JSDG 🦏 | Intro; Lexical Structure; Types, Values and Variables | chapters 1-3

`JavaScript: The Definitive Guide 7th EDITION • Master the World's Most-Used Programming Language • David Flanagan • 2021`

Дэвид Флэнаган занимается программированием и пишет о JavaScript, начиная с 1995 года. Он живет с женой и детьми на Тихоокеанском Северо-Западе между городами Сиэтл и Ванкувер, Британская Колумбия. Дэвид получил диплом в области компьютерных наук и инженерии в Массачусетском технологическом институте и работает инженером-программистом в VMware.

—

На протяжении большей части **2010**-х годов все веб-браузеры поддерживали версию **5 стандарта ECMAScript.**

В этой книге ES5 рассматривается как отправная точка совместимости и предшествующие ей версии языка больше не обсуждаются.

Стандарт **ES6** был выпущен в **2015** году и обзавелся важными новыми средствами,
включая синтаксис **классов и модулей**, которые превратили JavaScript из языка написания сценариев в серьезный универсальный язык, подходящий для крупномасштабной разработки ПО.

Начиная с ES6, спецификация ECMAScript перешла на ежегодный ритм выпусков и теперь версии языка —

ES2016,
ES2017,
ES2018,
ES2019 и ES2020 — идентифицируются по году выхода.

—

Литерал — это значение данных, находящееся прямо в программе. Все ниже перечисленное представляет собой

литералы

12 // Число двенадцать
1.2 // Число одна целая и две десятых
"hello world" // Строка текста
"HiR // Еще одна строка
true / / Булевское значение
false / Другое булевское значение
null // Отсутствие объекта

___

Идентификатор — это просто имя.

Идентификаторы в JavaScript применяются для именования **констант**, **переменных**, **свойств**, **функций** и **классов**, а также для того, чтобы **снабдить метками некоторые циклы в коде** JavaScript.

Идентификатор JavaScript должен начинаться с **буквы, подчеркивания (_) или знака доллара ($)**.
например - **vl3, _dummy, $str** etc

___

Зарезервированные слова

as	const	export	get	null	target	void
async	continue	extends	if	of	this	while
await	debugger	false	import	return	throw	with
break	default	finally	in	set	true	yield
case	delete	for	instanceof	static	try	
catch	do	from	let	super	typeof	
class	else	function	new	switch	var	

В JavaScript также зарезервированы или ограничены в применении определенные ключевые слова,
которые в текущее время не используются языком, но могут быть задействованы в будущих версиях:

enum implements interface package private protected public

—

Программы JavaScript пишутся с использованием набора символов Unicode и вы можете применять любые символы Unicode в строках и комментариях.

Для переносимости и легкости редактирования в идентификаторах принято исполь зовать только буквы ASCII и цифры.

Но это лишь соглашение в программировании, а сам язык допускает наличие в идентификаторах букв, цифр и идеограмм Unicode (но не эмотиконов).

—

Необязательные точки с запятой

let a = 1, b = 2;
(a+b).toString(); -> 2 is not a function

let a = 1, b = 2
;(a+b).toString(); -> '3'

—

3.Типы, значения и переменные

Одной из наиболее фундаментальных характеристик языка программирования является набор поддерживаемых типов.

Когда программе необходимо запомнить значение с целью использования в будущем, она присваивает значение переменной (или “сохраняет” его в ней).

Переменные имеют имена, которые можно употреблять в программах для ссылки на значения.

—

Типы JavaScript можно разделить на две категории:

элементарные типы* и *объектные типы.

Элементарные типы JavaScript включают **числа, строки текста (называемые просто строками) и булевские истинностные значения (называемые просто булевскими)**.

Специальные величины **null и undefined** в JavaScript относятся к элементарным значениям, но не являются числами, строками или булевскими значеними.

Каждое значение обычно считается единственным членом собственного особого типа.

В ES6 добавлен новый специализированный тип, известный как символ (Symbol), который делает возможным определение языковых расширений, не причиняя вреда обратной совместимости

——

Любое значение JavaScript, которое отличается от **числа, строки, булевского значения, символа**, **null** или **undefined**, **представляет собой объект.**

т.е. член типа Object — это коллекция *свойств*, где каждое свойство имеет имя и значение (либо элементарное значение, либо другой объект).

——

Обыкновенный объект JavaScript является неупорядоченной коллекцией именованных значений. В языке также определен специальный вид объекта, называемый массивом, который представляет упорядоченную коллекцию про нумерованных значений.

——

Вдобавок к базовым объектам и массивам в JavaScript определено несколько других полезных объектных типов.

Объект Set представляет множество значений.

Объект Map представляет отображение ключей на значения.

Разнообразные типы “типизированных массивов” облегчают операции на массивах байтов и других двоичных данных.

Тип RegExp представляет текстовые шаблоны и делает возможными сложно устроенные операции сопоставления, поиска и замены на строках.

Тип Date представляет дату и время плюс поддерживает элементарную арифметику с датами. Тип Error и его подтипы представляют ошибки, которые могут возникать при выполнении кода JavaScript.

___

Язык JavaScript поддерживает стиль объектно-ориентированного программирования.

В широком смысле это значит, что вместо наличия глобально определенных функций для оперирования значениями различных типов типы самостоятельно определяют методы для работы со значениями.

