> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Грамматика и типы

В данной главе рассматриваются базовая грамматика, объявление переменных, типы данных и литералы.

Основы

JavaScript заимствует большую часть синтаксиса из Java, но также испытал влияние таких языков, как Awk, Perl и Python.

JavaScript чувствителен к регистру и использует кодировку символов Unicode. Например, слово Früh ("рано" по-немецки) может использоваться в качестве имени переменной.

```
JS
var Früh = "foobar";
```

Но, переменная früh не то же самое что Früh потому что JavaScript чувствителен к

И _

В JavaScript инструкции называются statements и разделяются точкой с запятой (;). Пробел (space), табуляция (tab) и перевод строки (newline) называются пробельными символами (whitespace). Исходный текст скриптов на JavaScript сканируется слева направо и конвертируется в последовательность входных элементов, являющихся токенами (tokens), управляющими символами, символами конца строки,

комментариями или пробельными символами. ECMAScript также определяет некоторые ключевые слова и литералы и устанавливает правила для автоматической вставки точек с запятой (ASI), чтобы обозначить конец инструкций (statements). Однако, рекомендуется всегда ставить точку с запятой в конце каждой инструкции вручную, чтобы избежать побочных эффектов. Чтобы получить более подробную информацию, прочитайте Lexical Grammar.

Комментарии

Синтаксис комментариев является таким же, как и в C++ и во многих других языках:

JS

```
// Комментарий, занимающий одну строку.

/* Комментарий,
   занимающий несколько строк.
 */

/* Нельзя вкладывать /* комментарий в комментарий */ SyntaxError */
```

Объявления

В JavaScript существует три вида объявлений:

var

Объявляет переменную, инициализация переменной значением является необязательной.

let

Объявляет локальную переменную в области видимости блока, инициализация переменной значением является необязательной.

const

Объявляет именованную константу, доступную только для чтения.

Переменные

Вы можете использовать переменные как символические имена для значений в вашем приложении. Имена переменных называются identifiers и должны соответствовать определённым правилам.

Идентификатор в JavaScript должен начинаться с буквы, нижнего подчёркивания (_) или знака доллара ($); последующие символы могут также быть цифрами (0-9). Поскольку JavaScript чувствителен к регистру, буквы включают символы от "A" до "Z" (верхний регистр) и символы от "a" до "z" (нижний регистр).

Вы можете использовать в идентификаторах буквы ISO 8859-1 или Unicode, например, å или ü. Вы также можете использовать управляющие последовательности Unicode как символы в идентификаторах.

Некоторые примеры корректных имён: `Number_hits`, `temp99`, `_name`.

Объявление переменных

Вы можете объявить переменную тремя способами:

- Используя ключевое слово `var`. Например, `var x = 42`. Данный синтаксис может быть использован для объявления как локальных, так и глобальных переменных.
- Просто присвоить переменной значение. Например, `x = 42`. Переменные, объявленные данным способом, являются глобальными. Такое объявление генерирует строгое предупреждение (`strict mode`). Не рекомендуется использовать данный способ.
- Используя ключевое слово `let`. Например, `let y = 13`. Данный синтаксис может быть использован для объявления локальной переменной в области видимости блока.

Присваивание значений

Переменная, объявленная через `var` или `let` без присвоения начального значения, имеет значение `undefined`.

При попытке доступа к необъявленной переменной или переменной до её объявления будет выброшено исключение `ReferenceError`:

JS

```
var a;
console.log("The value of a is " + a); //Значение переменной a undefined

console.log("The value of b is " + b); //Uncaught ReferenceError: b не определена

console.log("The value of c is " + c); //Значение переменной c undefined
var c;

console.log("The value of x is " + x); //Uncaught ReferenceError: x не определена
let x;
```

Вы можете использовать `undefined`, чтобы определить, имеет ли переменная значение. В следующем примере переменной `input` не присвоено значение, и оператор `if` будет вычислен как `true`:

JS

```
var input;
if (input === undefined) {
  doThis();
} else {
  doThat();
}
```

Значение `undefined` ведёт себя как `false`, когда используется в логическом контексте. Например, следующий код выполняет функцию `myFunction`, т.к. элемент `myArray` не определён:

