МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ЕЛЕЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. И.А. БУНИНА»

С.Е.Попов

JavaScript.
Основы программирования.
Учебно-методическое пособие

Елец – 2020

УДК 002
ББК 32.97я73
Т 19

Печатается по решению редакционно-издательского совета
Елецкого государственного университета имени И.А. Бунина
от 29.09.2020 г., протокол № 1

Рецензенты:
Каверин Олег Владимирович, ООО «Шанс Энтерпрайз», системный администратор.
Тарова Инна Николаевна, кандидат педагогических наук, доцент.

С.Е.Попов

Т 19 JavaScript.Основы программирования: учебно-методическое пособие. –
Елец: Елецкий государственный университет им. И.А. Бунина, 2020. – 116 с.

В курсе описываются основные технологии разработки программного кода на JavaScript. Дается разъяснение основных концепций. Рассматриваются архитектура программного кода, , способы реализации различный программных конструкций и другие вопросы.

Учебное пособие предназначено для студентов второго и третьего курсов СПО и бакалавриата, обучающихся по направлению «Программирования» и изучающих методы разработки ПО. Учебно-методическое пособие также может быть использовано студентами, аспирантами и преподавателями, а также специалистами, которые планируют работать программистами или желают углубить знания технологий разработки и научиться разрабатывать веб-приложения.

УДК 002
ББК 32.97я73

Содержание

Введение.

В современном мире информационные технологии развиваются крайне быстрыми темпами, поэтому на сегодняшний день очень актуальна проблема подготовки высококвалифицированных кадров. Действующие образовательные программы высших учебных заведений часто не справляются с этой задачей, поскольку в большей степени ориентированы на изучение фундаментально-теоретических и академических знаний, из-за чего студенты после завершения обучения не имеют практического производственного опыта работы в реальных условиях. Данное учебно-методическое пособие призвано решить проблему недостатка практических знаний в области разработки фронтенд-приложений у выпускников вузов и ориентировано на студентов, обучающихся по программам бакалавриата и СПО по специальности «Программирование».

1.Введение в программирование.

1.1.Математика и константы.

- Математика в JavaScript выглядит и воспринимается как привычная математика:
  - Символы +, -, *, / означают то, что вы думаете.
  - Ещё есть символ % — остаток от деления. Он вычисляет остаток от деления первого операнда на второй. Например, 11%5 это 1, а 10%2 это 0.
  - Есть Infinity и -Infinity.
  - Когда вычисление оказывается не числом, JavaScript использует NaN, который означает "не число" (Not a Number). Например: 0/"word" это NaN
  - Если в вычислении присутствует NaN, то результатом всегда будет NaN. Например: 120 + 5 / NaN это NaN
- Заставьте компьютер "помнить" что-нибудь, создав константу. Например: const age = 39;

Конечно, компьютеры отлично справляются с вычислениями чисел. И когда дело касается простой математики, JavaScript довольно прямолинеен. Есть пять основных операций: сложение, вычитание, умножение, деление и остаток от деления. Ещё есть скобки, как и в обычной математике, которые помогают явно указывать последовательность вычислений.

Взгляните на это: 25 * 91. 25 и 91 называются **операндами**, а звёздочка - **оператором** умножения.

Вот немного более усложнённый пример: ((12 * 5) - 4) / 12.

В начале JavaScript производит умножение, затем вычитает 4, поскольку есть скобки, а потом делит результат на 12.

Все операторы здесь **инфиксные**: они находятся между операндами (в данном случае — между числами). Ещё есть **префиксные** операторы (например, знак минус, который обозначает отрицательное число: -5) и **постфиксные** (например, быстрое увеличение на один: x++). Мы изучим их позже.

В какой-то момент вы столкнётесь с необычной проблемой: если вы попытаетесь сложить 0.1 + 0.2 в JavaScript, результатом будет 0.30000-много-много-нулей-4, а не 0.3. Это потому что компьютеры

хранят числа в другом формате. Глубоко внутри все числа — это множество единиц и нулей, подчинённых определённым правилам и это не лучший формат для хранения любых чисел.

Это может показаться нелепым — почему мы позволяем компьютерам использовать такую плохую систему? На самом деле она не настолько плохая или глупая. Такой формат удобен для определённого набора задач и не слишком удобен для другого.

Также в JavaScript есть несколько терминов, которые нужны для формулировки определённых значений: разделите положительное число на 0 и получится "Бесконечность" — Infinity; разделите отрицательное число на 0 и получится "-Бесконечность" — -Infinity. В своих программах вы можете использовать Бесконечности как числа с другими операторами. Например, с Бесконечностью можно производить сложение.

Иногда вычисления не производят конкретного числа. Разделите 0 на строку и получится что-то не числовое. Нельзя сказать, что это ничто, это... просто не число. У JavaScript есть термин для такого понятия — NaN, который образован из "Not a Number" (не число).

Как и Бесконечность, NaN можно использовать в вычислениях с другими математическими операторами. Но Not a Number как бы всех подводит: если он присутствует в вычислении, результатом всегда будет NaN.

Вот случайный вопрос: какого размера Марс? Его радиус — 3390 километров, он почти в два раза меньше Земли. Но мы, конечно, заинтересованы там жить, поэтому нам важно сколько у нас будет поверхности. Другими словами, какая площадь поверхности у Марса?

Может вы помните формулу: площадь поверхности сферы равна $4\pi r^2$. r — это радиус, а π примерно 3.14.

Давайте вычислим это в JavaScript:

```
4 * 3.14 * 3390 * 3390;
```

Теперь представьте как вычислить площадь поверхности другой планеты. Например, Меркурия:

```
4 * 3.14 * 2440 * 2440;
```

Этот новый код точно такой же, как и предыдущий, изменился только радиус. Если мы продолжим в таком же стиле, нам понадобится писать значение π самим каждый раз. Так не пойдёт, мы не хотим повторений в своих программах.

Мы можем заставить компьютер "помнить", что такое π и использовать это в вычислениях. Такой механизм называется "константы". Давайте создадим новую константу со значением числа π:

```
const pi = 3.14;
```

const — это специальное ключевое слово, после него — идентификатор — название вашей константы, затем знак равенства и значение.

Теперь мы можем писать "pi" вместо того, чтобы вводить вручную 3.14.

```
4 * pi * 3390 * 3390; // surface area of Mars
4 * pi * 2440 * 2440; // surface area of Venus
```

Кстати, двойной слеш и текст после него — это **комментарии**: JavaScript просто игнорирует их и они не влияют на работу кода. Мы пишем комментарии для себя и других людей, чтобы легче понимать код.

Давайте поместим площадь поверхности Марса в другую константу:

```
const surface = 4 * pi * 3390 * 3390;
```

Теперь surface это другой идентификатор, в нём хранится **результат** вычисления. Тогда как это вычисление произвелось? Ну, во-первых, JavaScript должен вспомнить что такое pi, а потом заменить его на число, значение:

```
4 * pi * 3390 * 3390;
4 * 3.14 * 3390 * 3390;
```

А затем идут умножения слева направо, потому что у нас нет скобок:

```
4 * 3.14 * 3390 * 3390;
12.56 * 3390 * 3390;
42578.4 * 3390;
144340776;
```

Мы можем вывести результат на экран с console.log: console.log(surface). Заметьте, что в этот раз мы не писали кавычки. Мы не будем выводить слово "surface". Это не строка. Мы выводим значение константы, называемой "surface".

Возведение в степень

Стандарт es7 вводит новый оператор **, полезный при возведении в степень:

```
2 ** 4; // 16
3 ** 2; // 9
```

Другой способ возвести число в степень:

```
Math.pow(2, 4); // 16
Math.pow(3, 2); // 9
```

Почему константы?

Константы гораздо проще, чем переменные. Они всегда однозначно определены и никогда не меняются. В школе и в ВУЗе, в математике и физике мы имели дело только с константами. Единственное место, где требуются переменные — это циклы. Практически во всех других случаях они не нужны, и их присутствие усложняет.

Арность

Арность — это количество аргументов. Арность всех +, -, * и / составляет 2, поэтому мы можем называть эти операторы *бинарными*. Есть ещё один оператор, который выглядит как двоичный минус,

но он унарный (одинарный). Это когда - используется для обозначения отрицательного числа, например -327. Ещё бывают *тернарные* (троичные) операторы, но вы их не часто встретите.

Ассоциативность

Ассоциативность (или фиксированность) определяет, как операторы группируются при отсутствии скобок. Рассмотрим выражение a ~ b ~ c. Если у оператора ~ левая ассоциативность, это выражение будет трактоваться как (a ~ b) ~ c. Почитайте подробнее об ассоциативности в википедии.

Undefined

Рассмотрим следующий код:

```
const a;
console.log(a);
```

Что в этом случае выведется на экран? Современный JavaScript вообще не позволит вам создать константу без значения, но если позволит, полученный console.log выведет undefined. Это специальный идентификатор.

Вы можете установить значение undefined сами, вот таким способом:

```
const a = undefined;
```

Но самому этого делать не стоит.

1.2. Условия принятия решений.

Условие формально выглядит так:

```
if (условие) then
  выполнить что-то
else if (другое_условие) then
  выполнить что-то другое
else (ни одного из тех условий) then
  выполнить что-то ещё
```

JavaScript-функция, которая принимает значение и возвращает абсолютное значение:

```
const abs = (num) => {
  if (num > 0) {
    return num;
  } else if (num < 0) {
    return -num;
  } else {
    return 0;
  }
}
```

Условия могут быть либо истинным (true) либо ложным (false). Например, (num > 0) истинно, когда num равно 9 или 15, но ложно, когда num -18 или, скажем, 0.

То, что даёт ответ TRUE или FALSE, называется **предикатом**.

Другие математические предикаты в JavaScript:

```
===
!==
>
<
>=
<=
```

Примеры:

```
512 === 512;    // true
512 === 988;    // false

512 !== 378;    // true
512 !== 512;    // false

512 > 500;      // true
512 > 689;      // false

512 < 900;      // true
512 < -30;      // false

512 >= 512;     // true
512 >= 777;     // false

512 <= 512;     // true
512 <= 600;     // true
512 <= 5;       // false
```

AND (&&):

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

OR (||):

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

NOT (!):

A	NOT A
TRUE	FALSE
FALSE	TRUE

Альтернативный способ реализации функции abs:

```
const abs = (num) => {
  if (num === 0 || num > 0) {
    return num;
  } else {
    return -num;
  }
}
```

Можно придумать другой способ реализации той же функции, с использованием символа >= .

Та же функция значит: "назначение функции то же, но внутренность (реализация) может отличаться".

Важные заметки

Тернарный оператор

В JavaScript и многих других языках есть сокращённая версия if: она называется **тернарным оператором** (ternary operator):

```
condition ? expression : expression
```

В этом случае есть только два условия и два варианта: один для true и один для false.

```
const absValue = (num === 0 || num > 0) ? num : -num;
```

Мы создаём absValue и присваиваем ему значение. Это значение зависит от условия: если условие истинно, тогда используется num , в противном случае используется -num.

Тернарная операция VS условная конструкция if

Между этими двумя элементами языка есть различия. Тернарная, или условная, операция **вычисляет и возвращает значение**, то есть является **выражением**. Это значит, что мы можем сохранить результат вычисления этого выражения в константе, например:

```
const absValue = (num >= 0) ? num : -num;
```

Условная конструкция if в JavaScript выражением НЕ является. Это **инструкция** — она выполняет **действие**, ничего не вычисляя и не возвращая. Как следствие, мы НЕ можем с помощью конструкции if сделать так же:

```
const absValue = if (num >= 0) { ... }; // unknown: Unexpected token
```

Такая строчка кода приведёт к ошибке. Кстати, в некоторых других языках, например в Ruby, if реализован как выражение, поэтому там подобное присваивание возможно.

Чтобы сделать тоже самое, придётся "попотеть":

```
let absValue;

if (num >= 0) {
  absValue = num;
} else {
  absValue = -num;
}
```

Как видно, код стал более громоздким и сложным. Более того, нам пришлось ввести изменяемое состояние — вместо константы const использовать переменную (let). То есть мы позволили всему остальному коду изменять значение absValue, хотя объективно это больше нигде не понадобится. Тем самым мы заложили потенциальную возможность для будущих ошибок.

Результат вычисления тернарной операции можно вернуть из функции, поставив её после инструкции return. Особенно хорошо она сочетается с функциями-однострочниками, когда надо вернуть то или иное значение в зависимости от логического условия:

```
const getAbs = num => (num >= 0) ? num : -num;
```

Вместо унылого:

```
const getAbs = (num) => {
  if (num >= 0) {
    return num;
  }
  return -num;
};
```

Таким образом, тернарная операция позволяет писать более лаконичный и простой код. Но не стоит увлекаться, у неё есть свои минусы и ограничения по сравнению с конструкцией if, всё зависит от конкретной ситуации:

"Вложенные" тернарные операции выглядят эффектно, но ухудшают читабельность кода:

```
const type = (num > 0) ? 'positive' : (num < 0) ? 'negative' : 'zero';
```

Тернарная операция не подойдёт, если в зависимости от условия надо выполнить несколько (а не одно выражение) строчек кода (блок кода). Нужна условная конструкция if:

```
if (condition) {
  statement;
  ... ;
  ... ;
} else {
  statement;
  ...;
  ...;
}
```

Используя в теле функции, мы можем легко вернуть результат вычисления тернарника, поставив его после инструкции return. Ведь это обыкновенное выражение. Однако, инструкцию return нельзя использовать **внутри** тернарной операции. Интерпретатор вас не поймёт и выкинет ошибку:

```
const getAbs = (num) => {
  (num >= 0) ? return num : return -num;
};

const result = getAbs(-3); // unknown: Unexpected token
```

1.3.Визуализация рекурсивного процесса

https://goo.gl/gWCcgm

Представляем функции

- Можно представить функции как чёрные коробки: коробка забирает объект, производит внутри какие-то действия, а потом выплёвывает что-то новое
    - Некоторые функции ничего не забирают (не принимают аргументы), некоторые вообще ничего не делают (они пустые), некоторые не возвращают значения.
    - Наш surfaceAreaCalculator принимает один аргумент (radius), вычисляет площадь поверхности и возвращает результат этого вычисления.
- Функции могут вызывать другие функции
- surfaceAreaCalculator может вызывать функцию square, чтобы получить радиус, возведённый в квадрат, вместо того, чтобы умножать радиус на радиус.
- Мы пишем функции, чтобы облегчить жизнь:
    - такой код легче понимать
    - функции могут переиспользоваться несколько раз

Сравните:

```
const surfaceOfMars = surfaceAreaCalculator(3390);  // это "ЧТО", в таком виде легче понять суть
const surfaceOfMars = 4 * 3.14 * 3390 * 3390;       // это "КАК"
```

Две функции вместе:

```
const surfaceAreaCalculator = (radius) => {
  return 4 * 3.14 * square(radius);
}

const square = (num) => {
  return num * num;
}
```

Функции, которые вызывают сами себя

- **Определение функции** — это описание коробки
- Оригинал коробки формируется при **вызове функции**
- Когда функция вызывает сама себя, создаётся новая идентичная коробка

Перестановки:

- Количество способов перестановки n объектов равно n! (перестановки)
- n! определяется таким способом: если n = 0, то **n! = 1**; если n > 0, то **n! = n * (n-1)!**

Функция, вычисляющая факториал:

```
const factorial = (n) => {
  if (n === 0) {
    return 1;
  }
  else {
    return n * factorial(n - 1);
```

```
  }
}

const answer = factorial(3);
```

Требования рекурсии

1. Простой базовый случай, или терминальный сценарий, или терминальное условие. Простыми словами, когда остановиться. В нашем примере это была 1: мы остановили вычисление факториала, когда достигли 1.
2. Правило двигаться по рекурсии, углубляться. В нашем случае, это было `n * factorial(n-1)`.

Ожидание умножения

Ничего не умножается, пока мы спускаемся к базовому случаю `factorial(1)`. Затем мы начинаем подниматься обратно, по одному шагу.

```
factorial(3);
3 * factorial(2);
3 * 2 * factorial(1);
3 * 2 * 1;
3 * 2;
6;
```

Примечание

Заметьте, что 0! это 1, а простой базовый случай для n! это 0! В этом уроке мы пропустили такой случай, чтобы сократить рекурсию на один вызов и на одну коробку, поскольку 1 * 1 — это, в любом случае — 1.

Просто ради забавы

У программистов есть одна шутка: "Чтобы понять рекурсию, нужно понять рекурсию". Google, кажется, любит такие шутки. Попробуйте погуглить "рекурсия" и зацените верхний результат поиска ;-)

Рекомендуем посмотреть

- What on Earth is Recursion? - Computerphile
- Recursion (Part 7 of Functional Programming in JavaScript) from Fun Fun Function

Опциональный текст

- Properties of recursive algorithms

Опциональные видео

- Fibonacci Programming - Computerphile

1.4. Функции.

Способы записи функций

Определение функции:

```
// const <name> = (<argument>) => {
//   return <expressions>;
// };

const identity = (value) => {
  return value;
};
```

Существует несколько других. Например, если у вас функция-однострочник, то можно использовать сокращенный синтаксис:

```
// const <name> = (<argument>) => <expressions>;
const identity = value => value;
```

В коде выше мы опустили фигурные скобки и слово return. Заодно мы опустили скобки вокруг аргумента — это можно делать только если у функции один аргумент.

Так же можно определять функции используя ключевое слово function:

```
// Устаревший синтаксис, предпочтительным является () => {}. Кроме синтаксической разницы есть и семантическая.
// Она связана с пока не изученной темой this.
const identity = function(value) {
  return value;
};
```

или

```
// Такую функцию можно использовать до её определения (в этом же файле).
function identity(value) {
  return value;
}
```

Мы изучаем новый стандарт ES6 и используем стрелочные функции. Поэтому мы будем придерживаться такой () => {} формы записи по многим причинам. Во-первых, она лаконичнее, во-вторых, обладает одним важным свойством, которое будет изучено позже, ну а в-третьих, такой способ записи визуально стирает грань между функциями и данными, что очень пригодится нам в будущем.

Некоторые особенности такой формы записи вместо function выходят за рамки базовых курсов, вы о них узнаете в будущем. Если любопытно, то можете почитать статьи по запросу «стрелочные функции es6»,
например https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Формальные и Фактические параметры функции

Формальными параметрами функции называются имена переменных в *определении функции*. Например у функции const f = (a, b) => a - b; формальные параметры — это a и b.

Фактические параметры — это то, что было *передано в функцию* в момент вызова. Например если предыдущую функцию вызвать так f(5, z), где const z = 8, то фактическими параметрами

являются 5 и z. Результатом этого вызова будет число -3, а внутри функции на момент конкретного вызова параметр a становится равным 5, а b становится равным 8.

```
const f = x => x * x;

const y = 5;
console.log(f(y)); // 25

const z = 3;
console.log(f(z)); // 9
```

Как видите, нет никакой связи между именами формальных и фактических параметров. Более того, у фактических параметров вообще может не быть имен, как в примере выше, где мы сразу передали число 5 в функцию. Что имеет значение, так это **позиция**. Во время вызова функции параметры должны передаваться в правильном порядке, и только тогда функция отработает, как предполагается.

```
const f = (a, b) => a - b;

const x = 5;
const y = 8;

console.log(f(x, y)); // -3
console.log(f(y, x)); // 3
```

Return

Вызов оператора return приводит к изменению течения программы. Последующие инструкции никогда не будут выполнены:

```
const identity = (value) => {
  return value;
  const a = 3 + 5; // этот код никогда не будет достигнут
};
```

- Функции подобны специальным машинам, которые мы создаём, чтобы они выполняли что-нибудь для нас.
- Функции можно представить в виде ящиков, которые **принимают что-то** и **что-то выплёвывают**.
    - Функции могут принимать **аргументы**.
    - И могут **возвращать** что-то.

Вот пример определения и вызова функции:

```
const surfaceAreaCalculator = (radius) => {
  return 4 * 3.14 * radius * radius;
};

const surfaceOfMars = surfaceAreaCalculator(3390);
```

Здесь radius — это аргумент функции surfaceAreaCalculator.