Скажем, что бы отсортировать элементы массива a, мы не передаем a в функцию sort(). Взамен мы вызываем метод sort() массива a:

a.sort(); // Объектно-ориентированная версия sort(a)

___

Формально методы имеют толь ко объекты JavaScript. Но числовые, строковые, булевские и символьные значения ведут себя так, будто они располагают методами.

В JavaScript нельзя вызывать методы лишь на значениях null и undefined.

___

Объектные типы JavaScript являются *изменяемыми*,

элементарные типы — *не изменяемыми*.

Строки можно рассматривать как массивы символов и ожидать от них изменяемости.

Однако строки в JavaScript неизменяемы: вы можете получать доступ к тексту по любому индексу строки, но нет какого-либо способа модификации текста существующей строки.

—

JavaScript свободно преобразует значения из одного типа в другой.

Скажем, если программа ожидает строку, а вы предоставляете ей число, то произойдет автоматическое преобразование числа в строку.

И если вы используете не булевское значение там, где ожидается булевское, тогда JavaScript соответствующим образом преобразует его.

—

Либеральные правила преобразования значений JavaScript влияют на определение равенства, и операция равенства == выполняет преобразования типов

Вместо операции равенства == рекомендуется применять операцию строгого равенства ===, которая не делает никаких преобразований типов.

—

Основной числовой тип JavaScript, Number, служит для представления целых чисел и аппроксимации вещественных чисел. Числа в JavaScript представляются с применением 64-битного формата с плавающей точкой, определенного стан дартом ШЕЕ 754,1т.е. он способен представлять числа в диапазоне от ±5x10-324 До ±1.7976931348623157 x Ю308

Формат для чисел типа double в Java, C++ и большинстве

современных языков программирования.

Когда число находится прямо в программе JavaScript, оно называется *числовым литералом*.

___

Разделители в числовых литералах

В длинных числовых литералах можно применять подчеркивания, чтобы разбивать их на порции, которые легче для восприятия:

let billion = 1_000_000__000; // Подчеркивание как разделитель тысяч,
let bytes = 0x89_AB_CD_EF; // Подчеркивание как разделитель байтов,
let bits=0b0001_1101_0111; // Подчеркивание как разделитель полубайтов
let fraction = 0.123__456_789; // Работает и в дробной части.

На момент написания главы в начале 2020 года подчеркивания в числовых литералах еще не были формально стандартизированы в качестве части JavaScript. Но они находятся на поздней фазе процесса стандартизации и внедрены во всех основных браузерах и Node.

___

В ES2016 добавляется ** для возведения в степень.

___

```
Math.pow(2,53)     // => 9007199254740992: 2 в степени 53
Math.round(.6)     // => 1.0: округляет до ближайшего целого
Math.ceil(.6)      // => 1.0: округляет в большую сторону до целого
Math.floor(.6)     // => 0.0: округляет в меньшую сторону до целого
Math.abs(-5)       // => 5: абсолютная величина
Math.max(x,y,z)    // Возвращает наибольший аргумент
Math.min(x,y,z)    // Возвращает наименьший аргумент
Math.random()      // Псевдослучайное число x, где 0 <= x < 1.0
Math.PI            // π: длина окружности, деленная на диаметр
Math.E             // e: основание натурального логарифма
Math.sqrt(3)       // => 3**0.5: квадратный корень из 3
Math.pow(3, 1/3)   // => 3**(1/3): кубический корень из 3
Math.sin(0)        // Тригонометрия: также есть Math.cos, Math.atan и т.д.
Math.log(10)       // Натуральный логарифм 10
Math.log(100)/Math.LN10    // Десятичный логарифм 100
Math.log(512)/Math.LN2     // Двоичный логарифм 512
Math.exp(3)                // Math.E в кубе
```

В ES6 определены дополнительные функции объекта Math:

```
Math.cbrt(27)      // => 3: кубический корень
Math.hypot(3, 4)   // => 5: квадратный корень из суммы квадратов
                   //       всех аргументов
Math.log10(100)    // => 2: десятичный логарифм
Math.log2(1024)    // => 10: двоичный логарифм
Math.log1p(x)  // Натуральный логарифм (1+x); точен для очень малых x
Math.expm1(x)  // Math.exp(x)-1; инверсия Math.log1p()
Math.sign(x)   // -1, 0 или 1 для аргументов <, == или > 0
Math.imul(2,3)// => 6: оптимизированное умножение 32-битных целых чисел
Math.clz32(0xf)    // => 28: количество ведущих нулевых бит
                   //        в 32-битном целом числе
Math.trunc(3.9)    // => 3: преобразует в целое число,
                   //       отбрасывая дробную часть
Math.fround(x)     // Округляет до ближайшего 32-битного числа
                   // с плавающей точкой
Math.sinh(x)       // Гиперболический синус.
                   // Также есть Math.cosh(), Math.tanh()
Math.asinh(x)      // Гиперболический арксинус.
                   // Также есть Math.acosh(), Math.atanh()
```

Деление на ноль в JavaScript ошибкой не является: просто возвращается бесконечность или отрицательная бесконечность.

