14 мая 2023 г.

Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

- `setTimeout` позволяет вызвать функцию **один раз** через определённый интервал времени.
- `setInterval` позволяет вызывать функцию **регулярно**, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout

Синтаксис:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...);
```

Параметры:

`func|code`

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

`delay`

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

`arg1`, `arg2`…

Аргументы, передаваемые в функцию

Например, данный код вызывает `sayHi()` спустя одну секунду:

```
function sayHi() {
  alert('Привет');
}

setTimeout(sayHi, 1000);
```

С аргументами:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон
```

Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

Это также будет работать:

```
setTimeout("alert('Привет')", 1000);
```

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

```
setTimeout(() => alert('Привет'), 1000);
```

Передавайте функцию, но не запускайте её

Начинающие разработчики иногда ошибаются, добавляя скобки `()` после функции:

```
// неправильно!
setTimeout(sayHi(), 1000);
```

Это не работает, потому что `setTimeout` ожидает ссылку на функцию. Здесь `sayHi()` запускает выполнение функции, и *результат выполнения* отправляется в `setTimeout`. В нашем случае результатом выполнения `sayHi()` является `undefined` (так как функция ничего не возвращает), поэтому ничего не планируется.

Отмена через clearTimeout

Вызов `setTimeout` возвращает «идентификатор таймера» `timerId`, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

В коде ниже планируем вызов функции и затем отменяем его (просто передумали). В результате ничего не происходит:

```
let timerId = setTimeout(() => alert("ничего не происходит"), 1000);
alert(timerId); // идентификатор таймера


```

```
clearTimeout(timerId);
alert(timerId); // тот же идентификатор (не принимает значение null после отме
```

Как мы видим из вывода `alert`, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

Повторюсь, что нет единой спецификации на эти методы, поэтому такое поведение является нормальным.

Для браузеров таймеры описаны в разделе таймеров стандарта HTML5.

setInterval

Метод `setInterval` имеет такой же синтаксис как `setTimeout`:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...);
```

Все аргументы имеют такое же значение. Но отличие этого метода от `setTimeout` в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать `clearInterval(timerId)`.

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

```
// повторить с интервалом 2 секунды
let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Во время показа `alert` время тоже идёт

В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа `alert/confirm/prompt`.

Так что если вы запустите код выше и подождёте с закрытием `alert` несколько секунд, то следующий `alert` будет показан сразу, как только вы закроете предыдущий. Интервал времени между сообщениями `alert` будет короче, чем 2 секунды.

Вложенный setTimeout

Есть два способа запускать что-то регулярно.

Один из них `setInterval`. Другим является вложенный `setTimeout`. Например:

```
/** вместо:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');

```

```
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

Метод `setTimeout` выше планирует следующий вызов прямо после окончания текущего `(*)`.

Вложенный `setTimeout` – более гибкий метод, чем `setInterval`. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.

Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд… Вот псевдокод:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  ...отправить запрос...

  if (ошибка запроса из-за перегрузки сервера) {
    // увеличить интервал для следующего запроса
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

А если функции, которые мы планируем, ресурсоёмкие и требуют времени, то мы можем измерить время, затраченное на выполнение, и спланировать следующий вызов раньше или позже.

Вложенный `setTimeout` позволяет задать задержку между выполнениями более точно, чем `setInterval`.

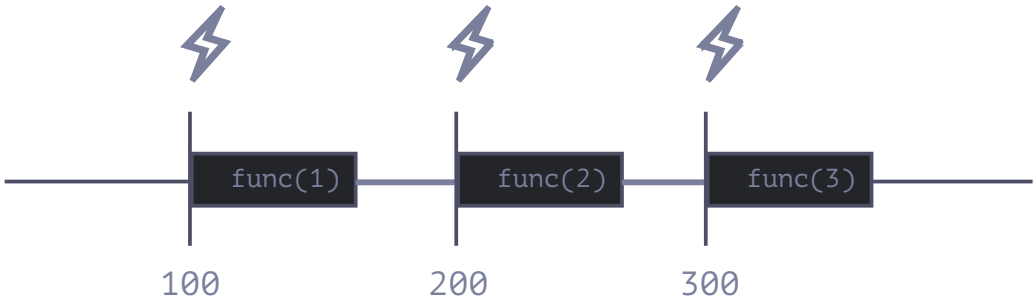
Сравним два фрагмента кода. Первый использует `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i);
}, 100);
```