Вы можете вызывать функцию и вкладывать вызов другой функции в качестве аргумента:

```
console.log(surfaceAreaCalculator(3390));
```

Опционально

- JavaScript variable name validator
- Valid JavaScript variable names

В прошлый раз мы упростили вычисление с помощью констант: мы создали константу для pi, и использовали её для вычисления площади поверхности разных планет. Таким образом мы сократили объём кода, но нам всё ещё требовалось множество повторений:

```
const surfaceOfMars    = 4 * pi * 3390 * 3390;
const surfaceOfMercury = 4 * pi * 2440 * 2440;
```

Строка кода называется **инструкцией**, в JavaScript инструкции должны заканчиваться точкой с запятой. Тут у нас две инструкции и нам нужно повторять всю формулу в каждой из них. Помните — мы не хотим, чтобы в наших программах что-то повторялось.

Было бы здорово, если бы у нас была какая-нибудь специфическая машина, которая вычисляет площадь поверхности любой сферы. Чтобы мы могли делать что-то подобное:

```
const surfaceOfMars    = surfaceAreaCalculator(3390);
const surfaceOfMercury = surfaceAreaCalculator(2440);
```

Чтобы можно было просто сказать ей: "эй, surfaceAreaCalculator, скажи мне площадь поверхности сферы с радиусом 3390" и она бы сразу давала ответ.

Отличные новости! Мы можем создавать подобные машины! Их называют **функциями**, их можно представить в виде чёрных ящиков. Положите что-нибудь внутрь ящика, и она выплюнет что-то другое. В данном случае ящик surfaceAreaCalculator возьмёт одно число — радиус, а выплюнет другое — площадь поверхности.

В этом примере

```
const surfaceOfMars = surfaceAreaCalculator(3390);
```

мы вызываем ящик surfaceAreaCalculator, даём ему 3390 и получаем ответ. Этот ответ затем сохраняется в константе surfaceOfMars.

Важно понимать разницу между определением функции и её вызовом. То, что мы только что сделали, называется вызовом — мы **вызвали** функцию surfaceAreaCalculator, другими словами, мы предположили, что она уже существует и использовали её.

Но она должна существовать, так что нужно создать её перед тем, как использовать. Нам нужно **определить** функцию surfaceAreaCalculator. Это определение функции:

```
(radius) => {
  return 4 * 3.14 * radius * radius;
};
```

Эта структура со скобками, стрелкой и фигурными скобками определяет функцию. Слово radius в скобках — то, как функция называет число, которое мы передаем в нее. Это **аргумент** функции. Эта конкретная функция принимает один **аргумент**, но другие функции могут принимать больше аргументов или вообще не принимать аргументов.

Фигурные скобки создают **блок**. В JavaScript и других языках вы часто сталкиваетесь с блоками. Они создают группу **инструкций**, таким способом мы понимаем где функция начинается и заканчивается.

Эта функция содержит всего одну инструкцию, она начинается с return, а дальше вы видите уже знакомую формулу.

Вы, возможно, уже догадались, что return заставляет ящик выплёвывать результат. Результат вычисления 4 * pi * radius * radius это то, что функция **возвращает** нам после того, как мы её вызываем.

Теперь нам нужно дать этой функции какое-нибудь название, чтобы по этому названию мы её вызывали. Мне захотелось назвать её surfaceAreaCalculator, но вы можете давать любой функции любое название. Есть определенные правила для названий: например, название не должно содержать пробелы и использовать некоторые специфические слова, как const, но в остальном и функции и константы можно называть почти как захочется.

Мы уже знаем как давать названия численным константам:

```
const pi = 3.14;
```
С функциями все точно также:

```
const surfaceAreaCalculator = (radius) => {
  return 4 * 3.14 * radius * radius;
};
```
const, название, которое вы хотите, знак равенства и сама функция

Давайте теперь соберём всё вместе:

```
const surfaceAreaCalculator = (radius) => {
  return 4 * 3.14 * radius * radius;
};

const surfaceOfMars = surfaceAreaCalculator(3390);
```
В верхней части — определение функции, а затем вызов функции.

Давайте заглянем внутрь коробки — что происходит, когда она вызывается с числом 3390 в качестве аргумента. return хочет возвратить результат, но он ещё не готов — в начале нужно произвести вычисление. В вычислении используется аргумент, так что вначале нужно заменить название radius на само значение, которое было передано в функцию при вызове, а затем уже выполнять умножения.

```
4 * 3.14 * radius * radius;
4 * 3.14 * 3390 * 3390;
12.56 * 3390 * 3390;
42578.4 * 3390;
144340776;
```
И функция возвращает число 144340776. Можно представить, что происходит снаружи, когда наша функция возвращает значение:

```
const surfaceOfMars = surfaceAreaCalculator(3390);
const surfaceOfMars = 144340776;
```
Теперь surfaceOfMars — это константа с конкретным числовым значением.

Давайте посмотрим на другой пример:

```
const percentageCalculator = (number, total) => {
  return number * 100 / total;
};
```
Эта функция принимает два числа и возвращает процент. Если вы дадите ей 40 и 80, она вернёт 50, потому что 40 это 50% от 80.

Как вы видите, когда есть несколько аргументов, они разделяются запятыми.

Давайте выведем результат на экран:

```
console.log("How much of December is gone already? ");
console.log(percentageCalculator(16, 31));
```

Сегодня 16 декабря, и я хочу узнать, какой процент этого месяца уже прошёл. Посмотрите на вторую строку — здесь есть два важных момента.

Первый: мы можем разместить вызов функции в любом месте, где можно вставить строку или константу. Вначале мы напечатали текст, а потом напечатали то, что возвращает функция percentageCalculator.

И второй важный момент: вызов функции может быть аргументом для других функций! console.log — это функция, вот почему мы ставим скобки, когда её используем. Функция console.log принимает аргумент и выводит его на экран. Мы ввели вызов функции, КАК аргумент и это сработало. Наша функция была вызвана вначале, затем был возвращён результат, затем console.log был вызван с этим результатом, как аргументом, и он был выведен на экран.

1.5. Переменные. Декларативное и императивное программирование.

Переменная определяется с помощью команды let

Например, let i=1.

Мы уже знакомы со способом называть что-то, используя константы. Например, это константа pi со значением 3.14.

```
const pi = 3.14;
```

После этой строки, каждый раз, когда мы видим pi, мы знаем, что её значение 3.14. Она называется константой, потому что, ээм, она неизменна, постоянна. После этой строки pi всегда будет 3.14, это никогда не изменится. Именно поэтому по аналогии с бумагой я использую ручку.

Это может показаться ограничением, но вообще — это довольно хорошее свойство. Изменение того, что мы уже создали — сложная задача. Представьте, что пишите код, используя константу, в значении которой не уверены.

Тем не менее, иногда вам может потребоваться изменить уже существующие данные или, другими словами, их значения. Допустим, вы хотите повторить что-то 5 раз. Один способ — выполнять повторы, отсчитывая до пяти, а затем остановиться. Для этого вам потребуется что-нибудь для хранения счётчика этого меняющегося числа. Константа не будет в таком случае работать — она неизменна, вы не можете изменить её значение после того, как уже создали её.

Поэтому в JavaScript и многих других языках программирования существует идея переменной. Можно представить ее в виде такой же бумажки, на которой имя написано ручкой, но значение — карандашом. В любой момент можно заменить значение на другое.

Можно посчитать факториал с помощью переменной вот так:

```
let factorial = 1;

factorial = factorial * 2;   // 2
factorial = factorial * 3;   // 6
factorial = factorial * 4;   // 24
factorial = factorial * 5;   // 120
```

Создать переменную — просто, она выглядит как константа, только вместо const мы пишем let. Мы позволяем ей быть чем-то и это не навсегда.

Затем мы изменяем значение factorial. Мы бы такого не смогли сделать, если бы factorial был константой. Эта строка означает "изменить значение перменной факториал, на результат умножения факториала на 2". Теперь JavaScript умножает factorial на 2 и хранит этот результат в переменной factorial. Раньше factorial был 1, а теперь это 2.

Мы повторяем это ещё трижды, каждый раз умножая полученное значение на следующе целое число: на 3, на 4 и на 5. Это то, что вы возможно делаете в уме, когда умножаете числа. Вероятно вы не думали об этом так чётко, но это неплохой способ описания процесса вычисления, если бы вам было нужно объяснить его.

Идея использования счётчика для повторения чего-то множество раз — распростанённая в программировании, и большинство языков программирования имеют для этого "циклы". Давайте рассмотрим один тип цикла — "цикл while". Это блок кода, который повторяется, пока удовлетворяется какое-то условие.

Представьте фермера, который работает от рассвета до заката. Другими словами, он работает пока солнце в небе. Вы можете записать:

```
while (sun is up) {
  work
}
```

Конечно, это не настоящий JavaScript, это просто чтобы показать идею. Эта строка "work" будет повторяться снова и снова, пока солнце над горизонтом. Это значит после каждого повторения нам нужно проверять, действительно ли солнце в небе, и остановиться если это не так. Другими словами: проверить — исполнить, проверить — исполнить, и так далее.

Вот функция факториала с переменными и циклом вместо рекурсии.

```
const factorial = (n) => {
  let counter = 1;
  let result  = 1;

  while (counter <= n) {
    result = result * counter;
    counter = counter + 1;
```

```
  }

  return result;
}
```

Что тут происходит? Во-первых, мы создали две переменные: одна для счётчика, чтобы считать от 1 до верхнего предела, а вторая для текущего результата.

Затем начинается главная часть: цикл while, который повторяется, пока счётчик меньше или равен n — числу, переданному в эту функцию. Код, который повторяется, простой: мы меняем значения наших двух переменных. Текущий результат умножается на счётчик, а счётчик увеличивается на 1.

В какой-то момент это условие — "счётчик меньше или равен n" — станет ложным, цикл больше не будет повторяться, а программа перейдёт к следующему этапу — return result. К этому моменту результат станет ответом, потому что за время всех повторов в цикле, результат умножался на 1, затем на 2, 3 и так далее, пока не достиг значения n, каким бы оно ни было.

Давайте посмотрим, что компьютер делает шаг за шагом, когда мы вызываем факториал 3.

1. Взять один аргумент — 3, известный внутри, как n
2. Создать переменную counter, установить значение 1
3. Создать переменную result, установить значение 1
4. Проверить: в счётчике — 1, это меньше или равно n, поэтому
5. Умножить result на counter и положить ответ — 1 — в result
6. Добавить 1 к counter и положить ответ — 2 — в counter
7. Вернуться и проверить: counter — 2, это меньше или равно n, поэтому
8. Умножить result на counter и положить ответ — 2 — в result
9. Добавить 1 к counter и положить ответ — 3 — в counter
10. Вернуться и проверить: counter — 3, это меньше или равно n, поэтому
11. Умножить result на counter и положить ответ — 6 — в result
12. Добавить 1 к counter и положить ответ — 4 — в counter
13. Вернуться и проверить: counter — 4, это не меньше и не равно n, поэтому остановить повтор и перейти к следующей строке
14. Вернуть result — 6

Компьютер выполняет такие операции в миллиарды раз быстрее, но по сути это выглядит именно так. Общее название такого вида сформулированных повторений — "итерация". Наша программа использует итерацию, чтобы рассчитать факториал.

В прошлый раз мы рассматривали итеративный процесс с рекурсией, а в этот — итеративный процесс без рекурсии.

Оба используют технику итерации, но с рекурсивными вызовами нам не нужно менять значения, мы просто передаём новые значения в следующий вызов функции. А эта функция факториала не имеет рекурсивных вызовов вообще, поэтому все трансформации должны происходить внутри единственного экземпляра, единственной функциональной коробки. У нас нет выбора, кроме как менять значения содержимого.

Стиль программирования, который вы видели в предыдущих уроках, называется "декларативным". Сегодняшний стиль, с изменением значений, называется "императивным".

Сравните рекурсивный и нерекурсивный факториалы:

```
const recursiveFactorial = (n) => {
  if (n === 1) {
    return 1;
  }
  return n * recursiveFactorial(n-1);
}

const factorial = (n) => {
  let counter = 1;
  let result  = 1;

  while (counter <= n) {
    result = result * counter;
    counter = counter + 1;
  }

  return result;
}
```

Эта рекурсивная функция — декларативная — она как описание факториала. Она объясняет, что такое факториал.

Это нерекурсивная итеративная функция, и она императивная — она описывает, что делать, чтобы найти факториал.

Слово декларативный происходит от латинского "clarare" — *разъяснять, заявлять, делать объявление*. Вы разъясняете: я хочу, чтобы факториал n был n умножить на факториал n-1.

Слово императивный происходит от латинского "imperare", что значит "*командовать*". Вы приказываете чётко передвигаться по шагам — умножать это на это, пока идёт отсчёт и запоминать какие-то числа.

Декларативное — это что. Императивное — это как.

Писать декларативный код, в целом, лучший подход. Ваш код будет легче читать, понимать, и делать что-то новое опираясь на него. Некоторые языки провоцируют вас использовать тот или иной подход, а некоторые вообще не оставляют выбора.

Но в итоге вам нужно будет научиться оценивать, когда императивный подход принесет больше проблем чем решений.

Следить за изменениями – сложно, и буквально несколько переменных могут сделать систему очень сложной для понимания.

От изменения состояния* появляется гора багов, а **оператор присваивания** (**assignment statements**), который создает изменения, часто является причинами всего зла во вселенной.

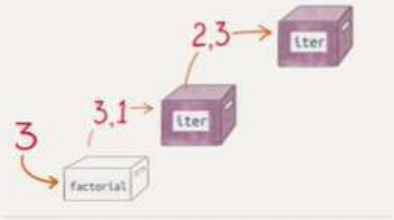
Декларативное vs. императивное программирование

Сравните рекурсивный факториал и нерекурсивный факториал:

Итеративный процесс с рекурсией

```
const factorial = (n) => {
  const iter = (counter, acc) => {
    if (counter === 1) {
      return acc;
    }
    return iter(counter-1, counter * acc);
  }

  return iter(n, 1);
}
```

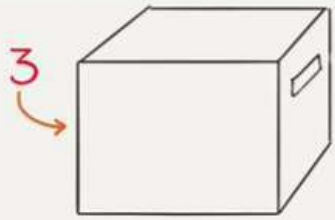


Итеративный процесс без рекурсии

```
factorial = (n) => {
  let counter = 1;
  let result  = 1;

  while (counter <= n) {
    result = result * counter;
    counter = counter + 1;
  }

  return result;
}
```



Эта рекурсивная функция — **декларативная** — она как бы определение (трактование, характеристика) факториала. Она декларирует, что такое факториал.

Эта нерекурсивная итеративная функция — **императивная** — описание того, что нужно делать, чтобы найти факториал.

Слово декларативный происходит от латинского "clarare" — *разъяснять, заявлять, делать объявление*. Вы разъясняете: я хочу, чтобы факториал n был n раз факториалом n-1.

Слово императивный происходит от латинского "imperare", что значит "*командовать*". Вы приказываете чётко передвигаться по шагам — умножать это на это, пока идёт отсчёт и запоминаются какие-то числа.

Декларативное — это что. Императивное — это как.

Писать декларативный код, в целом, лучший выбор. Ваш код будет легче читать, понимать, и делать что-то новое опираясь на него. Но иногда у вас нет выбора.

От изменения состояния* появляется гора багов, а **инструкции (операторы) присваивания** (**assignment statements**), которые создают изменения, часто являются коренными причинами всего зла во вселенной.

Поэтому, когда дело доходит до инструкций присваивания, **действуйте осторожно**.

Опционально

- Iteration on Wikipedia
- Fundamentals of Programming: Iteration

Важные заметки

var vs let

Кроме let существует другой способ определения переменных: var a = 5. Этот способ был единственным до появления стандарта ES6, но в современном JavaScript он является устаревшим, и let полностью заменил его.

let и var по-разному влияют на видимость переменной, и использование var сегодня нежелательно. let был создан как более правильная альтернатива старому способу.

Отладка: Как найти и исправить ошибки

Наделать ошибок очень легко, когда приходится справляться с переменными, изменять их, отслеживать и всякое такое. Особенно в зацикленном процессе. Прекрасный способ разобраться с тем, что происходит — использовать простейшую технику для отладки — console.log. Вспомните, эта функция выводит на экран то, что ей передают.

Например, я пытаюсь разобраться, что происходит внутри цикла while:

```
while (counter <= n) {
  result = result * result;
  counter = counter + 1;
}
```

Добавлю сюда console.log:

```
while (counter <= n) {
  result = result * result;
  counter = counter + 1;
  console.log(result)
}
```

Теперь, каждый раз, когда повторяется этот блок кода, переменная result будет выводиться на экран. Давайте посмотрим:

```
1
1
1
```

(Я запускаю функцию с n равным 3)

В каждом шаге result — это 1. Не верно — result должен возрастать на каждом шаге… Ок, значит вероятная проблема, в строке, где result меняется. Так и есть! У меня result = result * result, а мне нужно умножить result на counter, а не на result.

```
1
2
6
```

Теперь всё работает! Теперь я вижу шаги и последний шаг выдал верный ответ: 3! это 6.

Не задумываясь используйте console.log абсолютно везде. Это ваш лучший друг :-)

1.6.Строки и работа с символами

- Строка — это последовательность символов
- Пустая строка — это тоже строка (последовательность нуля символов)
- Обозначается единичными или двойными кавычками

Создание строки с константой:

```
const str1 = "Hello";
const str2 = 'Hello';
```

Возможно включить кавычку одного типа внутрь строки, окружив её кавычками другого типа:

```
const str1 = 'They call him "Harry", and he likes it';
const str2 = "They call him 'Harry', and he likes it";
```

Если в строке используются кавычки того же типа, они должны быть **экранированы** с помощью обратного слеша \:

```
const str1 = 'They call her \'Ann\', and she likes it';
const str2 = "They call her \"Ann\", and she likes it";
```

Если строка включает обратный слеш (именно как символ, который хочется иметь в строке), он должен быть экранирован другим обратным слешем:

```
const str = "This is a backslash \\ here"
// This is a backslash \ here
```

Так же существуют **управляющие символы** — специальные комбинации, которые генерируют невидимые детали:

```
const str = "There is a tab \t and here \ncomes the new line!"

// Here is a tab    and here
// comes the new line!
```

\t — это табуляция, \n это перенос на новую строку. Больше об экранировании (англ).

Конкатенация строк

Строки могут склеиваться друг с другом. Такой процесс называется **конкатенацией** и задаётся символом +:

```
const name = "Alex";
const age = 22;
console.log("His name is " + name + " and his age is " + age);

// His name is Alex and his age is 22
```

Строки будут склеены в том порядке, в котором они указаны: "mos" + "cow" → "moscow", а "cow" + "mos" → "cowmos"

Доступ к индивидуальным символам

str[i] это i-ый символ строки str, начинающейся с 0. Например, "hexlet"[0] это h, а "hexlet"[2] это x.

Вот функция, которая принимает строку и возвращает копию этой строки без каждой второй буквы. Например, "hexlet" становится "hxe".

```
const skip = (str) => {
  let i = 0;
  let result = '';

  while (i < str.length) {
    result = result + str[i];
    i = i + 2;
  }

  return result;
}
```

str.length это длина str, то есть количество символов. Это просто количество, поэтому мы **не** начинаем отсчёт от 0. Например, "food".length это 4.

Важные заметки

Неизменяемость

В JavaScript строки являются неизменяемыми, так же говорят "immutable". Это означает, что какие бы вы к ним не применяли функции, они не производят in-place замены (то есть не производят изменения самой строки). Любые строковые функции, примененные к строкам, возвращают **новую** строку. Это верно и в том случае, когда мы обращаемся к конкретному символу в строке.

Пример:

```
const str = 'hello';

str.toUpperCase(); // HELLO
console.log(str); // hello

str[0].toUpperCase(); // H
console.log(str); // hello

str[0] = 'W';
console.log(str); // hello
```

Лексикографический порядок

Лексикографический порядок — это, другими словами, алфавитный порядок. Такой порядок используется в словарях, телефонных справочниках, записных книжках и так далее.

В JavaScript вы можете сравнивать строки с помощью > и <, и сравнение будет происходить именно лексикографически.

Помните, '8' это не число, а строка.

Интерполяция

Кроме одиночных '' и двойных кавычек "", современный JavaScript содержит обратные тики (backticks):

``
С обратными тиками вы можете использовать **интерполяцию**, вместо конкатенации. Вот, смотрите:

```
const name = "Alex";
const a = 10;
const b = 12;
console.log(`His name was ${name} and his age was ${a + b}`);
```
Такой код выведет на экран His name was Alex and his age was 22. Внутрь ${} вы можете поместить любое выражение.