JS

```
var myArray = [];
if (!myArray[0]) {
  myFunction();
}
```

Значение `undefined` конвертируется в `NaN`, когда используется в числовом контексте:

JS

```
var a;
a + 2; // NaN
```

Значение `null` ведёт себя как 0 в числовом контексте и как `false` в логическом контексте:

```
JS
var n = null;
console.log(n * 32); // В консоль выведется 0
```

Область видимости переменных

Когда вы объявляете переменную вне функции, то такая переменная называется *глобальной* переменной, т.к. доступна любому коду в текущем документе. Когда вы объявляете переменную внутри функции, то такая переменная называется *локальной* переменной, т.к. доступна только внутри данной функции.

До ECMAScript 6 в JavaScript отсутствовала область видимости блока; переменная, объявленная внутри блока, является локальной для _функции _(или *глобальной* области видимости), внутри которой находится данный блок. Например, следующий код выведет значение 5, т.к. областью видимости переменной `x` является функция (или глобальный контекст), внутри которой объявлена переменная `x`, а *не блок,* которым в данном случае является оператор `if`:

```
JS
if (true) {
  var x = 5;
}
console.log(x); // 5
```

Такое поведение меняется, если используется оператор `let`, введённый в ECMAScript 6:

```
JS
if (true) {
  let y = 5;
}
console.log(y); // ReferenceError
```

Поднятие переменных

Другим необычным свойством переменных в JavaScript является то, что можно сослаться на переменную, которая объявляется позже, и не получить при этом исключения. Эта концепция известна как `поднятие` (hoisting) переменных; переменные в JavaScript поднимаются в самое начало функции или выражения. Однако, переменные, которые ещё не были инициализированы, возвратят значение `undefined`:

JS

```
/*
 * Example 1
 */
console.log(x === undefined); // true
var x = 3;

/*
 * Example 2
 */
var myvar = "my value";

(function () {
  console.log(myvar); // undefined
  var myvar = "local value";
})();
```

Приведённые выше примеры будут интерпретироваться так же, как:

JS

```
/*
 * Example 1
 */
var x;
console.log(x === undefined); // true
x = 3;

/*
 * Example 2
 */
var myvar = "my value";

(function () {
  var myvar;
```

```
  console.log(myvar); // undefined
  myvar = "local value";
})();
```

Из-за поднятия переменных, все операторы `var` в функции следует размещать настолько близко к началу функции, насколько это возможно. Следование этому правилу улучшает ясность кода.

В ECMAScript 2015, `let (const)` не будет подниматься вверх блока. Однако, ссылки на переменную в блоке до объявления переменной вызовут `ReferenceError` . Переменная во "временной мёртвой зоне" в начале блока, до объявления.

JS

```
function do_something() {
  console.log(foo); // ReferenceError
  let foo = 2;
}
```

Поднятие функций

Для функций: только определения функций поднимаются наверх, но не функции, определённые через выражения.

JS

```
/* Определение функции */
foo(); // "bar"

function foo() {
  console.log("bar");
}

/* Определение функции через выражение */
baz(); // TypeError: baz is not a function

var baz = function () {
  console.log("bar2");
};
```

Глобальные переменные

Глобальные переменные на самом деле являются свойствами *глобального объекта*. На веб-страницах глобальным объектом является `window`, поэтому вы можете устанавливать глобальные переменные и обращаться к ним, используя синтаксис `window.variable`:

```
JS
window.foo = "bar";
```

Следовательно, вы можете обращаться к глобальным переменным, объявленным в одном объекте window или frame из другого объекта window или frame, указав имя window или frame. Например, если переменная `phoneNumber` объявлена в документе, то вы можете сослаться на эту переменную из iframe как `parent.phoneNumber`.

Константы

Вы можете создать именованную константу, доступную только для чтения, используя ключевое слово `const`. Синтаксис идентификатора константы является таким же, как и у идентификатора переменной: он должен начинаться с буквы, нижнего подчёркивания или знака $ и может содержать буквы, цифры или нижнее подчёркивание.

```
JS
const PREFIX = "212";
```

Нельзя изменить значение константы через присваивание или повторное объявление во время выполнения скрипта. Значение должно быть указано при инициализации.