Второй использует вложенный `setTimeout`:

```
let i = 1;
setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

Для `setInterval` внутренний планировщик будет выполнять `func(i)` каждые 100 мс:



Обратили внимание?

Реальная задержка между вызовами `func` с помощью `setInterval` меньше, чем указано в коде!

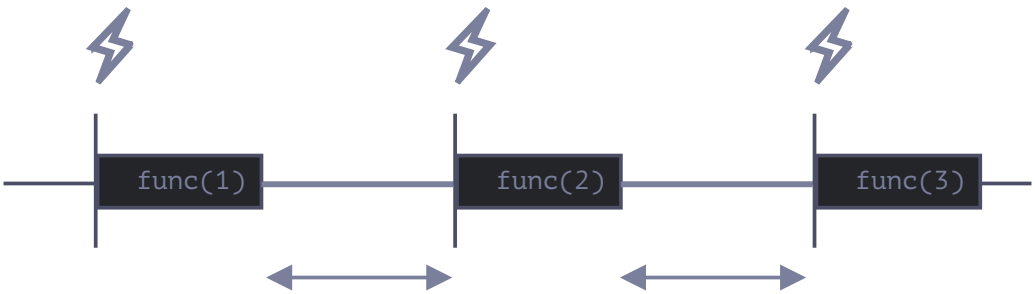
Это нормально, потому что время, затраченное на выполнение `func`, использует часть заданного интервала времени.

Вполне возможно, что выполнение `func` будет дольше, чем мы ожидали, и займёт более 100 мс.

В данном случае движок ждёт окончания выполнения `func` и затем проверяет планировщик и, если время истекло, *немедленно* запускает его снова.

В крайнем случае, если функция всегда выполняется дольше, чем задержка `delay`, то вызовы будут выполняться без задержек вообще.

Ниже представлено изображение, показывающее процесс работы рекурсивного `setTimeout`:



Вложенный `setTimeout` гарантирует фиксированную задержку (здесь 100 мс).

Это потому, что новый вызов планируется в конце предыдущего.

Сборка мусора и колбэк setTimeout/setInterval

Когда функция передаётся в `setInterval/setTimeout`, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик мусора, даже если на неё нет других ссылок.

```
// функция остаётся в памяти до тех пор, пока планировщик обращается к ней
setTimeout(function() {...}, 100);
```

Для `setInterval` функция остаётся в памяти до тех пор, пока не будет вызван `clearInterval`.

Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

setTimeout с нулевой задержкой

Особый вариант использования: `setTimeout(func, 0)` или просто `setTimeout(func)`.

Это планирует вызов `func` настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода.

Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

```
1  setTimeout(() => alert("Мир"));
2
3  alert("Привет");
```

Первая строка помещает вызов в «календарь» через 0 мс. Но планировщик проверит «календарь» только после того, как текущий код завершится. Поэтому `"Привет"` выводится первым, а `"Мир"` – после него.

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе Событийный цикл: микрозадачи и макрозадачи.

Минимальная задержка вложенных таймеров в браузере

В браузере есть ограничение на то, как часто внутренние счётчики могут выполняться. В стандарте HTML5 говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов `setTimeout` повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве `times`. Какова реальная задержка? Посмотрим:

```
1  let start = Date.now();
2  let times = [];
3
4  setTimeout(function run() {
5    times.push(Date.now() - start); // запоминаем задержку от предыдущего вы
6
7    if (start + 100 < Date.now()) alert(times); // показываем задержку через
8    else setTimeout(run); // если нужно ещё запланировать
9  });
10
11 // пример вывода:
12 // 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

Первый таймер запускается сразу (как и указано в спецификации), а затем задержка вступает в игру, и мы видим `9, 15, 20, 24...`.

Аналогичное происходит при использовании `setInterval` вместо `setTimeout`: `setInterval(f)` запускает `f` несколько раз с нулевой задержкой, а затем с задержкой 4+ мс.

Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам.

Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, setImmediate для Node.js. Так что это ограничение относится только к браузерам.

