Практическая работа №11 «JavaScript. Массивы. Методы массивов. Деструктуризация. Рекурсия и стек. Глобальный объект»

Массивы

Объекты позволяют хранить данные со строковыми ключами. Но довольно часто мы понимаем, что нам необходима *упорядоченная коллекция* данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

Объявление

Существует два варианта синтаксиса для создания пустого массива:

```
let arr = new Array();
let arr = [];

```

Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];
```

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива

```

Мы можем заменить элемент или добавить новый к существующему массиву:

```
fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]
fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]

```

В массиве могут храниться элементы любого типа. Например:

```
// разные типы значений
let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

// получить элемент с индексом 1 (объект) и затем показать его свойство
alert( arr[1].name ); // Джон

// получить элемент с индексом 3 (функция) и выполнить её
arr[3](); // привет

```

Методы pop/push, shift/unshift

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

- push добавляет элемент в конец.
- pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца». Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху:

- Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.
- В компьютерных науках структура данных, делающая это возможным, называется двусторонняя очередь.

Методы, работающие с концом массива:

pop

Удаляет последний элемент из массива и возвращает его:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин

```

И fruits.pop() и fruits.at(-1) возвращают последний элемент массива, но fruits.pop() также изменяет массив, удаляя его.

push

Добавляет элемент в конец массива:

```
let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша

```

Вызов fruits.push(...) равнозначен fruits[fruits.length] = ...

Методы, работающие с началом массива:

shift

Удаляет из массива первый элемент и возвращает его:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() ); // удаляем Яблоко и выводим его

alert( fruits ); // Апельсин, Груша

```

unshift

Добавляет элемент в начало массива:

```
let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша

```

Методы push и unshift могут добавлять сразу несколько элементов:

```
let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");
fruits.unshift("Ананас", "Лимон");

// ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]
alert( fruits );

```

Внутреннее устройство массива

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это по сути обычный синтаксис доступа по ключу, как obj[key], где в роли obj у нас arr, а в качестве ключа – числовой индекс.

Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length. Но в основе всё равно лежит объект.

Следует помнить, что в JavaScript существует 8 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект.

Варианты неправильного применения массива:

- Добавление нечислового свойства, например: arr.test = 5.
- Создание «дыр», например: добавление arr[0], затем arr[1000] (между ними ничего нет).
- Заполнение массива в обратном порядке, например: arr[1000], arr[999] и т.д.

Массив следует считать особой структурой, позволяющей работать с *упорядоченными данными*. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, пожалуйста, используйте

их именно в таких случаях. Если вам нужны произвольные ключи, вполне возможно, лучше подойдёт обычный объект {}.

Эффективность

Методы push/pop выполняются быстро, а методы shift/unshift – медленно.

Например, операция shift должна выполнить 3 действия:

1. Удалить элемент с индексом 0.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив 1 на 0, 2 на 1 и т.д.
3. Обновить свойство length .

Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с unshift: чтобы добавить элемент в начало массива, нам нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

push/pop не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length.

Метод pop не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро.

Аналогично работает метод push.

Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам, но для массивов возможен и другой вариант цикла, for..of:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям
for (let fruit of fruits) {
  alert( fruit );
}
```

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно.

Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, центральный элемент
```

Вызов new Array(number) создаёт массив с заданной длиной, но без элементов.

- Свойство length отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.
- Если мы уменьшаем length вручную, массив укорачивается.

Получение элементов:

- Мы можем получить элемент по его индексу, например arr[0].

- Также мы можем использовать метод at(i) для получения элементов с отрицательным индексом, для отрицательных значений i, он отступает от конца массива. В остальном он работает так же, как arr[i], если i >= 0.

Мы можем использовать массив как двустороннюю очередь, используя следующие операции:

- push(...items)добавляет items в конец массива.
- pop() удаляет элемент в конце массива и возвращает его.
- shift() удаляет элемент в начале массива и возвращает его.
- unshift(...items) добавляет items в начало массива.

Чтобы пройтись по элементам массива:

- for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.
- for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).
- for (let i in arr) – никогда не используйте для массивов!

Методы массивов

Массивы предоставляют множество методов.