Тем не менее, существует одно исключение: ноль, деленный на ноль, не имеет четко определенного значения, и результатом такой операции будет особое значение “не

число” (not-a-number), NaN

Значение “не число” в JavaScript обладает одной необычной особенностью: оно не равно никакому другому значению, включая самого себя.

Отсюда следует, что вы не можете записать **x === NaN,** чтобы выяснить, имеет ли переменная **x** значение **NaN.** Взамен вы должны записывать **x** != **x** или **Number. isNaN (x).**

Глобальная функция **isNaN**() аналогична **Number**. **isNaN**()

___

Связанная функция **Number. isFinite** () возвращает **true,** если аргумент представляет собой число, отличающееся от **NaN, Infinity** или **-Infinity .** Глобальная функция **isFinite**() возвращает **true,** если ее аргумент является или может быть преобразован в конечное число.

___

let zero = 0; // Нормальный ноль
let negz = -0; // Отрицательный ноль
zero === negz; // => true: ноль и отрицательный ноль равны
1/zero === 1/negz; // => false: Infinity и -Infinity не равны

___

Двоичное представление чисел с плавающей точкой и ошибки округления

Представление плавающей точки IEEE-754, используемое JavaScript (и почти любым другим современным языком программирования), является двоичным и может точно представлять дроби вроде 1/2, 1/8 и 1/1024. К сожалению, чаще всего (особенно при выполнении финансовых расчетов) мы применяем десятичные дроби: 1/10, 1/100 и т.д. Двоичные представления чисел с плавающей точкой не способны точно представлять даже такие простые числа, как 0.1.

Числа JavaScript обладают достаточной точностью и могут

обеспечить очень близкую аппроксимацию к 0.1.

let x=.3-.2;
let y=.2-.1;
x === y; / => false: два значения не одинаковы!
x === 0.1 / => false
У === 0.1 / => true

___

Одним из новейших средств JavaScript, определенных в ES2020, является числовой тип под названием **Bigint.**

Тип **Bigint** был добавлен в JavaScript главным образом для того, чтобы сделать возможным представление 64-битных целых чисел, которое требуется для совместимости со многими другими языками программирования и API-интерфейсами.

Однако обратите внимание, что реализации **Bigint** не подходят для криптографии, поскольку они не пытаются предотвратить атаки по времени.

Литералы Bigint записываются как строка цифр, за которой следует буква n в нижнем регистре. По умолчанию они десятичные, но можно применять префиксы 0Ь, Оо и Ох для двоичных, восьмеричных и шестнадцатеричных Bigint

BigInt(Number.MAX_SAFE_INTEGER) // 9007199254740991n
let string = "1" + "0".repeat(100); // 1 со следующими 100 нулями
BigInt(string) // => 10n100n: один гугол**

BigInt способен представлять только целые числа, что делает более общим обычный числовой тип JavaScript

___

Дата и время

В JavaScript определен простой класс Date для представления и манипулирования числами, которые представляют дату и время. Экземпляры класса Date являются объектами, но они также

имеют числовое представление в виде *отметок времени*

let timestamp = Date.nowO ; //Текущее время как отметка времени (число) .
let now = new Date(); // Текущее время как объект Date.
let ms = now.getTime(); // Преобразовать в миллисекундную отметку времени.
let iso = now.toISOString(); // Преобразовать в строку со стандартным // форматом.

——

Текст

Строка — это неизменяемая упорядоченная последовательность 16-битных значений, каждое из которых обычно представляет символ Unicode.

***Длиной* строки является количество содержащихся в ней 16-битных значений.**

Строки (и массивы) в JavaScript используют индексацию, начинающуюся с нуля: первое 16- битное значение находится в позиции 0, второе — в позиции 1 и т.д.

Пустая строка — это строка с длиной 0.

В JavaScript не предусмотрен особый тип, который представлял бы одиночный элемент строки.

Для представления одиночного 16-битного значения просто применяйте строку, имеющую длину 1.

Однако в ES6 строки являются *итерируемыми*и применение цикла fo r/of или операции . . . со строкой приведет к проходу по действительным символам строки, а не по 16-битным значениям.

let euro = '∈'
let love = '❤️'
euro.length // => 1: этот символ имеет один 16-битный элемент love.length
love.length // => 2

Строки в обратных кавычках являются возможностью ES6 и допускают наличие в строковых литералах встроенных (или *интерполированных)* выражений JavaScript.

Символ обратной косой черты (\) имеет специальное назначение в строках JavaScript.

Например, \n являет ся *управляющей последовательностью*, которая представляет символ новой строки.

Таблица 3.1. Управляющие последовательности JavaScript

Последова-тельность	Представляемый символ
\0	Символ NUL (\u0000)
\b	Забой (\u0008)
\t	Горизонтальная табуляция (\u0009)
\n	Новая строка (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Апостроф или одинарная кавычка (\u0027)
\\	Обратная косая черта (\u005C)
\x*nn*	Символ Unicode, указанный двумя шестнадцатеричными цифрами *nn*
\u*nnnn*	Символ Unicode, указанный четырьмя шестнадцатеричными цифрами *nnnn*
\u{*n*}	Символ Unicode, указанный с помощью кодовой точки *n*, где n — от одной до шести шестнадцатеричных цифр между 0 и 10FFFF (ES6)

Если символ \ предшествует любому символу, не перечисленному в табл. 3.1, тогда обратная косая черта просто игнорируется
Например, \# — то же самое, что и #

___

Работа со строками

Строки можно сравнивать с помощью стандартных операций равенства === и неравенства !==: две строки равны, если и только если они содержат в точности одну и ту же последовательность 16-битных значений.

