Массивы и многое другое

Самый интересный тип в JS

Кирилл Талецкий

TeachMeSkills
7 августа 2023

Что будет

- Методы примитивов (и как такое вообще возможно)
- Массивы и их методы
- Коллекции Map и Set
- Деструктурирующее присваивание, rest оператор
- Поверхностное копирование коллекций, spread оператор
- rest и spread операторы в функциях

Примитивы как объекты

Примитивы как объекты

- Примитивы — не объекты, но нам хотелось бы в них хранить функции полезные для работы с ними
- При этом, примитивы должны быть максимально лёгкими для быстродействия
- Поэтому JS делает следующее
  - Примитивы во всех операциях остаются примитивами
  - При этом, они позволяют обращаться к методам, которые в них хранятся
    - Чтобы это работало, в момент обращения к методу примитива для него создаётся специальный объект-обёртка, который затем подчищается из памяти

Объекты-обёртки примитивов

- Объекты-обёртки для примитивов создаются с помощью соответствующих функций-конструкторов

```
String()
Number()
Boolean()
Symbol()
BigInt()
```

- По историческим причинам, конструкторы примитивов можно вызывать через `new` руками

```
const num = new Number(0);
```

- Но лучше этого не делать

```
typeof num; // object (!)
```

Пример

```
const str = 'Привет';

str.toUpperCase(); // ПРИВЕТ
```

Пример

```
const str = 'Привет';

str.toUpperCase(); // ПРИВЕТ
```

```
// (неявно)
let tempStr = new String(str); // создаётся объект-обёртка
tempStr.toUpperCase(); // вызывается метод объекта
```

Методы примитивов

Методы строк

- Длина строки

```
const str = 'Привет';
console.log(str.length); // 6
```

- Доступ к символам

```
const str = 'Привет';

// первый символ
console.log(str[0]); // П
console.log(str.at(0)); // П

// последний символ
console.log(str[str.length - 1]); // т
console.log(str.at(-1)); // т
```

Методы строк

- Поиск подстроки — indexOf

```
const str = 'человек человеку кек'

console.log(str.indexOf('человек')); // 0
console.log(str.indexOf('Человек')); // -1


console.log(str.indexOf('человек', 1)); // 8
/**
 *   Начинаем поиск отсюда
 *   v
 *  012345678
 * 'человек человеку кек'
 *         ^
 */
```

Методы строк

- Поиск подстроки — includes, startsWith, endsWith

```
const str = 'человек человеку кек'

console.log(str.includes('кек')); // true
console.log(str.includes('лол')); // false

console.log(str.includes('чел', 11)); // false


console.log(str.startsWith('чел')); // true
console.log(str.endsWith('ек')); // true
```

Строки неизменяемы

- Это важно - мы не можем добавлять/изменять/удалять символы в строках
- Если мы хотим этого добиться, мы должны создать новую строку, и записывать в неё нужные нам символы

- Следствие - все методы умеющие изменять строки всегда возвращают новую строку

Методы строк

- Изменение регистра — toUpperCase, toLowerCase

```
"лол".toUpperCase(); // ЛОЛ
"пРиВеТиКи".toLowerCase(); // приветики
```

Методы строк

- Получение подстроки — slice, substring

```
const str = 'человек человеку кек';

console.log( str.slice(0, 3) ); // чел
console.log( str.slice(0, 1) ); // ч

console.log( str.slice(8) ); // человеку кек

console.log( str.slice(-6, -1) ); // ку ке
```

```
const str = 'человек человеку кек';

console.log(str.substring(4, 7)); // век
console.log(str.substring(7, 4)); // век

console.log(str.slice(4, 7)); // век
console.log(str.slice(7, 4)); // "" (пустая строка)
```

Массивы

Массивы

- *Упорядоченная* коллекция данных
- В JS массив является объектом с набором специальных свойств и методов

Объявление

```
const arr = new Array(); // конструктор
const arr = []; // литерал
```