Интерполяция предпочтительнее конкатенации. Мы советуем не использовать конкатенацию вообще. Вот некоторые из причин:

- Такой код заставляет больше думать, потому что синтаксически + больше смахивает на сложение.
- Из-за слабой типизации можно легко получить не тот результат. Конкатенация может породить ошибки.
- Сложные строки при использовании конкатенации невозможно нормально разобрать в голове и понять, как они устроены.

1.7. Цикл FOR и изменение переменных

Вызовем функцию факториала с циклом while:

```
const factorial = (n) => {
  let counter = 1;
  let result  = 1;

  while (counter <= n) {
    result = result * counter;
    counter = counter + 1;
  }

  return result;
}
```
Когда мы работаем с переменными, мы часто поступаем так: меняем их значения, прибавляя к ним сколько-нибудь или умножая их на что-то. Или просто прибавляем или вычитаем единицу.

Как и во многих других языках программирования в JavaScript есть для этого упрощенные формы.

Вместо result = result * counter вы можете сказать result *= counter. Результат будет тот же самый, это просто способ сократить запись. Точно так же вы можете поступить со знаками плюс, минус и разделить:

```
result *= counter;    // то же, что result = result * counter
result += counter;    // то же, что result = result + counter
result -= counter;    // то же, что result = result - counter
result %= counter;    // то же, что result = result % counter
```

Добавление единицы к переменной — тоже очень типичная операция, поэтому вместо counter = counter + 1 можно записать counter++. Так же для минуса — counter = counter - 1 равносильно counter--. Это операторы инкрементирования и декрементирования.

Есть два вида, проще их понять на примере:

```
// Postfix
let a = 3;
let b;
b = a++;   // b = 3, a = 4

// Prefix
let a = 3;
let b;
b = ++a;   // b = 4, a = 4
```

Если вы поставите ++ после имени переменной — это постфиксная нотация — то фактическое сложение произойдёт после того, как значение вернётся. Вот почему b тут 3: оно получает значение перед тем как меняется a.

Если вы поставите ++ перед именем переменной — это префиксная нотация — то фактическое сложение произойдёт перед тем, как значение вернётся. Вот почему b тут 4: оно получает значение после того как меняется a.

Но в конце в обоих случаях a становится 4.

Это обновлённая функция факториала:

```
const factorial = (n) => {
  let counter = 1;
  let result  = 1;

  while (counter <= n) {
    result *= counter;
    counter++;
  }

  return result;
}
```

Здесь не имеет значения, используем мы префикс или постфикс когда инкрементируем counter, потому что значение не хранится больше нигде.

Этот код немного проще и короче. Иметь цикл — этот повторяющийся код — со счётчиком контролирующим повторения — распространённый в программировании приём. Поэтому кроме цикла **while** существует цикл **for**. В нем есть встроенные счетчики.

Это та же функция факториала, но с циклом **for** вместо цикла **while**:

```
const factorial = (n) => {
  let result  = 1;
```

```
  for (let counter = 1; counter <= n; counter++) {
    result *= counter;
  }

  return result;
}
```

Здесь три момента:

1. Инициализация счётчика.
2. Условие цикла. Так же как и в цикле while, этот цикл будет повторяться пока это условие истинно.
3. Обновление счётчика. Как менять счётчик в каждом шаге.

А затем следует тело, код, который должен повторяться. Нам не нужно менять счётчик в теле, потому что он будет меняться, благодаря этому выражению сверху.

Тут следует упомянуть о том, что все 3 выражения в цикле for не обязательны.

Например, в блоке инициализации не требуется определять переменные:

```
let counter = 1;
for (; counter <= n; counter++) {
  // любой код
}
```

Как и блок инициализации, блок условия не обязателен. Если пропустите это выражение, вы должны быть уверены, что прервете цикл где-то в теле, а не создадите бесконечный цикл.

```
for (let counter = 1;; counter++) {
  if (counter <= n) break;
  // любой код
}
```

Вы можете пропустить все 3 блока. Снова убедитесь, что используете break, чтоб закончить цикл, а также изменить счётчик так, чтоб условие для break было истинно в нужный момент.

```
let counter  = 1;
for (;;) {
  if (counter <= n) break;
  // любой код
  counter++;
}
```

Скрытые сложности

Операции декремента и инкремента кажутся мощными механизмами, но их использование привносит ненужную сложность в программы. Код, написанный с их использованием, часто превращается в ребус. Попробуйте ответить, чему равно значение:

```
let x = 5;
```

```
let y = 10;

console.log(x++ + ++y);
```
Как видите, этот код заставляет думать, так как кроме арифметических выражений, мы имеем дело с побочными эффектами.

Во многих языках таких операций нет в принципе. Линтеры (программы, проверяющие код на соответствие стандартам) в JS настроены так, чтобы "ругаться" при виде этих операций в коде. Вместо них предлагается делать так:

```
x += 1; // x = x + 1;
```
Что гораздо проще и понятнее. Да, не получится записать выражение в одну строку, но сам код будет очевидным и без сюрпризов.

Соответствующее правило в eslint: http://eslint.org/docs/rules/no-plusplus

Switch

Конструкция switch может заменить собой несколько условий if. Вот пример обычного условия с if:

```
let answer;

if (num === 1) {
  answer = "One";
} else if (num === 2) {
  answer = "Two";
} else {
  answer = "Nothing";
}
```
А вот как его можно переписать с помощью switch:

```
switch(num) {
  case 1:  // if (num === 1)
    answer = "One";
    break;

  case 2:  // if (num === 2)
    answer = "Two";
    break;

  default:
    answer = "Nothing";
    break;
}
```
break необходим, чтобы выйти из блока switch. Если break отсутствует, то выполнение пойдёт ниже по следующим случаям, игнорируя проверки. **break также можно использовать в циклах for для мгновенного выхода из цикла.**

Если в примере выше убрать все break'и, а num равен 1, то выполнятся все строки:

```
answer = "One";
answer = "Two";
```

```
answer = "Nothing";
}
```
Так что в итоге answer будет иметь значение "Nothing".

Несколько значений case можно группировать.

```
switch(num) {
  case 1:  // if (num === 1)
    answer = "One";
    break;

  case 2:  // if (num === 2)
  case 3:  // if (num === 3)
  case 4:  // if (num === 4)
    answer = "Two to four";
    break;

  default:
    answer = "Nothing";
    break;
}
```

1.8. Модули.

Вы можете дробить код на отдельные модули. В JavaScript один модуль — это один файл.

Объединение кода, расположенного в разных модулях, происходит через:

1. Экспорт чего-то из модуля.
2. Импорт в другой модуль.

Экспорт и два способа импорта

Поставьте export перед тем, что вы хотите экспортировать. Такая операция сделает это импортируемым куда угодно:

```
export const pi = 3.14;
export const e = 2.718;

export const square = (x) => {
  return x * x;
};

export const surfaceArea = (r) => {
  return 4 * pi * square(r);
};
```

Импортируйте функции следующим способом:

```
import { surfaceArea, square } from './math';

const surfaceOfMars = surfaceArea(3390);
const surfaceOfMercury = surfaceArea(2440);
const yearSquared = square(2017);
```

'./math' означает "из файла math.js, расположенного в той же (текущей) папке".

Или импортируйте всё:

```
import * as mathematics from './math';

const surfaceOfMars = mathematics.surfaceArea(3390);
const surfaceOfMercury = mathematics.surfaceArea(2440);
const yearSquared = mathematics.square(2017);
```

Это значит: "импортировать весь модуль и назвать его mathematics в этом модуле". Вот почему к импортированным сущностям обращение происходит через mathematics вот так: mathematics.surfaceArea.

Экспорт по умолчанию

Вы можете сделать одну позицию экспортируемой по умолчанию.

```
const pi = 3.14;
const e = 2.718;

const square = (x) => {
  return x * x;
};

const surfaceArea = (r) => {
  return 4 * pi * square(r);
};

export default surfaceArea;
```

Можно также экспортировать функцию или константу без имени:

```
const pi = 3.14;
const e = 2.718;
```

```
const square = (x) => {
  return x * x;
};

export default (r) => {
  return 4 * pi * square(r);
};
```

Импортирование чего-то, что было экспортировано по умолчанию:

```
import surfaceArea from './math';

const surfaceOfMars = surfaceArea(3390);
```

При экспорте функции без имени, её имя в модуле будет определяться в момент импорта, т.е. один и тот же экспорт может иметь разные имена в разных модулях:

math.js

```
export default () => {
  ///
};
```

import1.js:

```
import something1 from './math';
```

Import2.js:

```
import something2 from './math';
```

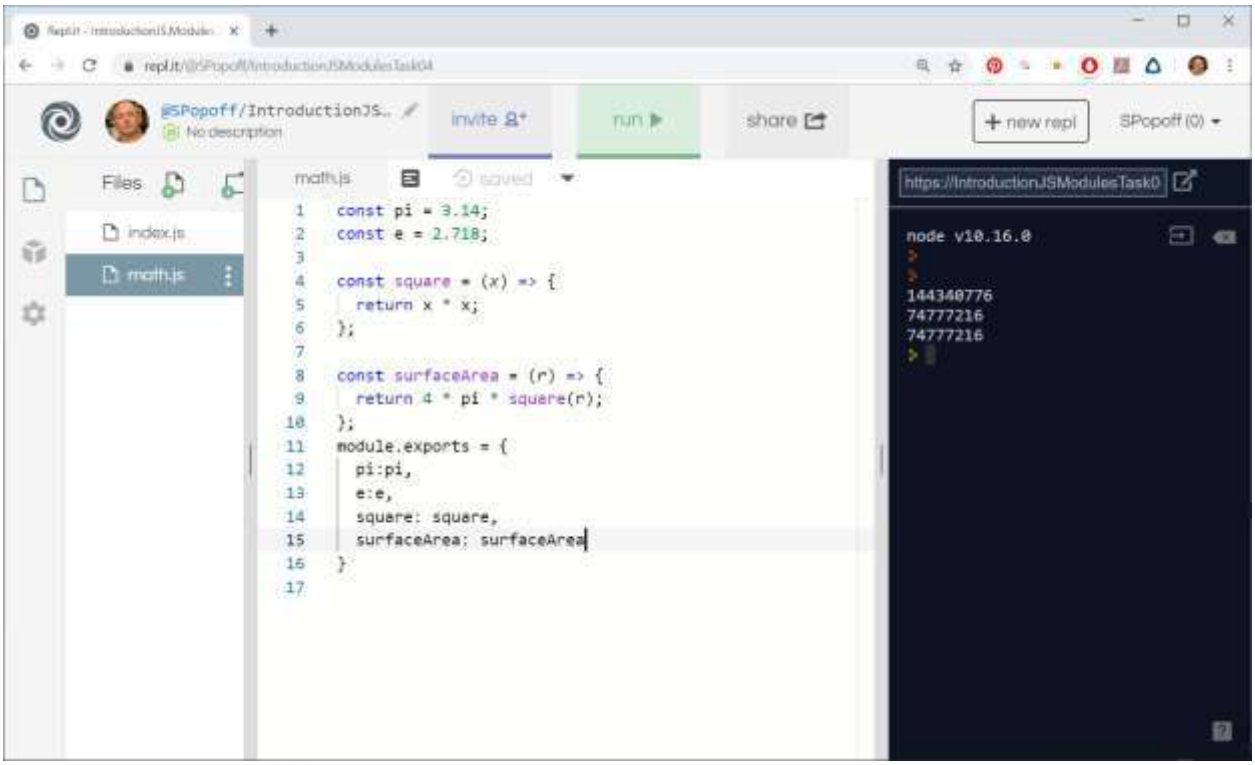
Опциональное чтение

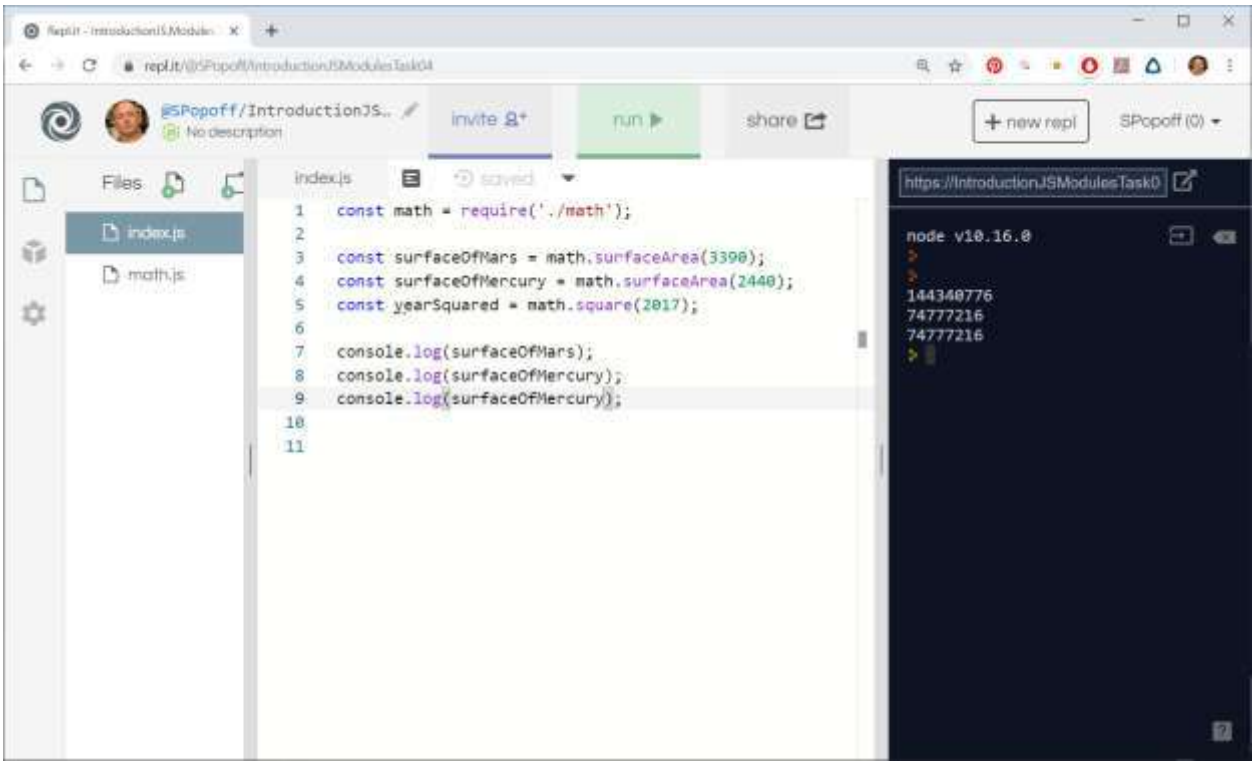
1. Тонкости модульной системы ECMAScript 2015 (ES6)
2. Exploring JS: Modules
3. Understanding ES6 Modules

В нашем случае на платформе Repl.It произошли серьезные изменения (в качестве транслятора для JavaScript выбрана платформа Node.js). Далее идет ссылка на мой репл.

https://repl.it/@SPopoff/IntroductionJSModulesTask04

Поэтому для нас есть единственный способ использовать модули:





Для файла math.js мы используем следующую конструкцию:

```
const pi = 3.14;
const e = 2.718;

const square = (x) => {
  return x * x;
};
```

```
const surfaceArea = (r) => {
  return 4 * pi * square(r);
};
module.exports = {
  pi:pi,
  e:e,
  square: square,
  surfaceArea: surfaceArea
}
```

Здесь мы добавляем секцию module.exports, где перечисляем экспортируемые константы.

В файле index.js мы создаем константу как результат выполнения функции require(), и данную константу используем для разыменования импортируемых функций и констант.

```
const math = require('./math');

const surfaceOfMars = math.surfaceArea(3390);
const surfaceOfMercury = math.surfaceArea(2440);
const yearSquared = math.square(2017);

console.log(surfaceOfMars);
console.log(surfaceOfMercury);
console.log(surfaceOfMercury);
```

1.9.Выражения и инструкции.

Полезные ссылки

- Expressions / Mozilla Developer Network

Выражением (expression) является любой корректный блок кода, который возвращает значение. (источник).

Ниже 5 это expression, оно выражается в значение 5:

const x = 5;
Ниже getAnswer() — это вызов функции — другое выражение. Этот вызов возвращает значение, то есть этот вызов функции выразится в значение:

const y = getAnswer();
Ниже пример выражения, которое состоит из нескольких *подвыражений*, и пошаговый процесс превращения каждого выражения по порядку, пока целое выражение не превратится в одно значение:

12 + square(7 + 5) + square(square(2));

```
12 + square(12) + square(square(2));
12 + 144 + square(square(2));
12 + 144 + square(4);
12 + 144 + 16;
156 + 16;
172;
```

JavaScript различает **выражения (expressions)** и **инструкции (statements)**. Инструкция — это (грубо говоря) команда, действие.

if, while, for, const — примеры инструкций. Они производят или контролируют действия, но не превращаются в значения.

Опционально

- Выражения / MDN
- Приоритет операторов
- Expressions versus statements in JavaScript

Взгляните на эту простую строчку кода:

```
const x = 5;
```

Вы точно знаете, что в ней происходит, верно? Создадим новую константу x, зададим ей значение 5. Ничего особенного тут нет.

Вот другая строчка кода:

```
const y = getAnswer();
```

Создадим новую константу y, зададим ей то значение, которое возвращает функция getAnswer. Теперь представьте, что getAnswer — это на самом деле невероятно сложная функция с миллионом строчек кода и потребуется 12 лет для её вычисления.

Насколько отличаются эти строчки? Оказывается, в информатике более важный и полезный воспрос: "насколько они схожи?".

И ответ, конечно — "всё относительно".

Если вы рассуждаете о том, что действительно, буквально, происходит — они вообще не похожи друг на друга. Одна устанавливает числовое значение, другая вызывает какую-то функцию. А мы уже хорошо понимаем, что это разные вещи. Мы знакомы с функциями, аргументами и всем, что с связано с функциями.

Но иногда полезно оперировать другими понятиями, на другом уровне. Конечно, бегать и управлять самолётом — это очень разные виды активности, но на определённом уровне они подобны — и та и другая подразумевают передвижение из одной точки в другую.

Эти две строчки подобны, потому что справа от знака равно в обоих случаях находится выражение. Выражение — это фрагмент кода, который превращается в значение. Другими словами — становится значением. Да, знаю, 5 — это уже значение, но для интерпретатора JavaScript это выражение, которое превращается в значение 5. Другое выражение, которое превращается в значение 5 — это, например, 2 + 3.

Вызов функции getAnswer() — это тоже выражение, потому что функция что-то возвращает. Этот вызов будет заменён на значение, которое она возвращает. Другими словами, вызов функции превратится в значение, а поэтому он является выражением.

Не всё в коде становится значением. Так что не всё в коде — это выражение, хотя большая часть его — именно выражения.

JavaScript различает выражения и инструкции. Инструкция — это команда, действие. Помните условие с if, циклы с while и for — всё это — инструкции, потому что они только производят и контролируют действия, но не становятся значениями.

Это что, какие-то мутные технические термины из учебников? Может так показаться, но в реальности очень важно понимать и видеть разницу между выражениями и инструкциями.

Это помогает правильно понимать процесс вычисления и выполнения программы. Посмотрите на пример:

```
12 + square(7 + 5) + square(square(2));
```
Это выражение состоит из нескольких подвыражений.

Первое — 12 — выражается в 12. Следующее состоит из множества подвыражений:

- 7 выражается в 7
- 5 выражается в в 5
- 7 + 5 выражается в 12
- square(12) выражается в 144

К этому моменту в процессе JavaScript видит такую картинку:

```
12 + 144 + square(square(2));
```
Это еще не конец, остались необработанные выражения. Это будет продолжаться пока всё выражение не превратится в единое значение.

square(square(2)) решается таким способом:

- 2 выражается в 2
- square(2) выражается в 4
- square(4) выражается в 16

Давайте заглянем в мозг интерпретатора снова:

```
12 + 144 + 16;
```
Все внутренние выражения решены, так что теперь сложение происходит в два шага:

```
12 + 144 + 16;
156 + 16;
172;
```
Теперь решено всё выражение.

Кстати, оператор сложения имеет левую ассоциативность. Это значит, что в случае с составными сложениями процесс пойдёт слева направо, вот почему мы вначале видим 12 + 144, а потом 156 + 16.

