Поделиться

Редактировать на GitHub

Числа

В современном JavaScript существует два типа чисел:

1. 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.
2. `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 2^{53} или менее чем -2^{53}. Так как `BigInt` числа нужны достаточно редко, мы рассмотрим их в отдельной главе BigInt.

В данной главе мы рассмотрим только первый тип чисел: числа типа `number`. Давайте глубже изучим, как с ними работать в JavaScript.

Способы записи числа

Представьте, что нам надо записать число 1 миллиард. Самый очевидный путь:

```
1  let billion = 1000000000;
```

Но в реальной жизни мы обычно опускаем запись множества нулей, так как можно легко ошибиться. Укороченная запись может выглядеть как `"1млрд"` или `"7.3млрд"` для 7 миллиардов 300 миллионов. Такой принцип работает для всех больших чисел.

В JavaScript можно использовать букву `"e"`, чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

```
1  let billion = 1e9;  // 1 миллиард, буквально: 1 и 9 нул
2
3  alert( 7.3e9 );  // 7.3 миллиардов (7,300,000,000)
```

Другими словами, `"e"` производит операцию умножения числа на 1 с указанным количеством нулей.

```
1  1e3 = 1 * 1000
2  1.23e6 = 1.23 * 1000000
```

Сейчас давайте запишем что-нибудь очень маленькое. К примеру, 1 микросекунду (одна миллионная секунды):

```
1  let ms = 0.000001;
```

Записать микросекунду в укороченном виде нам поможет `"e"`.

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

Если мы подсчитаем количество нулей `0.000001`, их будет 6. Естественно, верная запись `1e-6`.

Другими словами, отрицательное число после `"e"` подразумевает деление на 1 с указанным количеством нулей:

Редактировать на GitHub

```
1  // 1 делится на 1 с 3 нулями
2  1e-3 = 1 / 1000 (=0.001)
3
4  // 1.23 делится на 1 с 6 нулями
5  1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

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.

Например:

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

Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются `0b` для двоичных и `0o` для восьмеричных:

```
1  let a = 0b11111111; // бинарная форма записи числа 255
2  let b = 0o377; // восьмеричная форма записи числа 255
3
4  alert( a == b ); // true, с двух сторон число 255
```

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления мы рекомендуем использовать функцию `parseInt` (рассмотрим позже в этой главе).

toString(base)

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

Например:

```
1  let num = 255;
2
3  alert( num.toString(16) );  // ff
4  alert( num.toString(2) );   // 11111111
```

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

Часто используемые:

- **base=16** — для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть `0..9` или `A..F`.
- **base=2** — обычно используется для отладки побитовых операций, цифры `0` или `1`.
- **base=36** — максимальное основание, цифры могут быть `0..9` или `A..Z`. То есть, используется весь латинский алфавит для представления числа. Забавно, но можно использовать `36`-разрядную систему счисления для получения короткого представления большого числового идентификатора. К примеру, для создания короткой ссылки. Для этого просто преобразуем его в `36`-разрядную систему счисления:

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

Поделиться
Редактировать на GitHub

Две точки для вызова метода

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)`.

Округление

Одна из часто используемых операций при работе с числами – это округление.

В JavaScript есть несколько встроенных функций для работы с округлением:

`Math.floor`

Округление в меньшую сторону: `3.1` становится `3`, а `-1.1` — `-2`.

`Math.ceil`

Округление в большую сторону: `3.1` становится `4`, а `-1.1` — `-1`.

`Math.round`

Округление до ближайшего целого: `3.1` становится `3`, `3.6` — `4`, а `-1.1` — `-1`.

`Math.trunc` **(не поддерживается в Internet Explorer)**

Производит удаление дробной части без округления: `3.1` становится `3`, а `-1.1` — `-1`.

Ниже представлена таблица с различиями между функциями округления:

	`Math.floor`	`Math.ceil`	`Math.round`	`Math.trunc`
`3.1`	3	4	3	3
`3.6`	3	4	4	3
`-1.1`	-2	-1	-1	-1
`-1.6`	-2	-1	-2	-1

Эти функции охватывают все возможные способы обработки десятичной части. Что если нам надо округлить число до `n-ого` количества цифр в дробной части?

Например, у нас есть `1.2345` и мы хотим округлить число до 2-х знаков после запятой, оставить только `1.23`.

Есть два пути решения:

1. Умножить и разделить.

   Например, чтобы округлить число до второго знака после запятой, мы можем умножить число на `100`, вызвать функцию округления и разделить обратно.

   ```
   1  let num = 1.23456;
   2
   3  alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 1
   ```

2. Метод toFixed(n) округляет число до `n` знаков после запятой и возвращает строковое представление результата.

Редактировать на GitHub