Правила, касающиеся области видимости, для констант являются такими же, как и для переменных, объявленных через `let`. Если ключевое слово `const` не указано, то идентификатор будет являться переменной.

Нельзя объявить константу с таким же именем, как у функции или переменной в одной области видимости. Следующие примеры выбросят исключение `TypeError`:

```
JS
```

```
// Это вызовет ошибку
function f() {}
const f = 5;

// Это тоже вызовет ошибку
function f() {
  const g = 5;
  var g;

  // какие-то выражения
}
```

Однако, атрибуты объектов не защищены, так что следующее выражение выполнится без проблем

```
JS
const MY_OBJECT = { key: "value" };
MY_OBJECT.key = "otherValue";
```

Структуры и типы данных

Типы данных

Последний стандарт ECMAScript определяет семь типов данных:

- Шесть типов данных, которые являются примитивами:
  - Boolean. `true` и `false`.
  - null. Специальное ключевое слово, обозначающее нулевое или «пустое» значение. Поскольку JavaScript чувствителен к регистру, `null` не то же самое, что `Null`, `NULL` или любой другой вариант.
  - undefined. Свойство глобального объекта; переменная, не имеющая присвоенного значения, обладает типом `undefined`.
  - Number. `42` или `3.14159`.
  - String. "Howdy".
  - Symbol (ECMAScript 6)
- и Object

Хотя типов данных относительно немного, но они позволяют вам выполнять полезные функции в ваших приложениях. Объекты и функции являются другими фундаментальными элементами языка. Вы можете думать об объектах как об именованных контейнерах для значений и о функциях как о процедурах, которые ваше приложение может исполнять.

Преобразование типов данных

JavaScript — это динамически типизированный язык. Это означает, что вам не нужно указывать тип данных переменной, когда вы её объявляете, типы данных преобразуются автоматически по мере необходимости во время выполнения скрипта. Так, например, вы можете определить переменную следующим образом:

```
JS
var answer = 42;
```

А позже вы можете присвоить этой переменной строковое значение, например:

```
JS
answer = "Thanks for all the fish...";
```

Поскольку JavaScript является динамически типизированным, это присваивание не вызовет сообщения об ошибке.

В выражениях, включающих числовые и строковые значения с оператором `+`, JavaScript преобразует числовые значения в строковые. Например:

```
JS
x = "The answer is " + 42; // "The answer is 42"
y = 42 + " is the answer"; // "42 is the answer"
```

В выражениях с другими операторами JavaScript не преобразует числовые значения в строковые. Например:

```
JS
```

```
"37" - 7; // 30
"37" + 7; // "377"
```

Преобразование строк в числа

В том случае, если значение, представляющее число, хранится в памяти как строка, можно использовать методы для преобразования строк в числа:

- `parseInt(numString, [radix])`
- `parseFloat(numString)`

`parseInt` преобразует строку в целочисленное значение. Хорошей практикой является всегда указывать основание системы счисления (параметр `radix`).

Альтернативным способом для получения числа из строки является использование оператора "унарный плюс":

JS

```
"1.1" + "1.1"; // "1.11.1"
(+"1.1") + (+"1.1"); // 2.2
// Обратите внимание на то, что скобки не являются обязательными и используются для ясности.
```

Литералы

Литералы используются для представления значений в JavaScript. Они являются фиксированными значениями, а не переменными. В данной секции рассматриваются следующие типы литералов:

- Литерал массива
- Логический литерал
- Литерал целого числа
- Литерал числа с плавающей точкой
- Литерал объекта
- RegExp литерал
- Литерал строки

Литерал массива

Литерал массива — это список из нуля или более выражений, каждое из которых представляет элемент массива, заключённый в квадратные скобки (`[]`). Когда вы создаёте массив, используя литерал массива, он инициализируется с помощью переданных значений, которые будут являться его элементами, длина массива будет равна числу переданных аргументов.