Вы не можете ставить инструкции там, где должны быть выражения. Например, передача инструкции const как аргумента функции приведёт к ошибке. Как и попытка присвоить инструкциюif переменной. Подобное просто не имеет смысла в этом языке, потому что в таких случаях предполагаются только выражения:

```
console.log(const x);   // error!
let b = if (x > 10) { return 100; };    // error!
```

Зная такие вещи, вы скоро обретёте две важные суперспособности:

1. Вы будете способны замечать, что большая часть кода, даже та функция из миллиона строк на двенадцать лет, это просто горсть безделушек, которые становятся значениями.
2. Вы будете способны замечать, что иногда код просто не будет работать, потому что он не имеет смысла в контексте выражений и инструкций.

Окружение

ВАЖНО

Изначально в начале конспекта и урока речь шла о глобальном и локальном окружении, но объяснялась фактически глобальная и локальная область видимости. В видео эта тема до сих пор называется «глобальное и локальное окружение».

Под лексической областью видимости можно понимать просто механизм поиска значений: смотрим в текущей области, если нет — идём на уровень выше, и так далее. Слово «лексический» означает, что видимость задаётся исключительно текстом программы, исходным кодом. То есть можно смотреть на программу, не запуская её, и понять область видимости в любой точке.

В других языках может быть не лексический механизм, а динамический (dynamic scope).

Область видимости — это широкое понятие, означающее грубо говоря «интерпретатор в разных местах кода видит разные штуки». Лексическая область видимости — это конкретный механизм, одно из правил для области видимости. Этот механизм применяется в JavaScript и большинстве других языков.

Путаница возникает ещё потому, что в разделе про Область видимости мы смотрим на примеры, которые работают по правилу «лексическая область видимости». От этого никуда не деться, так как язык JavaScript работает именно так.

Конспект урока

- **Область видимости** (scope) компонентов — это местоположение, где эти компоненты доступны.
- Компоненты, созданные снаружи функций, инструкций с if, циклов и так далее, находятся в **глобальной области видимости**
- Фигурные скобки { } задают новую **локальную область видимости**

Глобальная против локальной

Локальные константы и переменные не видимы снаружи их области видимости:

```
const multiplier = (num) => {
  const x = 10;
  return num * x;
}

console.log(x);     // ReferenceError: x is not defined
```
Но если x представлен глобально, то он доступен:

```
const x = 55;

const multiplier = (num) => {
  const x = 10;
  return num * x;
}

console.log(x);     // 55
```
Возможен доступ к внешней области видимости:

```
let a = 0;

const changer = () => {
  a++;
}

console.log(a);   // 0
changer();
console.log(a);   // 1
```
Функция фактически производит что-то, только когда она вызывается, а не задаётся, поэтому изначально a это 0, но после того, как вызвана changer, a становится 1.

Лексическая область видимости

JavaScript пытается найти значение в текущем окружении. Но значение не находится и JavaScript выходит наружу, на один уровень за попытку, пока не найдёт значение или не поймет, что значение невозможно найти.

```
let a = 7;
let b = 10;

const multiplier = () => {
  let a = 5;
  return a * b;
}

multiplier(); // 50
```
Здесь, в выражении a * b , функция multiplier использует локальную a (потому что она обнаружена локально), и наружную b (потому что локально b найдена не была).

Замыкания

```
const createPrinter = () => {
  const name = "King";
```

```
  const printName = () => {
    console.log(name);
  }

  return printName;
}

const myPrinter = createPrinter();
myPrinter();   // King
```

myPrinter — это функция, которая была возвращена createPrinter. Несмотря на то, что вызов createPrinter окончен и константы name больше не существует, значение запомнено в myPrinter.

Это **замыкание**: сочетание функции и окружения, где она была заявлена.

Optional reading

- Variables and scoping / Exploring JS
- Scope / Wikipedia
- Closure / Wikipedia
- Closures in JS / MDN (тут в примерах используется ES5, предыдущая версия стандарта языка)

1.10. Окружение

Давайте поговорим об окружении. Наша планета огромна, но мы все делим её. Если вы построите химический завод, неплохо бы изолировать его от окружающего мира, чтобы то, что в нём происходит оставалось внутри. Вы можете сказать, что в этом здании своё окружение, микроклимат, изолированный от внешней окружающей среды.

Ваша программа имеет подобную структуру по похожим причинам. То, что вы создаёте снаружи — снаружи функций, инструкций if, циклов и других блоков кода — находится во внешнем, глобальном окружении.

```
const age = 29;

const multiplier = (num) => {
  const x = 10;
  return num * x;
}

let result = true;
```

Константа age , функция multiplier и переменная result — все во внешнем окружении. Эти компоненты имеют "глобальную область видимости". Область видимости означает "область, где компоненты доступны".

Внутри функции multiplier есть константа x. Поскольку она внутри блока кода, это локальная константа, а не глобальная. Она видна только внутри этой функции, но не снаружи. У неё локальная область видимости.

В функции multiplier есть ещё один компонент из локальной области видимости — аргумент num. Он не задан так же чётко, как константы или переменные, но ведёт себя почти как локальная переменная.

У нас нет доступа к x снаружи, как будто её там не существует:

```
const multiplier = (num) => {
  const x = 10;
  return num * x;
}

console.log(x);    // ReferenceError: x is not defined
```

console.log вызывается в глобальном окружении, а x не задан в этом глобальном окружении. Поэтому мы получаем Reference Error.

Мы можем задать x глобально:

```
const x = 55;

const multiplier = (num) => {
  const x = 10;
  return num * x;
}

console.log(x);    // 55
```

Теперь существует глобальный x и его значение было выведено на экран, но локальный x внутри функции multiplier по прежнему виден только внутри этой функции. Эти два x не имеют ничего общего друг с другом, они находятся в разных окружениях. Они не схлопываются в одно целое, не смотря на то, что у них одно и то же имя.

Любой блок кода окружёный фигурными скобками превращается в локальное окружение. Вот пример с блоком if:

```
let a = 0;

if (a === 0) {
  const local = 10;
}

console.log(local);
```

То же работает для циклов while и for.

Ок, локальное не видимо снаружи. Но глобальное видимо везде. Даже внутри чего-то? Да!

```
let a = 0;

const changer = () => {
  a++;
}

console.log(a);  // 0
changer();
console.log(a);  // 1
```

Эта глобальная переменная a изменилась внутри функции changer. Функция вообще выполняет что-то только когда её вызывают, а не когда её определяют, так что вначале a это 0, но после того как вызывается changer, a становится 1.

Хоть это и заманчиво всё помещать в глобальную область видимости и забыть о сложностях раздельных окружений — это ужасная практика. Глобальные переменные делают ваш код невероятно хрупким. В таком случае что угодно может сломать что угодно. Поэтому избегайте глобальной области видимости, храните вещи там, где им место.

Лексическая область видимости

Взгляните на эту программу:

```
let a = 7;
let b = 10;

const multiplier = () => {
  let a = 5;
  return a * b;
}

multiplier(); // 50
```

Функция multiplier возвращает произведение a и b. a задано внутри, а b — нет.

Пытаясь решить умножение a * b, JavaScript ищет значения a и b. Он начинает искать локально и выходит наружу, по одной области видимости за шаг, пока он не найдёт то, что ему нужно или пока не поймёт, что это невозможно найти.

Поэтому в данном примере JavaScript начинает с поиска a внутри локальной области видимости — внутри функции multiplier. Он находит значение сразу и переходит к b. Невозможно найти значение b в локальной области видимости, поэтому он переходит к наружной области. Тут он находит b — это 10. a * b превращается в 5 * 10, а затем в 50.

Весь этот кусок кода мог бы быть внутри другой функции, и ещё внутри другой функции. И если бы b не нашлась здесь, JavaScript продолжил бы искать b за пределами функции, слой за слоем.

Заметьте, что a = 7 не затронула вычисления, a была найдена внутри, поэтому внешняя a не сыграла роли.

Это называется лексической областью видимости. Область видимости любого компонента определяется местом расположения этого компонента внутри кода. И вложенные блоки имеют доступ к их внешним областям видимости.

Замыкания

Большинство языков программирования имеют что-то вроде области видимости или окружения, и этот механизм позволяет существовать замыканиям. Замыкание — это всего лишь модное название функции, которая запоминает внешние штуки, используемые внутри.

Перед тем, как мы продолжим, давайте вспомним как функции создаются и используются:

```
const f = () => {
  return 0;
}
```

f — довольно бесполезная функция, она всегда возвращает 0. Весь этот набор состоит из двух частей: константы и самой функции.

Важно помнить, что эти два компонента раздельны. Первый — константа с именем f. Её значение могло бы быть числом или строкой. Но в данном случае её значение — функция.

Мы использовали аналогию в предыдущих уроках: константы как листы бумаги — имя на одной стороне, значение на другой. Следовательно, f — лист бумаги с f на одной стороне и описанием запускаемой функции на другой.

Когда вы вызываете эту функцию, вот так:

```
f();
```
... создаётся новый ящик, основываясь на описании на этом листе бумаги.

Ок, вернёмся к замыканиям. Рассмотрим следующий код:

```
const createPrinter = () => {
  const name = "King";

  const printName = () => {
    console.log(name);
  }

  return printName;
}

const myPrinter = createPrinter();
myPrinter();    // King
```
Функция createPrinter создаёт константу name и затем функцию с именем printName. Обе они локальные для функции createPrinter, и доступны только внутри createPrinter.

У самой printName нет локальных компонентов, но у неё есть доступ к области видимости, где она сама находится, внешней области, где задана константа name.

Затем функция createPrinter возвращает функцию printName. Помните, определения функций — это описания запущенных функций, просто фрагменты информации, как числа или строки. Поэтому мы можем вернуть определение функции, как мы возвращаем число.

Во внешней области видимости мы создаём константу myPrinter и задаём ей значение, которое возвращает createPrinter. Он возвращает функцию, так что теперь myPrinter это функция. Вызовите ее, и на экран выведется "King".

Тут есть странная штука: эта константа name была создана внутри функции createPrinter. Функция была вызвана и исполнена. Как мы знаем, когда функция заканчивает работу, она больше не существует. Этот магический ящик исчезает со всеми своими внутренностями.

НО он возвратил другую функцию, и уже она как-то запомнила константу name. Поэтому когда мы вызывали myPrinter, она вывела "King" — запомненное значение, при том, что область видимости, где оно было задано больше не существует.

Функция, которая была возвращена из createPrinter, называется замыканием. Замыкание — это сочетание функции и окружения, где она была задана. Функция "замкнула" в себе некоторую информацию из области видимости.

Это может выглядеть как JavaScript-фокус, но замыкания, когда их используют разумно, могут сделать код приятней, чище и проще для чтения. И сама идея возврата функций тем же способом, которым можно возвращать числа и строки, даёт больше возможностей и гибкости.

2.Массивы

Программирование становится по-настоящему интересным, когда появляется возможность работать с наборами (коллекциями) элементов. Вот лишь некоторые примеры того, где они встречаются:

- Постраничный вывод данных на сайте
- Подсчёт общей суммы в заказе на основании каждой из позиций
- Вывод списка друзей, сообщений, фильмов и тому подобное
- Обработка набора DOM-узлов (HTML, фронтенд разработка)

Любые списки которые окружают нас в реальном или виртуальном мире, являются коллекциями элементов с точки зрения программирования. В JavaScript для их хранения используется *массив* – структура данных, позволяющая работать с набором как с единым целым.

```
// Определение массива друзей
const friends = ['petya', 'vasya', 'ivan'];
```

В отличие от примитивных типов данных, массивы в JavaScript могут изменяться. Причем как по содержимому так и по размеру самого массива. Это сильно влияет на работу с ними и добавляет с одной стороны больше возможностей, а с другой – ответственности. Используя массивы, одну и ту же задачу можно решить множеством разных способов. Только некоторые из них будут хорошими, остальные же, не эффективными, сложными в отладке и анализе.

Именно поэтому массивам посвящено не несколько уроков, а целый и довольно большой курс. В этом курсе рассматривается множество ситуаций, которые традиционно решаются с помощью массивов. Знания, полученные в этом курсе, станут тем фундаментом, на котором основана вся дальнейшая разработка. Основные темы этого курса:

- Манипуляции с массивами
- Обработка массивов в циклах
- Работа с вложенными массивами используя вложенные циклы
- Сортировка массивов
- Работа со строками через массивы

Помимо массивов, мы коснемся темы алгоритмов и структур данных. Познакомитесь с понятием алгоритмической сложности, узнаем, как реализовывать некоторые типичные алгоритмы, которые часто спрашивают на собеседованиях. Знание этих тем, хотя бы на базовом уровне, критично для написания эффективного кода.

Дополнительные материалы

1. Массивы

2.1.Синтаксис

Массивом в программировании представляют любые упорядоченные наборы (или коллекции) элементов, будь то курсы на Хекслете, студенты в группе или друзья в вашей любимой социальной сети. Задача массива представить такие коллекции в виде единой структуры, которая позволяет работать с ними как с единым целым.

Определение массива

```
// Создание пустого массива
const items = [];

// Создание массива с тремя элементами
const animals = ['cats', 'dogs', 'birds'];
```

В примере происходит определение массива `['cats', 'dogs', 'birds']`, который затем присваивается константе `animals`.

Обратите внимание на именование констант содержащих массивы. Они во множественном числе. Это подчеркивает природу константы и делает код проще для анализа.

Получение данных

Элементы в массиве упорядочены слева направо. Каждый элемент имеет порядковый номер, называемый **индексом**. Индексация массива начинается с нуля. То есть первый элемент массива доступен по индексу `0`, второй — по индексу `1` и так далее... Для извлечения элемента из массива по индексу используется особый синтаксис:

```
const animals = ['cats', 'dogs', 'birds'];
animals[0]; // 'cats'
animals[1]; // 'dogs'
// Последний индекс в массиве всегда меньше размера массива на единицу.
// В этом массиве три элемента, но последний индекс равен двум
animals[2]; // 'birds'
```

Узнать размер массива можно, обратившись к его свойству `length`.

```
const animals = ['cats', 'dogs', 'birds'];
// У массивов много других свойств и методов, с которыми мы познакомимся в процессе прохождения курсов.
animals.length; // 3
```

В реальных задачах индекс часто вычисляется динамически, поэтому обращение к конкретному элементу происходит с использованием переменных:

```
let i = 1;
const animals = ['cats', 'dogs', 'birds'];
animals[i]; // 'dogs'
```

И даже так:

```
let i = 1;
let j = 1;
const animals = ['cats', 'dogs', 'birds'];
animals[i + j]; // 'birds'
```

Такой вызов возможен по одной простой причине — внутри скобок ожидается *выражение*. А там, где ожидается выражение, можно подставлять всё, что вычисляется. В том числе вызовы функций:

```
const getIndexOfSecondElement = () => 1;
```

```
const animals = ['cats', 'dogs', 'birds'];
animals[getIndexOfSecondElement()]; // 'dogs'
```

Довольно часто, в задачах с использованием массивов, нужно взять последний элемент. Для этого вычисляется последний индекс массива по формуле *размер_массива - 1*, по которому и можно обратиться к последнему элементу:

```
const animals = ['cats', 'dogs', 'birds'];
animals[animals.length - 1]; // 'birds'
```

Дополнительные материалы

1. Документация

2.2.Модификация

Примитивные типы данных, с которыми мы работали до сих пор, невозможно изменить. Любые функции и методы над ними возвращают новые значения, но не могут ничего сделать со старым.

```
const name = 'Hexlet';
name.toUpperCase(); // 'HEXLET'
// Значение name не поменялось
console.log(name); // 'Hexlet'
```

С массивами это правило не работает. Массивы могут меняться: увеличиваться, уменьшаться, изменять значения по индексам. Ниже мы разберем все эти операции.

Изменение элементов массива

Синтаксис изменения элемента массива, практически такой же, как и при обращении к элементу массива. Разница лишь в наличии присваивания:

```
const animals = ['cats', 'dogs', 'birds'];
// Меняется первый элемент массива
animals[0] = 'horses';
console.log(animals); // => ['horses', 'dogs', 'birds']
```

Самое неожиданное в данном коде – изменение константы. Константы в JavaScript не совсем то, как мы себе это представляли раньше. Константы хранят ссылку на данные (подробнее об этом в следующих уроках), а не сами данные. Это значит что менять данные можно, но нельзя заменить ссылку. Технически это значит, что мы не можем заменить все значение константы целиком:

```
const animals = ['cats', 'dogs', 'birds'];
// Меняем данные, а сам массив остался тем же
// Такой код работает
animals[2] = 'fish';

// Произойдет ошибка, так как здесь идет замена константы
animals = ['fish', 'cats'];
// Uncaught TypeError: Assignment to constant variable.
```

Добавление элемента в массив

Метод push() добавляет элемент в *конец* массива:

```
const animals = ['cats', 'dogs', 'birds'];
animals.push('horses');

// массив animals изменён — стал больше
console.log(animals); // => ['cats', 'dogs', 'birds', 'horses']

// строка 'horses' была добавлена в конец массива (индекс = 3)
console.log(animals[3]); // => 'horses'
```

Метод array.unshift() добавляет элемент в *начало* массива:

```
const animals = ['cats', 'dogs', 'birds'];
animals.unshift('horses');

// массив animals изменён — стал больше
console.log(animals); // => ['horses', 'cats', 'dogs', 'birds']

// строка 'horses' была добавлена в начало массива (индекс = 0)
console.log(animals[0]); // => 'horses'
```

Иногда индекс добавления известен сразу и в таком случае добавление работает так же как и изменение:

```
const animals = ['cats', 'dogs', 'birds'];
animals[3] = 'horses';
console.log(animals); // => ['cats', 'dogs', 'birds', 'horses']
```

Удаление элемента из массива

Удалить элемент из массива можно с помощью специальной конструкции delete: delete arr[index].

Пример:

```
const animals = ['cats', 'dogs', 'birds'];
delete animals[1]; // удаляем элемент под индексом 1
console.log(animals); // => ['cats', <1 empty item>, 'birds']
```

Этот способ обладает рядом недостатков, завязанных на особенности внутренней организации языка JavaScript. Например, после такого удаления, можно с удивлением заметить, что размер массива не изменился:

```
animals.length; // 3
```

Есть и другие особенности и последствия использования этого оператора, в которые сейчас не будем углубляться. Здесь мы его привели лишь для примера и не рекомендуем использовать при написании кода. В общем случае уменьшение размера массива — нежелательная операция. Подробнее об этом поговорим в одном из следующих уроков.

2.3. Проверка существования значения

При работе с массивами, часто допускается ситуация, называемая "выход за границу массива". Она возникает при обращении к несуществующему индексу:

```
const animals = ['cats', 'dogs', 'birds'];
// Элемента с индексом 5 не существует
```

```
animals[5]; // undefined
```

В разных языках программирования поведение в случае выхода за границу реализовано совершенно по-разному. Иногда возникает ошибка, иногда нет, а иногда подобный выход возвращает случайные данные из соседнего блока памяти, как в Си, что может привести к катастрофе.

В JavaScript свой путь. Здесь дана большая свобода, допускающая почти любые вольности. Обращение по несуществующему индексу возвращает значение undefined. При этом никаких ошибок не возникает, это рассматривается как нормальная ситуация:

```
const animals = ['cats', 'dogs', 'birds'];

// Выход за границы массива
animals[5]; // undefined
animals[4]; // undefined
animals[3]; // undefined

// Ура, мы попали в границы массива :)
animals[2]; // 'birds'
```

В подавляющем большинстве ситуаций, выход за границу массива является нежелательным поведением. Он происходит из-за логических ошибок в программе. Программа, при этом, продолжает работать и, даже, иногда выдавать правильный результат. Со временем вы научитесь видеть такие ситуации и достаточно быстро исправлять их. Но даже опытные программисты регулярно ошибаются при обращении с массивами.

2.4.Цикл for

Работа с массивами, почти всегда, завязана на одновременную обработку всех его элементов. Это нужно при выводе списков на экран, при выполнении различных расчетов или проверке данных. Во всех этих случаях нужен механизм для перебора элементов массива. Самый простой способ сделать это – использовать цикл.

Обход

Циклы напрямую с массивами не связаны, но у циклов есть счетчик, который может выступать в качестве индекса массива. Поэтому соединить их не составляет никакого труда:

```
// Создаем массив
const userNames = ['petya', 'vasya', 'evgeny'];

// Определяем цикл
// Начальное значение счетчика i = 0 – вычисляется один раз перед началом выполнения
// Условие остановки i < userNames.length – выполняется перед каждой итерацией
// Изменение счетчика i += 1 – выполняется после каждой итерации
for (let i = 0; i < userNames.length; i += 1) {
  // Этот код выполняется для каждого элемента
  console.log(userNames[i]);
}

// => 'petya'
// => 'vasya'
// => 'evgeny'
```

https://repl.it/@hexlet/js-arrays-for-print

В данном коде создаём массив из трёх элементов — имён. Далее в цикле обходим массив и выводим на экран все имена так, что каждое имя оказывается на новой строке (console.log автоматически делает перевод строки).

