



СИМВОЛЫ, ИТЕРАТОРЫ И ГЕНЕРАТОРЫ



ИГОРЬ КАМЫШЕВ / BREADHEAD



ИГОРЬ КАМЫШЕВ

Tech Lead в Breadhead



igor@kamyshev.me



[igorkamyshev](https://t.me/igorkamyshev)

ПЛАН ЗАНЯТИЯ

- Символы
 - Историческая справка
 - Что такое символ
 - Как их использовать
- Итераторы
 - Для чего нужны итераторы
 - Как реализовать свой итератор
- Генератор
 - Подпрограммы
 - Функция, которую можно прервать
 - Генератор как итератор
 - Тонкие материи генераторов
- Резюме

СИМВОЛЫ

Новый примитивный тип данных `Symbol` служит для создания уникальных идентификаторов.

ИСТОРИЧЕСКАЯ СПРАВКА

Когда JS был молод и незрел, многие авторы библиотек применяли monkey patching стандартных вещей языка. Например, добавляли новые методы для массивов, чтобы с ними было удобнее работать.

```
1 | Array.prototype.print = function () {
2 |   console.log(this)
3 |
4 |
5 | ['ds'].print() // ["ds"]
```

В этом есть определенные опасности:

- Если разработчик установит две библиотеки, которые по-разному определяют один и тот же метод массива, одна из них перестанет работать.
- Если в стандарт языка добавят аналогичную функцию, которая работает иначе, библиотека тоже сломается.

Потому такой путь считается плохим. С этим нужно было как-то бороться, и решение нашлось.

ЧТО ТАКОЕ «СИМВОЛ»

Это примитивный тип данных. Любые два символа отличны друг от друга.

```
1 let sym1 = Symbol();
2 let sym2 = Symbol();
3
4 console.log(sym1 === sym2); // false
```

Это свойство символов дает большой простор для написания хорошего кода.

Мы рассмотрели создание «локальных» символов.

Локальный символ – символ сохраненный в переменную и недоступный никак иначе.

Бывают еще «глобальные» символы. Они хранятся в «специальном» реестре и могут быть получены (или созданы) по имени.

```
1 // создание символа в реестре
2 let name = Symbol.for("name");
3
4 // символ уже есть, чтение из реестра
5 console.log(Symbol.for("name") == name); // true
```

Глобальный символ – символ сохраненный в глобальном реестре и доступный через него.

КАК ИСПОЛЬЗОВАТЬ СИМВОЛЫ

Есть несколько сфер применения символов. Но чаще всего их используют как ключи для полей объекта.

```
1 let isAdmin = Symbol("isAdmin");
2
3 let user = {
4     name: "Вася",
5     [isAdmin]: true
6 };
7
8 console.log(user[isAdmin]); // true
```

Это как раз решает проблему, рассмотренную в начале. Так как, даже если авторы языка (или авторы другой библиотеки) добавят ЛЮБОМУ объекту поле `isAdmin`, это не повлияет на наш код.

Важно! Поля, имена которых являются символами, не участвуют в итерации.

```
1 let user = {  
2   name: "Вася",  
3   age: 30,  
4   [Symbol.for("isAdmin")]: true  
5 };  
6  
7 // в цикле for..in также не будет символа  
8 console.log(Object.keys(user)); // name, age  
9  
10 // доступ к свойству через глобальный символ – работает  
11 console.log(user[Symbol.for("isAdmin")]);
```

Благодаря использованию символов у разработчиков языка появляется возможность развивать язык, гарантированно не ломая уже существующий код.

Иногда нам все же нужно узнать, какие есть поля в объекте (включая символы). Для этого есть специальный синтаксис.

```
1 let obj = {  
2   iterator: 1,  
3   [Symbol.iterator]: function() {}  
4 }  
5  
6 // один символ в объекте  
7 console.log(Object.getOwnPropertySymbols(obj));  
8 // Symbol(Symbol.iterator)  
9  
10 // и одно обычное свойство  
11 console.log(Object.getOwnPropertyNames(obj));  
12 // iterator
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ

Пусть нам строго необходимо добавить к прототипу массива новый метод `head`, который возвращал бы первый элемент массива или `undefined`. Просто в прямую добавлять его опасно, вдруг когда-нибудь появиться такой метод в стандарте языка.

Создадим символ и используем его:

```
1 const headSymbol = Symbol.for("array-head");
2
3 Array.prototype[headSymbol] = function() {
4   console.log(this[0])
5 }
```

Теперь мы можем использовать его в любом месте абсолютно безопасно.

