c is not defined  
const b = 15;  
console.log(b); //15
```

Можно запомнить простое правило - **использование переменной до ее объявления вызовет ошибку.**

## 8 - Типы примитивов

JavaScript это динамический, слабо типизированный язык, поэтому сама переменная не ассоциируется с каким-либо типом, только ее значение. Тип данных описывает, какие данные хранятся. Другими словами, одна и та же переменная может хранить значения разных типов в любое время.

### 8.1 - Number

Целые числа и числа с плавающей запятой. После того, как вы объявили переменную, вы можете инициализировать ее числовым значением. Более детально описаны в разделе [Числа](#).

```
let age = 20;  
let number = 5.8;
```

## 8.2 - String

Строки в JavaScript - это просто текст, последовательность из нуля или более символов. Стока начинается и заканчивается либо одиночной ' , либо двойными кавычками " .

Ключевым моментом для запоминания является то, что строка, начинающаяся с двойной кавычки, должна заканчиваться двойной кавычкой, а строка, начинающаяся с одиночной кавычки, должна заканчиваться одиночной кавычкой.

```
let name = 'Mango';  
let hobby = "Coding";
```

Более детально описаны в разделе [Строки](#)

## 8.3 - Boolean

Логический тип данных, флаги состояния. Он имеет только два значения: `true` или `false`, в нижнем регистре. К примеру на вопрос включен ли свет в комнате вы можете ответить да (`true`) и нет (`false`).

Синтаксис	Значение
<code>true</code>	имеет смысл «да», «верно», «истина», 1
<code>false</code>	означает «нет», «неверно», «ложь», 0

Обратите внимание на имена переменных содержащих булевое значение. Они задают вопрос, и ответ на его - **да или нет**.

```
let isAuthenticated = true;
let canMerge = false;
let hasChildren = true;
```

## 8.4 - null

Особое значение в JavaScript, которое по сути значит "ничто". Используется в тех ситуациях когда необходимо явно указать отсутствие значения. К примеру если в базе данных не нашли пользователя, то можно сказать что значение `null`.

Переменная `apples` пуста и лишена значения.

```
let apples = null;
```

## 8.5 - undefined

Еще одно специальное значение в JavaScript. По умолчанию, когда переменная объявляется, но не инициализируется, ее значение не определено, ей присваивается значение `undefined`, потому что не было присвоено значение.

```
let apples;  
console.log( apples ); // undefined
```

## Исследование бездны null и undefined в JavaScript

### 8.6 - Оператор typeof

Чтобы получить текущий тип значения переменной, используйте оператор `typeof`, который возвращает на место своего вызова тип переменной или литерала указанного после него. Возвращаемый тип это просто строка.

```
// undefined  
let a;  
console.log(typeof a); // "undefined"  
  
// null  
// object это не примитивный тип данных,  
// о нем мы узнаем в следующих модулях  
let b = null;  
console.log(typeof null); // "object"  
  
// Number  
let c = 5;  
console.log(typeof c); // "number"  
  
// Чтобы представить число с плавающей запятой,  
// просто ставим точку  
let d = 5.3;  
console.log(typeof d); // "number"  
  
// String
```

```
let message = "Lets learn some JavaScript!";
console.log(typeof message); // "string"

// Boolean
let isActive = false;
console.log(typeof isActive); // "boolean"

let isHappy = true;
console.log(typeof isHappy); // "boolean"
```

## 9 - Взаимодействие с пользователем

В этом разделе мы разберем базовые приемы ввода/вывода, достаточные для получения и отображения данных до того как научимся взаимодействовать с HTML-документом.

### 9.1 - Вывод данных

Для вывода данных, на данном этапе, мы будем использовать 2 конструкции: `console.log()` и `alert()`. Это функции, о том что такая функция мы поговорим в следующих модулях, сейчас нам интересно только научиться использовать их для вывода данных.