Работа с массивом

```
const todo = ["покушать", "кодить", "поспать"];

console.log(todo[0]); // покушать
console.log(todo[1]); // кодить
console.log(todo[2]); // поспать

// замена элемента
todo[0] = "кодить";
console.log(todo); // кодить, кодить, поспать

// добавление ещё одного элемента
todo[3] = "ещё поспать"; // (!) так можно, но лучше так не делать
console.log(todo); // кодить, кодить, поспать, ещё поспать

// число элементов в массиве
console.log(todo.length); // 4

// последний элемент массива
console.log(todo[todo.length - 1]); // ещё поспать
console.log(todo.at(-1)); // то же самое, только надо подождать ещё пару лет
```

Устройство массива

- Массив — это объект, в котором есть
  - Свойство length
  - Специальные методы для работы с упорядоченной коллекцией
- С массивом можно поступать как и с любым другим объектом

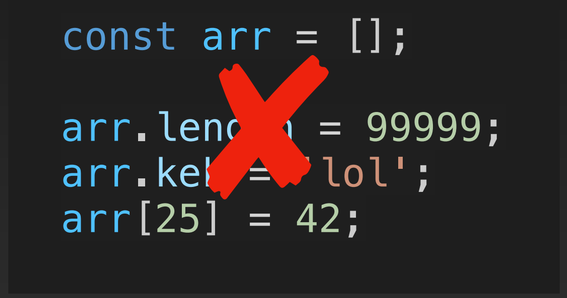
```
const arr = [];

arr.length = 99999;
arr.kek = 'lol';
arr[25] = 42;
```

Устройство массива

- JS понимает, когда перед ним массив, а когда обычный объект
- Движок JS оптимизирует работу с массивами
  - Например, хранит все элементы в непрерывной области памяти

Устройство массива

- **Как только мы начинаем работать с массивом как с объектом, JS больше не применяет к нему оптимизации производительности**



Методы массивов

Добавление/удаление элементов в массива

```
const pets = ['🐼', '🐈', '🐈'];

pets.push('🐸');    // ['🐼', '🐈', '🐈', '🐸']
pets.pop();         // ['🐼', '🐈', '🐈']

pets.unshift('🐸'); // ['🐸', '🐼', '🐈', '🐈']
pets.shift();       // ['🐼', '🐈', '🐈']
```

Добавляет элемент в конец массива

Удаляет последний элемент, возвращает его

Добавляет элемент в начало массива

Удаляет первый элемент, возвращает его

Производительность

```
const pets = ['🐼', '🐈', '🐈'];

pets.push('🐸');     // ['🐼', '🐈', '🐈', '🐸']
pets.pop();          // ['🐼', '🐈', '🐈']

pets.unshift('🐸'); // ['🐸', '🐼', '🐈', '🐈']
pets.shift();        // ['🐼', '🐈', '🐈']
```

Работают быстро — сложность O(1) для одного элемента

Работают медленно — сложность O(n) для одного элемента

(n - число элементов в исходном массиве)

