https://habr.com/ru/companies/vk/articles/533822/

Перебираемые (или итерируемые) объекты – это ?????????

**[Symbol.iterator](https://learn.javascript.ru/iterable" \l "symbol-iterator)**

Чтобы сделать например объект итерируемым (и позволить for..of работать с ним), нужно добавить в объект метод - Symbol.iterator.

1. Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден - TypeError: obj is not iterable). Этот метод должен вернуть *итератор* – объект с методом next.
2. Дальше for..of работает *только с этим возвращённым объектом*.
3. Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.
4. Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что цикл завершён, в противном случае value содержит очередное значение.

Задание: написать итератор для range

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// ...она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором,

// запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает!

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

* У самого range нет метода next().
* Вместо этого другой объект, так называемый «итератор», создаётся вызовом range[Symbol.iterator](), и именно его next() генерирует значения.

Таким образом, объект итератор отделён от самого итерируемого объекта.

Технически мы можем объединить их и использовать сам range как итератор, чтобы упростить код.

Задание: переписать итератор внутри range

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

this.current = this.from;

return this;

},

next() {

if (this.current <= this.to) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next(), и он запоминает текущее состояние итерации в this.current. Короче? Да. И иногда такой способ тоже хорош.

Недостаток такого подхода в том, что теперь мы не можем использовать этот объект в двух параллельных циклах for..of: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах for..of, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

**Бесконечные итераторы**

Можно сделать бесконечный итератор. Например, range будет бесконечным при range.to = Infinity. Или мы можем создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод next не имеет ограничений, он может возвращать всё новые и новые значения, это нормально.

Конечно же, цикл for..of с таким итерируемым объектом будет бесконечным. Но мы всегда можем прервать его, используя break.

## [Строка – перебираемый объект](https://learn.javascript.ru/iterable" \l "stroka-perebiraemyy-obekt)

Среди встроенных перебираемых объектов наиболее широко используются массивы и строки.

Для строки for..of перебирает символы:

for (let char of "test") {

// срабатывает 4 раза: по одному для каждого символа

alert( char ); // t, затем e, затем s, затем t

}

## [Явный вызов итератора](https://learn.javascript.ru/iterable" \l "yavnyy-vyzov-iteratora)

Можно перебрать строку точно так же, как цикл for..of, но вручную, прямыми вызовами.

**Задание: перебрать строковый итератор вруную**

Нижеприведённый код получает строковый итератор и берёт из него значения:

let str = "Hello";

// делает то же самое, что и

// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {

let result = iterator.next();

if (result.done) break;

alert(result.value); // выводит символы один за другим

}

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем for..of. Например, мы можем разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё и потом продолжить.

## [Итерируемые объекты и псевдомассивы](https://learn.javascript.ru/iterable" \l "array-like)

* *Итерируемые объекты* – это объекты, которые реализуют метод Symbol.iterator, как было описано выше.
* *Псевдомассивы* – это объекты, у которых есть индексы и свойство length, то есть, они выглядят как массивы

При использовании JavaScript в браузере или других окружениях мы можем встретить объекты, которые являются итерируемыми или псевдомассивами, или и тем, и другим.

Например, строки итерируемы (для них работает for..of) и являются псевдомассивами (они индексированы и есть length).

Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым.

Например, объект range из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и length.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

**Задание: написать пример неитерируемого псевдомассива**

let arrayLike = { // есть индексы и свойство length => псевдомассив

0: "Hello",

1: "World",

length: 2

};

// Ошибка (отсутствует Symbol.iterator)

for (let item of arrayLike) {}

Что у них общего? И итерируемые объекты, и псевдомассивы – это обычно не массивы, у них нет методов push, pop и т.д.

## [Array.from](https://learn.javascript.ru/iterable" \l "array-from)

Есть универсальный метод [Array.from](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/from), который принимает итерируемый объект или псевдомассив и делает из него «настоящий» Array. После этого мы уже можем использовать методы массивов.

Например псевдомассив в массив:

let arrayLike = {

0: "Hello",

1: "World",

length: 2

};

let arr = Array.from(arrayLike); // (\*)

alert(arr.pop()); // World (метод работает)

Array.from в строке (\*) принимает объект, проверяет, является ли он итерируемым объектом или псевдомассивом, затем создаёт новый массив и копирует туда все элементы.

То же самое происходит с итерируемым объектом

// range взят из примера в начале статьи

let arr = Array.from(range);

alert(arr); // 1,2,3,4,5 (преобразование массива через toString работает)

Полный синтаксис Array.from позволяет указать необязательную «трансформирующую» функцию:

Array.from(obj[, mapFn, thisArg])

Необязательный второй аргумент может быть функцией, которая будет применена к каждому элементу перед добавлением в массив, а thisArg позволяет установить this для этой функции.

// range взят из примера в начале статьи

// возводим каждое число в квадрат

let arr = Array.from(range, num => num \* num);

alert(arr); // 1,4,9,16,25

**Суррогатные пары**

Это когда закодированы смайлики например, и на один смайлик приходится несколько символов.

console.log('𝒳😂')

Например можно написать такое, и в консоли действительно отобразится смайлик.

Если такую строку перебрать классическим циклом for, в консоли будет следующее

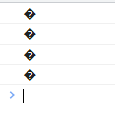
let str = '𝒳😂';

for(let i = 0; i < str.length; i++){

    console.log(str[i])

}

Вывод консоли:



А вот цикл for…of отрабатывает как нужно:

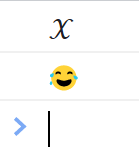
      let str = '𝒳😂';

      for(i of str){

        console.log(i)

      }

Вывод в консоль:



Чтобы превратить строку, содержащую суррогатные пары в массив, метод split не подойдет:

let str = '𝒳😂';

let arr = str.split('');

console.log(arr)

Результат:



Для таких случаев необходимо использовать Array.from:

        let str = '𝒳😂';

        let arr = Array.from(str);

        console.log(arr)

Результат:



В отличие от str.split, этот метод в работе опирается на итерируемость строки, и поэтому, как и for..of, он корректно работает с суррогатными парами.

Объекты, которые можно использовать в цикле for..of, называются *итерируемыми*.

* Технически итерируемые объекты должны иметь метод Symbol.iterator.
  + Результат вызова obj[Symbol.iterator] называется *итератором*. Он управляет процессом итерации.
  + Итератор должен иметь метод next(), который возвращает объект {done: Boolean, value: any}, где done:true сигнализирует об окончании процесса итерации, в противном случае value – следующее значение.
* Метод Symbol.iterator автоматически вызывается циклом for..of, но можно вызвать его и напрямую.
* Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод Symbol.iterator.
* Строковый итератор знает про суррогатные пары.

Объекты, имеющие индексированные свойства и length, называются *псевдомассивами*. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Если мы заглянем в спецификацию, мы увидим, что большинство встроенных методов рассчитывают на то, что они будут работать с итерируемыми объектами или псевдомассивами вместо «настоящих» массивов, потому что эти объекты более абстрактны.

Array.from(obj[, mapFn, thisArg]) создаёт настоящий Array из итерируемого объекта или псевдомассива obj, и затем мы можем применять к нему методы массивов. Необязательные аргументы mapFn и thisArg позволяют применять функцию с задаваемым контекстом к каждому элементу.