В следующем примере создаётся массив `coffees` с тремя элементам и длиной, равной трём:

```
JS
var coffees = ["French Roast", "Colombian", "Kona"];
```

> **Примечание:** Обратите внимание на то, что литерал массива является инициализатором объекта. Чтобы получить более подробную информацию, прочитайте Использование инициализаторов объекта.

Если массив создаётся с помощью литерала в скрипте верхнего уровня, то JavaScript интерпретирует массив каждый раз, когда вычисляет выражение, содержащее литерал. Кроме того, литерал, используемый в функции, создаётся каждый раз, когда вызывается функция.

Литералы массива также являются объектами `Array` . Чтобы получить более подробную информацию, прочитайте `Array` и упорядоченные наборы данных.

Лишние запятые в литералах array

Не обязательно указывать все элементы в литерале array. Если вы поставите две запятые подряд, то пропущенные элементы будут иметь значение `undefined` . Например:

```
JS
var fish = ["Lion", , "Angel"]; // ["Lion", undefined, "Angel"]
```

У этого массива есть 2 элемента со значениям и один пустой (`fish[0]` - "Lion", `fish[1]` - `undefined` , а `fish[2]` - "Angel").

Если вы поставите запятую в конце списка элементов, то она будет проигнорирована. В следующем примере, длина массива равна 3. Нет `myList[3]` . Все другие запятые в списке говорят о новом элементе.

> Примечание: Лишние запятые могут вызывать ошибки в старых версиях браузеров, поэтому лучше избегать их использования.

```
JS
var myList = ["home", , "school", ]; // ["home", undefined, "school"]
```

В следующем примере длина массива равна четырём, элементы `myList[0]` и `myList[2]` имеют значение `undefined` :

```
JS
var myList = [, "home", , "school"]; // [undefined, "home", undefined, "school"]
```

В следующем примере длина массива равна четырём, элементы `myList[1]` и `myList[3]` имеют значение `undefined` . Игнорируется только последняя запятая.

```
JS
var myList = ["home", , "school", ,]; // ["home", undefined, "school", undefined]
```

Понимание поведения лишних запятых важно для понимания JavaScript как языка. Однако, когда будете писать свой собственный код, то имейте в виду, что явное объявление отсутствующих элементов как `undefined` улучшает ясность кода и лёгкость его поддержки.

Логические литералы

Логический (Boolean) тип имеет два литеральных значения: `true` и `false` .

Не путайте примитивные логические значения `true` и `false` со значениями true и false объекта Boolean. Объект Boolean является объектом-обёрткой над примитивом логического типа. Чтобы получить более подробную информацию, прочитайте `Boolean`.

Литерал целого числа

Целые числа могут быть записаны в десятичной, шестнадцатеричной, восьмеричной и двоичной системах счисления.

- Десятичный целочисленный литерал состоит из последовательности цифр без ведущего нуля.
- Ведущий ноль в целочисленном литерале указывает на то, что он записан в восьмеричной системе счисления. Восьмеричные целые числа состоят только из цифр 0-7.
- Ведущие символы 0x (или 0X) указывают на то, что число шестнадцатеричное. Шестнадцатеричные целые числа могут состоять из цифр 0-9 и букв a-f и A-F.
- Ведущие символы 0b (или 0B) указывают на то, что число двоичное. Двоичные числа могут включать в себя только цифры 0 и 1.

Несколько примеров целочисленных литералов:

```
0, 117 и -345 (десятичная система счисления)
015, 0001 и -077 (восьмеричная система счисления)
0x1123, 0x00111 и -0xF1A7 (шестнадцатеричная система счисления)
0b11, 0b0011 и -0b11 (двоичная система счисления)
```

Для дополнительной информации смотрите числовые литералы в лексической грамматике.

Литерал числа с плавающей точкой

Числа с плавающей точкой могут состоять из следующих частей:

- Десятичное целое число, которое может иметь знак (символ "+" или "-", стоящий перед числом),

- Десятичная точка ("."),
- Дробная часть (другое десятичное число),
- Экспонента.

Экспонента состоит из символа "e" или "E", за которым следует целое число, которое может иметь знак. Число с плавающей точкой должно состоять по крайней мере из одной цифры и либо десятичной точки, либо символа "e" (или "E").

