

Fundamentals of JavaScript 1

What is JavaScript

JavaScript ≠ Java

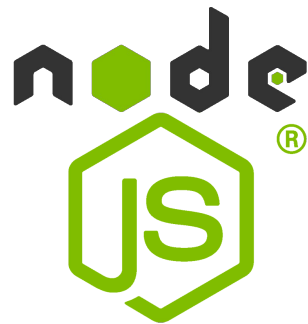
A yellow square containing the letters 'JS' in a large, dark blue, sans-serif font, representing the JavaScript logo.

JavaScript (далее JS) - это мультипарадигменный язык программирования, который является одной из основных технологий Всемирной паутины, наряду с HTML и CSS.

По состоянию на 2022 год 98% веб-сайтов используют JavaScript на стороне клиента для поведения веб-страниц. Все основные веб-браузеры имеют специальный движок JavaScript для выполнения кода на устройствах пользователей.

Движки JavaScript первоначально использовались только в веб-браузерах, но теперь (благодаря node.js) являются основными компонентами некоторых серверов и различных приложений.

What is Node.js



До 2009 года JS мог выполняться только в браузере.

В 2009 году вышла **Node.js** - программа, позволяющая запускать JS вне браузера.

Node.js сделала JavaScript языком общего назначения (как Python).

Итого на данный момент мы имеем две возможные среды выполнения JS: *браузерную* и *node.js*.



(Клиентская часть сайта)



(Серверная часть сайта, десктопные приложения)

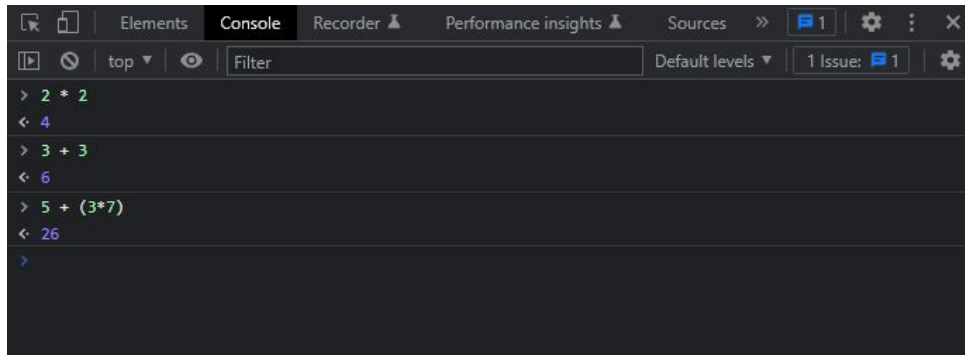
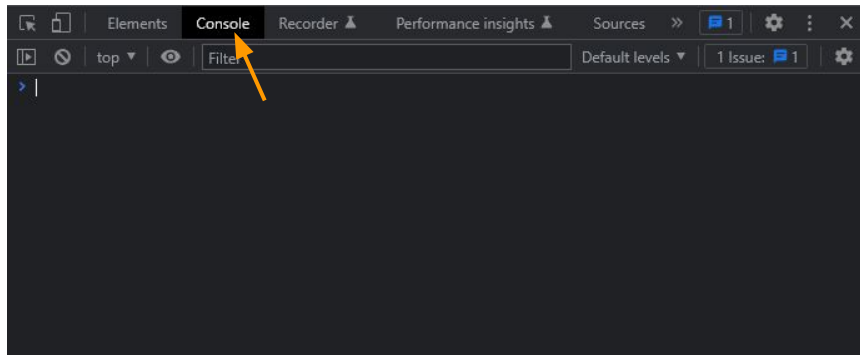
В браузерной среде выполняется клиентская часть всех сайтов. В node.js может выполняться серверная часть сайта, десктопные или любые другие приложения.

Browser runtime environment

Будем работать с браузерной средой.

Для открытия JS консоли в браузере нажмите F12 и перейдите во вкладку консоль.

Вы можете вбить в консоль простые арифметические выражения и исполнитель JS будет вам возвращать ответ. Такая форма организации интерактивной среды программирования называется **REPL** (read-eval-print loop). То есть js сначала читает выражение (read), затем обрабатывает и вычисляет его (eval) и после вывода результата (print) ожидает следующее выражение (loop).



Browser runtime environment

Но писать большой код в консоли неудобно. Поэтому js код обычно пишут в файле, который затем открывают в браузере.

Создадим файл **script.html** и поместим в него следующее содержимое:

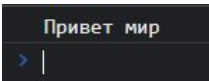
```
<script>  
console.log( "Привет мир" );  
</script>
```

Между тегами script будем помещать наш JavaScript код

(Впоследствии для краткости теги script будем опускать)

Мы по сути создали обычный html файл с единственным тегом `<script>`, внутри которого и помещается JavaScript код. Функция `console.log()` используется для вывода различных данных в консоль.

Если вы откроете полученный html файл в браузере и посмотрите в консоль (F12, вкладка Консоль), то увидите там сообщение:



<script> element

Тег **<script>** может иметь атрибут **src**, который должен содержать URL файла с расширением .js. При указании **src** вместо кода, заключенного между открывающим(<script>) и закрывающим(</script>) тегами, браузер будет выполнять js код, который находится в указанном файле. Таким образом мы можем подключать к нашей html странице много скриптов:

```
<script src="script1.js"></script>
<script src="script2.js"></script>
<script src="script3.js"></script>
```

Также у тега **<script>** есть ещё два важных атрибута: **async** и **defer**.

Атрибут **async** устанавливает асинхронную загрузку скрипта, не блокирующую процесс загрузки страницы. Как только скрипт загрузится, браузер его выполнит. (Подключение большого скрипта без async приведёт к блокировке загрузки остальной части станицы. Пока скрипт не загрузится и не исполнится, браузер не перейдет к обработке следующего html элемента)

Атрибут **defer** откладывает выполнение скрипта до полной загрузки остальной страницы. Также при defer(в отличие от async) сохраняется порядок выполнения скриптов в соответствии с порядком их подключения в html файле.

Comments

Комментарии - это пометки в коде, которые игнорируются интерпретатором.

JavaScript поддерживает как однострочные, так и многострочные комментарии.