```
1  let num = 12.34;
2  alert( num.toFixed(1) ); // "12.3"
```

Округляет значение до ближайшего числа, как в большую, так и в меньшую сторону, аналогично методу `Math.round`:

```
1  let num = 12.36;
2  alert( num.toFixed(1) ); // "12.4"
```

Обратите внимание, что результатом `toFixed` является строка. Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:

```
1  let num = 12.34;
2  alert( num.toFixed(5) ); // "12.34000", добавлены нул
```

Мы можем преобразовать полученное значение в число, используя унарный оператор `+` или `Number()`, пример с унарным оператором: `+num.toFixed(5)`.

Неточные вычисления

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 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранение знака.

Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

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

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

Посмотрите на это (неверное!) сравнение:

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

Да-да, сумма `0.1` и `0.2` не равна `0.3`.

Странно! Что тогда, если не `0.3`?

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

Ой! Здесь гораздо больше последствий, чем просто некорректное сравнение. Представьте, вы делаете интернет-магазин и посетители формируют заказ из 2-х позиций за `$0.10` и `$0.20`. Итоговый заказ будет `$0.30000000000000004`. Это будет сюрпризом для всех.

Но почему это происходит?

Число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как `0.1`, `0.2`, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.

Другими словами, что такое `0.1`? Это единица делённая на десять — `1/10`, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: `1/3`, которая становится бесконечной дробью `0.33333(3)`.

Деление на `10` гарантированно хорошо работает в десятичной системе, но деление на `3` – нет. По той же причине и в двоичной системе счисления,

деление на `2` обязательно сработает, а `1/10` становится бесконечной дробью.

В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления.

Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют нам увидеть эту «крошечную потерю точности», но она существует.

Пример:

```
1  alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

И когда мы суммируем 2 числа, их «неточности» тоже суммируются.

Вот почему `0.1 + 0.2` – это не совсем `0.3`.

> **Не только в JavaScript**
>
> 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, Ruby.

Можно ли обойти проблему? Конечно, наиболее надёжный способ — это округлить результат используя метод toFixed(n):

```
1  let sum = 0.1 + 0.2;
2  alert( sum.toFixed(2) ); // 0.30
```

Помните, что метод `toFixed` всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. Также это удобно для форматирования цен в интернет-магазине `$0.30`. В других случаях можно использовать унарный оператор `+`, чтобы преобразовать строку в число:

```
1  let sum = 0.1 + 0.2;
2  alert( +sum.toFixed(2) ); // 0.3
```

Также можно временно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:

```
1  alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
2  alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000
```

Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.

Иногда можно попробовать полностью отказаться от дробей. Например, если мы в нашем интернет-магазине начнём использовать центы вместо долларов. Но что будет, если мы применим скидку 30%? На практике у нас не получится полностью избавиться от дроби. Просто используйте округление, чтобы отрезать «хвосты», когда надо.

Поделиться

Редактировать на GitHub

Забавный пример

Попробуйте выполнить его:

```
1  // Привет! Я – число, растущее само по себе!
2  alert( 9999999999999999 ); // покажет 10000000000(
```

Причина та же – потеря точности. Из 64 бит, отведённых 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».

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.

Проверка: isFinite и isNaN

Помните эти специальные числовые значения?

- `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`:

```
1  alert( isNaN(NaN) ); // true
2  alert( isNaN("str") ); // true
```

Нужна ли нам эта функция? Разве не можем ли мы просто сравнить `=== NaN`? К сожалению, нет. Значение `NaN` уникально тем, что оно не является равным ни чему другому, даже самому себе:

```
1  alert( NaN === NaN ); // false
```

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

```
1  alert( isFinite("15") ); // true
2  alert( isFinite("str") ); // false, потому что специа
3  alert( isFinite(Infinity) ); // false, потому что спе
```

Иногда `isFinite` используется для проверки, содержится ли в строке число:

```
1  let num = +prompt("Enter a number", '');
2
```

```
3  // вернёт true всегда, кроме ситуаций, когда аргумент -
4  alert( isFinite(num) );
```


Поделиться

Редактировать на GitHub

Помните, что пустая строка интерпретируется как `0` во всех числовых функциях, включая `isFinite`.

Сравнение Object.is

Существует специальный метод Object.is, который сравнивает значения примерно как `===`, но более надёжен в двух особых ситуациях:

1. Работает с `NaN`: `Object.is(NaN, NaN) === true`, здесь он хорош.
2. Значения `0` и `-0` разные: `Object.is(0, -0) === false`, это редко используется, но технически эти значения разные.

Во всех других случаях `Object.is(a, b)` идентичен `a === b`.