В дополнение к свойству length в JavaScript предлагается богатый API-интерфейс для работы со строками:

```
let s = "Hello, world";      // Начать с некоторого текста.

// Получение порций строки
s.substring(1,4)             // => "ell": 2-й, 3-й и 4-й символы
s.slice(1,4)                 // => "ell": то же самое
s.slice(-3)                  // => "rld": последние 3 символа
s.split(", ")                // => ["Hello", "world"]: разбивает по
                             // строке разделителя

// Поиск в строке
s.indexOf("l")               // => 2: позиция первой буквы l
s.indexOf("l", 3)            // => 3: позиция первой буквы l,
                             // начиная с 3-й позиции
s.indexOf("zz")              // => -1: s не включает подстроку "zz"
s.lastIndexOf("l")           // => 10: позиция последней буквы l

// Булевские функции поиска в ES6 и последующих версиях
s.startsWith("Hell")         // => true: строка начинается с этого
s.endsWith("!")              // => false: s не оканчивается этим
s.includes("or")             // => true: s включает подстроку "or"

// Создание модифицированных версий строки
s.replace("llo", "ya")       // => "Heya, world"
s.toLowerCase()              // => "hello, world"
s.toUpperCase()              // => "HELLO, WORLD"
s.normalize()  // Нормализация Unicode NFC (Normalization Form C): ES6
```

```
s.normalize("NFD") // Нормализация Unicode NFD (Normalization Form D).
                   // Также есть "NFKC", "NFKD"

// Инспектирование индивидуальных (16-битных) символов строки
s.charAt(0)        // => "H": первый символ
s.charAt(s.length-1)     // => "d": последний символ
s.charCodeAt(0)    // => 72: 16-битное число в указанной позиции
s.codePointAt(0)   // => 72: ES6, работает с кодовыми точками > 16 бит

// Функции дополнения строк в ES2017
"x".padStart(3)    // => "  x": добавляет пробелы слева до длины 3
"x".padEnd(3)      // => "x  ": добавляет пробелы справа до длины 3
"x".padStart(3,"*")// => "**x": добавляет звездочки слева до длины 3
"x".padEnd(3, "-") // => "x--": добавляет дефисы справа до длины 3

// Функции усечения пробелов. trim() введена в ES5; остальные в ES2019
" test ".trim()        // => "test": удаляет пробелы в начале и конце
" test ".trimStart()   // => "test ": удаляет пробелы слева.
                       // Также есть trimLeft
" test ".trimEnd()     // => " test": удаляет пробелы справа.
                       // Также есть trimRight

// Смешанные методы строк
s.concat("!")          // => "Hello, world!": взамен просто
                       // используйте операцию +
"<>".repeat(5)         // => "<><><><><>": выполняет конкатенацию
                       // n копий. ES6
```

Не забывайте, что строки в JavaScript неизменяемы. Методы

вроде replace() и toUpperCase () возвращают новые строки: они не модифицируют строку, на которой вызываются.

___

Шаблонные литералы

```
let name = 'Bill';
let greeting = `Hello ${ name }.`;
greeting //  == "Hello Bill."
```

Все, что расположено между символами ${ и соответствующим символом }, интерпретируется как выражение JavaScript. Все, что находится вне фигурных скобок, является нормальным текстом строкового литерала. Выражение внутри скобок оценивается и затем преобразуется в строку, которая вставляется в шаблон, замещая знак доллара, фигурные скобки и все, что между ними.

___

Мощная, но менее часто применяемая возможность шаблонных литералов, заключается в том, что если сразу после открывающей обратной кавычки находится имя функции (или “тег”), тогда текст и выражения внутри шаблонного литерала передаются этой функции. Значением такого “тегового шаблонного ли терала” (tagged template literal) будет возвращаемое значение функции. Данную особенность можно использовать, скажем, для отмены действия определенных символов в значениях HTML или SQL перед их вставкой в текст.

`\n`.length // => 1: строка содержит одиночный символ новой строки

String.raw`\n` .length // => 2: строка содержит символ обратной косой черты и букву n

В ES6 имеется одна встроенная теговая функция: String.raw() . Она возвращает текст внутри обратных кавычек, не обрабатывая управляющие символы в обратных кавычках

Возможность определения собственных теговых функций

шаблонов — мощное средство JavaScript. Такие функции не обязаны возвращать строки и их можно применять подобно конструкторам, как если бы они определяли новый литеральный синтаксис для языка.

=====

14.5. Теги шаблонов

Строки внутри обратных кавычек известны как “шаблонные литералы* и были раскрыты в подразделе 3.3.4.

Когда за выражением, значением которого является функция, следует шаблонный литерал, выражение превращается в вызов функции, называемый “тегированным шаблонным литералом” (tagged template literal).

Определение новой теговой функции для использования с тегированными шаблонными литералами можно считать метапрограммированием, поскольку тегированные шаблоны часто применяются для определения **проблемно-ориентированных** языков (**domain-specific language - DSL**), **и определение новой теговой функции подобно добавлению нового синтаксиса к JavaScript.**

Тегированные шаблонные литералы были приняты многими интерфейсными пакетами JavaScript.