Рассмотрим этот этап подробнее. При обходе массива циклом for счётчик, как правило, играет роль индекса в массиве. Он инициализируется нулём и увеличивается до userNames.length - 1, что соответствует индексу последнего элемента. Именно поэтому мы используем строгое сравнение < (*меньше*) в условном выражении i < userNames.length, а не <= (*меньше либо равно*).

А что, если нам нужно вывести значения в обратном порядке? Для этого есть два способа. Один — идти в прямом порядке, то есть от нулевого индекса до последнего, и каждый раз вычислять нужный индекс по такой формуле размер массива - 1 - текущее значение счётчика.

```
const userNames = ['petya', 'vasya', 'evgeny'];

for (let i = 0; i < userNames.length; i += 1) {
  const index = (userNames.length - 1) - i;
  console.log(userNames[index]);
}
```

Другой способ подразумевает обход в обратном порядке, от верхней границы до нижней, то есть от последнего индекса массива к первому (нулю, так как индексирование начинается с нуля). В такой ситуации код меняется на следующий:

```
const userNames = ['petya', 'vasya', 'evgeny'];

// Начальное значение i соответствует последнему индексу в массиве
for (let i = userNames.length - 1; i >= 0; i -= 1) {
  console.log(userNames[i]);
}
```

При таком обходе проверка остановки должна быть именно на >=, иначе элемент с индексом 0 не попадет в цикл.

Изменение

Во время обхода массива его можно не только читать, но и модифицировать. Предположим, что перед нами стоит задача нормализации списка электронных адресов — например, приведение их к нижнему регистру. Тогда код будет выглядеть так:

```
const emails = ['VASYA@gmAil.com', 'iGoR@yandex.RU', 'netiD@hot.CoM'];

console.log(emails);
// => [ 'VASYA@gmAil.com', 'iGoR@yandex.RU', 'netiD@hot.CoM' ]

for (let i = 0; i < emails.length; i += 1) {
  const email = emails[i];

  // toLowerCase() — стандартный метод js,
  // преобразующий строку в нижний регистр
  const normalizedEmail = email.toLowerCase();
  // Заменяем значение
  emails[i] = normalizedEmail;
}
```

```
console.log(emails);
// => [ 'vasya@gmail.com', 'igor@yandex.ru', 'netid@hot.com' ]
```
https://repl.it/@hexlet/js-arrays-for-update

Ключевая строчка: emails[i] = normalizedEmail;. В ней происходит перезапись элемента под индексом i.

Резюме

Цикл for можно комбинировать с массивами в любых вариантах. Массив не обязательно перебирать полностью и от начала до конца. Можно например смотреть только каждый второй элемент или двигаться до половины. Все это зависит от конкретной задачи.

Точно так же массивы сочетаются с while. Единственное что нужно массивам – индекс.

Дополнительные материалы

1. Официальная документация

2.5.Ссылки

Переменные (и константы) в JavaScript могут хранить два вида данных: примитивные и ссылочные. К примитивным относятся все примитивные типы: числа, строки, булеан и так далее. К ссылочным – объекты. Объекты, в общем смысле, изучаются только в следующем курсе, но массив это тоже объект внутри:

```
typeof []; // 'object'
```
В чем разница между ссылочными и примитивными типами данных и почему об этом нужно знать?

С точки зрения прикладного программиста, разница проявляется при изменении данных, их передаче и возврате из функций. Мы уже знаем, что массив можно менять даже если он записан в константу. Здесь как раз и проявляется ссылочная природа. Константа хранит ссылку на массив, а не сам массив и эта ссылка не меняется. А вот массив поменяться может. С примитивными типами такой трюк не пройдет.

Другой способ убедиться в том, что массивы ссылки – создать несколько констант содержащих один массив и посмотреть как они меняются:

```
const items = [1, 2];
// Ссылаются на один и тот же массив
const items2 = items;
items2.push(3);

console.log(items2); // => [1, 2, 3]
console.log(items); // => [1, 2, 3]
items2 === items; // true
```
Сравнение массивов тоже происходит по ссылке. Это может быть очень неожиданно с непривычки. Одинаковые массивы по структуре имеют разные ссылки и не равны друг другу:

```
[1,2,3] === [1,2,3]; // false

// Проверить два массива на равенство их значений можно с помощью lodash
```

Более того, если передать массив в какую-то функцию, которая его изменяет, то массив тоже изменится. Ведь в функцию передается именно ссылка на массив. Посмотрите на пример:

```
const f = (coll) => coll.push('wow');

const items = ['one'];
f(items);
console.log(items); // => ['one', 'wow']
f(items);
console.log(items); // => ['one', 'wow', 'wow']
```

Проектирование функций

Проектируя функции, работающие с массивами, есть два пути: менять исходный массив или формировать внутри новый и возвращать его наружу. Какой лучше? В подавляющем большинстве стоит предпочитать второй. Это безопасно. Функции, возвращающие новые значения, удобнее в работе, а поведение программы становится в целом более предсказуемым, так как отсутствуют неконтролируемые изменения данных.

Изменение массива может повлечь за собой неожиданные эффекты. Представьте себе функцию last, которая извлекает последний элемент из массива (такая функция есть в lodash). Она могла бы быть написана так:

```
// Метод .pop извлекает последний элемент из массива
// Он изменяет массив удаляя оттуда этот элемент
const last = (coll) => coll.pop();
```

Где-то в коде, вы просто хотели посмотреть последний элемент. А в дополнение к этому, функция для извлечения этого элемента, взяла и удалила его оттуда. Это поведение очень неожиданно для подобной функции. Оно противоречит большому количеству принципов построения хорошего кода (например cqs, этот принцип рассматривается в курсе по функциям). Правильная реализация данной функции выглядит так:

```
const last = (coll) => coll[coll.length - 1]
```

В каких же случаях стоит менять сам массив? Есть ровно одна причина по которой так делают – производительность. Именно поэтому некоторые встроенные методы массивов меняют их, например reverse() или sort():

```
const items = [3, 8, 1];

// Нет присвоения результата, массив изменяется напрямую
items.sort();
console.log(items); // => [1, 3, 8]

items.reverse();
console.log(items); // => [8, 3, 1]
```

https://repl.it/@hexlet/js-arrays-references-sort

Обычно в документации каждой функции отдельно подчёркивают, изменяет ли она исходный массив или возвращает результатом новый массив, не модифицируя исходный. Например, метод concat(), в отличие от sort(), возвращает новый массив, о чём написано в документации.

Несмотря на то, что подход, меняющий массивы напрямую, сложнее в отладке, его используют в некоторых языках для увеличения эффективности работы. Если массив достаточно большой, то

полное копирование окажется дорогой операцией. В реальной жизни (веб-разработчика) это почти никогда не является проблемой, но знать об этом полезно.

Дополнительные материалы

1. Продуманная оптимизация

2.6.Агрегация

Распространённый вариант использования циклов с массивами — **агрегация**. Агрегацией называются любые вычисления, которые, как правило, строятся на основе всего набора данных, например, поиск максимального, среднего, суммы и так далее. Процесс агрегации не требует знания нового синтаксиса, но влияет на алгоритм решения задач. Поэтому имеет смысл рассмотреть его отдельно. Начнем с поиска максимального.

```
const calculateMax = (coll) => {
  // Если коллекция пустая, то у нее не может быть максимального
  // В подобных ситуациях принято возвращать null
  if (coll.length === 0) {
    return null;
  }

  // Сравнение элементов нужно начать с какого-то первого элемента
  let max = coll[0]; // Принимаем за максимальное первый элемент
  // Обход начинаем со второго элемента
  for (let i = 1; i < coll.length; i += 1) {
    const currentElement = coll[i];
    // Если текущий элемент больше максимального,
    // то он становится максимальным
    if (currentElement > max) {
      max = currentElement;
    }
  }

  // Не забываем вернуть максимальное число
  return max;
};

console.log(calculateMax([])); // => null
console.log(calculateMax([3, 2, -10, 38, 0])); // => 38
```

https://repl.it/@hexlet/js-arrays-aggregation-max

Почему это пример агрегации? Здесь мы видим *вычисление*, которое включает в себя сравнение всех элементов для поиска одного, которое станет результатом этой операции.

Обратите внимание, что начальным значением max взят первый элемент, а не, скажем, число 0. Ведь может оказаться так, что все числа в массиве меньше 0, и тогда мы получим неверный ответ.

Теперь рассмотрим поиск суммы:

```
const calculateSum = (coll) => {
  // Начальное значение суммы
  let sum = 0;
  for (let i = 0; i < coll.length; i += 1) {
    // Поочередно складываем все элементы
    sum += coll[i];
  }

  return sum;
};

// Сумма элементов всегда возвращает какое-то число
// Если массив пустой, то сумма его элементов 0
console.log(calculateSum([])); // => 0

console.log(calculateSum([3, 2, -10, 38, 0])); // => 33
// Процесс вычислений
let sum = 0;
sum = sum + 3; // 3
sum = sum + 2; // 5
sum = sum + -10; // -5
sum = sum + 38; // 33
sum = sum + 0; // 33
```

https://repl.it/@hexlet/js-arrays-aggregation-sum

Алгоритм поиска суммы значительно проще, но обладает парой важных нюансов.

Чему равна сумма элементов пустого массива? С точки зрения математики такая сумма равна 0. Что в принципе совпадает со здравым смыслом. Если у нас нет яблок, значит у нас есть 0 яблок (количество яблок равно нулю). Функции в программировании работают по этой логике.

Второй момент связан с начальным элементом суммы. У переменной sum есть начальное значение равное 0. Зачем вообще задавать значение? Любая повторяющаяся операция начинается с какого-то значения. Нельзя просто так объявить переменную и начать с ней работать внутри цикла. Это приведет к неверному результату:

```
// начальное значение не задано
// js автоматически делает его равным undefined
let sum;

// первая итерация цикла
sum = sum + 2; // ?
```

В результате такого вызова, внутри sum окажется NaN, то есть не-число. Оно возникает из-за попытки сложить 2 и undefined. Значит какое-то значение все же нужно. Почему в коде выше выбран 0? Очень легко проверить, что все остальные варианты приведут к неверному результату. Если начальное значение будет равно 1, то результат получится на 1 больше чем нужно.

В математике существует понятие **нейтральный элемент операции** (у каждой операции свой элемент). Это понятие имеет очень простой смысл. Операция с этим элементом не изменяет то значение, над которым проводится операция. В сложении любое число плюс ноль дает само число. При вычитании тоже самое. Даже у конкатенации есть нейтральный элемент – это пустая строка: '' + 'one' будет 'one'.

Агрегация далеко не всегда означает, что коллекция элементов сводится к некоторому простому значению. Результатом агрегации может быть сколь угодно сложная структура, например, массив. Подобные примеры часто встречаются в реальной жизни. Самый простой пример – это список уникальных слов в тексте.

2.7.Цикл for...of

for относится к низкоуровневым циклам. Он требует задания счетчика, правил его изменения и условия остановки. Было бы значительно удобнее обходить элементы коллекции напрямую, без счетчика. Многие языки программирования решают это введением специального вида цикла. В JavaScript тоже есть такой: for...of.

```
const userNames = ['petya', 'vasya', 'evgeny'];

// name на каждой итерации свой собственный (локальный), поэтому используется const
for (const name of userNames) {
  console.log(name);
}
// => "petya"
// => "vasya"
// => "evgeny"
```

https://repl.it/@hexlet/js-arrays-for-of-example

Как видно из примера, код использующий for...of получается значительно чище, чем с использованием цикла for. for...of знает о том как перебирать элементы и знает о том когда они закончатся.

Этот цикл отлично подходит для задач агрегации:

```
const calculateSum = (coll) => {
  let sum = 0;
  for (const value of coll) {
    sum += value;
  }

  return sum;
};
```

for...of — это больше, чем просто цикл для массивов. Для полного понимания принципов его работы, нужно разбираться в темах, которые мы еще не проходили, среди них объекты, упаковка/распаковка и итераторы. Если по-простому, то разные данные в JavaScript могут притворяться коллекциями элементов. Самый простой пример — это строка: for...of перебирает строку посимвольно.

```
const greeting = 'Hello';
// В этот момент со строкой происходит магия, которая разбирается в курсе ООП
for (const symbol of greeting) {
  console.log(symbol);
}
// => "H"
// => "e"
// => "l"
// => "l"
// => "o"
```

Однако не следует путать строку с массивом. Несмотря на внешнюю схожесть доступа к элементам строки по индексу, строка массивом не является.

Применимость

В большинстве задач использующих цикл, предпочтительнее for...of. Иногда его бывает недостаточно, и требуется ручное управление обходом. В таких случаях можно возвращаться к использованию for. Например, когда нужно идти не по каждому элементу массива, а через один:

```
for (let i = 0; i < items.length; i += 2) {
  // какой-то код
}
```

Иногда нужно обходить массив в обратном порядке. for...of здесь бессилен и снова нужен for:

```
for (let i = items.length - 1; i >= 0; i -= 1) {
  // какой-то код
}
```

Другие задачи вообще с массивами напрямую не связаны. К последним относятся ситуации, когда нужно перебирать числа в определённом диапазоне. В этом случае нет массива, по которому можно было бы пройтись с помощью for...of.

```
for (let i = 5; i < 10; i += 1) {
  // какой-то код
}
```

Ну и наконец, встречаются задачи в которых нужно во время обхода менять исходный массив:

```
for (let i = 0; i < items.length; i += 1) {
  items[i] = /* что-то делаем */
}
```

Если заглядывать совсем в будущее и то как пишется реальный код на JavaScript, то там появляются функции высшего порядка. То есть на практике циклы, можно сказать, не нужны за редким исключением. Однако, невозможно перепрыгнуть работу с циклами, так как это база. А функции высшего порядка требуют знание тем, которые за один присест не изучаются.

Дополнительные материалы

1. Официальная документация

2.8.Удаление элементов массива

В JavaScript не существует настоящего способа удалить элемент из массива. Инструкция delete лишь очищает значение, но сама ячейка никуда не девается:

```
const numbers = [1, 10];
delete numbers[0];
console.log(numbers);
// => [ <1 empty item>, 10 ]
```

При этом задача удаления возникает регулярно. Причем, обычно, удаляется не один элемент, а набор элементов по определенным правилам. Например довольно распространена операция *compact* – удаление null значений из массива. Как правильно ее реализовать?

В подавляющем большинстве ситуаций, изменение массива должно трансформироваться в создание нового массива, в котором отсутствуют удаляемые элементы. Ниже пример реализации функции compact()

```
const compact = (coll) => {
  // Инициализация результата
  // Для пустой входной коллекции результатом будет пустой массив
  const result = [];

  for (const item of coll) {
    if (item !== null) {
      result.push(item);
    }
  }

  return result;
};

console.log(compact([0, 1, false, null, true, 'wow', null]));
// => [ 0, 1, false, true, 'wow' ]
console.log(compact([]));
// => []
```

https://repl.it/@hexlet/js-arrays-removing-compact

Главное, на что нужно обратить внимание, — не происходит модификаций исходного массива coll. Вместо этого создаётся новый массив result, который наполняется только подходящими под условие значениями. Именно так нужно воспринимать фразу "удалить из массива что-то". Код, использующий новый массив, меньше подвержен ошибкам, проще в отладке и оставляет больше возможностей для анализа. Вы всегда можете посмотреть исходный массив, если что-то пошло не так. Вы всегда можете наблюдать за процессом наполнения результирующего массива, что позволит чётко отследить правильность поставленных условий.

По сути, код выше — пример агрегации. Только в отличие от предыдущих примеров, в которых результатом был примитивный тип, здесь результат — массив. Это совершенно нормально. Как вы увидите в дальнейшем, результат может быть и более сложной структурой. Сама операция прореживания (удаления элементов по определенным условиям) массива обычно называется **фильтрацией**.

2.9.Управляющие инструкции

В циклах JavaScript доступны для использования две инструкции, влияющие на их поведение: break и continue. Их использование не является необходимым, но все же они встречаются на практике и поэтому про них нужно знать.

Break

Инструкция break производит *выход из цикла*. Не из функции, а из цикла. Встретив её, интерпретатор перестаёт выполнять текущий цикл и переходит к инструкциям, идущими сразу за циклом.

```
const coll = ['one', 'two', 'three', 'four', 'stop', 'five'];

for (const item of coll) {
  if (item === 'stop') {
    break;
```

```
  }
  console.log(item);
}
```

То же самое легко получить, используя цикл while. Этот цикл семантически лучше подходит для такой задачи, так как подразумевает неполный перебор:

```
const coll = ['one', 'two', 'three', 'four', 'stop', 'five'];

let i = 0;
while (coll[i] !== 'stop') {
  console.log(coll[i]);
  i += 1;
}
```

Continue

Инструкция continue позволяет пропустить итерацию цикла. Ниже пример с функцией myCompact, которая удаляет null элементы из массива:

```
const myCompact = (coll) => {
  const result = [];

  for (const item of coll) {
    if (item === null) {
      continue;
    }

    result.push(item);
  }

  return result;
};
```

Код без continue получается проще:

```
const myCompact = (coll) => {
  const result = [];

  for (const item of coll) {
    if (item !== null) {
      result.push(item);
    }
  }

  return result;
};
```

Резюме

break и continue призваны добавить гибкости в управление процессом обхода. На практике же, всегда проще построить код без них и, скорее, он будет даже проще. По возможности избегайте этих конструкций.

Дополнительные материалы

1. Break
2. Continue

2.10.Вложенные массивы

Значением массива может быть всё, что угодно, в том числе другой массив. Создать массив в массиве можно так:

```
const arr1 = [[3]];
arr1.length; // 1

const arr2 = [1, [3, 2], [3, [4]]];
arr2.length; // 3
```

https://repl.it/@hexlet/js-arrays-nested-arrays-multidimensional-array

Каждый элемент, являющийся массивом, рассматривается как единое целое. Это видно по размеру второго массива. Синтаксис JavaScript позволяет размещать элементы создаваемого массива построчно, перепишем для наглядности создание второго массива:

```
const arr2 = [
  1,       // первый элемент (число)
  [3, 2],   // второй элемент (массив)
  [3, [4]], // третий элемент (массив)
];
arr2.length; // 3
```

Вложенность никак не ограничивается. Можно создавать массив массивов массивов и так далее.

Обращение ко вложенным массивам выглядит немного необычно, хотя и логично:

```
const arr1 = [[3]];
arr1[0][0]; // 3

const arr2 = [1, [3, 2], [3, [4]]];
arr2[2][1][0]; // 4
```

https://repl.it/@hexlet/js-arrays-nested-arrays-access

Возможно, с непривычки вы не всегда сразу точно увидите, как добраться до нужного элемента, но это всего лишь вопрос тренировок:

```
const arr2 = [
  1,
  [3, 2],
  [3, [4]],
];

arr2[2];      // [3, [4]]
arr2[2][1];    // [4]
arr2[2][1][0]; // 4
```

Изменение и добавление массивов в массив:

```
const arr1 = [[3]];
```

```
arr1[0] = [2, 10];
arr1.push([3, 4, 5]);

// [[2, 10], [3, 4, 5]]
```
https://repl.it/@hexlet/js-arrays-nested-arrays-change-and-adding-elements

Вложенные массивы можно изменять напрямую, просто обратившись к нужному элементу:

```
const arr1 = [[3]];
arr1[0][0] = 5;
// [[5]]
```
То же самое касается и добавления нового элемента:

```
const arr1 = [[3]];
arr1[0].push(10);
// [[3, 10]]
```
Для чего же могут понадобиться вложенные массивы? Таких примеров довольно много: начиная от математических концепций, например, матриц, заканчивая представлением игровых полей. Помните игру крестики-нолики? Это как раз тот самый случай:

Разберём такую задачку. Дано игровое поле для крестиков-ноликов. Нужно написать функцию, которая проверяет, есть ли на этом поле хотя бы один крестик или нолик, в зависимости от того, что попросят проверить.

```
// Инициализируем поле
const field = [
  [null, null, null],
  [null, null, null],
  [null, null, null],
];

// Делаем ход:
field[1][2] = 'x';
// [
//     [null, null, null],
//     [null, null, 'x'],
//     [null, null, null],
// ]
```
Теперь реализуем функцию, которая выполняет проверку:

```
const didPlayerMove = (field, symbol) => {
  // Обходим поле. Каждый элемент — это строчка в игровом поле.
  for (const row of field) {
    // метод includes проверяет присутствует ли элемент в массиве,
    if (row.includes(symbol)) { // Если присутствует, значит мы нашли то, что искали.
      return true;
    }
  }

  // Если поле было просмотрено, но ничего не нашли,
  // значит ходов не было.
  return false;
```

```
}
```
Проверим:

```
didPlayerMove(field, 'x'); // true
didPlayerMove(field, 'o'); // false
```
https://repl.it/@hexlet/js-arrays-nested-arrays-tic-tac-toe

Дополнительные материалы

1. Метод массива includes

2.11. Генерация строки в цикле.