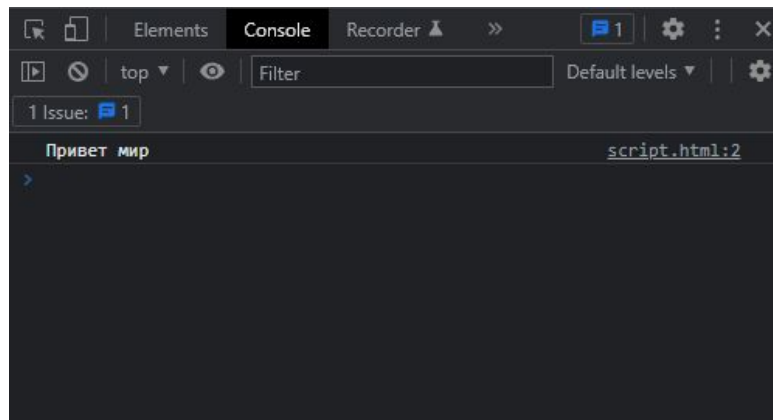
```
<script>

//    однострочный комментарий

/*
    Многострочный
    комментарий
*/

console.log("Привет мир");

</script>
```



Variables and literals

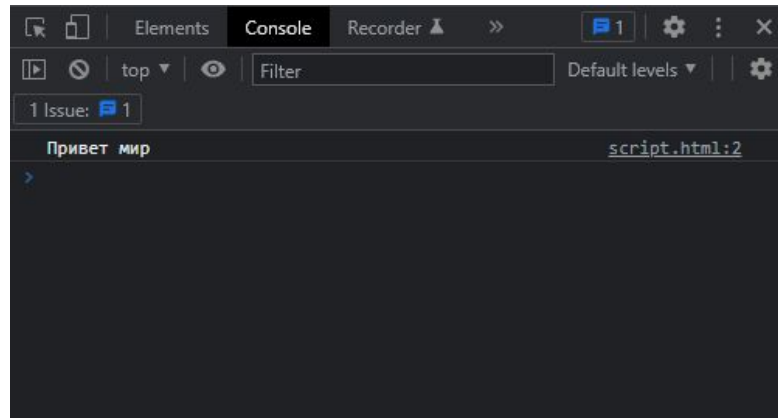
Скрипту часто бывает нужно какое-то время сохранять фрагменты информации, необходимые для выполнения задачи. Эти данные можно хранить в переменных.

Переменная – это «именованное хранилище» для данных. Для создания переменной в JavaScript используется ключевое слово **let**.

переменная

литерал

```
let greet;  
greet = "Привет мир";  
console.log(greet);
```



Variable initialization

Прежде чем пользоваться переменной, нам необходимо её объявить, указав после ключевого слова `let` имя новой переменной (Оно должно содержать только буквы, цифры или символы `$` и `_`. Цифра не может быть первой).

Затем с помощью оператора `"="` мы можем присвоить переменной конкретное значение.

```
let greet;
```

<- Объявление

```
greet = "Привет мир";
```

<- Присваивание

Одновременное объявление и присваивание называется инициализацией.

```
let greet = "Привет мир";
```

<- Инициализация (Объявление + присваивание)

const

С помощью ключевого слова **const** мы можем объявить немодифицируемую переменную. Такую переменную можно только проинициализировать при создании и нельзя переприсвоить.

```
> const pi = 3.14
    console.log(2*pi*100)
    628
< undefined
> pi = 3.16
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at <anonymous>:1:4
>
```

Переменные, объявленные через const, **нельзя переприсвоить**

```
> const pi
✖ Uncaught SyntaxError: Missing initializer in const declaration
>
```

Переменные, объявленные через const, **обязательно нужно проинициализировать**

Semicolons

Приложения на JavaScript состоят из инструкций с соответствующим синтаксисом (Весь список инструкций JS можно посмотреть [на MDN](#)). И так как в каждой инструкции присутствует определённое ключевое слово (или оператор), то инструкции нужно как-то разделять. Иначе интерпретатор просто не поймёт, где кончается одна инструкция и начинается вторая:

let greet1let greet2 Нужно добавить разделители → let greet1;
let greet2;

(интерпретатор видит объявление переменной “greet1let”, а затем неизвестный идентификатор “greet2”)

(В JS единичные инструкции разделяются точкой с запятой)

Вместо точек с запятой инструкции может разделять символ перехода на новую строку.

(Но возникают случаи, когда это не работает и опускание точки с запятой приводит к ошибке)

```
let greet;  
greet = "Привет мир";  
console.log(greet);
```

Точки с запятой в конце каждой инструкции в JS не обязательны

Data types

Data types

В JS есть 8 типов данных:

Number

`let a = 2`

Числа (как целые, так и дробные)

BigInt

`let a = 123123123123123123123123n`

Целые числа, которые могут превышать $2^{53} - 1$.
Создаются путем добавления "n" в конец
целочисленного литерала

String

`let a = "Привет мир"`

Строки

Boolean

`let a = false`

Логические значения: true и false

Null

`let a = null`

Специальный объект null, показывающий
намеренное отсутствие какого-либо
значения

Undefined

`let a` (или `let a = undefined`)

Специальный объект undefined. Он
автоматически присваивается переменным,
которым не было присвоено иного значения.