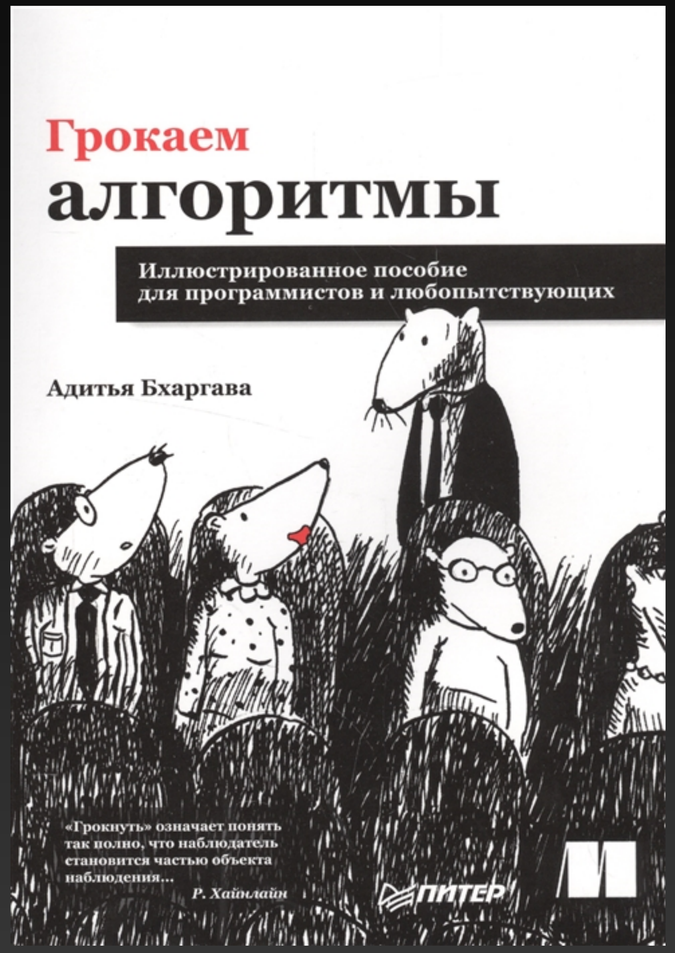
Почему так

- В массиве, чтобы добавить элемент в начало, нужно сначала подвинуть все существующие элементы на один индекс вправо

```
const pets = ['🐼', '🦮', '🐕'];
pets.unshift('🐇');

// (*)  ['🐼', '🦮', '🐕', _пусто_]

// (1)  ['🐼', '🦮', _пусто_, '🐕']
// (2)  ['🐼', _пусто_, '🦮', '🐕']
// (3)  [_пусто_, '🐼', '🦮', '🐕']

// (**) ['🐇', '🐼', '🦮', '🐕']
```

- Для массива из n элементов, нужно сделать n операций — сложность O(n)

Больше информации — в книге



Методы массивов

- Как удалить элемент из массива по индексу?
- Пробуем `delete`

```
const pets = ['🐸', '🐼', '🦙'];

delete pets[1];
console.log(pets[1]);     // undefined – похоже, что сработало

console.log(pets.length); // 3 – нет, не сработало
console.log(pets);        // ['🐸', empty, '🦙']  – точно не сработало
```

Удаление и вставка по индексу

- Удаление и вставка элементов в массив по индексу — splice

```
// Синтаксис
arr.splice(начало[, сколькоУдалить, элемент1, ..., элементN])


const pets = ['🐼', '🐕', '🐕', '🐇'];

// начиная с индекса 1, удалить 1 элемент
pets.splice(1, 1);
console.log(pets);                                  // ['🐼', '🐕', '🐇']

// удалить 2 первых элемента и заменить их другими тремя
const deleted = pets.splice(0, 2, '🐸', '🦩', '🦁');
console.log(pets);                                  // ['🐸', '🦩', '🦁', '🐇'];

// splice возвращает массив из удалённых элементов
console.log(deleted);                               // ['🐼', '🐕']
```

splice изменяет исходный массив!

Все последующие методы возвращают новый массив

Все* последующие методы всегда возвращают новый массив

Возвращаем часть массива

- slice

```
// Синтаксис
arr.slice(начиная_с_индекса, по_индекс__не_включительно);


const arr = ["Я", "очень", "люблю", "JavaScript"];
//            0      1        2           3

const arr1 = arr.slice(1, 3); // очень, люблю
const arr2 = arr.slice(-2); // люблю, JavaScript
```

Соединяем массивы

- concat

```
// Синтаксис
arr.concat(элемент_или_массив, элемент_или_массив, ...);


const arr = ['🐼', '🐈'];

const newArr = arr.concat(['🐸', '🦩'], '🐕', ['🦁', '🐇']);

console.log(newArr); // ['🐼', '🐈', '🐸', '🦩', '🐕', '🦁', '🐇']
```