```
const message = 'Let\'s learn some JavaScript!';
// В круглых скобках мы можем указать имя переменной
// значение которой необходимо вывести в консоль
console.log(message); // Let's learn some JavaScript!
```

```
const name = 'Mango';
// Так же мы можем сначала указать какую-то
// произвольную строку, к примеру описывающую
// переменную или дополняющую ее, после чего
// поставить запятую и указать имя переменной
console.log('My name is', name); // My name is Mango

// alert показывает модальное окно, текст которого
// соответствует значению переменной (или просто строку/числ
// которую мы укажем в скобках.
alert(message);
```

## 9.2 - Получение данных

Для получения данных от пользователя мы, пока что, будем использовать `prompt()` и `confirm()`. Это тоже функции. Их особенность в том, что результатом своего выполнения они возвращают то, что было введено пользователем. Результат работы этих функций можно записать в переменную для дальнейшего использования.

- `confirm` – выводит модальное окно с сообщением, и 2 кнопки, `Ok` и `Cancel`. При нажатии на `Ok`, на место вызова функции возвращается `true`, при нажатии на `Cancel` возвращается `false`.
- `prompt` – выводит модальное окно с полем ввода и кнопками `Ok` и `cancel`. При `Ok` возвращает то, что было введено в поле ввода, при `Cancel` возвращает `null`.

```
// Результат работы confirm и prompt,
// мы можем записать в переменную.
```

```
// В дальнейшем мы научимся проверять  
// введенные данные, выбранную опцию и т.д.  
  
// Мы можем попросить клиента подтвердить бронь на отель  
const isComing = confirm('Please confirm hotel reservation');  
console.log(isComing);  
  
// Тут мы можем спросить имя отеля  
// в котором хотел бы остановится клиент  
const hotelName = prompt('Please enter desired hotel name:');  
console.log(hotelName);  
  
// Важной особенностью prompt есть то, что не зависимо  
// что ввел пользователь, всегда вернется строка.  
// То есть если пользоваль ввел 5, то вернется не число 5,  
// а строка "5".  
// Об этом всегда необходимо помнить.  
  
const number = prompt('Please enter a number!');  
console.log( typeof number ); // 'string'  
console.log(number); // '5'
```

## 10 - Основные операторы

### 10.1 - Математические операторы

Ничем не отличаются от школьного курса элементарной алгебры.  
Порядок вычисления математических выражений и т. п. это все  
привычная алгебра.

**Синтаксис****Операция**

$a + b$	Сложение
$a - b$	Вычитание
$a * b$	Умножение
$a / b$	Деление
$a \% b$	Остаток от деления

Важно запомнить правильное именование составляющих выражения.

**+ - \* / %** называются **операторами**, а то на что они применяются (в данном случае числа) называются **операндами**.

Операторы возвращают значение на место операции, к примеру JavaScript видит операцию  $2 + 2 + 2$  как  $2 + 2 + 2 = 4 + 2 = 6$

```
// Операции с числами
const x = 10;
const y = 5;
let result;

result = x + y;
console.log(result); // 15

result = x - y;
console.log(result); // 5

result = x * y;
console.log(result); // 50

result = x / y;
console.log(result); // 2

result = x % y;
console.log(result); // 0
```



```
console.log(9 % 4); // 1  
console.log(13 % 5); // 3
```

## 10.2 - Операторы сравнения

Операторы сравнения используются... для сравнения значений.

Результатом своего исполнения они возвращают були, `true` или `false`.

Синтаксис	Операция
<code>a &gt; b, a &lt; b</code>	больше/меньше
<code>a &gt;= b, a &lt;= b</code>	больше/меньше или равно
<code>a == b</code>	равенство
<code>a != b</code>	неравенство
<code>a === b</code>	строгое равенство
<code>a !== b</code>	строгое неравенство

**Всегда** используйте строгое равенство `==` и строгое неравенство `!=` при написании кода. Операторы `==` и `!=` выполняют преобразование типов сравниваемых значений, что может привести к ошибкам, особенно у начинающих.

```
const x = 5;  
const y = 10;  
const z = 5;  
let result;  
  
result = x > y;
```

```
console.log('x > y:', result); // false

result = x < y;
console.log('x < y:', result); // true

result = x < z;
console.log('x < z:', result); // false

result = x <= z;
console.log('x <= z:', result); // true

result = x === y;
console.log('x === y:', result); // false

result = x === z;
console.log('x === z:', result); // true

result = x !== y;
console.log('x !== y:', result); // true

result = x !== z;
console.log('x !== z:', result); // false
```

## 11 - Числа

Все числа в JavaScript, как целые так и дробные, имеют тип `Number` и хранятся в 64-битном формате. В JavaScript можно записывать числа не только в десятичной системе счисления.

В ES6, `Number` получил большое количество улучшений. Материя этого курса, руководствуясь лучшими практиками, использует именно

улучшеные методы работы с числами. Отличную статью по **Number** [обязательно прочитать тут](#).

## 11.1 - Приведение к числу

Большинство арифметических операций и математических функций преобразуют значение в число автоматически. Для того чтобы сделать это явно, обычно перед значением ставят унарный плюс + .

В новой версии языка можно использовать более наглядный и приятный способ приведения к числу используя **Number()** .

При этом, если строка не является в точности числом, то результат будет **Nan** (Not a Number). Аналогичным образом происходит преобразование и в других математических операторах и функциях.