Язык запросов GraphQL использует теговую функцию **gqi4'** для того, чтобы запросы можно было встраивать в код JavaScript.

Библиотека Emotion применяет теговую функцию c s s 4', чтобы сделать возможным встраивание стилей CSS в код JavaScript.

Если шаблонный литерал - просто постоянная строка, не содержащая интерполяций, тогда теговая функция будет вызвана с массивом из одной строки и без дополнительных аргументов.

В общем случае, если шаблонный литерал имеет n интерполированных значений, тогда теговая функция будет вызвана с n+1 аргументов. В первом аргументе передается массив из n+1 строк, а в остальных - n интерполированных значений в порядке, в котором они следуют в шаблонном

литерале.

```
function html(strings, ...values) {
   // Преобразовать каждое значение в строку
   // и отменить специальные символы HTML
   let escaped = values.map(v => String(v)
                           .replace("&","&amp;")
                           .replace"&lt;")
                           .replace(">", "&gt;")
                           .replace('"', "&quot;")
                           .replace("'", "&#39;"));

   // Возвратить объединенные строки и отмененные значения
   let result = strings[0];

   for (let i = 0; i < escaped, length; i++) {
      result += escaped[i] + strings[i+1];
   }

   return result;
}

let operator = "<";

html`4<b>x ${operator} y</b>4` // => "<b>x &lt; y</b>"


let kind = "game", name = "D&D";

html`<div class="${kind}">$(name}</div>N` // =>'<div
class=,gameM>D&amp;D</div>1
```

=====

Булевские значения

Показанные ниже значения преобразуются и потому работают подобно false :
undefined null 0 -0 **NaN "" // пустая строка**

Все остальные значения, включая все объекты (и массивы)

преобразуются и потому работают как true. Временами значение false и шесть значений, которые в него преобразуются, называют *ложными*, тогда как все остальные значения — *истинными*

Булевские значения имеют метод **toString ()**, который можно использо вать для их преобразования в строки **"true"** или **"false",** но какими-то другими полезными методами они не обладают.

Несмотря на тривиальный API- интерфейс, есть три важных булевских операции.

Операция && выполняет булевскую операцию “И”. Она дает истинное значение тогда и только тогда, когда оба ее операнда истинны, а в противном случае оценивается в ложное значение.

Операция | | — это булевская операция “ИЛИ”: она вычисляется в истинное значение, если один из ее операндов истинный (или оба), и в ложное значение, когда оба операнда ложные. Наконец, унарная операция ! выполняет булевскую операцию “НЕ”: она дает true

, если ее операнд ложный, и false , если операнд истинный. Вот примеры:

if ((x === 0 && y === 0) || !(z === 0)) { // x и y равны нулю или z не равно нулю }

—

null и undefined

null — это ключевое слово языка, оцениваемое в особое значение, которое обычно применяется для указания на отсутствие значения.

Использование операции typeof на null возвращает строку "object", указывая на то, что null можно считать особым объектным значением, которое служит признаком “отсутствия объекта”.

Однако на практике null обычно рассматривается как единственный член собственного типа и может применяться с целью индикации “отсутствия значения” **для чисел и строк, а также объектов.**

В JavaScript есть и второе значение, которое указывает на отсутствие значения. Значение undefined представляет более глубокий вид отсутствия. Это значение переменных, которые не были инициализированы, и то, что получается при запрашивании значений свойств объекта или элементов массива, которые не существуют. Значение undefined также является возвращаемым значением функций, явно не возвращающих значение, и значением параметров функций, для которых аргументы не передавались, undefined представляет собой заранее определенную глобальную константу (не ключевое слово языка вроде n u ll, хотя практически такое различие не особенно важно), которая инициализируется значением undefined. Если вы примените к значению undefined операцию typeof, то она возвратит "undefined", указывая на то, что оно — единственный член специального типа.

Вопреки описанным отличиям n u ll и undefined служат признаком отсутствия значения и часто могут использоваться взаимозаменяемо. Операция равенства == считает их равными. (Для их различения применяйте операцию строгого равенства ===.) Оба являются ложными значениями: они ведут себя подобно false, когда требуется булевское значение. Ни null, ни undefined не имеют свойств либо методов. На самом деле использование . или [] для до ступа к какому-то свойству или методу значений null и undefined приводят к ошибке типа TypeEггor.

Я полагаю, что undefined предназначено для представления системного, непредвиденного или похожего на ошибку отсутствия значения, а null — для представления программного, нормального или ожидаемого отсутствия значения. Когда могу, я избегаю применения null и undefined, но если мне нуж но присвоить одно из этих значений переменной или свойству либо передать одно из них в функцию или возвратить из нее, то обычно использую null. Некоторые программисты стараются вообще избегать n u l l и применять на его месте undefined везде,

где только возможно.

___

Тип Symbol

Символы были введены в ES6, чтобы служить нестроковыми именами свойств.

Для понимания символов вы должны знать, что фундаментальный тип Object в JavaScript определен как неупорядоченная коллекция свойств, где каждое свойство имеет имя и значение.