Object

`let a = {}`

Объекты - это особые структуры в JS,
похожие на словари (ключ-значение)

Symbol

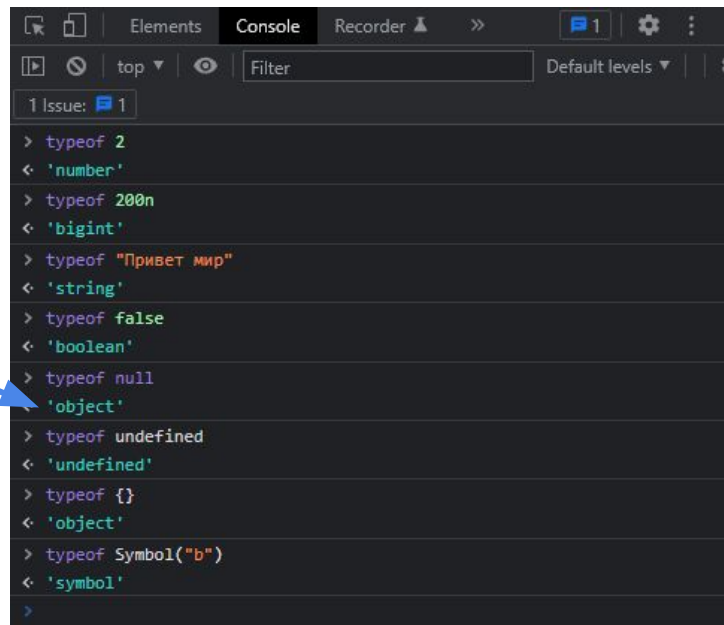
`let a = Symbol('b')`

Символы

Data types

Тип в JS определяется с помощью оператора **typeof**

Результатом вызова `typeof null` является "object". Это общепризнанная ошибка `typeof`, которую оставили для обратной совместимости. (null конечно не является объектом, это отдельный тип данных)



```
> typeof 2
< 'number'

> typeof 200n
< 'bigint'

> typeof "Привет мир"
< 'string'

> typeof false
< 'boolean'

> typeof null
< 'object'

> typeof undefined
< 'undefined'

> typeof {}
< 'object'

> typeof Symbol("b")
< 'symbol'
```

Basic operators

Arithmetic operators

Арифметические операторы работают только с численными значениями (как литералами, так и переменными)

<code>+, -, *, /</code>	сложение/вычитание/ умножение/деление	возвращает сумму/разность/произведение/частное двух операндов
<code>%</code>	остаток от деления	возвращает целочисленный остаток от деления двух операндов
<code>++</code>	инкремент	Добавляет единицу к операнду. При префиксном использовании возвращает значение операнда после прибавления единицы. При постфиксном использовании возвращает значение операнда до прибавления единицы.
<code>--</code>	декремент	вычитает единицу из операнда. Аналогичен инкременту
<code>-</code>	унарный минус	возвращает отрицание своего операнда.
<code>+</code>	унарный плюс	пытается преобразовать операнд в число. (Если операнд и так число, то оператор ничего не делает)
<code>**</code>	возведение в степень	возвращает возведение первого операнда в степень второго

Оператор “+” также может использоваться для конкатенации(склеивания строк)

Подробнее об арифметических операторах [на MDN](#)

Comparison operators

==	равно	возвращает true, если операнды равны
!=	не равно	возвращает true, если операнды не равны
===	строгое равно	возвращает true, если операнды равны и одного типа
!==	строгое не равно	возвращает true, если операнды не равны или разных типов
>	больше	возвращает true, если левый операнд больше правого
<	меньше	возвращает true, если левый операнд меньше правого
>=	больше и равно	возвращает true, если левый операнд больше правого или равен ему
<=	меньше или равно	возвращает true, если левый операнд меньше правого или равен ему

Подробнее о том, например, как операторы больше и меньше работают со строками можно посмотреть [на MDN](#)

== VS ===

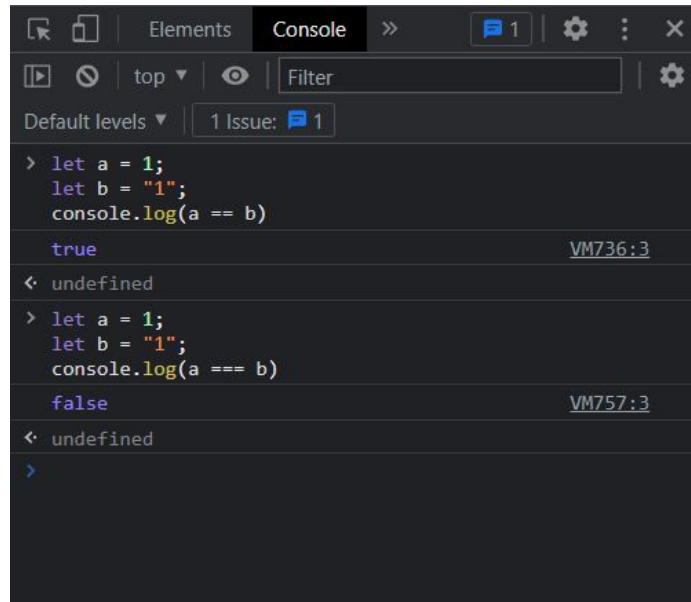
Разница между строгим сравнением и нестрогим:

Нестрогое.

“==” - сравнивает два операнда, предварительно пытаясь привести их к одному типу

Строгое.

“===” - сравнивает два операнда без приведения типов (если типы разные, то результат всегда **false**)



```
> let a = 1;
let b = "1";
console.log(a == b)
true VM736:3
< undefined

> let a = 1;
let b = "1";
console.log(a === b)
false VM757:3
< undefined
>
```

== VS ===

Оператор “==” выполняет преобразования типов данных перед сравнением. Это может привести к различным казусам и непредвиденным ошибкам, поэтому в современной разработке на JS *рекомендуется всегда использовать только “===”*.

На эту тему даже делали мемы в свое время.

Суть мема заключается в следующем:

Мы пытаемся сравнить `0 == "0"`. При сравнении числа со строкой оператор “==” сначала пытается привести строку к числу. Поэтому слева и справа получился просто ноль и они равны, т.е. результат `true`

При сравнении нуля с пустым массивом (`0 == []`) последний также приводится у числу ноль и результат `true`.

Тогда исходя из транзитивности логично предположить, что `"0" == []` тоже даст `true`. Но нет - пустой массив в этом случае приводится к строке “”, которая не равна “0”. Поэтому `false`.



Logical operators

&&	Логическое И	Возвращает true только если два оператора имеют значение true, иначе - false. (Если операторы не Boolean: Возвращает первый операнд, если он может быть преобразован к false; иначе возвращает второй операнд)
	Логическое ИЛИ	Возвращает false только если два оператора имеют значение false, иначе - true. (Если операторы не Boolean: Возвращает первый операнд, если он может быть преобразован к true; иначе возвращает второй операнд)
!	Логическое НЕ	Возвращает false, если оператор может быть преобразован к true; иначе - true

Lazy Evaluation:

Если `expr1 == false`, то `expr2` не вычисляется

`expr1 && expr2`

`expr1 || expr2`

Если `expr1 == true`, то `expr2` не вычисляется

Подробнее о логических операторах [на MDN](#).