```
1 const head = Symbol.for("array-head");
2
3 const arr1 = [0, 1, 2, 3, 4];
4 console.log(arr1[head]()); // 0
5
6 const arr2 = [];
7 console.log(arr2[head]()); // undefined
```

Все работает, как мы и планировали!

ИТОГО

- Символы – примитивный тип, предназначенный для уникальных идентификаторов.
- Все символы уникальны. Даже символы с одинаковым именем не равны друг другу.
- Существует глобальный реестр символов, доступных через метод `Symbol.for("name")`. Для глобального символа можно получить имя вызовом `Symbol.keyFor(sym)`.
- Основная область использования символов – это системные свойства объектов, которые задают разные аспекты их поведения. Системные символы позволяют разработчикам стандарта добавлять новые «особые» свойства объектов, при этом не резервируя соответствующие строковые значения.
- Системные символы доступны как свойства функции `Symbol`, например `Symbol.iterator`.
- Можно создавать и свои символы, использовать их в объектах. Записывать их как свойства `Symbol`, разумеется, нельзя. Если нужен глобально доступный символ, то используется `Symbol.for(имя)`.

ИТЕРАТОРЫ

ИТЕРАТОРЫ

Итерируемые объекты – это особенные структуры, которые позволяют перебирать содержимое в цикле. Еще их называют «перебираемыми» объектами. Такой концепт существует во многих языках, в том числе в JavaScript.

Примеры итерируемых объектов: массив, список DOM-узлов.

Так же на основе итерируемых объектов построена работа оператора spread `f(...args)`.

Массив – только частный случай итерируемого объекта. Потому перебираемые объекты не обязаны иметь длины `length` и других характеристик, присущих массивам.

СОЗДАЕМ ИТЕРАТОР

С помощью итераторов мы можем добавить «итерируемость» любому объекту. Итак, пусть у нас есть объект, который мы хотим перебрать.

Например, `range` – диапазон чисел от `from` до `to`. Нужно, чтобы `for (let num of range)` «перебирал» этот объект (перечислял числа от `from` до `to`).

Исходный объект:

```
1 let range = {  
2   from: 1,  
3   to: 5  
4 };
```

Добавим возможность итерироваться по нему. Для этого нужно просто создать в нём метод с названием `Symbol.iterator` (системный символ). При вызове такого метода перебираемый объект должен возвращать другой объект-итератор, который и осуществляет перебор. У такого объекта должен быть метод `next()`, который при каждом вызове возвращает очередное значение и проверяет, окончен ли перебор.

```
1 let range = {  
2   from: 1,  
3   to: 5  
4 }  
5  
6 // сделаем объект range итерируемым  
7 range[Symbol.iterator] = function() {  
8  
9   let current = this.from;  
10  let last = this.to;  
11  
12  // метод должен вернуть объект с методом next()  
13  return {  
14    next() {  
15      if (current <= last) {  
16        return {  
17          done: false,  
18          value: current++  
19        };  
20      } else {  
21        return {  
22          done: true  
23        };  
24      }  
25    }  
26  }  
27};  
28  
30 for (let num of range) {  
31   console.log(num);  
32 }  
33 // 1, затем 2, 3, 4, 5  
34  
35 console.log(Math.max(...range)); // 5
```

Можно сделать и бесконечный итератор. Например, наш объект станет таким при `range.to = Infinity`. Другие примеры бесконечных итераторов: последовательность случайных чисел, последовательность простых чисел.

Ограничений на `next` не предусмотрено, он может возвращать значения сколько угодно раз.

Внимание! Цикл `for .. of` по такому итератору тоже будет бесконечным, нужно его прерывать в ручном режиме, например, через `break`.

ИТОГО

- Итератор – объект, предназначенный для перебора другого объекта.
- У итератора должен быть метод `next()`, возвращающий объект `{ done: Boolean, value: any }`, где `value` – очередное значение, а `done: true` указывает на окончание итерации.
- Метод `Symbol.iterator` предназначен для получения итератора из объекта. Цикл `for..of` делает это неявно, но можно и вызвать его напрямую.
- В JS есть много мест, где вместо массива используются более абстрактные итерируемые объекты, например оператор `spread ...`.
- Многие встроенные объекты, такие как массивы и строки, являются итерируемыми.

ГЕНЕРАТОР

ГЕНЕРАТОРЫ

В Java Script существуют функции, выполнение которых может быть приостановлено. После этого возвращается промежуточный результат и функция продолжает выполнение, когда это необходимо. Они называются **генераторами**.

Пример функции, создающей генератор:

```
1 function* generateNumbers() {  
2     yield 111;  
3     yield 222;  
4     return 333;  
5 }
```

Если запустить `generateNumbers`, то ее тело не выполниться. Она вернет объект-генератор, которым мы и будем пользоваться в дальнейшем.