Именами свойств обычно (а до ES6 единственно) являются строки. Но в ES6 и последующих версиях для такой цели можно использовать символы:

```
let strname = "string name";
let symname = Symbol("propname");

typeof strname; // 'string'
typeof symname; // 'symbol'

let o = {};
o[strname] = 1;
o[symname] = 2;
o[strname]  // 1
o[symname]  // 2
```

Символы не располагают литеральным синтаксисом. Чтобы получить зна чение символа, понадобится вызвать функцию Symbol (), которая никогда не возвращает то же самое значение дважды, даже когда вызывается с таким же аргументом. Это означает, что если вы вызываете Symbol () для получения зна чения символа, то при добавлении к объекту нового свойства можете безопас но применять полученное значение символа как имя свойства, не беспокоясь о возможности перезаписывания существующего свойства с тем же именем.

На практике символы выступают в качестве механизма расширения языка. Когда в ES6 был представлен цикл f o r / o f

(см. подраздел 5.4.4) и итерируемые объекты (см. главу 12), возникла необходимость определить стандартный метод, который классы могли бы реализовывать, чтобы делать себя итерируемыми. Но стандартизация любого индивидуального строкового имени для этого итера торного метода привела бы к нарушению работы существующего кода, так что взамен было решено применять символьное имя.

Symbol.iterator представляет собой символьное значение, которое можно использовать в качестве имени метода, делая объект итерируемым.

Функция Symbol () принимает необязательный строковый аргумент и возвращает уникальное значение типа Symbol.

Если вы предоставите строковый аргумент, то переданная в нем строка будет включена в выход метода toString () типа Symbol.

Тем не менее, имейте в виду, что вызов Symbol () два раза с той и то же самой строкой даст два совершенно разных значения Symbol

- **toString** () — единственный интересный метод в экземплярах **Symbol.**

Однако вы должны знать еще две функции, относящиеся к Symbol.

- **Symbol.iterator.**
- Функция **Symbol.for**() принимает строковый аргумент и возвращает значение **Symbol,** ассоциированное с предоставленной вами строкой.
  - Если никакого значения **Symbol** со строкой пока не ассоциировано, тогда создает ся и возвращается новое значение **Symbol,** иначе возвращается существующее значение.
  - То есть функция **Symbol.for** () совершенно отличается от функции **Symbol** (): **Symbol** () никогда не возвращает одно и то же значение дважды, но **Symbol. for** () всегда возвращает то же самое

значение, когда вызывается с од ной и той же строкой. Переданная функции **Symbol.for(**) строка присутствует в выводе **toString** () для возвращенного значения **Symbol;** вдобавок ее можно извлечь, вызвав **Symbol.keyFor** () на возвращенном значении **Symbol.**

—

Глобальный объект

Глобальный объект — это обыкновенный объект JavaScript, который служит крайне важной цели: его свойствами являются глобально определенные идентификаторы, доступные программе JavaScript.

Когда интерпретатор JavaScript запускается, он создает новый глобальный объект и предоставляет ему начальный набор свойств, которые определяют:

• глобальные константы вроде undefined, Infinity и NaN;
• глобальныефункции наподобие isNaN (), parselnt () (см. подраздел 3.9.2) и eval () (см. раздел 4.12);
• функции конструкторов вроде Date (), RegExp (), String (), Object () и Array () (см. подраздел 3.9.2);
• глобальные объекты наподобие Math и JSON (см. раздел 6.8).

В Node глобальный объект имеет свойство с именем **global,** значением которого будет сам глобальный объект, поэтому в программах Node вы всегда можете ссылаться на глобальный объект по имени **global.**

В веб-браузерах глобальным объектом для всего кода JavaScript, содержащегося в окне браузера, служит объект **Window,** который представляет это окно. Глобальный объект **Window** располагает ссылающимся на самого себя свойством **window,** которое можно применять для ссылки на глобальный объект.

Потоки веб-воркеров (см. раздел 15.13) имеют другой глобаль-ный объект, нежели объект **Window,** с которым они ассоциированы. Код в воркере может ссылаться на свой глобальный объект посредством **self .**

Наконец, в качестве стандартного способа ссылки на глобальный объект в любом контексте версия ES2020 определяет **globalThis .** К началу 2020 года данная возможность была реализована всеми современными браузерами и Node.

___

Неизменяемые элементарные значения и изменяемые объектные ссылки

В JavaScript существует фундаментальное отличие между элементарными значениями **(undefined, null,** булевские, числа и строки) и объектами (включая массивы и функции).

Элементарные значения неизменяемы: никакого способа модифицировать элементарное значение не предусмотрено.

Элементарные значения также сравниваются *по величине:* два значения бу дут одинаковыми, только если они имеют одну и ту же величину.

При сравнении двух отдельных строковых значений JavaScript трактует их как равные тогда и только тогда, когда они имеют одну и ту же дли ну и одинаковые символы по каждому индексу.

Объекты не сравниваются по величине: два отдельных объекта не равны, даже когда имеют те же самые свойства и значения. И два отдельных массива не равны, даже если они содержат те же самые элементы в том же порядке:

Иногда объекты называют *ссылочными типами*, чтобы отличать их от эле ментарных типов JavaScript. В рамках такой терминологии объектные значения являются *ссылками* и мы говорим, что объекты сравниваются *по ссылке:* два объектных значения будут одинаковыми, если и только если они *ссылаются* на тот же самый внутренний объект.

