ИТЕРАТОРЫ

ВСПОМНИМ ПРО ИТЕРАТОРЫ

У любого массива по символьному ключу `Symbol.iterator` доступен метод, который вернет объект-итератор:

```
const names = ['Иван', 'Николай', 'Татьяна'];
const iterator = names[Symbol.iterator]();
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

У итератора в свою очередь есть метод `next` который листает элементы:

```
{ value: 'Иван', done: false }
{ value: 'Николай', done: false }
{ value: 'Татьяна', done: false }
{ value: undefined, done: true }
```

ИТЕРАТОРЫ ВСТРОЕНЫ

Итераторы есть не только у массива, но и у других стандартных объектов в которых есть что перечислять. Например у строк:

```
const city = 'Уфа';
const iterator = city[Symbol.iterator]();
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

У строк перечисляются символы:

```
{ value: 'У', done: false }
{ value: 'ф', done: false }
{ value: 'а', done: false }
{ value: undefined, done: true }
```

for...of ИСПОЛЬЗУЕТ ИТЕРАТОРЫ

```
const names = ['Иван', 'Николай', 'Татьяна'];
for (let name of names) {
  console.log(name);
}
// Иван
// Николай
// Татьяна
```

```
const city = 'Уфа';
for (let letter of city) {
  console.log(letter);
}
// У
// ф
// а
```

SPREAD ОПЕРАТОР ИСПОЛЬЗУЕТ ИТЕРАТОР

```
const city = 'Саратов';
const letters = [...city];
console.log(letters);
// [ 'С', 'а', 'р', 'а', 'т', 'о', 'в' ]
```

ВСТРАИВАЕМЫЕ ИТЕРАТОРЫ

Мы можем реализовать итератор для наших классов/объектов:

```
class Shuffler {
  constructor(list) {
    this.list = list;
  }
  [Symbol.iterator]() {
    const list = this.list.slice();
    return {
      next() {
        const i = Math.floor(Math.random() * list.length);
        return {
          done: list.length === 0,
          value: list.splice(i, 1).shift()
        };
      }
    };
  }
}
```

ИСПОЛЬЗУЕМ НАШ ИТЕРИРУЕМЫЙ ОБЪЕКТ

```
const names = ['Иван', 'Олег', 'Петр'];
const shuffled = new Shuffler(names);

console.log(...shuffled);
// Петр Олег Иван
console.log(...shuffled);
// Олег Петр Иван

for (let name of shuffled) {
  console.log(name);
}
// Иван
// Петр
// Олег
```

ЧТО ТАКОЕ ГЕНЕРАТОР?

РАЗБЕРЕМ РЕАЛИЗАЦИЮ ИТЕРАТОРА

Тут я реализовал его просто как отдельную функцию:

```
function getIterator(list) {
  list = list.slice();
  return {
    next() {
      const i = Math.floor(Math.random() * list.length);
      return {
        done: list.length === 0,
        value: list.splice(i, 1).shift()

      };
    }
  };
}
```

ИСПОЛЬЗУЕМ НАШ «ИТЕРАТОР»

```
const iterator = getIterator(['Иван', 'Олег', 'Петр']);

console.log(iterator.next());
// { done: false, value: 'Петр' }
console.log(iterator.next());
// { done: false, value: 'Олег' }
console.log(iterator.next());
// { done: false, value: 'Иван' }
console.log(iterator.next());
// { done: true, value: undefined }
```

ПРИНЦИП РАБОТЫ

Всё самое важное происходит в методе `next`, остальное «ритуал».
Алгоритм можно описать так:

1. Берем случайное число от `0` до максимальноего индекса `ist.length - 1.`
2. Извлекаем из массива элемент с таким индексом.
3. «Возвращаем» его.
4. И так пока в массиве есть элементы.

«Ритуальные» действия:

— Выносим список `list` за пределы метода, чтобы иметь к нему доступ и после создания. Функция не хранит состояние между вызовами.