Добавление/удаление элементов

Методы, которые добавляют и удаляют элементы из начала или конца:

- arr.push(...items) – добавляет элементы в конец,
- arr.pop() – извлекает элемент из конца,
- arr.shift() – извлекает элемент из начала,
- arr.unshift(...items) – добавляет элементы в начало.

Удаление элемента из массива?

Метод arr.splice(str) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

Пример:

```
let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 3 первых элемента и заменить их другими
arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]
```

Здесь видно, что splice возвращает массив из удалённых элементов:

```
let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 2 первых элемента
let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" <-- массив из удалённых элементов
```

Отрицательные индексы разрешены

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца, как тут:

```
let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом)
// удалить 0 элементов,
// затем вставить числа 3 и 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5

```

slice

Метод arr.slice намного проще, чем похожий на него arr.splice.

Метод arr.slice используется для создания нового массива, содержащего выбранную часть исходного массива.

Его синтаксис:

```
arr.slice([start], [end])
```

Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива. Это похоже на строковый метод str.slice, но вместо подстрок возвращает подмассивы.

Например:

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (копирует с 1 до 3)

alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)

```

Можно вызвать slice и вообще без аргументов: arr.slice() создаёт копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

concat

Метод arr.concat создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

```
arr.concat(arg1, arg2...)
```

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями. В результате мы получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и т.д.

Если аргумент argN – массив, то все его элементы копируются. Иначе скопируется сам аргумент.

Например:

```
let arr = [1, 2];

// создать массив из: arr и [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// создать массив из: arr и [3,4] и [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// создать массив из: arr и [3,4], потом добавить значения 5 и 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6

```

Обычно он копирует только элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть.

Перебор: forEach

Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

```
arr.forEach(function(item, index, array) {
  // ... делать что-то с item
});
```

Например, этот код выведет на экран каждый элемент массива:

```
// Вызов alert для каждого элемента
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
А этот вдобавок расскажет и о своей позиции в массиве:
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} имеет позицию ${index} в ${array}`);
});
```

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

Поиск в массиве

indexOf/lastIndexOf и includes

Методы arr.indexOf, arr.lastIndexOf и arr.includes имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

- arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
- arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
- arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

```

Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes. Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (должен быть 0, но === проверка на равенство не работает для NaN)
alert( arr.includes(NaN) );// true (верно)

```

find и findIndex

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод arr.find.

Его синтаксис таков:

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

```

Функция вызывается по очереди для каждого элемента массива:

- item – очередной элемент.
- index – его индекс.
- array – сам массив.

Если функция возвращает true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined.

Например, у нас есть массив пользователей, каждый из которых имеет поля id и name. Попробуем найти того, кто с id == 1:

```
let users = [
  {id: 1, name: "Вася"},
  {id: 2, name: "Петя"},
  {id: 3, name: "Маша"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // Вася

```

Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко. Метод arr.findIndex – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

filter

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true. На тот случай, если найденных элементов может быть много, предусмотрен метод arr.filter(fn). Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:

```
let results = arr.filter(function(item, index, array) {
  // если true - элемент добавляется к результату, и перебор продолжается
  // возвращается пустой массив в случае, если ничего не найдено
});

```

Преобразование массива

map

Метод arr.map является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции. Например, здесь мы преобразуем каждый элемент в его длину:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

sort(fn)

Вызов arr.sort() сортирует массив *на месте*, меняя в нём порядок элементов (сортирует элементы массива в порядке возрастания или убывания).

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr.

По умолчанию элементы сортируются как строки.

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок. Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort(). Функция должна для пары значений возвращать:

```
function compare(a, b) {
  if (a > b) return 1; // если первое значение больше второго
  if (a == b) return 0; // если равны
  if (a < b) return -1; // если первое значение меньше второго
}
```

Например, для сортировки чисел:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15

```

reverse

Метод arr.reverse меняет порядок элементов в arr на обратный.

Например:

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

alert( arr ); // 5,4,3,2,1

```

Он также возвращает массив arr с изменённым порядком элементов.

split и join

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Метод str.split(delim) именно это и делает. Он разбивает строку на массив по заданному разделителю delim.

```
let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

for (let name of arr) {
  alert( `Сообщение получат: ${name}.` ); // Сообщение получат: Вася (и другие имена)
}
```

Разбивка по буквам

Вызов split(' ') с пустым аргументом s разбил бы строку на массив букв:

```
let str = "тест";

alert( str.split('') ); // m,e,c,m

```

Вызов arr.join(glue) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними.