Генерация строки в цикле

Генерация строк в циклах — задача, часто возникающая на практике. Типичный пример — функция, помогающая генерировать HTML-списки. Она принимает на вход коллекцию элементов и возвращает HTML-список из них:

```
const coll = ['milk', 'butter'];

buildHTMLList(coll);
// <ul><li>milk</li><li>butter</li></ul>
```
Как можно решить эту задачу "в лоб":

1. Создать переменную result и записать в нее <ul>.
2. Пройтись циклом по элементам коллекции и дописать в результирующую строку очередной элемент <li>.
3. Добавить в конце и вернуть result из функции.

```
const buildHTMLList = (coll) => {
  let result = '<ul>';
  for (const item of coll) {
    result = `${result}<li>${item}</li>`;
    // либо так: result += `<li>${item}</li>`;
  }
  result = `${result}</ul>`;

  return result;
}
```
Такой способ вполне рабочий, но для большинства языков программирования максимально неэффективный. Дело в том, что конкатенация и интерполяция порождают новую строчку вместо старой, — и подобная ситуация повторяется на каждой итерации. Причём строка становится всё больше и больше. Копирование строк приводит к серьёзному расходу памяти и может влиять на производительность. Конечно, для большинства приложений данная проблема неактуальна из-за малого объёма прогоняемых данных, но более эффективный подход не сложнее в реализации и обладает рядом плюсов. Поэтому стоит сразу приучить себя работать правильно.

Правильно, в случае с динамическими языками, – формировать массив, который затем с помощью метода join() превратить в строку:

```
const buildHTMLList = (coll) => {
```

```
  const parts = [];
  for (const item of coll) {
    parts.push(`<li>${item}</li>`);
  }

  // Метод join объединяет элементы массива в строку
  // В качестве разделителя между значениями
  // используется то, что передано первым параметром
  const innerValue = parts.join('');
  const result = `<ul>${innerValue}</ul>`;
  return result;
}
```

Размер кода практически не изменился, но способ формирования результата стал другим. Вместо строки, сначала собирается массив, который затем превращается в строку с помощью метода join(). Помимо эффективности, у такого подхода есть дополнительные плюсы:

- Такой код проще отлаживать. Данные, представленные массивом, легче вычленять визуально и программно.
- Массив это структура, с ним можно производить дополнительные манипуляции. С готовой строкой уже ничего особо не сделать.

Регулируя разделитель, строки можно объединять разными способами. Например, через запятую с пробелом:

```
const parts = ['JavaScript', 'PHP', 'Python'];
const output = parts.join(', ');

console.log(output); // => JavaScript, PHP, Python
```

Если каждое слово надо вывести на новой строчке, то в качестве разделителя используем символ перевода строки '\n':

```
const parts = ['JavaScript', 'PHP', 'Python'];

 // теперь каждое слово будет начинаться с новой строки
const output = parts.join('\n');

console.log(output); // =>
// JavaScript
// PHP
// Python
```

Последний пример особенно важен. Новички часто допускают ошибку и добавляют перевод строки в момент формирования массива, а не в join(). Посмотрите на пример с нашей функцией buildHTMLList:

```
// Неправильно

const parts = [];
for (const item of coll) {
  parts.push(`\n<li>${item}</li>`);
}
const innerValue = parts.join(''); // разделителя нет

// Правильно
```

```
const parts = [];
for (const item of coll) {
  parts.push(`<li>${item}</li>`);
}
const innerValue = parts.join('\n'); // перевод строки
```

Дополнительные материалы

1. Джоэль Спольски. Назад к основам (английская версия)

2.12. Обработка строк через преобразование в массив.

На собеседованиях часто задают подобные задачки:

Дана строка текста. Нужно сделать заглавной первую букву каждого слова в тексте. Для простоты считаем что мы работаем с текстом, который не содержит знаков препинаний.

```
const text = 'hello hexlet';
capitalizeWords(text); // 'Hello Hexlet'
```

Решить её можно многими способами. Чем больше называет человек — тем лучше. К ним относятся:

1. Посимвольный перебор строки.
2. Через преобразование в массив.
3. Регулярные выражения. Рассматриваются в отдельном курсе.

Разберем решение через массив. Для этого воспользуемся методом строки split(), который разделяет строку на части.

```
const capitalizeWords = (sentence) => {
  // определяем разделитель — пробел
  const separator = ' ';
  // split разделяет строку по указанному разделителю
  const words = sentence.split(separator);
  // ...
};
```

Следующем шагом нужно обойти массив получившихся слов и преобразовать первую букву каждого слова к верхнему регистру. Строки в JavaScript не имеют встроенного метода для этого, поэтому воспользуемся библиотекой *lodash*. В нее входит функция upperFirst(), которая делает то, что нам нужно.

```
import _ from 'lodash';

const capitalizeWords = (sentence) => {
  const separator = ' ';
  const words = sentence.split(separator);
  // Формируем массив обработанных слов
  const capitalizedWords = [];
  for (const word of words) {
```

```
    capitalizedWords.push(_.upperFirst(word));
  }

  // Соединяем обработанные слова обратно в предложение
  return capitalizedWords.join(separator);
};
```

https://repl.it/@hexlet/js-arrays-strings-capitalize-words

Последнее действие обратно первому. Нужно соединить слова и вернуть получившуюся строку наружу.

Обратите внимание на интересную деталь. Преобразование к верхнему регистру происходит не в исходном массиве words, а в новом. Почему? Такой код значительно упрощает отладку. Если алгоритм работает неверно, то в всегда можно посмотреть что находится в массиве words и что в массиве capitalizedWords. Изменяя массив words мы бы потеряли эту информацию.

2.13. Вложенные циклы.

Во многих языках программирования есть очень полезная функция *flatten*. В определённых задачах она сильно упрощает жизнь и сокращает количество кода. Эта функция принимает на вход массив и выпрямляет его: если элементами массива являются массивы, то *flatten* сводит всё к одному массиву, раскрывая каждый вложенный. В js эта функция реализована как метод flat() у массивов:

```
[[3, 2], 5, 3, [3, 4, 2], 10].flat();
// [3, 2, 5, 3, 3, 4, 2, 10]
```

Реализуем эту функцию самостоятельно. В общем случае эта функция раскрывает массивы на всех уровнях вложенности. Но мы для простоты сделаем вариант функции, в котором происходит раскрытие только до первого уровня. То есть, если элемент основного массива — массив, то он раскрывается без просмотра его внутренностей (там тоже могут быть массивы).

Логика работы функции выглядит так:

```
const flatten = (coll) => {
  const result = [];
  for (const item of coll) {
    // Array.isArray — функция-предикат стандартной библиотеки,
    // которая проверяет, является ли значение массивом.
    if (Array.isArray(item)) {
      // Если значение массив, то проходим по всем его элементам
      // Здесь появился вложенный цикл
      for (const subItem of item) {
        // и добавляем их в результирующий массив
        result.push(subItem);
      }
    } else {
      // Если значение не массив, то сразу добавляем его в результирующий массив
      result.push(item);
    }
  }

  return result;
};
```

```
console.log(flatten([3, 2, [], [3, 4, 2], 3, [123, 3]]));
// => [ 3, 2, 3, 4, 2, 3, 123, 3 ]
```
https://repl.it/@hexlet/js-arrays-nested-loops-flatten

Обратите внимание, что вложенный цикл запускается, только если текущий элемент — массив. Чисто технически во вложенных циклах нет ничего особенного. Их можно вкладывать внутрь любого блока и друг в друга сколько угодно раз. Но прямой связи между внешним и вложенным циклами нет. Внутренний цикл может использовать результаты внешнего, а может и работать по своей собственной логике независимо.

Вложенные циклы коварны. Их наличие может резко увеличить сложность кода, так как появляется множество постоянно изменяющихся переменных. Становится тяжело уследить за происходящими внутри процессами. Кроме того, вложенные циклы могут указывать на использование не эффективного алгоритма решения задачи. Это не всегда так, но вероятность такая есть.

Как избавиться от вложенных циклов? Есть три варианта. Первый – ничего не делать, иногда вложенные циклы это нормально, особенно в низкоуровневых алгоритмах. Второй – переписать алгоритм так, чтобы вложенного цикла не осталось вообще, даже в вызываемых функциях. Когда это невозможно – использовать третий вариант. Вынести вложенный цикл в функцию, либо заменить на встроенную функцию (или метод). Например в JavaScript у массивов есть метод includes(), который внутри себя представляет не что иное, как обход массива в цикле.

```
// Эта функция заменяет собой цикл
// Но не забывайте что внутри все равно остается полный обход массива
[1, 10, 3].includes(10); // true
```
Пример выноса в отдельную функцию кода на flatten:

```
// Изменяет первый массив напрямую
// В данном случае такая реализация оправдана
const append = (arr1, arr2) => {
  for (const item of arr2) {
    arr1.push(item);
  }
};

const flatten = (coll) => {
  let result = [];
  for (const item of coll) {
    if (Array.isArray(item)) {
      // Нет присваивания так как меняется сам result
      append(result, item);
    } else {
      result.push(item);
    }
  }

  return result;
};

flatten([3, 2, [], [3, 4, 2], 3, [123, 3]]);
// [3, 2, 3, 4, 2, 3, 123, 3]
```

Дополнительные материалы

1. Встроенный метод flat
2. Функция flatten из библиотеки Lodash
3. Функция flattenDeep из библиотеки Lodash
4. Функция flattenDepth из библиотеки Lodash
5. Метод Array.isArray, проверяющий, является ли значение массивом

https://lodash.com/docs/#flatten

2.14. Теория множеств.

Теория множеств – крайне важная математическая концепция для любых разработчиков. Данные, с которыми работают программы, часто представляются как множества, а значит к ним применимы правила теории множеств. В первую очередь это касается различных операций над множествами, например, пересечением или объединением.

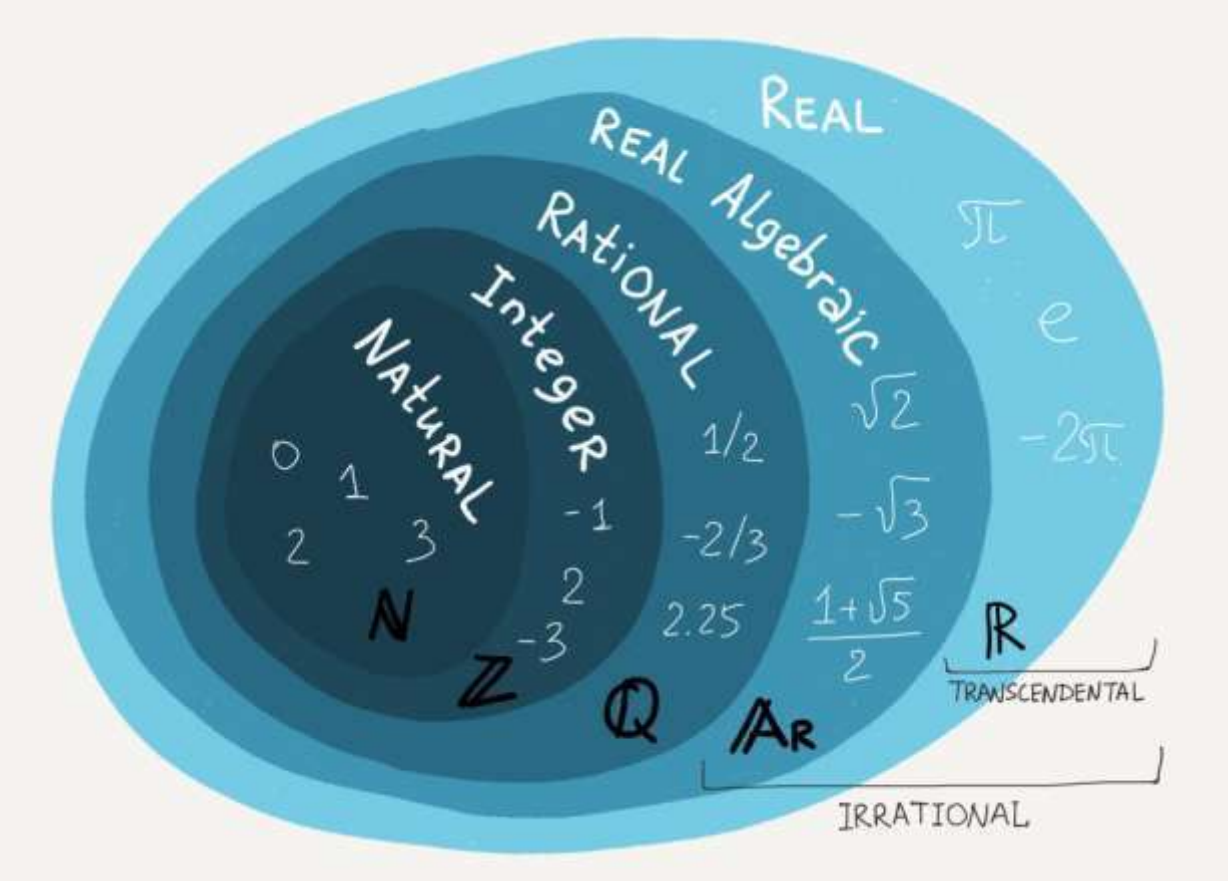
Это не значит, что нужно знать эту теорию от и до. Напротив, достаточно изучить ее основные понятия и некоторые операции. Этого хватит для эффективного решения подавляющего числа задач. Сама теория множеств относится к интуитивно понятным концепциям. Она хорошо ложится на здравый смысл и понятна людям даже без особой математической подготовки.

Краткая терминология

Основное понятие теории множеств **множество**. Множеством обозначают набор объектов произвольной природы, рассматривающихся как единое целое. Простейший пример — цифры. Множество всех цифр включает в себя 10 элементов (от 0 до 9).

Но не каждый набор объектов можно назвать множеством. Существует важное условие – все элементы множества должны быть уникальными. Например числа *1, 1 и 3* не могут называться множеством, а *1, 3, 5* могут.

Множества между собой могут находиться в определенных отношениях. Например множество натуральных чисел является подмножеством целых чисел, которые в свою очередь являются подмножеством рациональных чисел и так далее. Понятие «подмножество» означает, что все элементы одного множества также входят в другое множество, называемое **надмножеством**.



Представление множеств кружками довольно удобно. Можно быстро оценить как друг с другом соотносятся разные множества.

Но математические объекты, такие как числа, не единственные возможные объекты множеств. Множеством можно назвать группу людей, стоящих на остановке в ожидании своего автобуса, или жильцов квартир одного дома, города или страны.

В программировании в качестве множеств могут выступать массивы и таблицы в базе данных. В JavaScript, для представления множеств есть встроенный механизм Set. Но для работы с ним нужно немного понимать объектно-ориентированные возможности, которые рассматриваются в более поздних курсах.

Операции над множествами

На практике, представление данных в виде множеств полезно тогда, когда мы хотим что-то сделать с ними. Простой пример. Когда в Фейсбуке вы заходите на страницу другого человека, то Фейсбук показывает вам блок с общими друзьями. Если принять, что ваши друзья и друзья вашего друга — два множества, то общие друзья — множество, полученное как пересечение исходных множеств друзей.

Пересечение — один из ярких примеров операции над множествами, которая в программировании встречается повсеместно. То же самое можно сказать и о некоторых других операциях. Важно, что результатом всех этих операций являются множества, а значит они подчиняются тем же правилам, что и исходные множества. Например сохраняется уникальность элементов.

Пересечение

Пересечением множеств называется множество, в которое входят элементы, встречающиеся во всех данных множествах одновременно.

Пример с общими друзьями:

```
// Друзья одного человека
const friends1 = ['vasya', 'kolya', 'petya'];

// Друзья другого человека
const friends2 = ['igor', 'petya', 'sergey', 'vasya', 'sasha'];

// Общие друзья
// Эта функция принимает любое количество массивов.
// То есть вы можете находить пересечение любого количества массивов за один вызов.
_.intersection(friends1, friends2); // ['vasya', 'petya']
```

Объединение

Объединением множеств называется множество, в которое входят элементы всех данных множеств.

```
const friends1 = ['vasya', 'kolya', 'petya'];
const friends2 = ['igor', 'petya', 'sergey', 'vasya', 'sasha'];

_.union(friends1, friends2); // ['vasya', 'kolya', 'petya', 'igor', 'sergey', 'sasha']
```

Каждый друг в объединении встречается ровно один раз.

Дополнение (разность)

Разностью двух множеств называется множество, в которое входят элементы первого множества, не входящие во второе. В программировании такая операция часто называется *diff* (разница).

```
const friends1 = ['vasya', 'kolya', 'petya'];
const friends2 = ['igor', 'petya', 'sergey', 'vasya', 'sasha'];

_.difference(friends1, friends2); // ['kolya']
```

Принадлежность множеству

Проверку принадлежности элемента множеству можно выполнить с помощью встроенного метода includes():

```
const terribleNumbers = [4, 13];

if (terribleNumbers.includes(10)) {
  console.log('woah!');
}
```

Дополнительные материалы

1. Рассказы о множествах (pdf)
2. Встроенный метод includes
3. Функция intersection из библиотеки Lodash
4. Функция union из библиотеки Lodash
5. Функция difference из библиотеки Lodash

2.15. Сортировка массивов.

https://zen.yandex.ru/media/id/5ab37eee77d0e62bb2b0354a/12-otlichnyh-rasshirenii-dlia-javascript-vscode-kotorye-pomogut-kodit-bystree-5ad87e672f578c173b63fdfa?utm_source=serp

12 отличных расширений для JavaScript VSCode, которые помогут кодить быстрее

Сортировка массивов — базовая алгоритмическая задача, которую нередко спрашивают на собеседованиях. Однако, в реальном коде массивы сортируют, используя уже готовые функции стандартной библиотеки. В JavaScript сортировка выполняется с помощью метода sort() массивов:

```
const numbers = [8, 3, 10];
// sort изменяет массив!
numbers.sort((a, b) => a - b); // сортировка по возрастанию
console.log(numbers); // => [3, 8, 10]

// В обратную сторону можно через reverse()
// Тоже изменяет массив
numbers.reverse();
console.log(numbers); // => [10, 8, 3]
```

Тогда для чего задают подобные вопросы? Обычно собеседующий хочет узнать следующее:

1. Насколько кандидат вообще в курсе о существовании алгоритмов.
2. Способен ли он программировать (составлять программу сам, думая своей головой).
3. Как работает его алгоритмическое мышление.

Знание алгоритмов действительно влияет на то, как мы думаем и насколько быстро соображаем. И хотя невозможно знать все алгоритмы, нужно хотя бы иметь представление о самых ключевых и в идеале уметь их реализовывать. В нашем списке рекомендуемых книг есть как минимум одна книга, полностью посвящённая алгоритмам.

Кроме того, Роберт Мартин в своей книге "Идеальный программист" рассказывает о подходе Ката — понятии, взятом из боевых искусств.

Принцип изучения боевого искусства на основе ката состоит в том, что повторяя ката многие тысячи раз, практик боевого искусства приучает своё тело к определённого рода движениям, выводя их на бессознательный уровень. Таким образом, попадая в боевую ситуацию, тело работает "само" на основе рефлексов, вложенных многократным повторением ката. Также считается, что ката обладают медитативным воздействием.

Роберт Мартин рекомендует время от времени решать классические алгоритмические задачки для поддержания формы. Эта тема стала настолько популярной, что если загуглить *javascript github kata*, то вы увидите множество репозиториев с подобными задачками.

Сортировка

Способов сортировать массив достаточно много. Самый популярный для обучения — пузырьковая сортировка (bubble sort).