Assignment operators

=	присваивание	Присваивает левому операнду значение правого. $x = f()$
+=	присваивание со сложением	$x += f()$ эквивалентно $x = x + f()$
-=	присваивание с вычитанием	$x -= f()$ эквивалентно $x = x - f()$
*=	присваивание с умножением	$x *= f()$ эквивалентно $x = x * f()$
/=	присваивание с делением	$x /= f()$ эквивалентно $x = x / f()$
%=	присваивание со взятием остатка	$x \% = f()$ эквивалентно $x = x \% f()$
**=	присваивание с возведением в степень	$x ** = f()$ эквивалентно $x = x ** f()$

Полный список всех операторов и их приоритет можно также посмотреть [на MDN](#).

Interaction with the user (not through the console)

alert(msg) - метод объекта window, вызывающий всплывающее окно вывода.

Принимает в качестве аргумента строку, которую надо вывести (если аргумент не является строкой, то JS сперва попыбует преобразовать его в строку). Возвращает undefined

prompt(msg) - метод объекта window, вызывающий всплывающее окно ввода.

Принимает в качестве аргумента строку, которую надо вывести в качестве пояснения над полем ввода (если аргумент не является строкой, то JS сперва попыбует преобразовать его в строку). Возвращает строку, которую пользователь ввёл в поле ввода (если пользователь закроет это окно, то возвращается null).

confirm(msg) - метод объекта window, вызывающий всплывающее окно подтверждения “да/нет”

Принимает в качестве аргумента строку, которую надо вывести в качестве пояснения над полем ввода (если аргумент не является строкой, то JS сперва попыбует преобразовать его в строку). Возвращает true, если пользователь нажал кнопку согласия и false, если нажал кнопку отклонения.

Interaction with the user (not through the console)

Все эти функции *работают синхронно*. То есть дойдя до одной из этих функций, JS будет ждать реакции пользователя. Только после того, как пользователь провзаимодействует с модальным окном, скрипт продолжит свое выполнение с того же места.

(Поэтому нужно помнить, что пока активно модальное окно, пользователь не может взаимодействовать с остальной страницей, а выполнение скрипта приостанавливается)

<Выполнение кода>

...

```
let name = prompt("Ваш никнейм: ")
```

```
alert("Привет, " + name)
```

...

<Ожидание ввода пользователя>

<Вывод имени пользователя
Ожидание нажатия OK>

<Продолжение выполнения кода>

This page says

Ваш никнейм:

OK Cancel



This page says

Привет, Artemon

OK

Conditional constructions

Conditional constructions: If Else

Условные конструкции позволяют управлять потоком выполнения скрипта и принимать различные решения на основе определённых условий.

Самой базовой конструкцией является if else:

```
if(условие) {  
    //код, если условие истинно  
} else {  
    //код, если условие ложно  
}
```

```
if(условие) {  
    //код, если условие истинно  
}  
//ветвь else может отсутствовать
```

```
let isAdult = confirm("Вам есть 18 лет?")  
if(isAdult === true) {  
    alert("Добро пожаловать")  
} else {  
    alert("Вы не можете пользоваться сайтом")  
}
```

This page says

Вам есть 18 лет?

OK

Cancel

This page says

Добро пожаловать

OK

This page says

Вы не можете пользоваться сайтом

OK

Conditional constructions: switch

Конструкция **switch case** позволяет выполнять различные инструкции в зависимости от значения переменной:

```
switch(переменная) {  
  case значение1: <инструкция 1> break;  
  case значение2: <инструкция 2> break;  
  ...  
  case значениеN: <инструкция N> break;  
  default: <инструкция default> break;  
}
```

тут может идти несколько инструкций, разделенных точкой с запятой. Они будут выполняться до тех пор, пока не встретится break;

Инструкции выполняемые при для различных значениях переменной

Инструкция, выполняемая если ни одна из других не подошла

```
let number = prompt("Введите номер дня недели")  
switch(number) {  
  case "1": alert("Понедельник"); break;  
  case "2": alert("Вторник"); break;  
  case "3": alert("Среда"); break;  
  case "4": alert("Четверг"); break;  
  case "5": alert("Пятница"); break;  
  case "6": alert("Суббота"); break;  
  case "7": alert("Воскресенье"); break;  
  default: alert("Указан неверный номер"); break;  
}
```

Сравнение происходит строгое "===", поэтому нужно указывать строки(т.к. prompt возвращает строку)

This page says

Введите номер дня недели

OK Cancel



This page says

Пятница

OK

Ternary Operator

Тернарный оператор проверяет условие и на его основе возвращает либо значение1, либо значение2:

```
let variable = (условие) ? <значение1> : <значение2>
```

сохраним значение возвращенное
тернарным оператором в переменную

условие

вернётся, если условие истинно

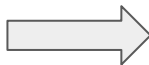
вернётся, если условие ложно

```
let number = Number(prompt("Введите число"))  
let answer = (number % 2 === 0) ? "Чётное" : "Нечетное"  
alert(answer)
```

This page says

Введите число

OK Cancel



This page says

Нечетное

OK

Functions

Functions

Функции являются одним из основных строительных блоков в JavaScript. Функция в JavaScript похожа на процедуру - набор инструкций, который выполняет задачу или вычисляет значение, но для того, чтобы процедура квалифицировалась как функция, она должна принимать некоторые входные данные и возвращать выходные данные.

```
function имяФункции (аргумент1, аргумент2, ..., аргументN) {  
    //набор инструкций, возвращающий значение с помощью команды return  
}
```

Возврат значения происходит с помощью команды **return**. Весь код, который идёт в функции после return, выполняться не будет, так как интерпретатор перейдет обратно к исполнению вызывающего кода.