— Оборачиваем все в объект, чтобы на другом конце была доступна вся информация о состоянии.

РЕШЕНИЕ МАКСИМАЛЬНО БЛИЗКОЕ К АЛГОРИТМУ

Такой код хорошо ассоциируется с решаемой задачей, в нем нет ничего лишнего и никаких «ритуалов»:

```
// Осторожно, псевдокод!
function getIterator(list) {
  list = list.slice();
  while (list.length) {
    const i = Math.floor(Math.random() * list.length);
    return list.splice(i, 1).shift();
  }
  return;
}
```

Одна только проблема: функции не могут возвращать значения несколько раз.

А ГЕНЕРАТОР МОЖЕТ!

Генератор — новый вид функции, которая может приостановить своё выполнение, вернув промежуточный результат, и потом возобновить работу. Переделаем нашу функцию в генератор:

```
function* getIterator(list) {
  list = list.slice();
  while (list.length) {
    const i = Math.floor(Math.random() * list.length);
    yield list.splice(i, 1).shift();
  }
  return;
}
```

Что изменилось? Мы добавили символ `*` после слова `function` и `return` в теле цикла заменили на `yeild`.

IT`S ALIVE!

Всё работает. Нам ничего не пришлось менять в той части где мы вызывали `getIterator`:

```
const iterator = getIterator(['Иван', 'Олег', 'Петр']);

console.log(iterator.next());
// { done: false, value: 'Олег' }
console.log(iterator.next());
// { done: false, value: 'Петр' }

console.log(iterator.next());
// { done: false, value: 'Иван' }
console.log(iterator.next());
// { done: true, value: undefined }
```

Выходит что генератор возвращает итератор.

СОЗДАНИЕ ГЕНЕРАТОРА

СИНТАКСИС

Генератор записывается как функция, только со звездочкой:

```
function* create() {
  // тело генератора
}

const iterator = create();
console.log(iterator.next());
// { value: undefined, done: true }
```

КОМАНДА yield

Позволяет приостановить выполнение функции и вернуть промежуточный результат:

```
function* create() {
  yield 42;
}

const iterator = create();
console.log(iterator.next());
// { value: 42, done: false }
console.log(iterator.next());
// { value: undefined, done: true }
```

МОЖНО СТАВИТЬ НА ПАУЗУ НЕСКОЛЬКО РАЗ

Тело функции выполняется от `yield` до `yield`:

```
1  function* create() {
2    yield 1;
3    yield 2;
4  }
5
6  const iterator = create();
7  console.log(iterator.next());
8  // { value: 1, done: false }
9  console.log(iterator.next());
10 // { value: 2, done: false }
11 console.log(iterator.next());
12 // { value: undefined, done: true }
```

ЗАГЛЯНИМ ЗА ШИРМУ

Добавим вывод в консоль в тело функции:

```
function* create() {
  console.log('=> До первого yield');
  yield 1;
  console.log('=> От первого  до второго yield');
  yield 2;
  console.log('=> После второго yield');
}

const iterator = create();

iterator.next(); // => До первого yield
iterator.next(); // => От первого  до второго yield
iterator.next(); // => После второго yield
```

ГЕНЕРАТОР С НЕСКОЛЬКИМИ `yield`

1. При вызове функции генератора (строка 9) тело функции вообще не выполняется.
2. При первом вызове метода `.next()` итератора (строка 11) выполняется код от начала тела функции до первого `yield` (строки 2-3).
3. Функция встаёт на паузу. Итератор возвращает текущее значение которое вернул первый `yield`.
4. При втором вызове метода `.next()` итератора (строка 12) выполняется код от первого `yield` до второго (строки 3-5).
5. Функция опять встаёт на паузу. Итератор опять возвращает текущее значение переданное второму `yield`.
6. При третьем вызове метода `.next()` итератора (строка 13) выполняется код от второго `yield` до конца тела функции (строки 5-6).