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится

на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде. Отсюда и название алгоритма).

```
// Функция изменяет входящий массив items
const bubbleSort = (items) => {
  let stepsCount = items.length - 1;
  // Объявляем переменную swapped, значение которой показывает был ли
  // совершен обмен элементов во время перебора массива
  let swapped;
  // do..while цикл. Работает почти идентично while
  // Разница в проверке. Тут она делается не до выполнения тела, а после.
  // Такой цикл полезен там, где надо выполнить тело хотя бы раз в любом случае.
  do {
    swapped = false;
    // Перебираем массив и меняем местами элементы, если предыдущий
    // больше, чем следующий
    for (let i = 0; i < stepsCount; i += 1) {
      if (items[i] > items[i + 1]) {
        // temp – временная переменная для хранения текущего элемента
        let temp = items[i];
        items[i] = items[i + 1];
        items[i + 1] = temp;
        // Если сработал if и была совершена перестановка,
        // присваиваем swapped значение true
        swapped = true;
      }
    }
    // Уменьшаем счетчик на 1, т.к. самый большой элемент уже находится
    // в конце массива
    stepsCount -= 1;
  } while (swapped); // продолжаем, пока swapped === true

  return items;
};

console.log(bubbleSort([3, 2, 10, -2, 0])); // => [ -2, 0, 2, 3, 10 ]
```

https://repl.it/@hexlet/js-arrays-sorting-bubble

Весь код этой функции делится на два уровня:

1. Внутренний цикл *for*, который проходит по массиву от начала до конца, меняя элементы попарно, если нужно сортировать.
2. Внешний цикл *do...while*, который определяет, когда нужно остановиться. Обратите внимание, что в худшем случае этот цикл выполнится items.length раз, что совпадает с теоретическим худшим случаем этого алгоритма, при котором самый большой или маленький элемент находятся в противоположных концах массива от сортированного варианта.

Пузырьковая сортировка – самый простой и интуитивно понятный алгоритм сортировки. Очень полезно уметь реализовывать по памяти. Попробуйте сделать это на собственном компьютере не подсматривая в теорию.

Дополнительные материалы

1. Сортировка средствами JavaScript
2. Цикл do...while
3. Визуализация алгоритмов сортировок

2.16. Стек.

Тема алгоритмов не существует сама по себе. Важно не только уметь составлять правильный алгоритм решения, но не менее важно использовать для работы с данными правильную структуру.

Структура данных — это конкретный способ хранения и организации данных. В зависимости от решаемых задач, удобным оказывается либо один способ организации данных, либо другой. Как минимум, одну структуру данных вы уже знаете достаточно хорошо — это массив. С точки зрения организации, массив представляет собой совокупность элементов, к которым имеется индексированный доступ (доступ по индексу), а вот с точки зрения физического хранения в памяти — всё сложнее. Массивы бывают разные и внутри языка реализуются тоже по-разному.

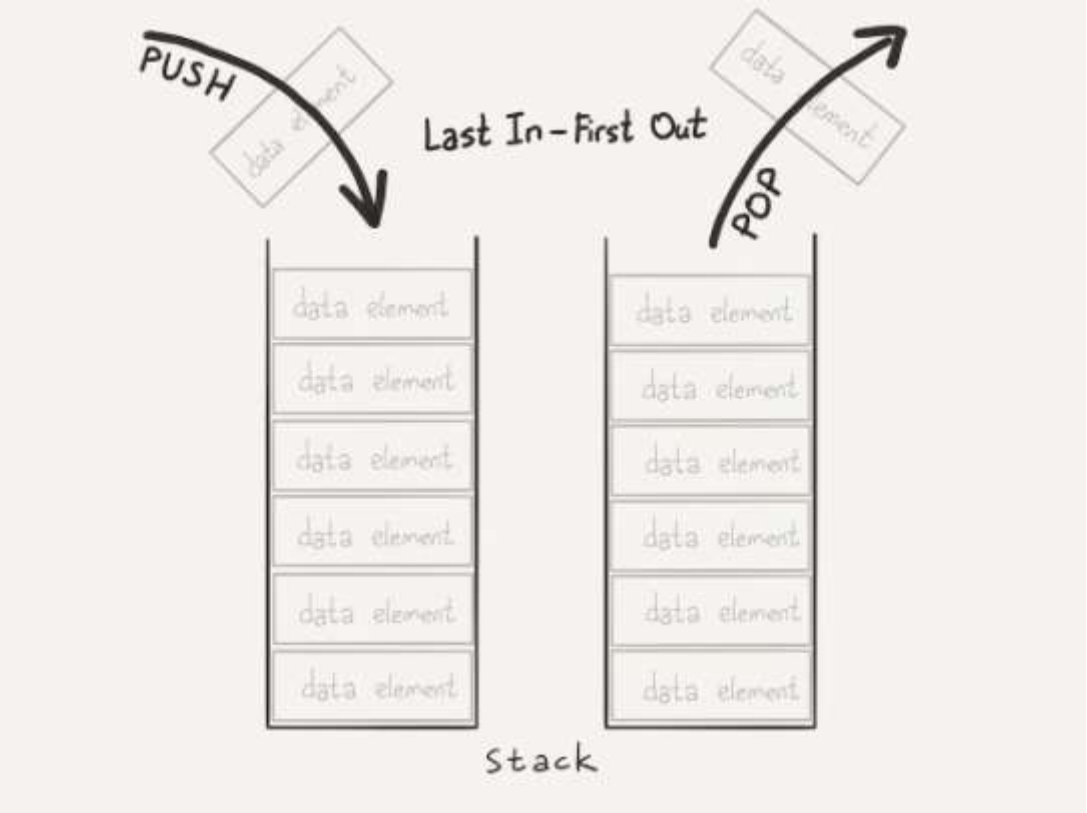
Кроме массивов существует множество других структур данных, таких как списки, хеш-таблицы, деревья, графы, стек, очередь и другие. Использование структуры данных, подходящей под решаемую задачу, позволяет кардинально упростить код, устраняя запутанную логику.

Некоторые из перечисленных структур данных мы рассмотрим в процессе прохождения курсов и проектов, другие вам нужно будет подтянуть самостоятельно из книг. В любом случае алгоритмы и структуры данных (без фанатизма) составляют базу, на которую нанизывается всё остальное в разработке.

В этом уроке мы разберем одну из самых простых и важных структур данных – стек (stack, переводится как стопка).

Стек

Стек — упорядоченная коллекция элементов, в которой добавление новых и удаление старых элементов всегда происходит с одного конца коллекции. Обычно его называют вершиной стека.



У стека есть аналоги из реальной жизни. Например магазин автомата. Патроны добавляются в магазин только друг за другом. Извлекаются тоже, только в обратном порядке. Последний вставленный патрон выйдет из магазина первым.

Практически любая стопка может рассматриваться как стек. Если не применять грубую физическую силу, то со стопками мы работаем двумя способами. Либо кладём новый элемент (например, книгу) на верхушку стопки, либо снимаем элемент с верхушки. Поэтому стек ещё называют "Last In First Out" (LIFO), то есть "последний зашёл, первый вышел".

Перед тем, как разбирать конкретную задачу, посмотрим на роль, которую играет стек в программировании. Вспомните, как исполняется любая программа. Одни функции вызывают другие, которые, в свою очередь, вызывают третьи, и так далее. После того, как выполнение заходит в самую глубокую функцию, та возвращает значение, и начинается обратный процесс. Сначала идёт выход из наиболее глубоких функций, затем из тех, что уровнем выше, и так далее до тех пор, пока не дойдёт до самой внешней функции. Вызов функций — не что иное, как добавление элемента в стек, а возврат – извлечение из стека. Именно так всё устроено на аппаратном уровне. К тому же, если в процессе

выполнения программы происходит ошибка, то её вывод часто называют *Stack Trace* (трассировка стека).

Другой пример, связанный с программированием — кнопка "назад" в браузере. История посещений представляет собой стек, каждый новый переход по ссылке добавляет её в историю, а кнопка «назад» извлекает из стека последний элемент.

Работа со стеком включает в себя следующие операции:

- Добавить в стек (push)
- Взять из стека (pop)
- Вернуть элемент с вершины стека без удаления (peek)
- Проверить на пустоту (isEmpty)
- Вернуть размер (size)

Первые две – базовые, остальные нужны время от времени. В JavaScript стек можно создать на основе массивов. Для этого используются методы push(), pop() и свойство length.

```
const stack = [];

stack.push(3);
console.log(stack); // => [ 3 ]
stack.push('Winterfall');
console.log(stack); // => [ 3, 'Winterfall' ]
stack.push(true);
console.log(stack); // => [ 3, 'Winterfall', true ]

const element1 = stack.pop();
console.log(element1); // => true
console.log(stack); // => [ 3, 'Winterfall' ]

const element2 = stack.pop();
console.log(element2); // => Winterfall
console.log(stack); // => [ 3 ]
```

https://repl.it/@hexlet/js-arrays-stack

Обратите внимание, что методы pop() и push() изменяют исходный массив. pop не только изменяет его, но и возвращает элемент, снятый со стека.

Рассмотрим задачку, решение которой тривиально при использовании стека. Кстати, её нередко задают на собеседованиях, как раз чтобы убедиться, хорошо ли вы знаете базовые структуры данных.

Задача:

Необходимо реализовать функцию, которая проверяет, что парные скобки сбалансированы. То есть каждая открывающая скобка имеет закрывающую: (), (()())). А вот пример несбалансированных скобок: (, (()),)(. Для проверки баланса недостаточно считать количество, важен так же порядок в котором они идут.

У этой задачи есть простое решение и без стека, но стек тоже хорошо подходит (и даже нагляднее)

Решение со стеком выглядит так:

1. Если перед нами открывающий элемент, то заносим его в стек
2. Если закрывающий, то достаём из стека элемент (очевидно, последний добавленный) и смотрим, что он открывающий для данного закрывающего. Если проверка провалилась, значит выражение не соответствует требуемому формату.
3. Если мы дошли до конца строки и стек пустой, то всё хорошо. Если в стеке остались элементы, то проверка не прошла. Такое может быть, если в начале строки были открывающие элементы, но в конце не было закрывающих.

Разберём его построчно:

```
const checkIfBalanced = (expression) => {
  // Инициализация стека
  const stack = [];
  // Проходим по каждому символу в строке
  for (const symbol of expression) {
    // Смотрим открывающий или закрывающий
    if (symbol === '(') {
      stack.push(symbol);
    } else if (symbol === ')') {
      // Если для закрывающего не нашлось открывающего, значит баланса нет
      if (!stack.pop()) {
        return false;
      }
    }
  }

  return stack.length === 0;
};

export default checkIfBalanced;
```

https://repl.it/@hexlet/js-arrays-stack-balancing

Предположим, что на вход функции попала следующая строка: ((). Ниже описание ключевых шагов при выполнении функции проверки:

1. Первая скобка (заносится в стек, так как она открывающая
2. Следующая скобка (также заносится в стек по той же самой причине
3. Последняя) является закрывающей, из стека извлекается последняя добавленная скобка
4. В стеке остался один элемент, значит баланса нет

Семантика

Может возникнуть соблазн использовать эти функции в повседневной практике. Например, чтобы извлечь из массива последний элемент. Несмотря на то, что метод pop() действительно позволяет это сделать, такой вариант использования крайне нежелателен по нескольким причинам:

1. Побочный эффект данной операции — изменение исходного массива. Даже если массив потом не используется, такой код вносит потенциальные проблемы и заставляет его переписывать в будущем.
2. Нарушается семантика. Инструменты нужно использовать по назначению, иначе рождается код, который декларирует одно, но в реальности делает другое. Любой опытный программист, который видит pop() сразу считает, что массив в данной части программы используется как

стек. Если это не так, то понадобятся дополнительные умственные усилия для понимания происходящего.

Для извлечения последнего элемента лучше использовать метод last() библиотеки lodash.

Дополнительные материалы

1. метод push
2. метод pop

2.17. Big O.

Когда заходит речь про алгоритмы, нельзя не упомянуть понятие «сложность алгоритма» и нотацию О-большое (Big O notation). Она не только полезна при прохождении собеседований, но и даёт понимание того, как вообще оценивать эффективность кода (очень относительно).

Как вы помните, алгоритмов сортировок существует много, я бы сказал очень много. Все они выполняют одну и ту же задачу, но при этом отличаются друг от друга. В информатике алгоритмы сравниваются друг с другом (или классифицируются) по их вычислительной (или алгоритмической) сложности. Сложность оценивается по количеству выполняемых операций. Понятно, что конкретное количество операций зависит от входных данных, например, если массив отсортирован, то количество операций будет минимальным (но они все равно будут, потому что алгоритм должен убедиться в том, что массив отсортирован). Если не отсортирован, то для каждого алгоритма можно подобрать такие входные данные, при которых он будет работать долго и не эффективно. Эти случаи называют соответственно верхней и нижней границей.

Нотация Big O как раз придумана для описания алгоритмической сложности. Она призвана показать, как сильно увеличится количество операций при увеличении размера данных.

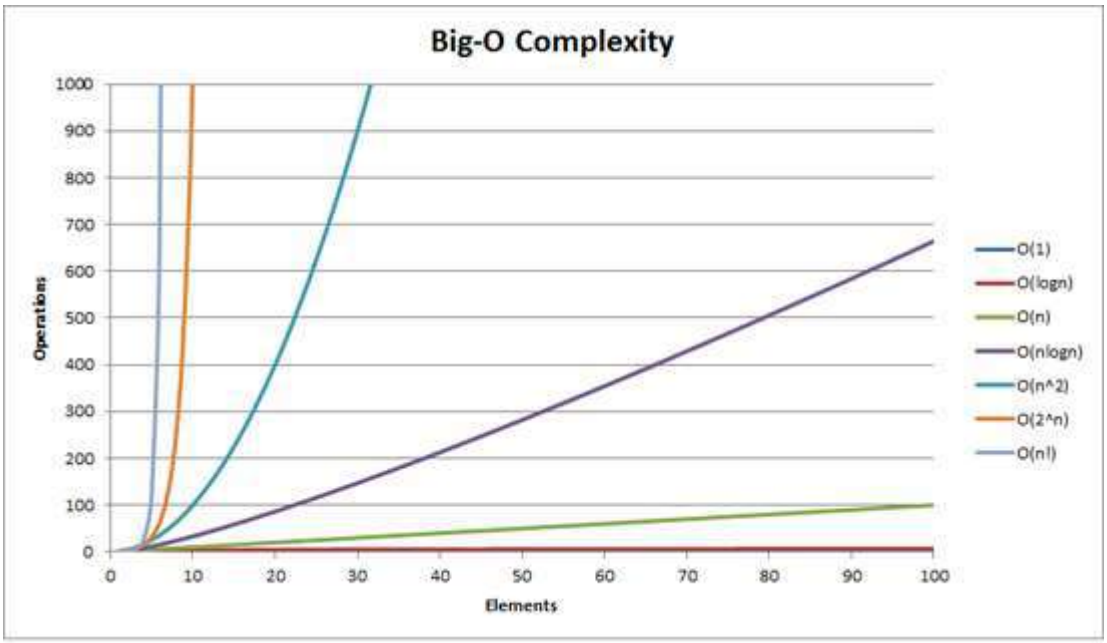
Вот некоторые примеры того, как записывается сложность: *O(1)*, *O(n)*, *O(nlog(n))*.

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

O(1) описывает так называемую константную сложность. Например обращение к элементу массива по индексу оценивается константой, другими словами оно не зависит от размера массива, поэтому внутри О записывается единица, символизирующая константу. А вот функция, которая печатает на экран все элементы переданного массива, используя обычный перебор имеет сложность *O(n)* (линейная сложность). То есть количество выполняемых операций будет равно количеству элементов массива. Именно это количество символизирует символ *n* в скобках.

Ещё один простой пример — вложенные циклы. Вспомните как работает поиск пересечений в неотсортированных массивах. Для каждого элемента из одного массива проверяется каждый элемент другого массива (либо через цикл, либо с помощью метода includes(), чья сложность *O(n)*, ведь в худшем случае он просматривает весь массив). Если принять, что размеры обоих массивов одинаковы и равны *n*, то получается, что поиск пересечений имеет квадратичную сложность или $O(n^{2})$ (n в квадрате). Существуют как очень эффективные, так и абсолютно неэффективные алгоритмы. Первые, как правило, имеют логарифмическую сложность, последние — степенную, такую, при которой *n* находится в степени. Скорость работы подобных алгоритмов падает с катастрофической скоростью даже при небольшом количестве элементов.

Нередко более быстрые алгоритмы быстрее не потому, что они лучше, а потому что они потребляют больше памяти или имеют возможность запускаться параллельно (и если это происходит, то работают крайне эффективно). Как и всё в инженерной деятельности, эффективность — компромисс. Выигрывая в одном месте, мы проиграем где-то в другом.



Big O, во многом, теоретическая оценка, на практике всё может быть по-другому. Реальное время выполнения зависит от множества факторов среди которых: архитектура процессора, операционная система, язык программирования, доступ к памяти (последовательный или произвольный) и многое другое.

Вопрос эффективности кода довольно опасен. В силу того, что многие начинают учить программирование именно с алгоритмов (особенно в университете), им начинает казаться, что эффективность — это главное. Код должен быть быстрым.

Такое отношение к коду гораздо чаще приводит к проблемам, чем делает его лучше. Важно понимать, что эффективность — враг понимаемости. Такой код всегда сложнее, больше подвержен ошибкам, труднее модифицируется, дольше пишется. А главное, настоящая эффективность редко когда нужна сразу или вообще не нужна. Обычно тормозит не код, а, например, запросы к базе данных или сеть. Но даже если код выполняется медленно, то вполне вероятно, что именно тот участок, который вы пытаетесь оптимизировать, вызывается за все время жизни программы всего лишь один раз и ни на что не влияет, потому что работает с небольшим объёмом памяти, а где-то в это время есть другой кусок, который вызывается тысячи раз, и приводит к реальному замедлению.

Программисты тратят огромное количество времени, размышляя и беспокоясь о некритичных местах кода, и пытаются оптимизировать их, что исключительно негативно сказывается на последующей

отладке и поддержке. Мы должны вообще забыть об оптимизации в, скажем, 97% случаев. Поспешная оптимизация является корнем всех зол. И, напротив, мы должны уделить все внимание оставшимся 3%. — Дональд Кнут

Перед тем, как пытаться что-то оптимизировать, обязательно прочитайте небольшую онлайн-книжку, которая хорошо объясняет суть всех оптимизаций.

Дополнительные материалы

1. Big-O Cheat Sheet

2.18. Деструктуризация.

Деструктуризация (destructuring) – синтаксическая возможность "раскладывать" элементы массива (и не только) в отдельные константы или переменные. Деструктуризация относится к необязательным, но очень приятным возможностям языка. Рассмотрим ее на примерах.

Представьте, что у нас есть массив из двух элементов, которыми мы хотим оперировать в нашей программе. Самый простой вариант использования его элементов — постоянное обращение по индексу point[0] и point[1].

```
const point = [3, 5];

console.log(`${point[0]}:${point[1]}`);
```

Индексы ничего не говорят о содержимом, и для понимания этого кода придется прикладывать дополнительные усилия. Гораздо лучше сначала присвоить эти значения переменным с хорошими именами. Тогда код станет читаемым:

```
const x = point[0];
const y = point[1];

console.log(`${x}:${y}`);
```

Код стал значительно понятнее, хотя и длиннее. С помощью деструктуризации то же самое можно сделать короче:

```
const [x, y] = point;
// Слева массив повторяет структуру правого массива
// но вместо значений используются идентификаторы
// они заполняются значениями, стоящими на тех же позициях в правом массиве
// [x, y] = [3, 5]
// x = 3, y = 5

console.log(`${x}:${y}`);
```

Получилось и короче, и понятнее (особенно если привыкнуть к этому способу записи). Деструктуризация позволяет извлечь из массива практически любые части, используя очень короткий и интуитивно понятный синтаксис. Она работает даже в том случае, когда нас интересует только часть массива. Причем как начало, так и его конец:

```
// Извлекаем первый элемент
// Остальные игнорируются
const [x] = point;
```

```
// Извлекаем второй элемент
// Для этого просто не указываем первый
const [, y] = point;

// и даже так
const [, secondElement, , fourthElement, fifthElement] = [1, 2, 3, 4, 5, 6];

console.log(secondElement); // => 2
console.log(fourthElement); // => 4
console.log(fifthElement);  // => 5
```

https://repl.it/@hexlet/js-arrays-destructuring-positional

Мы извлекли из массива [1, 2, 3, 4, 5, 6] значения второго, четвёртого и пятого элементов, сохранив их, соответственно, в константах secondElement, fourthElement и fifthElement. При этом на месте первого (нулевой индекс) и третьего (второй индекс) элементов мы сделали пропуски, не указав никаких переменных, потому что значения этих элементов в данном случае нам были не нужны.