В более сжатой форме синтаксис выглядит следующим образом:

```
[(+|-)][digits][.digits][(E|e)[(+|-)]digits]
```

Примеры:

```
3.14
-3.1E+12
-.3333333333333333
.1e-23
```

Литерал объекта

Литерал объекта — это список из нуля или более пар, состоящих из имён свойств и связанных с ними значений, заключённый в фигурные скобки (`{}`). Вам не следует использовать литерал объекта в начале выражения, т.к. это приведёт к ошибке или к поведению, которого вы не ожидаете, потому что символ "{" будет интерпретироваться как начало блока.

В следующем примере свойству `myCar` объекта `car` присваивается строка `"Saturn"`, свойству `getCar` — результат вызова функции `CarTypes("Honda")`, свойству `special` — значение переменной `Sales`:

JS

```
var Sales = "Toyota";

function CarTypes(name) {
  if (name == "Honda") {
    return name;
```

```
  } else {
    return "Извините, мы не продаём " + name + ".";
  }
}

var car = { myCar: "Saturn", getCar: CarTypes("Honda"), special: Sales };

console.log(car.myCar); // Saturn
console.log(car.getCar); // Honda
console.log(car.special); // Toyota
```

Кроме того, вы можете использовать числовой или строковой литералы в именах свойств или вкладывать один объект в другой. Например:

JS

```
var car = { manyCars: { a: "Saab", b: "Jeep" }, 7: "Mazda" };

console.log(car.manyCars.b); // Jeep
console.log(car[7]); // Mazda
```

Именем свойства объекта может быть любая строка, в том числе пустая строка. Если имя свойства не является корректным JavaScript идентификатором, то оно должно быть заключено в кавычки. Для обращения к таким именам следует использовать квадратные скобки (`[]`), а не точку (`.`):

JS

```
var unusualPropertyNames = {
  "": "An empty string",
  "!": "Bang!"
}
console.log(unusualPropertyNames.""); // SyntaxError: Unexpected string
console.log(unusualPropertyNames[""]); // "An empty string"
console.log(unusualPropertyNames.!); // SyntaxError: Unexpected token !
console.log(unusualPropertyNames["!"]); // "Bang!"
```

В ES2015 литералы объектов расширены до поддержки установки прототипа в конструкции короткой записи для `foo:` задание `foo` , определение методов, создание супер вызовов и вычисление имён свойств в выражениях. Вместе, они

делают похожими объектные литералы и объявления классов, а также позволяют объектному дизайну получать выгоду одинаковых возможностей.

```
JS
var obj = {
  // __proto__
  __proto__: theProtoObj,
  // Короткая запись для ‘handler: handler’
  handler,
  // Методы
  toString() {
    // Супер вызовы
    return "d " + super.toString();
  },
  // Динамическое вычисление имён свойств
  ["prop_" + (() => 42)()]: 42,
};
```

Обратите внимание на следующий пример:

```
JS
var foo = { a: "alpha", 2: "two" };
console.log(foo.a); // alpha
console.log(foo[2]); // two
// console.log(foo.2);            // SyntaxError: Unexpected number
// console.log(foo[a]);           // ReferenceError: a is not defined
console.log(foo["a"]); // alpha
console.log(foo["2"]); // two
```

RegExp литерал

Литерал regexp - шаблон между слешами. Следующий пример литерал regex:

```
JS
var re = /ab+c/;
```

Строковый литерал

Строковый литерал — это ноль или более символов, заключённых в двойные (") или одинарные (') кавычки. Строка должна быть ограничена кавычками одного

типа, т.е. либо обе одинарные, либо обе двойные. Например:

```
"foo"
'bar'
"1234"
"one line \n another line"
"John's cat"
```

Вы можете вызвать любой из методов объекта `String` для строкового литерала: JavaScript автоматически преобразует строковой литерал во временный объект `String`, вызовет метод, а затем уничтожит временный объект `String`. Вы также можете использовать свойство `String.length` со строковым литералом:

JS

```
console.log("John's cat".length);
// Число символов в строке, включая пробел.
// В данном случае длина строки равна 10.
```

В ES2015 также доступны шаблоны строк. Шаблоны строк представляют собой синтаксический сахар для конструирования строк. Это похоже на возможности интерполяции строк в Perl, Python и других. Дополнительно, может быть добавлен тег, позволяющий настраивать конструирование строк, избегая атак внедрения и построения структур данных высокого уровня из содержимого строки.