Поиск

- Поиск и возврат индекса — indexOf
- Проверка наличия элемента — includes
- Оба метода проверяют на строгое равенство `===`

```
const pets = ['🐼', '🦙', '🐸', '🦩', '🐕', '🦁', '🐇'];

pets.indexOf('🦙'); // 1

pets.includes('🦙'); // true
```

Поиск по условию

- find / findIndex

```
// Синтаксис
const result = arr.find(function(item, index, array) {
  // если true – возвращается текущий элемент и перебор прерывается
  // если все итерации оказались ложными, возвращается undefined
});


const profiles = [
  { id: 0, name: 'Pete' },
  { id: 1, name: 'John' },
  { id: 2, name: 'Jane' },
];

const profile = profiles.find(profile => profile.id === 2);

console.log(profile); // {id: 2, name: 'Jane'}
```

Перебор

- forEach

```
const pets = ['🐼', '🐈', '🐸', '🦩', '🐕', '🦁', '🐇'];

pets.forEach((pet, idx, pets) => {
  console.log(`Питомец с индексом ${idx} - это ${pet}, в списке ${pets}`)
});
```

Преобразование массива
Сортировка

- sort
- изменяет исходный массив!

```
// Синтаксис
arr.sort(function (a, b) {
  // если вернуть число > 0, то a > b
  // если вернуть число < 0, то a < b
  // если вернуть 0, то a == b

  // если a и b числа, то работает `return a - b`;
})

const profiles = [
  { id: 0, name: 'Pete', likes: 42 },
  { id: 1, name: 'John', likes: 15 },
  { id: 2, name: 'Jane', likes: 101 },
  { id: 3, name: 'Eva', likes: 95 },
];

profiles.sort((p1, p2) => p2.likes - p1.likes);
```

Преобразование массива
Фильтр

- filter

```
// Синтаксис
arr.filter(function (элемент) {
  // если вернуть `true`, то элемент останется в массиве
  // если вернуть `false`, то элемент будет удалён
});


const profiles = [
  { id: 0, name: 'Pete', likes: 42 },
  { id: 1, name: 'John', likes: 15 },
  { id: 2, name: 'Jane', likes: 101 },
  { id: 3, name: 'Eva', likes: 95 },
];

profiles.filter(profile => profile.likes > 50);
```

Преобразование массива
Отображение

- map

```
// Синтаксис
arr.map(function (элемент) {
  // то, что мы вернём, будет записано вместо элемента
});

const profiles = [
  { id: 0, name: 'Pete', likes: 42 },
  { id: 1, name: 'John', likes: 15 },
  { id: 2, name: 'Jane', likes: 101 },
  { id: 3, name: 'Eva', likes: 95 },
];

profiles.map(profile => profile.likes);
// 42, 15, 101, 95
```

Преобразование массива
Приведение к единичному значению

- reduce

```
// Синтаксис
arr.reduce(
  function (аккумулятор, элемент, индекс, массив) {
    // то, что мы вернём, будет записано в аккумулятор в следующей итерации
    // return some_result -- будет передано в аккумулятор
  },
  начальное_значение_аккумулятора
);

const profiles = [
  { id: 0, name: 'Pete', likes: 42 },
  { id: 1, name: 'John', likes: 15 },
  { id: 2, name: 'Jane', likes: 101 },
  { id: 3, name: 'Eva', likes: 95 },
];

profiles.reduce((acc, profile) => acc + profile.likes, 0);
// 253 (= 42 + 15 + 101 + 95)
```

Преобразование массива
Разворот

- reverse
- изменяет исходный массив!

```
const arr = [1, 2, 3, 4, 5];
arr.reverse();

console.log(arr); // [5, 4, 3, 2, 1];
```

