JavaScript - number,string

number, методы number, string, методы string, объект math

Примитив как объект

JavaScript позволяет нам работать с примитивными типами данных – строками, числами и т.д., как будто они являются объектами. У них есть и методы.

Вот парадокс, с которым столкнулся создатель JavaScript:

Есть много всего, что хотелось бы сделать с примитивами, такими как строка или число. Было бы замечательно, если бы мы могли обращаться к ним при помощи методов.

Примитивы должны быть легкими и быстрыми насколько это возможно.

Выбранное решение, хотя выглядит оно немного неуклюже:

Примитивы остаются примитивами. Одно значение, как и хотелось.

Язык позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов.

Чтобы это работало, при таком доступе создается специальный «объект-обёртка», который предоставляет нужную функциональность, а после удаляется.

Пример

```
let str = "Привет";

alert( str.toUpperCase() ); // ПРИВЕТ
```

Каждый примитив имеет свой собственный «объект-обертку», которые называются: String, Number, Boolean, Symbol и BigInt. Таким образом, они имеют разный набор методов.

Строка str – примитив. В момент обращения к его свойству, создается специальный объект, который знает значение строки и имеет такие полезные методы, как toUpperCase().

Этот метод запускается и возвращает новую строку (показывается в alert).

Специальный объект удаляется, оставляя только примитив str.

Получается, что примитивы могут предоставлять методы, и в то же время оставаться «легкими».

Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта. Однако, он всё же должен придерживаться спецификаций и работать так, как будто он его создаёт.

Конструкторы String/Number/Boolean предназначены только для внутреннего пользования

Некоторые языки, такие как Java, позволяют явное создание «объектов-оберток» для примитивов при помощи такого синтаксиса как new Number(1) или new Boolean(false).

В JavaScript, это тоже возможно по историческим причинам, но очень не рекомендуется. В некоторых местах последствия могут быть катастрофическими.

С другой стороны, использование функций String/Number/Boolean без оператора new – вполне разумно и полезно. Они превращают значение в соответствующий примитивный тип: в строку, в число, в булевый тип.

```
alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"!
```

```
let zero = new Number(0);

if (zero) {
  // zero возвращает "true", так как является объектом
  alert( "zero имеет «истинное» значение?!?" );
}
```

```
let num = Number("123"); // превращает строку в число
```

null/undefined не имеют методов

Особенные примитивы null и undefined являются исключениями. У них нет соответствующих «объектов-оберток», и они не имеют никаких методов. В некотором смысле, они «самые примитивные».

Попытка доступа к свойствам такого значения возвратит ошибку:

```
alert(null.test); // ошибка
```

Строки

В JavaScript любые текстовые данные являются строками.

Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы можем вставлять произвольные выражения, обернув их в ${…}, Еще одно преимущество обратных кавычек — они могут занимать более одной строки

`` - шаблонная строка, “”, ‘’ - обычная строка

Спецсимволы

все спецсимволы начинаются с обратного слеша, \ — так называемого «символа экранирования».

Он также используется, если необходимо вставить в строку кавычку.

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \

Но что, если нам надо добавить в строку собственно сам обратный слеш \?

Это можно сделать, добавив перед ним… еще один обратный слеш!

Символ	Описание
`\n`	Перевод строки
`\r`	В текстовых файлах Windows для перевода строки используется комбинация символов `\r\n`, а на других ОС это просто `\n`. Это так по историческим причинам, ПО под Windows обычно понимает и просто `\n`.
`\'`, `\"`, `\``	Кавычки
`\\`	Обратный слеш
`\t`	Знак табуляции
`\b`, `\f`, `\v`	Backspace, Form Feed и Vertical Tab — оставлены для обратной совместимости, сейчас не используются.

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

```
alert( 'I\'m the Walrus!' ); // I'm the Walrus!
```

```
alert( `The backslash: \\` ); // The backslash: \
```

Длина строки (свойство строки как объекта)

Свойство length содержит длину строки.

Пример:

Обратите внимание, \n — это один спецсимвол, поэтому тут всё правильно: длина строки 3

```
alert( `My\n`.length ); // 3
```

Доступ к символам

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод str.at(pos). Первый символ занимает нулевую позицию.

Также можно перебрать строку посимвольно, используя for..of.

```
let str = `Hello`;

// получаем первый символ
alert( str[0] ); // H
alert( str.at(0) ); // H

// получаем последний символ
alert( str[str.length - 1] ); // o
alert( str.at(-1) ); // o
```

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char — сначала "H", потом "e", потом "l" и т.д.)
}
```

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана — она такая навсегда.

Можно создать новую строку и записать её в ту же самую переменную вместо старой.

```
let str = 'Hi';

str[0] = 'h'; // ошибка
alert( str[0] ); // не работает
```

```
let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi
```

Сравнение строк

Строки кодируются в UTF-16. Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

```
// одна и та же буква в нижнем и верхнем регистре
// будет иметь разные коды
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

str.codePointAt(pos)

Метод codePointAt() возвращает неотрицательное целое число, являющееся закодированным в UTF-16 значением кодовой точки.

Методы строк (методы экземпляра строки)

- concat() / оператор “+” - сложение строк
- includes() - проверяет, содержит ли строка заданную подстроку
- indexOf() / lastIndexOf() - поиск индекса
- match() / matchAll() - возвращает получившиеся совпадения при сопоставлении строки с регулярным выражением
- replace() / replaceAll() - заменяет что то на что то в строке
- padStart() / padEnd() - заполняет оставшееся место
- repeat() - повтор строки
- slice() - извлекает часть строки
- toLowerCase() / toUpperCase() - меняет регистр строки
- trim() / trimRight() / trimLeft() - убирает пробелы

Числа

В современном JavaScript существует два типа чисел:

- Обычные числа в JavaScript хранятся в 64-битном формате IEEE-754, который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers). Это числа, которые мы будем использовать чаще всего. Мы поговорим о них в этой главе.
- BigInt числа дают возможность работать с целыми числами произвольной длины. Они нужны достаточно редко и используются в случаях, когда необходимо работать со значениями более чем (253-1) или менее чем -(253-1).