Например:

```
let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша

```

Array.isArray

Массивы не образуют отдельный тип языка. Они основаны на объектах. Поэтому typeof не может отличить простой объект от массива:

```
alert(typeof {}); // object
alert(typeof []); // може object

```

Однако массивы используются настолько часто, что для этого придумали специальный метод: Array.isArray(value). Он возвращает true, если value массив, и false, если нет.

```
alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

```

Небольшая шпаргалка по методам массива:

- Для добавления/удаления элементов:
    - push (...items) – добавляет элементы в конец,
    - pop() – извлекает элемент с конца,
    - shift() – извлекает элемент с начала,
    - unshift(...items) – добавляет элементы в начало.
    - splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items.
    - slice(start, end) – создаёт новый массив, копируя в него элементы с позиции start до end (не включая end).
    - concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
- Для поиска среди элементов:
    - indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
    - includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.

    - find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
    - findIndex похож на find, но возвращает индекс вместо значения.
- Для перебора элементов:
    - forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
- Для преобразования массива:
    - map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
    - sort(func) – сортирует массив «на месте», а потом возвращает его.
    - reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
    - split/join – преобразует строку в массив и обратно.
    - reduce/reduceRight(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
- Дополнительно:
    - Array.isArray(arr) проверяет, является ли arr массивом.

Обратите внимание, что методы sort, reverse и splice изменяют исходный массив.

Полный список есть в справочнике MDN.

Деструктурирующее присваивание

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array.

- Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам.
- Массивы позволяют нам собирать элементы данных в упорядоченный список.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию и так далее. Скоро мы увидим это.

Деструктуризация массива

Вот пример деструктуризации массива на переменные:

```
// у нас есть массив с именем и фамилией
let arr = ["Ilya", "Kantor"];

// деструктурирующее присваивание
// записывает firstName = arr[0]
// и surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor

```

Теперь мы можем использовать переменные вместо элементов массива:

```
let [firstName, surname] = "Ilya Kantor".split(' ');
alert(firstName); // Ilya
alert(surname);  // Kantor

```

Как вы можете видеть, синтаксис прост. Однако есть несколько странных моментов. Давайте посмотрим больше примеров, чтобы лучше понять это.

«Деструктуризация» не означает «разрушение».

«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

Пропускайте элементы, используя запятые

Нежелательные элементы массива также могут быть отброшены с помощью дополнительной запятой:

```
// второй элемент не нужен
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul

```

В примере выше второй элемент массива пропускается, а третий присваивается переменной title, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

Работает с любым перебираемым объектом с правой стороны

Трюк обмена переменных

Существует хорошо известный трюк для обмена значений двух переменных с использованием деструктурирующего присваивания:

```
let guest = "Jane";
let admin = "Pete";

// Давайте поменяем местами значения: сделаем guest = "Pete", а admin = "Jane"
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane (успешно заменено!)

```

Здесь мы создаём временный массив из двух переменных и немедленно

деструктурируем его в порядке замены. Таким образом, мы можем поменять местами даже более двух переменных.

Деструктуризация объекта

Деструктурирующее присваивание также работает с объектами.

Синтаксис:

```
let {var1, var2} = {var1:…, var2:…}
```

У нас есть существующий объект с правой стороны, который мы хотим разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}.