Functions

Пример простейшей функции возвращающей квадрат числа:

```
function power2(num) {  
    return num*num  
}  
  
let number = Number(prompt("Введите число"))  
let answer = power2(number)  
alert(answer)
```

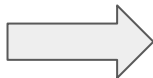
принимаем аргумент num и возвращаем значение num*num

Number() выполняет преобразование строки в число.
(Вместо него можно было использовать унарный плюс)

This page says

Введите число

OK Cancel



This page says

49

OK

Arrays and objects

Arrays

Массивы позволяют хранить упорядоченные коллекции.

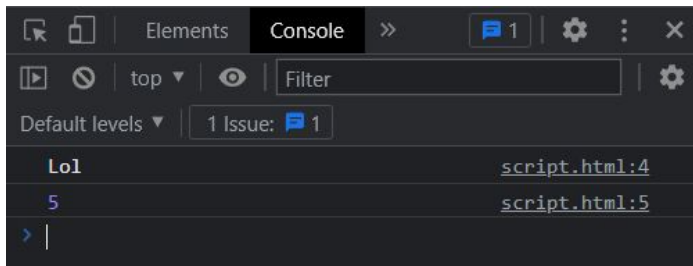
В отличие от многих языков, длина массива не зафиксирована. Также массив может хранить объекты разных типов одновременно.

Стандартный синтаксис

```
let arr = new Array(1,2,3,"Lol",true)
console.log(arr[3])
console.log(arr.length)
```

Упрощенный синтаксис

```
let arr =[1,2,3,"Lol",true]
console.log(arr[3])
console.log(arr.length)
```

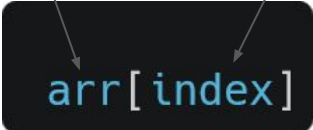


(При обращении к несуществующему индексу вернётся undefined)

Accessing array elements

Обращение к элементу массива:

имя массива индекс нужного элемента



`arr[index]`

Индексы начинаются с нуля

index: 0 1 2 3 4

```
> let arr = [100, 150, 200, 250, 300]

arr[3]

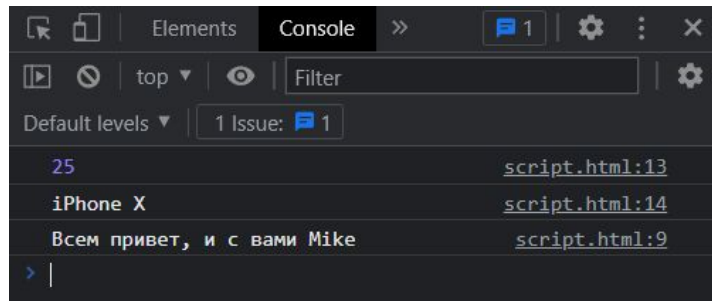
< 250
```

(кстати, со строками можно работать как с массивами символов)

Objects

Объекты в JS – особый тип данных, который используется для хранения коллекций различных значений и других сущностей.

```
let object = {  
  name: "Mike",  
  age: 25,  
  isStupid: true,  
  items: ["iPhone X", "Volkswagen Polo", "Brawl pass"],  
  sayHello: function() {  
    console.log("Всем привет, и с вами Mike")  
  }  
};  
  
console.log(object.age)  
console.log(object.items[0])  
object.sayHello()
```



Accessing object properties

Обращение к свойству объекта:

имя объекта имя нужного свойства



```
obj.property
```

Вызов функции, записанной в свойство объекта:

```
obj.property( )
```

```
> let obj = {  
  name: "Mike",  
  hello: function () {  
    console.log("Hello, I am Mike")  
  }  
}
```

```
obj.name  
< 'Mike'  
> obj.hello()  
Hello, I am Mike
```

Object as an associative array

```
let gradeDescriptions = {  
  A: "Отлично",  
  B: "Очень хорошо",  
  C: "Хорошо",  
  D: "Прикольно",  
  E: "Удовлетворительно",  
  F: "Неудовлетворительно"  
};  
  
let mark = prompt("Введите вашу оценку")  
alert( gradeDescriptions[mark] )
```

This page says

Введите вашу оценку

OK Cancel



This page says

Хорошо

OK

То есть **object.prop** эквивалентно **object["prop"]**

Operator “in”

Оператор `in` позволяет проверить, находится ли свойство в объекте, или цепочке его прототипов.

название свойства

объект

`"property" in obj`

- возвращает true/false

проверка нахождения свойств в объекте

свойства, наследуемые от прототипа,
также проверяются

```
> let obj = {  
  name: "Mike",  
  hello: function () {  
    console.log("Hello, I am Mike")  
  }  
}  
  
> "name" in obj  
< true  
> "hello" in obj  
< true  
> "lastname" in obj  
< false  
> "toString" in obj  
< true
```

Operator “?.”

Оператор “?.” позволяет безопасно обращаться к свойствам и методам объекта:

Мы ожидаем получить название книги, но что если в нашем объекте book окажется не задан

```
> let a = {  
  book: {  
    title: "The Adventures of Tom Sawyer"  
  }  
}  
  
a.book.title  
↵ 'The Adventures of Tom Sawyer'
```

Выпадает ошибка, останавливающая выполнение всего скрипта

```
> let a = {  
  book: null  
}  
  
a.book.title  
✖ ▶ Uncaught TypeError: Cannot read properties of null (reading 'title')  
   at <anonymous>:5:8
```

При обращении к свойствам несуществующего объекта через “?.”
возвращается undefined

a?.book?.title

аналогично

a && a.book && a.book.title

```
> let a = {  
  book: null  
}  
  
a?.book?.title  
↵ undefined
```

Operator “??”

Оператор ?? возвращает левый операнд, если он не null и undefined, иначе возвращает правый операнд.