Строка <-> массив

- str.split(sep) — разделение строки в массив
- arr.join(sep) — слияние элементов массива в строку

```
const str = "🐼, 🐈, 🐸, 🦩, 🐕, 🦁, 🐇";

const arr = str.split(', ');
console.log(arr); // ['🐼', '🐈', '🐸', '🦩', '🐕', '🦁', '🐇']

const newStr = arr.join('; ');
console.log(newStr); // 🐼; 🐈; 🐸; 🦩; 🐕; 🦁; 🐇
```

Объект <-> массив

- Object.keys(obj) – возвращает **массив** ключей
- Object.values(obj) – возвращает **массив** значений
- Object.entries(obj) – возвращает **массив** пар [ключ, значение]

- Object.fromEntries(iterable) — возвращает **объект** из массива пар [ключ, значение]

```
const prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};

const doubledPrices = Object.fromEntries(
  // преобразовать в массив
  // затем map
  // затем fromEntries обратно объект
  Object.entries(prices).map(
    ([key, value]) => [key, value * 2]
  )
);

console.log(doubledPrices.meat); // 8
```

Проверка на условие

- arr.some() — true, если хотя бы один элемент подходит по условию
- arr.every() — true, если все элементы подходят по условию

```
const profiles = [
  { id: 0, name: 'Pete', blocked: false },
  { id: 1, name: 'John', blocked: true },
  { id: 2, name: 'Jane', blocked: false },
  { id: 3, name: 'Eva', blocked: false },
];

const isBlocked = (profile) => profile.blocked;

const hasBlocked = profiles.some(isBlocked); // true
const allBlocked = profiles.every(isBlocked); // false
```

Перебор массивов

```
for (idx in arr) { }
for (element of arr) { }
for (let idx = 0; i < arr.length; i++) { }
```

Перебор массивов

Deprecated

```
for (idx in arr) { }
for (element of arr) { }
for (let idx = 0; i < arr.length; i++) { }
```

Good

```
arr.forEach();
arr.reduce();
arr.map();
arr.filter();
arr.sort();
arr.some();
arr.every();
```

Коллекции Map и Set

Map

- Как и Object — коллекция ключ-значение
- Позволяет использовать ключи любого типа, не только строки

Map

- new Map() – создаёт коллекцию.
- map.set(key, value) – записывает по ключу key значение value
- map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует
- map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false
- map.delete(key) – удаляет элемент (пару «ключ/значение») по ключу key
- map.clear() – очищает коллекцию от всех элементов
- map.size – возвращает текущее количество элементов.

```
const map = new Map();

map.set("1", "str1");    // строка в качестве ключа
map.set(1, "num1");      // цифра как ключ
map.set(true, "bool1");  // булево значение как ключ


// Map сохраняет тип ключей, так что в этом случае
// сохранится 2 разных значения:
alert(map.get(1));   // "num1"
alert(map.get("1")); // "str1"

alert(map.size);     // 3
```

Set

- *Множество* значений, без ключей
- Каждый элемент может появляться только один раз

- Подходит для переборов с проверкой на уникальность

Set

- new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set
- set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set
- set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false
- set.has(value) – возвращает true, если значение присутствует в множестве, иначе false
- set.clear() – удаляет все имеющиеся значения
- set.size – возвращает количество элементов в множестве

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set хранит только 3 уникальных значения
alert(set.size); // 3

for (let user of set) {
  alert(user.name); // John (потом Pete и Mary)
}
```

Деструктуризация

Деструктурирующее присваивание

- Специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных

```
const frame = ["100px", "250px"];

/**
 *    ["100px", "250px"]
 *        v         v        */
const [ width,  height ] = frame;

console.log(width); // 100px
console.log(height); // 250px
```

Деструктурирующее присваивание

- Это просто более короткая запись

```
const frame = ["100px", "250px"];

const [width, height] = frame;
```

```
const frame = ["100px", "250px"];

const width = frame[0];
const height = frame[1];
```

Деструктурирующее присваивание

- Можно пропускать элементы при помощи запятых