Этот способ сравнения часто используется в спецификации JavaScript. Когда внутреннему алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует `Object.is` (Определение SameValue).

parseInt и parseFloat

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

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

Единственное исключение — это пробелы в начале строки и в конце, они игнорируются.

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€"`. Так как нам получить числовое значение из таких строк?

Для этого есть `parseInt` и `parseFloat`.

Они «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:

```
1  alert( parseInt('100px') ); // 100
2  alert( parseFloat('12.5em') ); // 12.5
3
4  alert( parseInt('12.3') ); // 12, вернётся только целая
5  alert( parseFloat('12.3.4') ); // 12.3, произойдёт оста
```

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

```
1  alert( parseInt('a123') ); // NaN, на первом символе пр
```

Второй аргумент `parseInt(str, radix)`

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

Другие математические функции

В JavaScript встроен объект Math, который содержит различные математические функции и константы.

Несколько примеров:

`Math.random()`

Возвращает псевдослучайное число в диапазоне от 0 (включительно) до 1 (но не включая 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (любое количество псевдо
```

`Math.max(a, b, c...)` / `Math.min(a, b, c...)`

Возвращает наибольшее/наименьшее число из перечисленных аргументов.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

Возвращает число `n`, возведённое в степень `power`

```
alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024
```

В объекте `Math` есть множество функций и констант, включая тригонометрические функции, подробнее можно ознакомиться в документации по объекту Math.

Итого

Чтобы писать числа с большим количеством нулей:

- Используйте краткую форму записи чисел – `"e"`, с указанным количеством нулей. Например: `123e6` это `123` с 6-ю нулями `123000000`.
- Отрицательное число после `"e"` приводит к делению числа на 1 с указанным количеством нулей. Например: `123e-6` это `0.000123` (123 миллионных).

Для других систем счисления:

- Можно записывать числа сразу в шестнадцатеричной (`0x`), восьмеричной (`0o`) и бинарной (`0b`) системах счисления
- `parseInt(str, base)` преобразует строку в целое число в соответствии с указанной системой счисления: `2 ≤ base ≤ 36`.
- `num.toString(base)` представляет число в строковом виде в указанной системе счисления `base`.

Для преобразования значений типа `12pt` и `100px` в число:

- Используйте `parseInt/parseFloat` для «мягкого» преобразования строки в число, данные функции по порядку считывают число из строки до тех пор пока не возникнет ошибка.

Для дробей:

- Используйте округления `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` или `num.toFixed(precision)`.
- Помните, что при работе с дробями происходит потеря точности.

Ещё больше математических функций:

- Документация по объекту Math. Библиотека маленькая, но содержит всё самое важное.

Задачи

Сумма пользовательских чисел

важность: 5

Создайте скрипт, который запрашивает ввод двух чисел (используйте prompt) и после показывает их сумму.

Запустить демо

P.S. Есть «подводный камень» при работе с типами.

решение

Почему 6.35.toFixed(1) == 6.3?

важность: 4

Методы `Math.round` и `toFixed`, согласно документации, округляют до ближайшего целого числа: `0..4` округляется в меньшую сторону, тогда как `5..9` в большую сторону.

Например:

```
1  alert( 1.35.toFixed(1) ); // 1.4
```

Но почему в примере ниже 6.35 округляется до 6.3?

```
1  alert( 6.35.toFixed(1) ); // 6.3
```

Как правильно округлить `6.35`?

решение

Ввод числового значения

важность: 5

Создайте функцию `readNumber`, которая будет запрашивать ввод числового значения до тех пор, пока посетитель его не введёт.

Функция должна возвращать числовое значение.

Также надо разрешить пользователю остановить процесс ввода, отправив пустую строку или нажав «Отмена». В этом случае функция должна вернуть `null`.

Запустить демо

Открыть песочницу с тестами для задачи.

решение

Поделиться

Редактировать на GitHub

Бесконечный цикл по ошибке

важность: 4

Этот цикл – бесконечный. Он никогда не завершится, почему?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

решение

Случайное число от min до max

важность: 2

Встроенный метод `Math.random()` возвращает случайное число от `0` (включительно) до `1` (но не включая `1`)

Напишите функцию `random(min, max)`, которая генерирует случайное число с плавающей точкой от `min` до `max` (но не включая `max`).

Пример работы функции:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

решение

Случайное целое число от min до max

важность: 2

Напишите функцию `randomInteger(min, max)`, которая генерирует случайное *целое* (integer) число от `min` до `max` (включительно).

Любое число из интервала `min..max` должно появляться с одинаковой вероятностью.

Пример работы функции:

```
alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```

Можно использовать решение из предыдущей задачи.

решение

Проводим курсы по JavaScript и фреймворкам.

Комментарии

перед тем как писать...