`user = value ?? defaultValue`

аналогично

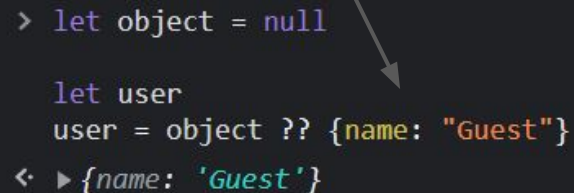
`user = (value !== null) ? value : defaultValue`

если object задан, то присваиваем его переменной user



```
> let object = {  
  name: "Mike"  
}  
  
let user  
user = object ?? {name: "Guest"}  
➤ {name: 'Mike'}
```

иначе присваиваем объект по умолчанию



```
> let object = null  
  
let user  
user = object ?? {name: "Guest"}  
➤ {name: 'Guest'}
```

Arrays and objects are passed by reference

Массивы и объекты **не копируются** при присваивании или передачи в функцию. Вместо них в новую переменную (или аргумент функции) записывается ссылка.

```
> let arr1 = [1,2,3,4,5]
let arr2 = arr1
```

Массивы и объекты копируются по ссылке

```
arr1[1] = 100
```

Если поменять первый массив, второй тоже изменится. Так arr1 и arr2 оба указывают на один и тот же массив.

```
arr2
```

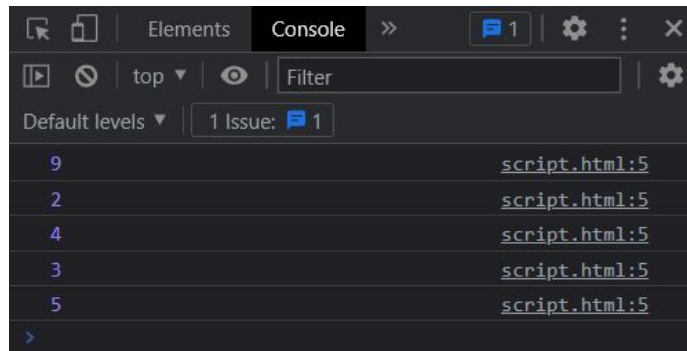
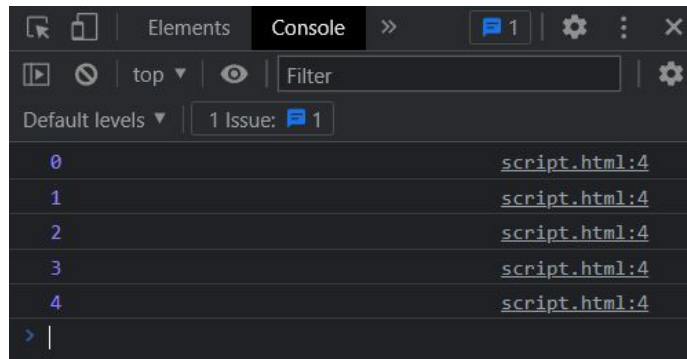
```
< ▶ (5) [1, 100, 3, 4, 5]
```


Loops

Loops: for

```
for(let i = 0; i<5; i++) {  
    console.log(i)  
}
```

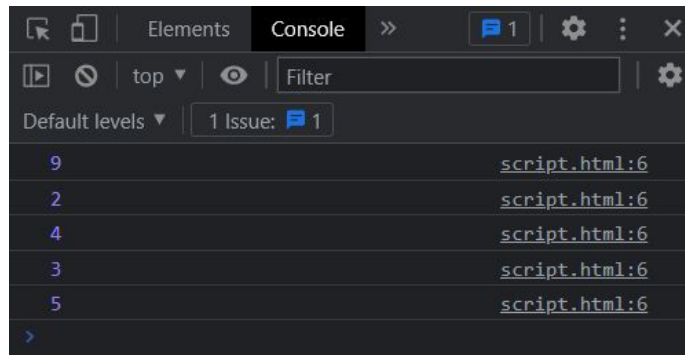
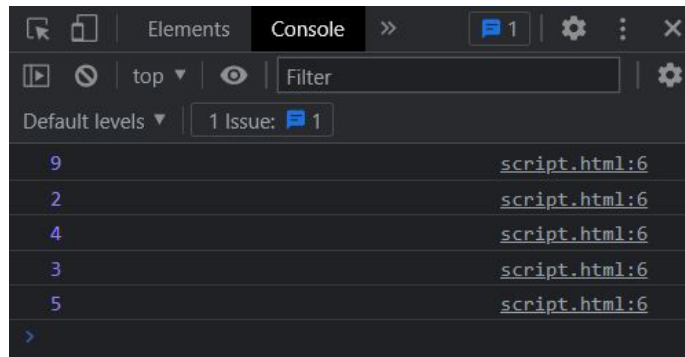
```
let a = [9,2,4,3,5]  
for(let i = 0; i < a.length; i++) {  
    console.log(a[i])  
}
```



Loops: while, do while

```
let a = [9,2,4,3,5]
let i = 0
while(a[i] !== undefined) {
  console.log(a[i])
  i++
}
```

```
let a = [9,2,4,3,5]
let i = 0
do {
  console.log(a[i])
  i++
} while(a[i] !== undefined)
```



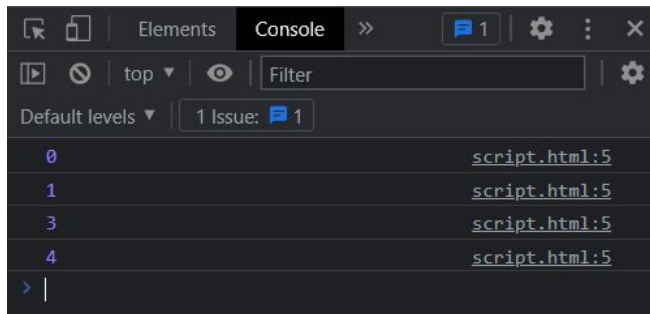
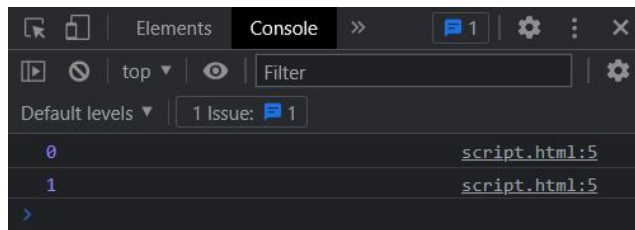
Interrupt loops

break; - полностью прерывает цикл

continue; - прерывает только текущую итерацию (и переходит к следующей)