Итого

- Методы `setInterval(func, delay, ...args)` и `setTimeout(func, delay, ...args)` позволяют выполнять `func` регулярно или только один раз после задержки `delay`, заданной в мс.
- Для отмены выполнения необходимо вызвать `clearInterval/clearTimeout` со значением, которое возвращают методы `setInterval/setTimeout`.
- Вложенный вызов `setTimeout` является более гибкой альтернативой `setInterval`. Также он позволяет более точно задать интервал между выполнениями.
- Планирование с нулевой задержкой `setTimeout(func,0)` или, что то же самое, `setTimeout(func)` используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.
- Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами `setTimeout`, а также для `setInterval`, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

- Перегружен процессор.
- Вкладка браузера в фоновом режиме.
- Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Задачи

Вывод каждую секунду

важность: 5

Напишите функцию `printNumbers(from, to)`, которая выводит число каждую секунду, начиная от `from` и заканчивая `to`.

Сделайте два варианта решения.

1. Используя `setInterval`.
2. Используя рекурсивный `setTimeout`.

решение

Используем `setInterval`:

```
function printNumbers(from, to) {
  let current = from;

  let timerId = setInterval(function() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
```

```
    current++;
  }, 1000);
}

// использование:
printNumbers(5, 10);
```

Используем рекурсивный `setTimeout` :

```
function printNumbers(from, to) {
  let current = from;

  setTimeout(function go() {
    alert(current);
    if (current < to) {
      setTimeout(go, 1000);
    }
    current++;
  }, 1000);
}

// использование:
printNumbers(5, 10);
```

Заметим, что в обоих решениях есть начальная задержка перед первым выводом. Она составляет одну секунду (1000мс). Если мы хотим, чтобы функция запускалась сразу же, то надо добавить такой запуск вручную на отдельной строке, вот так:

```
function printNumbers(from, to) {
  let current = from;

  function go() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }

  go();
  let timerId = setInterval(go, 1000);
}

printNumbers(5, 10);
```

Что покажет setTimeout?

важность: 5

В приведённом ниже коде запланирован вызов `setTimeout` , а затем выполняется сложное вычисление, для завершения которого требуется более 100 мс.

Когда будет выполнена запланированная функция?

1. После цикла.
2. Перед циклом.
3. В начале цикла.

Что покажет `alert` ?

```
let i = 0;

setTimeout(() => alert(i), 100); // ?

// предположим, что время выполнения этой функции >100 мс
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

решение

Любой вызов `setTimeout` будет выполнен только после того, как текущий код завершится.

Последним значением `i` будет: `100000000` .

```
let i = 0;

setTimeout(() => alert(i), 100); // 100000000

// предположим, что время выполнения этой функции >100 мс
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

Предыдущий урок

Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>` , для нескольких строк кода — тег `<pre>` , если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Юрий

месяц назад

⚡ **Хочешь понять как работают setTimeout и setInterval?**
Тогда переходи в ТГ-блог «Джун на фронте»!

👨‍💻 Автор - системный администратор, который с декабря 2021 года освоил HTML, CSS, JS, Vue, Nuxt, React Native, MongoDB и Node.js.

Следи за моим путем в мир разработки: от новичка до создателя 🤖 Телеграм-бота для автоматической отправки откликов!

💡 Ежедневно ты найдешь полезные ответы на насущные вопросы, анализ рынка вакансий и советы от человека, прошедшего этот путь.

Вбивайте «Джун на фронте» и присоединяйтесь к нам!

Zena

2 месяца назад edited

Решение через **setInterval**:

И через **setTimeout**:

0 0 Ответить

2 **29**

3 месяца назад edited

У меня проблема с return в теле функций setTimeout и setInterval

```
function i() {
    return 1
}
i1 = setTimeout(i, 1000) // не то что нужно
setTimeout(i1 = i(), 1000) // не то что нужно
```

Я не понимаю, как присвоить i1 значение, возвращаемое i() через 1 секунду?

upd: Окей ребят, я понял как сделать то что хотел, помощь больше не нужна
если что вот код который я пытался написать

```
dis = setInterval(
    function() {
        i = temp1.o[11].exports.player.gpData.getX(),
        dis1 = setTimeout(() =>
            {i1 = temp1.o[11].exports.player.gpData.getX(),
            console.log(i1-i),
            clearTimeout(dis1)}, 1000/6)},
    1000/3)
```

0 0 Ответить

0 **Odelschwank**

4 месяца назад edited

такое чудовище которое меняет и сам аргумент, но токо так получилось, с циклами не очень:

0 0 Ответить

Aristocrat

10 месяцев назад edited

"Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. " вызываю пояснительную бригаду.
Первая задача

```
 let timerId = setTimeout(function tick(from, to){
console.log(from);
if(from < to){
timerId = setTimeout(tick, 1000, from + 1, to);
}else{
clearInterval(timerId);
}
}, 1000, 1, 5);
```

0 0 Ответить

Maxim Aristocrat

10 месяцев назад

При поиске конкретного идентификатора переменной, происходит поиск через цепочку лексических окружений.