___

Преобразования типов

Таблица 3.2. Преобразования типов JavaScript

Значение	Преобразование в строку	Преобразование в число	Преобразование в булевское значение
undefined	"undefined"	NaN	false
null	"null"	0	false
true	"true"	1	
false	"false"	0	
"" (пустая строка)		0	false
"1.2" (непустая, числовое содержимое)		1.2	true
"one" (непустая, нечисловое содержимое)		NaN	true
0	"0"		false
-0	"0"		false
1 (конечное, ненулевое)	"1"		true
Infinity	"Infinity"		true
-Infinity	"-Infinity"		true
NaN	"NaN"		false
{} (любой объект)	См. подраздел 3.9.3	См. подраздел 3.9.3	true
[] (пустой массив)	""	0	true
[9] (один числовой элемент)	"9"	9	true
['a'] (любой другой массив)	Используйте метод join()	NaN	true
function(){} (любая функция)	См. подраздел 3.9.3	NaN	true

Явные преобразования

Перечисленные факты приводят к следующим идиомам преобразования типов, которые вы могли встречать в коде

x+"" //=>String(x)
+x // =>Number(x)
x-0 // =>Number(x)
!!x // => Boolean(x) : обратите внимание на два символа !

Преобразования объектов в элементарные значения

Одна из причин сложности преобразований объектов в

элементарные значения связана с тем, что некоторые типы объектов имеют несколько элементарных представлений. Скажем, объекты дат могут быть представлены как строки или как числовые отметки времени. В спецификации JavaScript определены три фундаментальных алгоритма для преобразования объектов в элементарные значения.

prefer-string
Этот алгоритм возвращает элементарное значение, отдавая предпочтение строке, если такое преобразование возможно.

prefer-number
Этот алгоритм возвращает элементарное значение, отдавая предпочтение числу, если такое преобразование возможно.

no-preference
Этот алгоритм не отдает никаких предпочтений относительно того, какой тип элементарного значения желателен, и классы могут определять собственные преобразования. Все встроенные типы JavaScript, исключая Date, реализуют данный алгоритм как *prefer-number*. Класс Date реализует его как *prefer-string*.

Методы toString() valueOf()

Все объекты наследуют два метода преобразования, которые применяются преобразованиями объектов в элементарные значения, и прежде чем можно будет объяснять алгоритмы преобразования *prefer-string, prefer-number* и no-*preference*, нужно взглянуть на эти два метода.

Первым методом является toString (), задача которого — возвратить строковое представление объекта.

Многие классы определяют более специфичные версии метода toString().

Скажем, метод toString () класса Array преобразует каждый элемент массива в строку и объединяет результирующие строки вместе, разделяя их запятыми.
Метод toString () класса Function преобразует определяемые пользователем функции в строки исходного кода JavaScript. Класс Date определяет метод toString (), который возвращает

пригодную для чтения человеком (и подда- ющуюся разбору интерпретатором JavaScript) строку с датой и временем. Класс RegExp определяет метод to S tr in g (), преобразующий объект RegExp в стро ку, которая выглядит подобно литералу RegExp:

[1/2,3] .toStringO // => "1/2/3"
(function(x) { f(x); }).toString() // => "function(x) { f(x); }"
/\d+/g.toString() // => "/\\d+/g"
let d = new Date (2020, 0,1) ; d.toString() //=>"Wed Jan 01 2020 00:00:00 GMT-0800 (Pacific Standard Time) "

——

Другая функция преобразования объектов называется **valueOf** ().

Задача этого метода определена менее четко: предполагается, что он должен преобразовывать объект в элементарное значение, которое представляет объект, если такое элементарное значение существует. Объекты являются составными значениями, причем большинство объектов на самом деле не могут быть представлены единственным элементарным значением, а потому стандартный метод **valueOf()** возвращает сам объект вместо элементарного значения. Классы-оболочки, такие как **String, Number** и **Boolean,** определяют методы **valueOf()**, которые просто возвращают содержащееся внутри элементарное значение. Массивы, функции и регулярные выражения наследуют стандартный метод. Вызов **valueOf()** для экземпляров упомянутых типов возвращает сам объект. Класс **Date** определяет метод **valueOf()**, возвращающий дату в ее внутреннем представлении — количество миллисекунд, прошедших с момента 1 января 1970 года:

let d = new Date(2010, 0, 1); //1 января 2010 года (тихоокеанское время)
d.valueOf() // => 1262332800000

В завершение данной темы полезно отметить, что детали преобразования *prefer-number* объясняют, почему пустые массивы преобразуются в число 0 и одноэлементные массивы также могут быть преобразованы в числа:

Number([]) // => 0: неожиданно! Number ([99]) // => 99: правда?

___

Объявление с помощью let и const

В главе 5 речь пойдет об операторах циклов for, for/in и for/of в JavaScript.

for (let i = 0, len = data.length; i < len; i++) console.log(data[i]);
for (let datum of data) console.log (datum);
for (let property in object) console.log(property);

Может показаться удивительным, но при объявлении “переменных” циклов for/in и for/o f допускается использовать также и const,
если в теле цикла им не присваивается новое значение.