Например:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200

```

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения.

Если мы хотим присвоить свойство объекта переменной с другим названием, например, свойство options.width присвоить переменной w, то мы можем использовать двоеточие:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

Двоеточие показывает «что : куда идёт». В примере выше свойство width сохраняется в переменную w, свойство height сохраняется в h, а title присваивается одноимённой переменной.

Вложенная деструктуризация

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства. В приведённом ниже коде options хранит другой объект в свойстве size и массив в свойстве items. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// деструктуризация разбита на несколько строк для ясности
let {
  size: { // положим size сюда
    width,
    height
  },
  items: [item1, item2], // добавим элементы к items
  title = "Menu" // отсутствует в объекте (используется значение по умолчанию)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

Весь объект options, кроме свойства extra, которое в левой части отсутствует, присваивается в соответствующие переменные:

- В итоге у нас есть width, height, item1, item2 и title со значением по умолчанию.
- Заметим, что переменные для size и items отсутствуют, так как мы взяли сразу их содержимое.

Итого

- Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
- Полный синтаксис для объекта:

  let {prop : varName = default, ...rest} = object

Свойство prop объекта object здесь должно быть присвоено переменной varName. Если в объекте отсутствует такое свойство, переменной varName присваивается значение по умолчанию.

Свойства, которые не были упомянуты, копируются в объект rest.

- Полный синтаксис для массива:

  let [item1 = default, item2, ...rest] = array
  Первый элемент отправляется в item1; второй отправляется в item2, все остальные элементы попадают в массив rest.

- Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

Рекурсия и стек

Рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. В процессе выполнения задачи в теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает *сама себя*. Это как раз и называется *рекурсией*.

Два способа мышления

В качестве первого примера напишем функцию pow(x, n), которая возводит x в натуральную степень n. Иначе говоря, умножает x на само себя n раз.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

Рассмотрим два способа её реализации.

1. Итеративный способ: цикл for:

```
function pow(x, n) {
  let result = 1;

  // умножаем result на x n раз в цикле
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

2. Рекурсивный способ: упрощение задачи и вызов функцией самой себя:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

Обратите внимание, что рекурсивный вариант отличается принципиально.

Когда функция pow(x, n) вызывается, исполнение делится на две ветви:

```
        if n==1  = x
       /
pow(x, n) =
       \
        else     = x * pow(x, n - 1)
```

1. Если n == 1, тогда всё просто. Эта ветвь называется *базой* рекурсии, потому что сразу же приводит к очевидному результату: pow(x, 1) равно x.

2. Мы можем представить pow(x, n) в виде: x * pow(x, n - 1). Что в математике записывается как: $x^n = x * x^{n-1}$. Эта ветвь – *шаг рекурсии*: мы сводим задачу к более простому действию (умножение на x) и более простой аналогичной задаче (pow с меньшим n). Последующие шаги упрощают задачу всё больше и больше, пока n не достигает 1.

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – к ещё более простому и так далее, пока значение не станет очевидно. Рекурсивное решение задачи обычно короче, чем итеративное.

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

Это ограничивает применение рекурсии, но она всё равно широко распространена: для решения большого числа задач рекурсивный способ решения даёт более простой код, который легче поддерживать.

Контекст выполнения, стек

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого заглянем «под капот» функций.

Информация о процессе выполнения запущенной функции хранится в её *контексте выполнения* (execution context).

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные

функции, значение this (мы не используем его в данном примере) и прочую служебную информацию. Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

Когда функция производит вложенный вызов, происходит следующее:

- Выполнение текущей функции приостанавливается.
- Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – *стеке контекстов выполнения*.
- Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
- После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Для выполнения вложенного вызова JavaScript запоминает текущий контекст выполнения в *стеке контекстов выполнения*. Для любых функций процесс одинаков:

1. Текущий контекст «запоминается» на вершине стека.
2. Создаётся новый контекст для вложенного вызова.
3. Когда выполнение вложенного вызова заканчивается – контекст предыдущего вызова восстанавливается, и выполнение соответствующей функции продолжается.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.

Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции в зависимости от условий используются различные рекурсивные подвызовы, результаты которых объединяются, или когда ветвление более сложное. Оптимизация может быть ненужной и совершенно нестоящей усилий.

Часто код с использованием рекурсии более короткий, лёгкий для понимания и поддержки. Оптимизация требуется не везде, как правило, нам важен хороший код, поэтому она и используется.

Рекурсивные обходы

Другим отличным применением рекурсии является рекурсивный обход.