Вот ортодоксальный пример:

```
const closure = () => {
  let value = 0;
  return () => {
    return value++;
  };
}

const nextValue = closure();
nextValue(); /// 0
nextValue(); /// 1
nextValue(); /// 2
```

Данный пример хорош тем что функция возвращает другую функцию. Возвращаемая функция определяется внутри другой функции и соответственно когда вложенная функция использует переменную из другого окружения она опирается на механизм поиска переменной то есть происходит циклическое [[OuterEnv]] для получения Environment Record.

Надеюсь теперь все стало понятно.

1 0 Ответить

O **Odelschwank** Maxim

4 месяца назад

подскажи пожалуйста: почему setTimeout выводит абсолютно последним всегда, например:
function secondItem(){
setTimeout(() => alert("hello"), 10)

alert("first");
alert("next")
}
secondItem()

alert("last")

0 0 Ответить

Maxim Odelschwank

4 месяца назад edited

Ну во-первых вы должны знать что такое **макрозадача**.
Далее если вы понимаете что это такое, то функция, которая передается как аргумент в **setTimeout**, будет исполнена по завершению таймера и в соответствии с **очередью макрозадач**.

По сути функция **setTimeout** лишь планирует вызов функции, которая будет передана в качестве аргумента. А вот ее исполнение будет гораздо позже.

Когда вы берете код и запускаете его (тот что вы продемонстрировали) он является **макрозадачей**. Когда эта **макрозадача** завершится, (ее завершение это выполнение последней команды `alert("last")`), начнется ряд других специфических этапов **цикла событий** (намеренно их не раскрываю), финальным этапом **цикла событий** можно считать отрисовку кадра или пропуск отрисовки. Далее **цикл событий** повторяется он достает из очереди новую макрозадачу и исполняет ее (в вашем случае это функция `() => alert("hello")`), а далее все тоже самое (происходят определенные этапы **цикла событий** с отрисовкой кадра или пропуском).

P.S Если вы не знаете что такое **цикл событий**, **макро** и **микро задачи**, то настоятельно рекомендую найти главу в данном учебнике, которая вам это объяснит

показать больше

1 0 Ответить

O **Odelschwank** Maxim

4 месяца назад

спасибо, изучу

0 0 Ответить

A **Армен Черноморских** Maxim

9 месяцев назад edited

```
for(var i = 0; i < 5; i++){
setTimeout(function(){
console.log(i);
}, 100)
}

})
```

Подскажите, пожалуйста, если у каждой итерации создается новое Лексическое окружение(Environment Record если точнее) с текущим значением i, то из-за чего в консоле выводиться последнее значение i?Разве функция не должна запоминать Environment Record в котором она определена и выводить текущее значение i?

0 0 Ответить

A **Artyom Evsyukov** Армен Черноморских
3 месяца назад

Чтобы этот код заработал правильно, нужно добавить уникальную задержку, в противном случае все вызовы setTimeout получат одинаковую задержку. Все задачи будут добавлены в очередь с одинаковой задержкой, и все они будут выполняться почти одновременно после завершения цикла.

```
function printNumbers(from, to) {
for (let current = from; current <= to; current++) {
setTimeout(() => {
console.log(current);
}, 500 * (current - from + 1));
}
}
printNumbers(-3, 9);
```

0 0 Ответить

Maxim Армен Черноморских
9 месяцев назад edited

Конструкция for (не in/of) может создавать разное поведение:

1) Когда вы используете var для объявления переменных.
2) Когда вы используете let/const для переменных.

Собственно два эти случая порождают разные цепочки Environment Record.

Итак выполнение for делится на 2 этапа, это выполнение головы и выполнение тела.
За голову отвечает все то что находится в for(...), за тело все то что находится после круглых скобок.