```
let valueA = '5';
let valueAsNumber = +valueA;
let typeOfValueAsNumber = typeof valueAsNumber;

console.log( typeOfValueAsNumber ); // 'number'
console.log( valueAsNumber ); // 5

valueAsNumber = Number(valueA);
typeOfValueAsNumber = typeof valueAsNumber;

console.log( typeOfValueAsNumber ); // 'number'
console.log( valueAsNumber ); // 5

let valueB = 'random string';
valueAsNumber = +valueB;
typeOfValueAsNumber = typeof valueAsNumber;
```

```

console.log( typeOfValueAsNumber ); // 'number'
console.log( valueAsNumber ); // NaN

valueAsNumber = Number(valueB);
typeOfValueAsNumber = typeof valueAsNumber;

console.log( typeOfValueAsNumber ); // 'number'
console.log( valueAsNumber ); // NaN

```

## 11.1.1 - parseInt и parseFloat

В HTML/CSS многие значения не являются числами, только одна их составляющая. Например `10px`. Оператор `+` или `Number()` для таких значений возвращают `NaN`.

Функция `Number.parseInt(val)` и ее аналог `Number.parseFloat(val)` преобразуют строку символ за символом, пока это возможно. При возникновении ошибки возвращается число, которое получилось. Функция `Number.parseInt(val)` читает из строки целое число, а `Number.parseFloat(val)` – дробное.

```

console.log( Number.parseInt('5px') ); // 5
console.log( Number.parseInt('12qwe74') ); // 12
console.log( Number.parseInt('12.46qwe79') ); // 12
console.log( Number.parseInt('qweqwe') ); // NaN

console.log( Number.parseFloat('5px') ); // 5
console.log( Number.parseFloat('12qwe74') ); // 12
console.log( Number.parseFloat('12.46qwe79') ); // 12.46
console.log( Number.parseFloat('qweqwe') ); // NaN

```

## 11.2 - Проверка на число

Для проверки на число можно использовать метод `Number.isNaN(val)`. Он проверяет, является ли указанное значение `NaN` или нет. Этот метод отвечает на вопрос "Это Not A Number?", и возвращает:

- `true` - если значение `val` это `NaN`
- `false` - если значение `val` это не `NaN`

Для всех значений `val` кроме `NaN`, при передаче в `Number.isNaN(val)` вернёт `false`. Этот метод не производит попытку преобразовать `val` к числу, а просто выполняет проверку на `NaN`. Отличные примеры [посмотрите тут](#).

```
const validNumber = Number('51'); // 51
console.log( Number.isNaN(validNumber) ); // false

const invalidNumber = Number('qweqwe'); // NaN
console.log( Number.isNaN(invalidNumber) ); // true
```

## 11.3 - Сложение чисел с плавающей запятой

Есть очень важная "особенность" при сложении не целых чисел в JavaScript. В кратце `0.1 + 0.2` не равно `0.3`. Результат сложения `0.1` и `0.2` немного больше чем `0.3`. В то время как мы считаем в десятичной системе, машина считает в двоичной системе.

Число `0.1` в двоичной системе счисления это бесконечная дробь так как единица на десять в двоичной системе так просто не дели. Также бесконечной дробью является `0.2`. Двоичное значение бесконечных дробей хранится только до определенного знака, поэтому

возникает неточность. Когда мы складываем **0.1** и **0.2**, то две неточности складываются, получаем незначительную, но всё же ошибку в вычислениях.

Конечно, это не означает, что точные вычисления для таких чисел невозможны. Они возможны. И даже необходимы. Есть несколько методов решения этой проблемы:

- Сделать их целыми, умножив на 100, сложить, а потом результат разделить на 100
- Сложить, а затем округлить до разумного знака после запятой. Округления до 10-го знака обычно бывает достаточно, чтобы отсечь ошибку вычислений