```
const list = ['Иванов', 'Петров', 'Вован', 'Сидоров']

const [first, second, , fourth] = list;
/**                     ^
 *                    (!)
 */

console.log(fourth); // Сидоров
```

Деструктурирующее присваивание

- Можно взять только несколько элементов, а остальные проигнорировать

```
const list = ['Иванов', 'Петров', 'Вован', 'Сидоров']

const [first, second] = list;


console.log(second); // Петров
```

Остаточные параметры

rest оператор — `...`

- Можно взять только несколько первых элементов
- Если мы хотим забрать остальные значения, то используем оператор `...`

```
const list = ['Иванов', 'Петров', 'Вован', 'Сидоров']

const [first, second, ...rest] = list;

console.log(rest); // ['Вован', 'Сидоров']
```

Значения по умолчанию

```
const [width, height] = [];

console.log(width); // undefined
console.log(height); // undefined
```

```
const [width = '0px', height = '0px'] = [];

console.log(width); // 0px
console.log(height); // 0px
```

Деструктуризация объекта

Деструктуризация объекта

- Почти такой же синтаксис, как и в массивах, но
- Название создаваемой переменной обязательно совпадает с ключом

```
const profile = {
  name: 'John',
  age: 42,
  isAdmin: false,
}

const { name, age, isAdmin } = profile;

console.log(name); // John
```

Деструктуризация объекта

- Переименовать переменную можно через `:`
- Синтаксис `что : куда идёт`

```
const profile = {
  name: 'John',
}

const { name: userName } = profile;

// name -> userName

console.log(userName); // John
```

Деструктуризация объекта

- Есть возможность задавать значения по умолчанию

```
const profile = {}

// значение по умолчанию
const { name = 'Unknown' } = profile;
console.log(name);

// значение по умолчанию и переименование
const { name: userName = 'Unknown' } = profile;
console.log(userName);
```

Деструктуризация объекта

- Если значений много, то можно взять только часть из них, а остальные проигнорировать

```
const profile = {
  name: 'John',
  age: 42,
  isAdmin: false,
}

const { name } = profile;

console.log(name); // John
```

Деструктуризация объекта

- Оставшиеся значения можно присвоить в отдельную переменную

```
const profile = {
  name: 'John',
  age: 42,
  isAdmin: false,
}

const { name, ...rest } = profile;

console.log(rest); // { age: 42, isAdmin: false }
```

Аргументы функций

- Если в вашей функции больше двух параметров, то имеет смысл вместо отдельных параметров передать в неё объект
  - При хороших именах ключей, пользователь быстрее поймёт, что нужно передать в функцию
  - Так будет проще пропустить необязательные аргументы
  - Упрощается рефакторинг кода

```
const profile = {
  name: 'John',
  age: 42,
  avatar: '👨',
  isAdmin: true,
}

function renderProfile({
  name = 'Unknown',
  age = '–',
  avatar = '👤',
  isAdmin = false
}) {
  console.log(`
    ${avatar}

    Name: ${name}
    Age: ${age}

    ${isAdmin ? '⭐ Super User!' : ''}
  `)
}

renderProfile(profile);
```

Копирование коллекций через spread

Массивы

- Оператор расширения (spread — `...`) позволяет копировать значения/ссылки в литералы массивов

```
const more = ['four', 'five'];

const all = ['one', 'two', 'three', ...more]; // ['one', 'two', 'three', 'four', 'five']
const all = ['one', ...more, 'two', 'three']; // ['one', 'four', 'five', 'two', 'three']
```

- Конкатенация массивов с использованием spread синтаксиса

```
const arr = ['one', 'two', 'three'];
const more = ['four', 'five'];

const all = [...arr, ...more]; // ['one', 'two', 'three', 'four', ‘five']
```

Массивы

- Объекты копируются по ссылке