Алгоритм выполнения головы зависит от того var или let/const вы там объявляете.
Если вы пишите var, то алгоритм головы никаких Environment Record не создает, он считает что объявление var для него такое же если бы вы этот var сделали до самой конструкции for. Но вот если у вас объявление let/const тогда алгоритм головы создает Environment Record, чтобы записать туда ваши объявления.

Промежуточное резюме: for с var не создает Environment Record, а let/const создает.

Насчет тела, вот тут алгоритм один, но есть тонкости. Итак, когда алгоритм тела запускается он создает дополнительный Environment Record, но только в том случае если из алгоритма головы

показать больше

4 0 Ответить

A **Армен Черноморских** Maxim
9 месяцев назад

Значит мой код для головы и тела не создает Enviroment Record, а после цикла функция будет искать переменную сразу в глобальном окружении где i = 10?

0 0 Ответить

Maxim Армен Черноморских
9 месяцев назад edited

Ваш код не создает конструкцией for дополнительных Environment Record, но как я упоминал BlockStatement создает и их будет 5 штук, на каждую итерацию. Так что никто не отменял

лексической природы остальных конструкций :)

И дабы доказать вам это я добавлю лишь одну строчку в ваш код:

```
for(var i = 0; i < 5; i++){
  let inc = i;
  setTimeout(function(){
    console.log(inc);
  }, 100)
}
```

После цикла ваша переменная var i будет доступна за пределами конструкции for, на уровне текущего Execution Context, а это может быть как глобальное окружение, так и функция.

Если "после цикла" вы имеете ввиду выполнение callback функции у setTimeout, то callback воспользуется полем [[Environment]] (это Environment Record - окружение в котором функция была определена), которое при вызове функции установит поле [[OuterEnv]] в значение [[Environment]]. Ну и так как у нас таких определений 5 (причина BlockStatement) то на каждую итерацию будет происходить определение [[Environment]] и в последствии установка [[OuterEnv]] и все из-за

показать больше

1 0 Ответить

Армен Черноморских Maxim

6 месяцев назад

Как-то давно в этом учебнике прочитал интересную вещь, об операторах, и так и не понял, что означает "Большинство операторов в JavaScript возвращают значение"? Подставлют вместо своего вызова какое-то значение как функции?Не знаю у кого ещё можно спросить кроме вас.

0 0 Ответить

Maxim Армен Черноморских

6 месяцев назад edited

Да это просто, результат работы оператора это конкретный результат взаимодействия оператора с операндом (насчет большинства я сомневаюсь, я бы категорично сказал что любой оператор после работы с ним возвращает конкретное значение)

Примеры:
1) Оператор **typeof** *<value>*. Если вы вместо *<value>* подставите что угодно, то работа оператора **typeof** вернет вам конкретное значение, которое определяет спецификация.
2) Любой унарный оператор (**delete**, **void**, **typeof**, **+**, **-**, **~**, **!**, **await**), тоже что-то вернут в результате своей работы.
3) Условный оператор или еще его называют, тернарным оператором - тоже вернет результат, который будет зависеть от условия.
4) итд

И можно много приводить разных операторов они что-то да и будут возвращать (пример: `typeof (5 + 5)` - тут участвуют три оператора (**typeof**, **additive +** и оператор группировки **()**)). И даже конструкции утверждений (как пример **if...else**) тоже возвращают результат (напишите в консоли: `if (true) {"cool"}` - и вы увидите что возвращает данная языковая конструкция). Вы бы спросили ну а почему я не могу совместить операторы и конструкции вида **if...else** - дело все в грамматике языка,

показать больше

0 0 Ответить

Армен Черноморских → Maxim
6 месяцев назад

А в других языках програмрования операторы по такому же принципу работают ?

0 0 Ответить

Maxim → Армен Черноморских
6 месяцев назад

Да, а почему должно быть иначе?!

0 0 Ответить

Армен Черноморских → Maxim
6 месяцев назад

Да так, на всякий случай спросил.

0 0 Ответить

Maxim → Армен Черноморских
6 месяцев назад

Чтобы не спрашивать вам нужно изучить грамматику того или иного языка. Как правило это не сложно если хоть раз попробовать это сделать.

0 0 Ответить

Загрузить ещё комментарии