Представьте, у нас есть компания. Структура персонала может быть представлена как объект:

```
let company = {
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

Другими словами, в компании есть отделы.

- Отдел может состоять из массива работников. Например, в отделе sales работают 2 сотрудника: Джон и Алиса.

- Или отдел может быть разделён на подотделы, например, отдел development состоит из подотделов: sites и internals. В каждом подотделе есть свой персонал.
- Также возможно, что при росте подотдела он делится на подразделения (или команды).

  Например, подотдел sites в будущем может быть разделён на команды siteA и siteB. И потенциально они могут быть разделены ещё. Этого нет на картинке, просто нужно иметь это в виду.

Теперь, допустим, нам нужна функция для получения суммы всех зарплат. Итеративный подход не прост, потому что структура довольно сложная. Первая идея заключается в том, чтобы сделать цикл for поверх объекта company с вложенным циклом над отделами 1-го уровня вложенности. Но затем нам нужно больше вложенных циклов для итераций над сотрудниками отделов второго уровня, таких как sites… А затем ещё один цикл по отделам 3-го уровня, которые могут появиться в будущем? Если мы поместим в код 3-4 вложенных цикла для обхода одного объекта, то это будет довольно некрасиво. Давайте попробуем рекурсию.

Как мы видим, когда наша функция получает отдел для подсчёта суммы зарплат, есть два возможных случая:

1. Либо это «простой» отдел с *массивом* – тогда мы сможем суммировать зарплаты в простом цикле.
2. Или это *объект* с N подотделами – тогда мы можем сделать N рекурсивных вызовов, чтобы получить сумму для каждого из подотделов, и объединить результаты.

Случай (1), когда мы получили массив, является базой рекурсии, тривиальным случаем.

Случай (2), при получении объекта, является шагом рекурсии. Сложная задача разделяется на подзадачи для подотделов. Они могут, в свою очередь, снова разделиться на подотделы, но рано или поздно это разделение закончится, и решение сведётся к случаю (1).

Алгоритм даже проще читается в виде кода:

```
let company = { // тот же самый объект, сжатый для краткости
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600 }],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// Функция для подсчёта суммы зарплат
function sumSalaries(department) {
  if (Array.isArray(department)) { // случай (1)
    return department.reduce((prev, current) => prev + current.salary, 0); // сумма элементов массива
  } else { // случай (2)
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // рекурсивно вызывается для подотделов, суммируя результаты
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 6700
```

Код краток и прост для понимания. В этом сила рекурсии. Она работает на любом уровне вложенности отделов.

Схема вызовов:

Принцип прост: для объекта {...} используются рекурсивные вызовы, а массивы [...] являются «листьями» дерева рекурсии, они сразу дают результат.

Обратите внимание, что в коде используются возможности, о которых мы говорили ранее:

- Метод arr.reduce из главы Методы массивов для получения суммы элементов массива.
- Цикл for(val of Object.values(obj)) для итерации по значениям объекта: Object.values возвращает массив значений.

Связанный список

Представьте себе, что мы хотим хранить упорядоченный список объектов.

Естественным выбором будет массив, но у массивов есть недостатки. Операции «удалить элемент» и «вставить элемент» являются дорогостоящими. Например, операция arr.unshift(obj) должна переиндексировать все элементы, чтобы освободить место для нового obj, и, если массив большой, на это потребуется время. То же самое с arr.shift().

Единственные структурные изменения, не требующие массовой переиндексации – это изменения, которые выполняются с конца массива: arr.push/pop. Таким образом, массив может быть довольно медленным для больших очередей, когда нам приходится работать с его началом.

Или же, если нам действительно нужны быстрые вставка/удаление, мы можем выбрать другую структуру данных, называемую связанный список.

Элемент *связанного списка* определяется рекурсивно как объект с:

- value,
- next – свойство, ссылающееся на следующий *элемент связанного списка* или null, если это последний элемент.

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

```

Здесь мы можем ещё лучше увидеть, что есть несколько объектов, каждый из которых имеет value и next, указывающий на соседа. Переменная list является первым объектом в цепочке, поэтому, следуя по указателям next из неё, мы можем попасть в любой элемент.