```
console.log( 0.1 + 0.2 === 0.3);
```

```
console.log(0.1 + 0.2);
```

```
// Сделать их целыми, умножив на 100,  
// сложить, а потом результат разделить на 100  
console.log( (0.1 * 100 + 0.2 * 100) / 100 );
```

```
//Сложить, а затем округлить до разумного знака после запятой  
const result = 0.1 + 0.2;  
console.log( +result.toFixed(10) );
```

## 11.4 - Методы объекта Math

Объект **Math** является встроенным объектом JavaScript. Объект **Math** содержит ряд методов для работы с числами. Знание всех методов наизусть не требуется, только наиболее важных и часто используемых.

**Метод****Описание****Math.random()**

Псевдослучайное число в диапазоне [0, 1), т.е. включая 0, но исключая 1.

**Math.max(a, b, c)**

Возвращает наибольшее из чисел, перечисленных через запятую в круглых скобках (наибольший аргумент)

**Math.min(a, b, c)**

Возвращает наименьшее из чисел, перечисленных через запятую в круглых скобках (наименьший аргумент)

**Math.ceil(0.5)**

Округление вверх: 1

**Math.floor(0.5)**

Округление вниз: 0. Используется для точного распределения Math.random()

**Math.round(0.5)**

Округление до ближайшего целого: 1

**Math.pow(base, exponent)**

возведение числа base в степень exponent

[Документация Math на MDN](#)

## 12 - Строки

Строка (или строчный литерал) - это набор из нуля или больше символов, заключенных в одинарные или двойные кавычки.

```
const name = 'Mango';
```

Содержимое строки в JavaScript нельзя изменять. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда. Можно лишь создать целиком новую строку и присвоить ее переменную вместо старой.

## 12.1 - Конкатенация строк

Если применить оператор `+` к строке и любому другому типу данных, результатом операции "сложения" будет строка. Данная операция называется **конкатенация**, или сложение строк.

Во время конкатенации, любой тип данных приводится к строке и сшивается со строкой в выражении сложения, но есть особенность - последовательность записи операндов.

```
// Классическая конкатенация строк
const state = 'Mango ' + 'is' + ' happy';
console.log(state); // Mango is happy

// Давайте разберемся с более интересной конструкцией
let result;

result = 5 + '5';
console.log(result); // '55'
console.log(typeof result); // string

result = 5 + '5' + 5;
console.log(result); // '555'
console.log(typeof result); // string

// Обратите внимание, произошла математическая операция
// сложения для первых двух пятерок, после чего 10 было
// преобразовано в строку '10' и сшито с '5'.
// Последовательность операций имеет значение, преобразование
// типов к строке при сложении со строкой происходит только в
// тот момент, когда происходит операция сложения со строкой,
// до этого момента действуют привычные правила математики!
result = 5 + 5 + '5';
```

```
console.log(result); // '105'  
console.log(typeof result); // string
```

## 12.2 - Методы строк

У каждой строки есть встроенные методы. Что такое метод и почему он есть у строк будет более детально разобрано в модуле о объектах. Сейчас мы просто научимся ими пользоваться.

Для использования метода, необходимо поставить точку после строчного литерала или переменной которая содержит строку, после чего написать имя метода.

Индексация символов строки начинается **с нуля**, не с единицы! Всегда важно об этом помнить. К примеру в строке "JavaScript is amazing!" буква "J" стоит на позиции с индексом **0**.

### Синтаксис

**length**

**toLowerCase()** и **toUpperCase()**

**index0f()**

**includes()**

### Функционал

свойство строки, вернет длину строки

вернут строку в соответствующем регистре

вернет позицию (индекс) на которой находится подстрока или -1, если ничего не найдено

один из наиболее часто используемых методов, в большинстве случаев заменяет indexOf, проверяет входит ли подстрока в строку, возвращает **true** или **false**

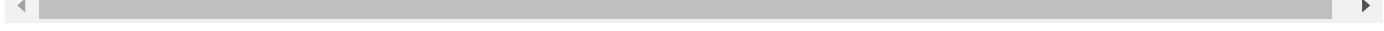
```
const message = 'Welcome to Bahamas!';

console.log(message.length); // 19
console.log(
  'There is nothing impossible to him who will try'.length
); // 47

console.log( message.toLowerCase() ); // welcome to bahamas!
console.log( message.toUpperCase() ); // WELCOME TO BAHAMAS!