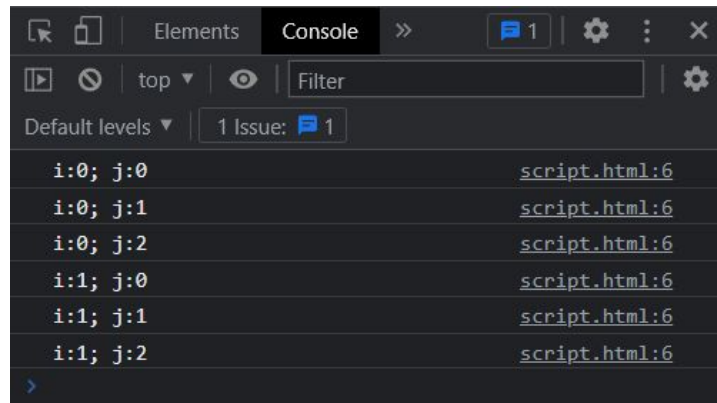
```
for(let i = 0; i<5; i++) {  
    if(i == 2) break;  
    console.log(i)  
}
```

```
for(let i = 0; i<5; i++) {  
    if(i == 2) continue;  
    console.log(i)  
}
```



Interrupting nested loops

```
myLabel: for(let i = 0; i<3; i++) {  
    for(let j = 0; j<3; j++) {  
        if(i == 2) break myLabel  
        console.log("i:" + i + "; j:" + j)  
    }  
}
```



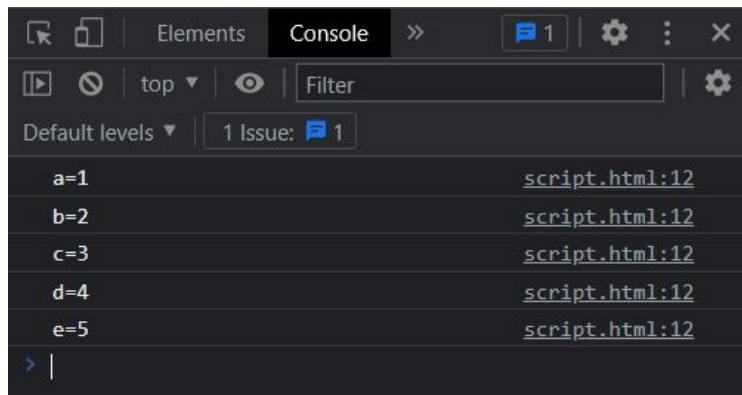
(“myLabel” - произвольное имя метки)

Loops: for in

цикл **for in** удобен для перебора свойств объекта

Выведем с его помощью всё содержимое объекта:

```
let object = {  
  a: 1,  
  b: 2,  
  c: 3,  
  d: 4,  
  e: 5  
};  
  
for(prop in object) {  
  console.log(prop + "=" + object[prop])  
}
```



(В случае с массивом, **for in** будет пробегать по индексам массива)

Loops: for of

Цикл **for of** предназначен для перебора значений итерируемых объектов (Array, Map, Set).

```
let arr = [100, 150, 200, 250, 300]

for(value of arr) {
  console.log(value)
}
```

100

150

200

250

300

> |

for of нельзя использовать с обычными (неитерируемыми) объектами

```
> let obj = {a:100, b:150, c:200, d:250, e:300}

for(value of obj) {
  console.log(value)
}
```

✖ ▶ Uncaught TypeError: obj is not iterable
at <anonymous>:3:14

Object methods, context

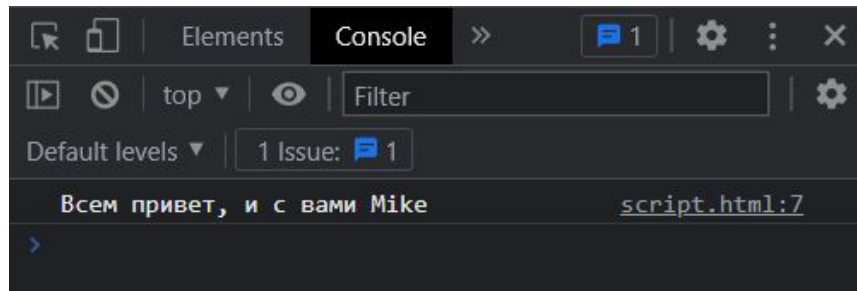
Methods

Мы уже видели пример, где функция записывалась в свойство объекта. Такие функции называют **методами**.

метод

```
let object = {  
  name: "Mike",  
  age: 25,  
  sayHello: function() {  
    console.log("Всем привет, и с вами Mike")  
  }  
};  
  
object.sayHello()
```

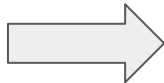
ВЫЗОВ МЕТОДА



Shortened syntax for creating methods

Существует более короткий синтаксис объявления методов внутри объекта:

```
let object = {  
  name: "Mike",  
  sayHello: function() {  
    console.log("Hello!")  
  }  
}
```

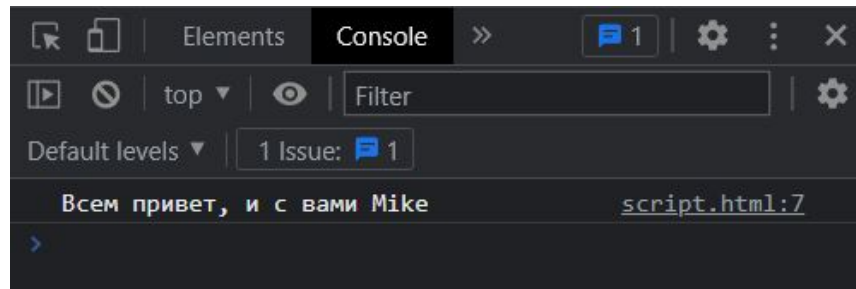



```
let object = {  
  name: "Mike",  
  sayHello() {  
    console.log("Hello!")  
  }  
}
```

Keyword this (context)

Каждый метод может получить доступ к свойствам объекта, к которому привязан, с помощью ключевого слова **this**.

```
let object = {  
  name: "Mike",  
  age: 25,  
  sayHello: function() {  
    console.log("Всем привет, и с вами " + this.name)  
  }  
};  
  
object.sayHello()
```



Built-in string methods

получение строки нижнего регистра (исходная строка не меняется)

получение строки верхнего регистра (исходная строка не меняется)

проверка наличия подстроки в строке

получение индекса начала подстроки в строке
(или -1 если подстроки в строке нет)