В рамках front-end разработки числа типа number более чем достаточно

Способы записи числа

Символ нижнего подчеркивания _ – это «синтаксический сахар», он делает число более читабельным. Движок JavaScript попросту игнорирует _ между цифрами, поэтому в примере выше получается точно такой же миллиард, как и в первом случае.

Однако в реальной жизни мы в основном стараемся не писать длинные последовательности нулей, так как можно легко ошибиться. Укороченная запись может выглядеть как "1 млрд" или "7.3млрд" для 7 миллиардов 300 миллионов. Такой принцип работает для всех больших чисел.

В JavaScript, чтобы сократить запись числа, мы можем добавить к нему букву "e" и указать необходимое количество нулей.

```
let billion = 1000000000;
```

```
let billion = 1_000_000_000
```

```
let billion = 1e9;  // 1 миллиард, буквально: 1 и 9 нулей
```

```
let mcs = 0.000001;
```

```
let ms = 1e-6; // шесть нулей слева от 1
```

Шестнадцатеричные, двоичные и восьмеричные числа

Шестнадцатеричные числа широко используются в JavaScript для представления цветов, кодировки символов и многого другого. Естественно, есть короткий стиль записи: 0x, после которого указывается число.

```
let a = 0b11111111; // двоичная (бинарная) форма записи числа 255
let b = 0o377; // восьмеричная форма записи числа 255

alert( a == b ); // true, с двух сторон число 255
```

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (то же самое, регистр не имеет значения)
```

toString(base)

Метод num.toString(base) возвращает строковое представление числа num в системе счисления base.

base может варьироваться от 2 до 36 (по умолчанию 10).

Внимание! Две точки в 123456..toString(36) это не опечатка. Если нам надо вызвать метод непосредственно на числе, как toString в примере выше, то нам надо поставить две точки .. после числа.

Если мы поставим одну точку: 123456.toString(36), тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод.

Также можно записать как (123456).toString(36)

```
alert( 123456..toString(36) ); // 2n9c
```

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

Неточные вычисления

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

Внутри JavaScript число представлено в виде 64-битного формата IEEE-754. Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 для хранения положения десятичной точки и один бит отведен на хранение знака.

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности.

Справедливости ради заметим, что ошибка в точности вычислений для чисел с плавающей точкой сохраняется в любом другом языке, где используется формат IEEE 754, включая PHP, Java, C, Perl и Ruby.

```
alert( 1e500 ); // Infinity
```

```
alert( 0.1 + 0.2 == 0.3 ); // false
```

В результате неточностей мы получаем

Два нуля

Другим забавным следствием внутреннего представления чисел является наличие двух нулей: 0 и -0.

Все потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль.

В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.

```
alert( 9999999999999999 ); // покажет 10000000000000000
```

Причина та же – потеря точности. Из 64 бит, отведенных на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит – знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

Проверка: isFinite и isNaN

- Infinity (и -Infinity) — особенное численное значение, которое ведёт себя в точности как математическая бесконечность ∞.
- NaN представляет ошибку.

Эти числовые значения принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки

- isNaN(value) преобразует значение в число и проверяет является ли оно NaN
- isFinite(value) преобразует аргумент в число и возвращает true, если оно является обычным числом, т.е. не NaN/Infinity/-Infinity.

toFixed()

```
numObj.toFixed([digits])
```

Метод toFixed() форматирует число, используя запись с фиксированной запятой.

digits

Необязательный параметр. Количество цифр после десятичной запятой; может быть значением между 0 и 20 включительно, хотя реализации могут поддерживать и больший диапазон значений. Если аргумент опущен, он считается равным 0.

Возвращает строку

parseInt и parseFloat

Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN.

```
alert( +"100px" ); // NaN
```

В реальной жизни мы часто сталкиваемся со значениями у которых есть единица измерения, например "100px" или "12pt" в CSS. Также во множестве стран символ валюты записывается после номинала "19€".

parseInt и parseFloat «читают» число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, вернётся только целая часть
alert( parseFloat('12.3.4') ); // 12.3, произойдёт остановка чтения на второй точке
```

Функции parseInt/parseFloat вернут NaN, если не смогли прочитать ни одну цифру

```
alert( parseInt('a123') ); // NaN, на первом символе происходит остановка
```

Функция parseInt() имеет необязательный второй параметр. Он определяет систему счисления, таким образом parseInt может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, без 0x тоже работает

alert( parseInt('2n9c', 36) ); // 123456
```

math

Объект Math является встроенным объектом, хранящим в своих свойствах и методах различные математические константы и функции. Объект Math не является функциональным объектом.

- abs() - модуль
- ceil() / floor() / round() - округление
- random() - псевдослучайное число (от 0 до 1)
- min() / max() - минимум и максимум (определение)
- pow() / sqrt() - возведение в степень и квадратный корень
- PI - математическая константы
- sin() / cos() / tan() / - тригонометрические функции

Ресурсы

Методы примитивов - ТЫК

Строки учебник - ТЫК

Числа учебник - ТЫК

Строки MDN документация - ТЫК

Числа MDN документация - ТЫК

Math MDN документация - ТЫК