// Все методы строки чувствительны к регистру
console.log( message.indexOf('to') ); // 28
console.log( message.indexOf('hello') ); // -1

console.log( message.includes('welcome') ); // false
console.log( message.includes('Welcome') ); // true
```



С полным списком методов строк Вы можете ознакомиться в [официальной документации](#).

## 12.3 - Шаблонные строки и интерполяция

Шаблонные строки это строчные литералы, допускающие использование выражений. Вы можете использовать многострочные литералы и возможности интерполяции.

Шаблонные строки заключены в обратные кавычки `` вместо двойных или одинарных. Они могут содержать местозаполнители, которые обозначаются знаком доллара и фигурными скобками \${переменная или выражение} .

```
// Представьте следующий код
const name = 'Mango';
const age = 2;
const mood = 'happy';
// А теперь нам надо из этих переменных составить
// результирующую строку в которую надо подставить их значения

const message = 'My name is ' + name + ", I'm " + age + ' years old and I am ' + mood;
console.log(message); // My name is Mango, I'm 2 years old and I am happy

// Ну как? Понравилось ставить море плюсов для конкатенации строк?
// Особенно хороша точка в конце.

// Представьте ситуацию когда у вас 10 переменных, значения которых
// необходимо подставить в строку. Понимаю, желание бросить Я
// сейчас дошло до пика.

// На помощь приходят шаблонные строки и интерполяция
const sameMessage = `My name is ${name}, I'm ${age} years old and I am ${mood}`;
console.log(sameMessage); // My name is Mango, I'm 2 years old and I am happy

// Ну как? Удобно? :)

// Можно также использовать выражения
console.log(`Результат сложения равен ${2 + 2}.`); // Результат сложения равен 4.
```

## 13 - Преобразование типов примитивов

Система преобразования типов в JavaScript довольно проста, но отличается от других языков. Всего есть 3 типа преобразования:

- Приведение к строке
- Приведение к числу
- Приведение к логическому типу

## 13.1 - Приведение к строке

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. То есть в строковом контексте или при сложении со строкой. Работает очевидным образом, как есть: `false` становится "`false`", `null` – "`null`" и т.п.

Например, его производит функция `alert`.

```
let isActive = false;  
alert( isActive ); // "false"
```

Можно также осуществить преобразование явным вызовом `String(val)`.

```
console.log( String(false) === "false" ); // true
```

Также для явного преобразования применяется оператор `+`, у которого один из аргументов строка. В этом случае он приводит к строке и другой аргумент.

```
console.log( true + "text" ); // "truetext"  
console.log( "100" + undefined ); // "100undefined"
```

## 13.2 - Приведение к числу

Происходит в численном контексте, в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений `==`, `!=`).

Для преобразования к числу в явном виде можно вызвать `Number(val)`, либо, что короче, поставить перед выражением унарный плюс `+`. Унарный плюс может не считаться лучшей практикой, так как усложняет чтение кода.

```
let a = +"100"; // 100
let b = Number("100"); // 100
```

Полноценное правило преобразования строки таково: пробельные символы по краям обрезаются; далее, если остаётся пустая строка, то она преобразовывается в число `0`, иначе из непустой строки "читывается" число, при ошибке результат `Nan`.

Значение	Преобразуется в...
<code>undefined</code>	<code>Nan</code>
<code>null</code>	<code>0</code>
<code>true/false</code>	<code>1/0</code>

```
console.log(+ "100text"); // NaN
console.log(+ undefined); // NaN
console.log(+ null); // 0
console.log(+ true); // 1
console.log(+ false); // 0
console.log( "1" == true); // true
console.log( "1" === true); // false
```

Интуитивно, значения `null/undefined` ассоциируются с нулём, но при преобразованиях ведут себя иначе. Для более очевидной работы кода и во избежание ошибок лучше не давать специальным значениям `null/undefined` участвовать в сравнениях `> >= < <=`.

## 13.3 - Приведение к логическому типу

Преобразование к `true/false` происходит в логическом контексте, таком как `if(value)`, и при применении логических операторов. Об этом детальнее в следующем модуле.

## 14 - Правила хорошего кода

Для стандартизации стиля кода используйте плагин [Prettier](#). Его реализации есть для таких редакторов как Atom, Sublime и др, их можно найти в [официальной документации](#).