получение подстроки по индексу начального и конечного символа
(последний индекс не включается)
(если последний индекс не указан, то подстрока берется до конца строки)

разбиение строки на массив строк по разделителю

склейка массива строк в одну строку с вставкой разделителя

получение строки без пробельных символов в начале и конце

```
> let str = " Hello, world! "
< undefined

> str.toLowerCase()
< ' hello, world! '

> str.toUpperCase()
< ' HELLO, WORLD! '

> str.includes("world")
< true

> str.includes("js")
< false

> str.indexOf("world")
< 9

> str.indexOf("js")
< -1

> str.slice(2, 7)
< 'Hello'

> str.slice(7)
< ', world! '

> str.split(',')
< ▶ (2) [' Hello', ' world! ']

> [' Hello', ' world! '].join('|')
< ' Hello| world! '

> str.trim()
< 'Hello, world!'
```

Built-in number methods

Перед вызовом метода число нужно обернуть в скобки, чтобы интерпретатор не воспринимал точку как десятичный разделитель числа

получение экспоненциального вида числа

```
> (123).toExponential()
```

```
< '1.23e+2'
```

округление числа

```
> (123.12).toFixed()
```

```
< '123'
```

округление числа до определённой позиции после запятой

```
> (123.12).toFixed(1)
```

```
< '123.1'
```

приведение числа к виду, принятому в определенном регионе

```
> (1000000.5).toLocaleString("ru-RU")
```

```
< '1 000 000,5'
```

```
> (1000000.5).toLocaleString("en-IN")
```

```
< '10,00,000.5'
```

Array methods

Basic Array Methods

```
> let arr;  
  arr = [10, 20, 30]  
◀ ▶ (3) [10, 20, 30]
```

Добавляем элементы в конец

```
> arr.push(40, 50)  
◀ 5  
  
> arr  
◀ ▶ (5) [10, 20, 30, 40, 50]
```

Извлекаем элемент с конца

```
> arr.pop()  
◀ 50  
  
> arr  
◀ ▶ (4) [10, 20, 30, 40]
```

Добавляем элементы в начало

```
> arr.unshift(-20, -10, 0)  
◀ 7  
  
> arr  
◀ ▶ (7) [-20, -10, 0, 10, 20, 30, 40]
```

Извлекаем элемент с начала

```
> arr.shift()  
◀ -20  
  
> arr  
◀ ▶ (6) [-10, 0, 10, 20, 30, 40]
```

indexOf and includes methods

Метод **includes** проверяет, есть ли элемент в массиве.

Метод **indexOf** ищет элемент в массиве и возвращает его и индекс (или -1, если элемент не найден)

проверка, есть ли элемент в массиве

получение индекса нужного элемента
(если таких элементов несколько, то
берется первый индекс)

```
> let arr = [100, 150, 200, 250, 300]
arr.includes(150)
< true
> arr.includes(175)
< false
> arr.indexOf(150)
< 1
> arr.indexOf(175)
< -1
```


splice method

Метод **splice** позволяет удалить часть элементов из массива:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

С какого элемента
начать удаление*

Сколько элементов
удалить

Какие элементы
поместить вместо
вырезанных

возвращает вырезанную часть массива

*отрицательный индекс допустим и
означает отсчёт от конца массива

```
> let arr;  
arr = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
< ▶ (10) [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]  
> arr.splice(3, 5, 400, 500, 600, 700, 800)  
< ▶ (5) [40, 50, 60, 70, 80]
```

```
> arr  
< ▶ (10) [10, 20, 30, 400, 500, 600, 700, 800, 90, 100]  
>
```

Начиная с 3-го индекса вырезать 5 элементов и вставить новые элементы: 400, 500, 600, ...

reverse method

Метод **reverse** оборачивает порядок элементов в массиве

```
> let arr;  
  arr = [10, 20, 30, 40, 50]  
< ▶ (5) [10, 20, 30, 40, 50]  
  
> arr.reverse()  
< ▶ (5) [50, 40, 30, 20, 10]  
  
> arr  
< ▶ (5) [50, 40, 30, 20, 10]
```

Переворачивает массив и возвращает его

slice method

Метод **slice** копирует и возвращает часть массива

```
arr.slice([start], [end])
```

Начальный индекс

Конечный индекс
(не включается)

```
> let arr;  
   arr = [10, 20, 30, 40, 50]  
↵ ▶ (5) [10, 20, 30, 40, 50]  
  
> arr.slice(1, 4)  
↵ ▶ (3) [20, 30, 40]
```

concat method

Метод **concat** создаёт новый массив, в который помещает текущий массив, и добавляет к нему другие массивы или элементы



```
arr.concat(arg1, arg2...)
```

Исходный массив

Добавочные массивы/элементы

```
> let arr;
  arr = [10, 20, 30, 40, 50]
< ▶ (5) [10, 20, 30, 40, 50]

> arr.concat([1, 2, 3], [4, 5, 6], 7)
< ▶ (12) [10, 20, 30, 40, 50, 1, 2, 3, 4, 5, 6, 7]

> arr
< ▶ (5) [10, 20, 30, 40, 50]
```

split and join methods

Метод **split** разбивает строку на массив по заданному разделителю.

Метод **join** создает строку из элементов массива, вставляя между ними разделитель.

```
str.split(separator[, limit])
```

Разделитель

Максимальное число элементов
итогового массива (хвост
массива будет обрезаться)

```
arr.join(separator)
```

Разделитель

```
> "Artem, Dana, Mike, Rachel".split(",")  
◀ ▶ (4) ['Artem', ' Dana', ' Mike', ' Rachel']
```

```
> ["Artem", "Dana", "Mike", "Rachel"].join("|")  
◀ 'Artem|Dana|Mike|Rachel'
```

Array.isArray method

Статический метод **Array.isArray** помогает определить, является ли переданный объект массивом.

```
Array.isArray(obj)
```

Объект для проверки

Для `typeof` массивы и объекты неотличимы (возвращает и для тех и для других `'object'`)

```
> typeof []  
< 'object'  
  
> typeof {}  
< 'object'
```

`Array.isArray` позволяет различать массивы и объекты

```
> Array.isArray([])  
< true  
  
> Array.isArray({})  
< false
```