```
1 function* generateNumbers() {  
2     yield 111;  
3     yield 222;  
4     return 333;  
5 }  
6  
7 let generator = generateNumbers();  
8  
9 let one = generator.next();  
10  
11 console.log(one); // { value: 111, done: false }
```

После этого функция останавливается на время и ожидает следующего вызова `next()`.

ГЕНЕРАТОР – ИТЕРАТОР

Любой генератор является итерируемым, его можно перебирать через

`for..of`:

```
1 function* generateNumbers() {
2     yield 111;
3     yield 222;
4     return 333;
5 }
6
7 let generator = generateNumbers();
8
9 for(let value of generator) {
10 console.log(value);
11 }
12 // 111, затем 222
```

Обратите внимание, что 3 не выведется. Так как `for..of` проходит только по тем значениям, где `done: false`, а у последнего значения `done: true`.

ТОНКИЕ МАТЕРИИ ГЕНЕРАТОРОВ

Композиция

Важно вкладывать один генератор в другой. Это называется композицией.

```
1 function* generateSequence(start, end) {
2     for (let i = start; i <= end; i++) yield i;
3 }
4
5 function* generateAlphaNum() {
6
7     // 0..9
8     yield* generateSequence(48, 57);
9
10    // A..Z
11    yield* generateSequence(65, 90);
12
13    // a..z
14    yield* generateSequence(97, 122);
15
16 }
17
18 let str = '';
19
20 for(let code of generateAlphaNum()) {
21     str += String.fromCharCode(code);
22 }
23
24 console.log(str); // 0..9A..Za..z
```

Работает это самым прямолинейным образом, просто выполняя генераторы по порядку.

Внимание! Конструкция `yield*` применима только к другому генератору.

ГЕНЕРАТОР – ТУДА И ОБРАТНО

Иногда требуется не только получать данные, но и отправлять какие-то данные в генератор. В языке предусмотрена такая возможность.

```
1 function* gen() {
2     // Передать вопрос во внешний код и подождать ответа
3     let result = yield "2 + 2?";
4
5     console.log(result);
6 }
7
8 let generator = gen();
9
10 let question = generator.next().value;
11 // "2 + 2?"
12
13 setTimeout(() => generator.next(4), 2000);
14
15 // Через 2 секунды выведется 4
```

Иногда может потребоваться вместо передачи генератору значения возбудить ошибку. Есть и такая возможность.

```
1 function* gen() {
2     try {
3         // в этой строке возникнет ошибка
4         let result = yield "Сколько будет 2 + 2?";
5
6         console.log("выше будет исключение ^^^");
7     } catch(e) {
8         console.log(e); // выведет ошибку
9     }
10}
11
12 let generator = gen();
13
14 let question = generator.next().value;
15
16 generator.throw(new Error("не знаю"));
```

ЗАДАЧА

Боевая задача – дан CSV файл со списком товаров магазина. Необходимо для каждой строки выполнять ряд действий, запрашивая подтверждение у пользователя.

Как мы хотим использовать генератор?

```
1 | for(let line of readFileByLines()) {  
2 |   // Разные манипуляции над строкой.  
3 | }
```

Теперь нужно реализовать функцию `readFileByLines`.

```
1 function* readFileByLines() {
2     let currentLine = 0;
3
4     const lineCount = getLineCount();
5     // Возвращает число строк в файле
6
7     while (currentLine < LineCount) {
8         yield readLine(currentLine);
9         // Получает указанную строку из файла
10        currentLine += 1;
11    }
12}
```

Теперь наше решение работает как и ожидается. Отлично!

ИТОГО

- Генераторы создаются при помощи особенных функций – функций-генераторов `function*(...)` `{...}`.
- Внутри генераторов и только внутри них разрешён оператор `yield`, он отдаёт одно из значений «наружу».
- Мы можем передать данные в генераторе при вызове метода `next`.

РЕЗЮМЕ

Зачем нужны символы? Почему нельзя просто писать как раньше?

Они позволяют избегать коллизий имен между разработчиками библиотек/разработчиками языка.

Зачем нужны итераторы, можно же делать всё «циклами и массивами»

Они позволяют писать более «семантичный код», который проще читать и понимать другим людям или самому разработчику через некоторое время.

Когда мне может понадобиться создавать итератор?

Когда создаю любой объект, по которому хочется итерироваться.

Чем итераторы отличаются от генераторов?

Генератор – это подвид итератора, реализует его интерфейс.

Когда использовать итераторы, а когда генераторы?

Итераторы – когда необходимо итерироваться по любому объекту.

Генераторы – когда нужно приостанавливать выполнение функции по той или иной причине.



Ваши вопросы?

ИГОРЬ КАМЫШЕВ



igor@kamyshev.me



[igorkamyshev](https://t.me/igorkamyshev)