Список можно легко разделить на несколько частей и впоследствии объединить обратно. И, конечно, мы можем вставить или удалить элементы из любого места. Например, для добавления нового элемента нам нужно обновить первый элемент списка:

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// добавление нового элемента в список
list = { value: "new item", next: list };
```

Чтобы удалить элемент из середины списка, нужно изменить значение next предыдущего элемента:

```
list.next = list.next.next;
```

list.next перепрыгнуло с 1 на значение 2. Значение 1 теперь исключено из цепочки. Если оно не хранится где-нибудь ещё, оно будет автоматически удалено из памяти. В отличие от массивов, нет перенумерации, элементы легко переставляются. Естественно, списки не всегда лучше массивов. В противном случае все пользовались бы только списками.

Главным недостатком является то, что мы не можем легко получить доступ к элементу по его индексу. В простом массиве: arr[n] является прямой ссылкой. Но в списке мы должны начать с первого элемента и перейти в next N раз, чтобы получить N-й элемент.

Списки могут быть улучшены:

- Можно добавить свойство prev в дополнение к next для ссылки на предыдущий элемент, чтобы легко двигаться по списку назад.
- Можно также добавить переменную tail, которая будет ссылаться на последний элемент списка (и обновлять её при добавлении/удалении элементов с конца).

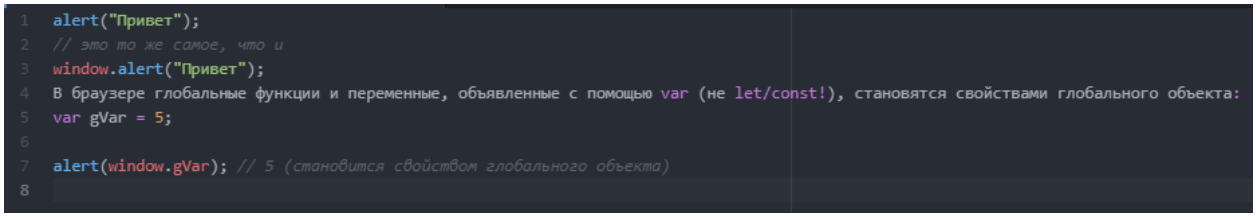
- Возможны другие изменения: главное, чтобы структура данных соответствовала нашим задачам с точки зрения производительности и удобства.

Глобальный объект

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js — global, в другой среде исполнения может называться иначе.

Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. Он поддерживается во всех основных браузерах.

Ко всем свойствам глобального объекта можно обращаться напрямую:



То же самое касается функций, объявленных с помощью синтаксиса Function Declaration (выражения с ключевым словом function в основном потоке кода, не Function Expression)

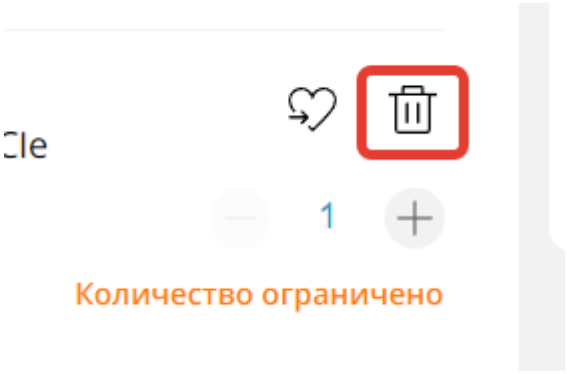
Однако, в современных проектах, использующих JavaScript-модули, такого не происходит. Если свойство настолько важное, что вы хотите сделать его доступным для всей программы, запишите его в глобальный объект напрямую:

```
// сделать информацию о текущем пользователе глобальной, для предоставления доступа всем скриптам
window.currentUser = {
  name: "John"
};

// где угодно в коде
alert(currentUser.name); // John

// или, если у нас есть локальная переменная с именем "currentUser",
// получим её из window явно (безопасно!)
alert(window.currentUser.name); // John
```

При этом обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надёжнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

Практическое задание

1. Создайте массив с товарами/элементами сайта (например, для корзины). По нажатию кнопки один из элементов массива должен меняться на другой (1.1), по нажатию на другую кнопку массив очищается (1.2).



   1.1. При нажатии на иконку «корзины» один элемент массива удаляется (его местоположение в массиве не важно), другие элементы занимают его место.

   1.2. При очистке массива он остается пустым до ввода новых элементов.

2. Добавить возможность увеличивать количество «товара» в корзине с изменением итоговой «стоимости» товаров.

3. Создайте фильтр, который принимает массив, ищет элементы со значениями больше или равными a и меньше или равными b и возвращает результат в виде массива. (Функция должна возвращать новый массив и не изменять исходный.)

4. Создайте сортировку на сайте элементов в порядке возрастания/убывания.