```
const profile1 = { id: 1 };
const profile2 = { id: 2 };
const profile3 = { id: 3 };
const profile4 = { id: 4 };
const profile5 = { id: 5 };

const admins = [profile1, profile2, profile3];
const moderators = [profile4, profile5];

const adminsAndModerators = [...admins, ...moderators];

console.log(adminsAndModerators);
// [{id: 1}, {id: 2}, {id: 3}, {id: 4}, {id: 5}]

profile2.id = 42;

console.log(adminsAndModerators);
// [{id: 1}, {id: 42}, {id: 3}, {id: 4}, {id: 5}]
```

Объекты

- Оператор расширения (spread — `...`) позволяет копировать пары ключ-значение в литералы объектов

```
const privileges = {
  isAdmin: true,
  isModerator: false,
};

const profile = {
  name: 'John',
  age: 42,
  ...privileges
};

console.log(profile);
// {name: "John", age: 42, isAdmin: true, isModerator: false}
```

Объекты

- При копировании, значения в совпавших ключах переписываются
- Приоритет отдаётся самому последнему значению

```
const privileges = {
  isAdmin: true,
};

const profile = {
  name: 'John',
  age: 42,
  isAdmin: false,
  ...privileges
};

console.log(profile.isAdmin); // true
```

```
const privileges = {
  isAdmin: true,
};

const profile = {
  ...privileges,
  name: 'John',
  age: 42,
  isAdmin: false,
};

console.log(profile.isAdmin); // false
```

Объекты

- Spread копирование удобно, например, для заполнения полей значениями по умолчанию

```
const DEFAULT_PROFILE = {
  name: 'Unknown',
  age: '-',
  isAdmin: false
}

const profileData = {
  name: 'Иван',
}

const profile = {
  ...DEFAULT_PROFILE,
  ...profileData,
}
```

Объекты

- Spread оператор копирует значения-объекты по ссылке
- Так называемое поверхностное копирование (shallow copy)

```
const DEFAULT_PRIVILEGES = {
  isAdmin: false,
};

const DEFAULT_PROFILE = {
  name: 'Unknown',
  age: '-',
  privileges: DEFAULT_PRIVILEGES,
};

const profile = {
  ...DEFAULT_PROFILE,
  name: 'Иван',
};

console.log(profile.privileges.isAdmin); // false

DEFAULT_PRIVILEGES.isAdmin = true; // случайно поменяли значение

console.log(profile.privileges.isAdmin);
// true - поменяются все профили, использующие дефолтный объект,
// потому что он был скопирован по ссылке
```

Rest/spread в функциях

Остаточные параметры

rest оператор — `...`

- Можно взять только несколько первых параметров
- Если мы хотим забрать остальные значения, то используем оператор `...`

```
function sum(first, second, ...rest) {
  console.log(first);
  console.log(second);
  console.log(rest);
};

sum(1, 2, 3, 4, 5, 6, 7);
/**
 * 1
 * 2
 * [3, 4, 5, 6, 7]
 */
```

Остаточные параметры

rest оператор — `...`

- Оператор `...` позволяет создавать функции, которые могут принимать произвольное число параметров

```
function sum(...args) {
  return args.reduce((sum, arg) => sum + arg, 0)
}

console.log(sum(2, 2));                  // 4
console.log(sum(2, 5, 5));               // 12
console.log(sum(2, 5, 5, 5, 5, 5, 5));  // 32
```

Проблема

- Мы рассмотрели, как получить массив из аргументов
- А как быть, когда нам нужно обратное?

```
function sum(...args) {
  return args.reduce((sum, arg) => sum + arg, 0)
}

console.log(sum(2, 2));    // 4
console.log(sum(2, 5, 5)); // 12


const args = [2, 5, 5];
// Как передать `args` в sum() ?
```

Решение

```
sum(...[2, 5, 5]);
// получаем:
// sum(2, 5, 5);
```

Распаковка массива в аргументы

spread оператор `...`

```
function sum(...args) {
  return args.reduce((sum, arg) => sum + arg, 0)
}

console.log(sum(2, 2));    // 4
console.log(sum(2, 5, 5)); // 12


const args = [2, 5, 5];
console.log(sum(...args));
```