В таком случае объявление const просто говорит о том, что значение будет константой в течение одной итерации цикла:

for (const datum of data) console.log (datum) ;
for (const property in object) console.log (property);

___

Область видимости переменных и констант

Переменные и константы, объяв ленные с помощью let и const, имеют *блочную область видимости*.

Другими словами, они определены только внутри блока кода, в котором находятся оператор let или const.

Определения классов и функций JavaScript являются блоками, равно как и тела операторов if/else, циклов while, for и т.д.

Грубо говоря, если переменная или константа объявляется внутри набора фигурных скобок, то эти фигурные скобки ограничивают область кода, где переменная или константа

определена (хотя, конечно же, нельзя ссылаться на переменную или константу из строк кода, который выполняется до оператора let или const, объявляющего переменную или константу).

Переменные и константы, объявленные как часть цикла for, for/in или for/of, имеют в качестве области видимости тело цикла, даже если формально они появляются за рамками фигурных скобок.

___

Объявление переменных с помощью var

Хотя var и let имеют одинаковый синтаксис, существуют важные отличия в том, как они работают.

- **Переменные, объявленные с помощью var, не имеют блочной области видимости. Взамен область видимости таких переменных распространяется на тело содержащей функции независимо от того, насколько глубоко их объявления вложены внутрь этой функции.**
- Если вы применяете var вне тела функции, то объявляется глобальная пе ременная. Но глобальные переменные, объявленные посредством var, от личаются от глобальных переменных, объявленных с использованием l e t , одним важным аспектом. Глобальные переменные, объявленные с помо щью var, реализуются в виде свойств глобального объекта (см. раздел 3.7). На глобальный объект можно ссылаться как на globalThis. Таким обра зом, если вы записываете var x = 2; за пределами функции, то это подоб но записи globalThis.x = 2;. Тем не менее, следует отметить, что анало гия не идеальна: свойства, созданные из глобальных объявлений var, не могут быть удалены посредством операции delete (см. подраздел 4.13.4). Глобальные переменные и константы, объявленные с применением l e t и const, не являются свойствами глобального объекта.
- В отличие от переменных, объявленных с помощью l e t , с использова нием var вполне законно объявлять ту же самую переменную много раз. И поскольку переменные var имеют область видимости функции, а не блока,

фактически общепринято делать повторное объявление такого рода. Переменная i часто применяется для целочисленных значений особенно в качестве индексной переменной циклов for. В функции с множеством циклов for каждый цикл обычно начинается с for (var i = 0; .... Из-за того, что var не ограничивает область видимости этих переменных телом цикла, каждый цикл (безопасно) повторно объявляет и заново инициали зирует ту же самую переменную.

- Одна из самых необычных особенностей объявлений var известна как подъем (hoisting).
  - Когда переменная объявляется с помощью var, объявле ние поднимается к верхушке объемлющей функции. Инициализация пере менной остается там, где вы ее записали, но определение переменной пере мещается к верхушке функции. Следовательно, переменные, объявленные посредством var, можно использовать безо всяких ошибок где угодно в объемлющей функции. Если код инициализации пока еще не выполнен, тогда значением переменной может быть undefined, но вы не получите ошибку в случае работы с переменной до ее инициализации. (Это может стать источником ошибок и одной из нежелательных характеристик, кото рую исправляет let: если вы объявите переменную с применением let, но попытаетесь ее использовать до выполнения оператора l e t , то получи те действительную ошибку, а не просто увидите значение undefined.)

Деструктурирующее присваивание

В ES6 реализован своего рода составной синтаксис объявления и присваивания, известный как *деструктурирующее присваивание* (destructuring assignment).

```
let [x,y] = [1,2];  / То же, что и let x=1, y=2
[X,y] = [x+1,y+1];  // То же, что и let x=1, y=2
[x,y] = [y,x];   // // Поменять местами значения двух переменных
```

[x,y] / / => [3,2] : инкрементированные и поменявшиеся // местами значения

```
let o = { x: 1, y: 2 }; // Объект, по которому будет делаться проход в цикле
for(const [паше, value] of Object.entries(o)) {
   console.log(name, value); // Выводится "x 1 y 2"
}
```

Если при деструктуризации массива вы хотите собрать все неиспользуемые или оставшиеся значения в единственную переменную, тогда применяйте три тоочки (. . .) перед последним именем переменной с левой стороны:

```
let [x, ...y] = [1,2,3,4]; // y == [2,3,4]
let [a, [b, c]] = [1, [2,2.5], 3]; // a == 1; b == 2; c == 2.5
let [first, ...rest] = "Hello”; // rest = ["e”,”l”,"l”,"o”]

let [{x: x1, y: y1}, {x: x2, y: y2}] = [{x: 1, y: 2}, {x: 3, y: 4}];
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // true

let { p1: [x1, y1], p2: [x2, y2] } = {p1: [1,2], p2: [3,4]}; // (x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) => true
```

——

Понимание сложной деструктуризации. Полезная законо-мерность

```
/ / Начать со структуры данных и сложной деструктуризации
let points = [{x: 1, y: 2}, {x: 3, y: 4}];
let [{x: xl, y: yl), {x: x2, y: y2}] = points;
/ / Проверить синтаксис деструктуризации,
/ / поменяв местами стороны присваивания
let points2= [{x: xl, y: yl}, {x: x2, y: y2}]; //points2 == points
```