JS

```
// Простое создание строки через литерал string
`In JavaScript '\n' is a line-feed.`;

// Мультистроковые строки
`In JavaScript this is
 not legal.`;

// Интерполяция строк
var name = "Бобби",
  time = "сегодня";
`Привет ${name}, как ты ${time}?`;

// Строим префикс HTTP запроса, используемый для интерпретации замен и конструирования
POST`http://foo.org/bar?a=${a}&b=${b}
```

```
    Content-Type: application/json
    X-Credentials: ${credentials}
    { "foo": ${foo},
      "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

Вам следует использовать строковые литералы до тех пор, пока вам специально не понадобится объект `String`. Чтобы получить более подробную информацию об объекте `String`, прочитайте `String`.

Использование специальных символов в строках

Кроме обычных символов вы также можете включать специальные символы в строки.

JS

```
"one line \n another line";
```

В следующей таблице перечислены специальные символы, которые вы можете использовать.

Символ	Значение
`\b`	Возврат (Backspace)
`\f`	Перевод или прогон страницы (Form feed)
`\n`	Перевод строки (New line)
`\r`	Возврат каретки (Carriage return)
`\t`	Табуляция (Tab)
`\v`	Вертикальная табуляция (Vertical tab)
`\'`	Апостроф или одинарная кавычка
`\"`	Двойная кавычка
`\\`	Обратная косая черта (Backslash)
`\XXX`	Символ в кодировке Latin-1, представленный тремя восьмеричными числами *XXX* от 0 до 377. Например, \251 (символ ©).

Символ	Значение
`\xXX`	Символ в кодировке Latin-1, представленный двумя шестнадцатеричными числами *XX* от 00 до FF. Например, \xA9 (символ ©).
`\uXXXX`	Символ в Unicode, представленный четырьмя шестнадцатеричными числами *XXXX*. Например, \u00A9 (символ ©).
`\u{XXXXX}`	Символ в UTF-32BE. Например, \u{2F804} обозначает то же, что обычная запись \uD87E\uDC04.

Экранирующие символы

Для символов, не перечисленных в вышеприведённой таблице, предваряющая обратная косая черта игнорируется. Такое использование не является рекомендованным (deprecated) и вам следует избегать его.

Вы можете вставить кавычку в строку, если поставите перед ней обратную косую черту. Это называется экранированием кавычек. Например:

```
JS
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service.";
console.log(quote); // He read "The Cremation of Sam McGee" by R.W. Service.
```

Чтобы включить обратную косую черту в строку, перед ней нужно поставить ещё одну обратную косую черту. Например:

```
JS
var home = "c:\\temp"; // c:\temp
```

Вы также можете экранировать перевод строки. Обратная косая черта и перевод строки будут удалены из содержимого строки. Например:

```
JS
var str =
  "this string \
is broken \
across multiple\
```

```
lines.";
console.log(str); // this string is broken across multiplelines.
```

Хотя JavaScript не поддерживает синтаксис "heredoc" (форматированный текст в одной строковой переменной), но вы можете эмулировать его, добавив перевод строки и обратную косую черту в конец каждой строки:

JS

```
var poem =
  "Roses are red,\n\
Violets are blue.\n\
I'm schizophrenic,\n\
And so am I.";
```

Дополнительная информация

Данная глава сфокусирована на базовом синтаксисе для объявлений и типов. Чтобы получить более подробную информацию о конструкциях JavaScript, прочитайте:

- Порядок выполнения и обработка ошибок
- Циклы и итерация
- Функции
- Выражения и операторы

В следующей главе рассматриваются управляющие конструкции и обработка ошибок.

This page was last modified on 10 авг. 2023 г. by MDN contributors.