Если при деструктуризации указать переменную так, что ей не будет соответствовать ни один элемент массива, то в переменную запишется значение undefined:

```
const [firstElement, secondElement, thirdElement] = [1, 2];

console.log(firstElement);  // => 1
console.log(secondElement); // => 2
console.log(thirdElement);  // => undefined
```

Исходный массив состоит всего из двух элементов, поэтому в thirdElement сохранилось undefined. Но в таких случаях можно подстраховаться и определить значение по умолчанию:

```
const [firstElement, secondElement, thirdElement = 3] = [1, 2];

console.log(firstElement);  // => 1
console.log(secondElement); // => 2
console.log(thirdElement);  // => 3
```

В массиве [1, 2] нет соответствия для thirdElement, поэтому в константу thirdElement было записано значение по умолчанию 3.

Деструктуризация работает на любом уровне вложенности. Например с ее помощью можно извлекать данные из массивов внутри массивов:

```
const [one, [two, three]] = [1, [2, 3]];
```

Количество возможных комбинаций и вариантов использования деструктуризации – бесконечное множество. Чем больше вы будете экспериментировать с ней, тем больше найдете вариантов использования.

Деструктуризация в циклах

Разложение массива можно использовать не только как отдельную инструкцию в коде, но и, например, в циклах:

```
const points = [
  [4, 3],
  [0, -3],
];
```

```
for (const [x, y] of points) {
  console.log([x, y]);
}

// => [ 4, 3 ]
// => [ 0, -3 ]
```
https://repl.it/@hexlet/js-arrays-destructuring-for-of

На самом деле такое разложение можно сделать почти во всех местах, где явно или неявно ожидается массив. Входные параметры и возвращаемые значения функций, циклы и некоторые другие ситуации, с которыми вы обязательно будете сталкиваться при написании кода.

```
const swapValues = ([a, b]) => [b, a];

swapValues([1, 2]); // [2, 1]
```

Дополнительные материалы

1. Официальная документация

2.19.Rest-оператор и деструктуризация.

Мощь деструктуризации больше всего проявляется там, где она используется вместе с rest-оператором. Rest-оператор позволяет "свернуть" часть элементов во время деструктуризации. Например с его помощью можно разложить массив на первый элемент и все остальные:

```
const fruits = ['apple', 'orange', 'banana', 'pineapple'];

// ... – rest-оператор
const [first, ...rest] = fruits;
console.log(first); // 'apple'
console.log(rest); // ['orange', 'banana', 'pineapple']
```
Запись ...rest, означает что нужно взять все элементы, которые остались от деструктуризации и поместить их в массив с именем rest. rest-оператор срабатывает в самом конце, когда все остальные данные уже разложены по своим константам (или переменным). Именно поэтому он называется *rest* (оставшиеся).

Подобным образом любой массив раскладывается на любое количество элементов + остальные. У rest-оператора есть ограничения. Он не может появляться нигде кроме конца массива.

```
const [first, second, ...rest] = fruits;
// rest = ['banana', 'pineapple']

const [first, second, third, ...rest] = fruits;
// rest = ['pineapple']

// Если элементов нет, то в rest окажется пустой массив
const [first, second, third, oneMore, ...rest] = fruits;
// rest = []
```

В ситуациях когда нас интересует только часть массива, но не важны первые элементы, лучше воспользоваться методом массива slice():

```
// slice возвращает новый массив, а не изменяет старый
const rest = fruits.slice(1);
console.log(rest); // ['orange', 'banana', 'pineapple'];
```

2.20.Spread-оператор и создание новых массивов.

У оператора rest есть оператор-компаньон – spread-оператор. Этот оператор имеет такой же синтаксис, но выполняет противоположную задачу. Он не сворачивает элементы, а наоборот, растягивает их. С его помощью обычно копируют или сливают массивы.

Представьте, что нам нужно определить массив, добавив туда элементы из другого массива. Такая задача часто встречается при работе со значениями по умолчанию:

```
const russianCities = ['moscow', 'kazan'];
const cities = ['milan', 'rome', ...russianCities];
// ['milan', 'rome', 'moscow', 'kazan']

// Массив russianCities при этом никак не меняется

// То же самое без spread-оператора
const cities = ['milan', 'rome'].concat(russianCities);
```

... в данном случае spread-оператор. Он растянул массив, добавив все его элементы в новый массив. Как отличить его от rest-оператора? Все дело в контексте использования. Rest-оператор появляется слева от знака равно, там, где происходит деструктуризация. Spread-оператор – справа от знака равно, там где массив формируется.

Spread-оператор, в отличии от rest-оператора может появляться в любой части массива. Например, мы можем дополнить исходный массив не справа, а слева:

```
const cities = [...russianCities, 'milan', 'rome'];
// ['moscow', 'kazan', 'milan', 'rome']

// То же самое без spread-оператора
const cities = russianCities.concat(['milan', 'rome']);
```

И даже посередине:

```
const cities = ['milan', ...russianCities, 'rome'];
// ['milan', 'moscow', 'kazan', 'rome']

// Без spread-оператора подобный код нельзя выразить одной операцией
```

Spread-оператор работает с любым количеством массивов:

```
const russianCities = ['moscow', 'kazan'];
const ukrainianCities = ['kiev', 'odessa']
// слияние двух массивов
const cities = [...russianCities, ...ukrainianCities];
// ['moscow', 'kazan', 'kiev', 'odessa']
```

```
// То же самое без spread-оператора
const cities = russianCities.concat(ukrainianCities);
```

Копирование массива

Spread-оператор нередко используется для копирования массива. Копирование предотвращает исходный массив от изменения, в том случае, когда необходимо менять его копию:

```
const russianCities = ['moscow', 'kazan'];
const copy = [...russianCities];
copy.push('samara');
console.log(copy); // => ['moscow', 'kazan', 'samara']
console.log(russianCities); // => ['moscow', 'kazan']

// То же самое без spread-оператора
const russianCities = ['moscow', 'kazan'];
const copy = russianCities.slice();
```

2.21.Массивы в памяти компьютера.

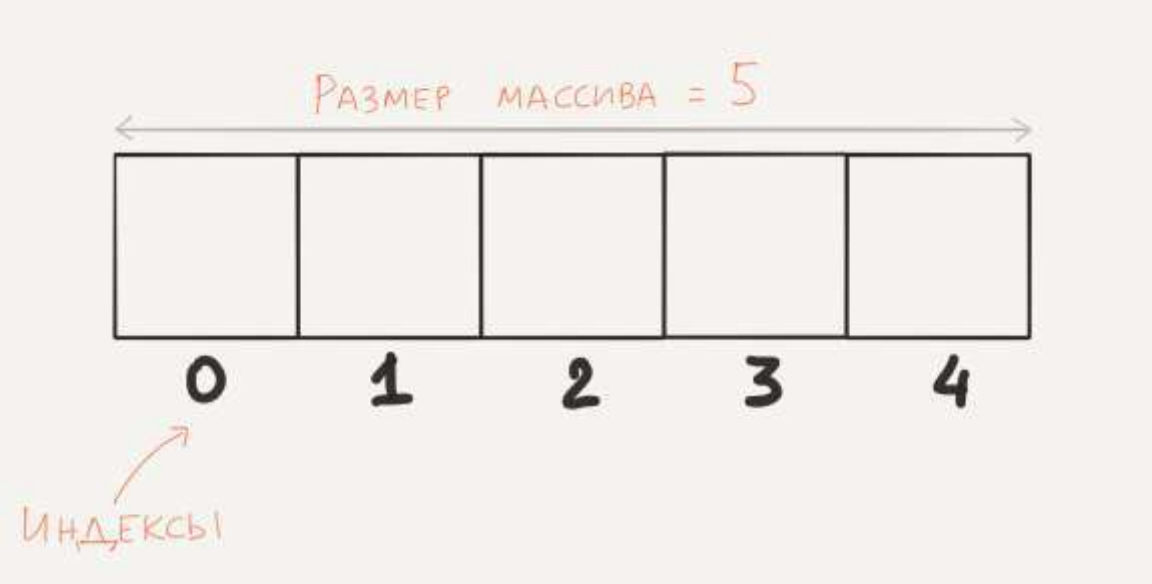
Массивы в памяти компьютера

Работая на таких высокоуровневых языках, как JavaScript, позволительно не знать устройство массивов для решения повседневных задач. С другой стороны, подобное понимание делает код менее магическим и даёт возможность заглядывать чуть дальше.

Массивы в си

Реальные массивы лучше всего рассматривать на языке **Си**, который, с одной стороны, достаточно простой и понятный, с другой — очень близок к железу и не скрывает от нас практически ничего. Когда мы говорим про примитивные типы данных, такие как "строка" или "число", то на интуитивном уровне всё довольно понятно: под каждое значение выделяется некоторый размер памяти (в соответствии с типом), в которой и хранится само значение. А как должна выделиться память под хранение массива? И что такое массив в памяти? На уровне хранения понятия массив не существует. Массив представляется цельным куском памяти, размер которого вычисляется по следующей формуле: *количество элементов * количество памяти под каждый элемент*. Из этого утверждения есть два интересных вывода:

- Размер массива — фиксированная величина. Те динамические массивы (изменяющие свой размер во время работы), с которыми мы имеем дело во многих языках, реализованы уже внутри языка, а не на уровне железа.
- Все элементы массива имеют один тип и занимают одно и то же количество памяти. Благодаря этому появляется возможность простым умножением (по формуле, описанной выше) получить адрес той ячейки, в которой лежит нужный нам элемент. Именно это происходит под капотом, при обращении к элементу массива под определённым индексом.



Фактически, индекс в массиве — смещение относительно начала куска памяти, содержащего данные массива. Адрес, по которому расположен элемент под конкретным индексом, рассчитывается так: *индекс * количество памяти, занимаемое одним элементом (для данного типа данных на данной архитектуре)*. Пример на Си:

```
// Инициализация массива из пяти элементов типа int
// Предположим, что int занимает 2 байта
// Общее количество памяти выделенное под массив int * 5 = 2 * 5 = 10 байт
int numbers[] = {19, 10, 8, 17, 9};
numbers[3]; // 17
```

Если предположить, что тип int занимает в памяти 2 байта, то адрес элемента, соответствующего индексу 3, вычисляется так: *начальный адрес + 3 * 2*. Начальный адрес — это адрес ячейки памяти, начиная с которой располагается массив. Он формируется во время выделения памяти под массив. Ниже пример рассчета адресов памяти под разные элементы массива numbers:

```
// Первый элемент
// Начальный адрес + 2 * 0 = начальный адрес
numbers[0]; // 19

// Начальный адрес + 2 * 1 = начальный адрес + 2
// То есть сместились на 2 байта
numbers[1]; // 10

// Начальный адрес + 2 * 2 = начальный адрес + 4
// То есть сместились на 4 байта
numbers[2]; // 8

// Последний элемент
// Начальный адрес + 2 * 4 = начальный адрес + 8
// То есть сместились на 8 байта
// И сам элемент занимает 2 байта. В сумме как раз 10
numbers[4]; // 9
```

Теперь должно быть понятно, почему индексы в массиве начинаются с нуля. 0 — означает отсутствие **смещения**.

Но не все данные имеют одинаковый размер. Как будет храниться массив строк? Строки ведь имеют разную длину, а значит требуют разное количество памяти для своего хранения. Один из способов сохранить строки в массиве на языке Си – создать массив массивов (тут нужно понимать, что любая строка в Си это массив символов). Вложенные массивы обязательно должны быть одного размера, невозможно обойти физические ограничения массивов. Хитрость в том, что этот размер должен быть достаточно большой, чтобы туда поместились необходимые строки.

```
// Массив из трех элементов, внутри которого массивы по 10 элементов
// Это значит, что здесь можно хранить 3 строки длиной не больше 10 символов
char strings[3][10] = {
  "spike",
  "tom",
  "jerry"
};

strings[0]; // spike
```

Безопасность

В отличие от высокоуровневых языков, в которых код защищён от выхода за границу массива, в таком языке, как **Си**, выход за границу не приводит к ошибкам (на самом деле он может приводить к segfault, но это здесь не важно). Обращение к элементу, индекс которого находится за пределами массива, вернёт данные, которые лежат в той самой области памяти, куда его попросили обратиться (в соответствии с формулой выше). Чем они окажутся — никому не известно (но они будут проинтерпретированы в соответствии с типом массива. Если массив имеет тип int, то вернётся число). Выход за границу массива активно эксплуатируется хакерами для взлома программ.

Массивы в JavaScript

Устройство массивов в JavaScript значительно сложнее чем в си. JavaScript динамический язык, это значит, что типы данных вычисляются автоматически во время выполнения кода. Массив в такой среде не может работать как в си. Неизвестно данные каких типов окажутся внутри в процессе работы.

Массивы в JavaScript содержат не сами данные, а ссылки (адреса в памяти) на них. Тогда становится не важно, что хранить. Любое значение в массиве – адрес, имеющий одинаковый размер независимо от данных, на которые он указывает. Такой подход делает массивы гибкими, но с другой стороны, более медленными.

Кроме того, массивы в JavaScript динамические. То есть их размер может увеличиваться или уменьшаться в процессе работы программы. Технически это работает так: если ссылки (помним, что данные там не хранятся) в массив не помещаются, то интерпретатор внутри себя создает новый массив большего размера (обычно в два раза) и переносит все ссылки туда. Динамические массивы очень упрощают процесс разработки, но за это тоже приходится платить скоростью.

2.22.Дополнительные материалы.

Стандартные встроенные объекты. Array.

Массив (**Array**) в JavaScript является глобальным объектом, который используется для создания массивов; которые представляют собой высокоуровневые спископодобные объекты.

Создание массива

```
var fruits = ['Яблоко', 'Банан'];

console.log(fruits.length);
// 2
```

Доступ к элементу массива по индексу

```
var first = fruits[0];
// Яблоко

var last = fruits[fruits.length - 1];
// Банан
```

Итерирование по массиву

```
fruits.forEach(function(item, index, array) {
  console.log(item, index);
});
// Яблоко 0
// Банан 1
```

Добавление элемента в конец массива

```
var newLength = fruits.push('Апельсин');
// ["Яблоко", "Банан", "Апельсин"]
```

Удаление последнего элемента массива

```
var last = fruits.pop(); // удалим Апельсин (из конца)
// ["Яблоко", "Банан"];
```

Удаление первого элемента массива

```
var first = fruits.shift(); // удалим Яблоко (из начала)
// ["Банан"];
```

Добавление элемента в начало массива

```
var newLength = fruits.unshift('Клубника') // добавляет в начало
// ["Клубника", "Банан"];
```

Поиск номера элемента в массиве

```
fruits.push('Манго');
// ["Клубника", "Банан", "Манго"]

var pos = fruits.indexOf('Банан');
// 1
```

Удаление элемента с определённым индексом

```
var removedItem = fruits.splice(pos, 1); // так можно удалить элемент

// ["Клубника", "Манго"]
```

Удаление нескольких элементов, начиная с определённого индекса

```
var vegetables = ['Капуста', 'Репа', 'Редиска', 'Морковка'];
console.log(vegetables);
// ["Капуста", "Репа", "Редиска", "Морковка"]

var pos = 1, n = 2;

var removedItems = vegetables.splice(pos, n);
// так можно удалить элементы, n определяет количество элементов для удаления,
```

```
// начиная с позиции(pos) и далее в направлении конца массива.

console.log(vegetables);
// ["Капуста", "Морковка"] (исходный массив изменён)

console.log(removedItems);
// ["Репа", "Редиска"]
```

Создание копии массива

```
var shallowCopy = fruits.slice(); // так можно создать копию массива
// ["Клубника", "Манго"]
```

Синтаксис

```
[element0, element1, ..., elementN]
new Array(element0, element1[, ...[, elementN]])
new Array(arrayLength)
```

element*N*

Массив в JavaScript инициализируется с помощью переданных элементов, за исключением случая, когда в конструктор Array передаётся один аргумент и этот аргумент является числом (см. ниже). Стоит обратить внимание, что этот особый случай применяется только к JavaScript-массивам, создаваемым с помощью конструктора Array, а не к литеральным массивам, создаваемым с использованием скобочного синтаксиса.

arrayLength

Если конструктору Array передаётся единственный аргумент, являющийся целым числом в диапазоне от 0 до 2^{32}-1 (включительно), будет возвращён новый пустой JavaScript-массив, длина которого установится в это число (**примечание**: это означает массив, содержащий arrayLength пустых ячеек, а не ячеек со значениями undefined). Если аргументом будет любое другое число, возникнет исключение RangeError.

Описание

Массивы являются спископодобными объектами, чьи прототипы содержат методы для операций обхода и изменения массива. Ни размер JavaScript-массива, ни типы его элементов не являются фиксированными. Поскольку размер массива может увеличиваться и уменьшаться в любое время, то нет гарантии, что массив окажется плотным. То есть, при работе с массивом может возникнуть ситуация, что элемент массива, к которому вы обратитесь, будет пустым и вернёт undefined. В целом, это удобная характеристика; но если эта особенность массива не желательна в вашем специфическом случае, вы можете рассмотреть возможность использования типизированных массивов.
Некоторые полагают, что вы не должны использовать массив в качестве ассоциативного массива. В любом случае, вместо него вы можете использовать простые объекты, хотя у них есть и свои

подводные камни. Смотрите пост Легковесные JavaScript-словари с произвольными ключами*(англ.)* в качестве примера.

Доступ к элементам массива

Массивы в JavaScript индексируются с нуля: первый элемент массива имеет индекс, равный 0, а индекс последнего элемента равен значению свойства массива length минус 1.

```
var arr = ['первый элемент', 'второй элемент', 'последний элемент'];
console.log(arr[0]);              // напечатает 'первый элемент'
console.log(arr[1]);              // напечатает 'второй элемент'
console.log(arr[arr.length - 1]); // напечатает 'последний элемент'
```

Элементы массива являются свойствами, точно такими же, как, например, свойство toString, однако попытка получить элемент массива по имени его свойства приведёт к синтаксической ошибке, поскольку имя свойства не является допустимым именем JavaScript:

```
console.log(arr.0); // синтаксическая ошибка
```

Это не особенность массивов или их свойств. В JavaScript к свойствам, начинающимся с цифры, невозможно обратиться посредством точечной нотации; к ним можно обратиться только с помощью скобочной нотации. Например, если у вас есть объект со свойством, названным '3d', вы сможете обратиться к нему только посредством скобочной нотации. Примеры:

```
var years = [1950, 1960, 1970, 1980, 1990, 2000, 2010];
console.log(years.0);   // синтаксическая ошибка
console.log(years[0]);  // работает как положено
renderer.3d.setTexture(model, 'character.png');     // синтаксическая ошибка
renderer['3d'].setTexture(model, 'character.png');  // работает как положено
```

Обратите внимание, что во втором примере 3d заключено в кавычки: '3d'. Индексы можно заключать в кавычки (например years['2'] вместо years[2]), но в этом нет необходимости. Значение 2 в выражении years[2] будет неявно приведено к строке движком JavaScript через метод преобразования toString. Именно по этой причине ключи '2' и '02' будут ссылаться на два разных элемента в объекте years и следующий пример выведет true:

```
console.log(years['2'] != years['02']);
```

Аналогично, к свойствам объекта, являющимся зарезервированными словами(!) можно получить доступ только посредством скобочной нотации:

```
var promise = {
  'var'  : 'text',
  'array': [1, 2, 3, 4]
};

console.log(promise['array']);
```

Взаимосвязь свойства length с числовыми свойствами

Свойство массивов length взаимосвязано с числовыми свойствами. Некоторые встроенные методы массива (например, join, slice, indexOf и т.д.) учитывают значение свойства length при своём вызове. Другие методы (например, push, splice и т.д.) в результате своей работы также обновляют свойство length массива.

```
var fruits = [];
fruits.push('банан', 'яблоко', 'персик');
```

```
console.log(fruits.length); // 3
```

При установке свойства в массиве, если свойство имеет действительный индекс и этот индекс выходит за пределы текущих границ массива, движок соответствующим образом обновит свойство length:

```
fruits[5] = 'манго';
console.log(fruits[5]);           // 'манго'
console.log(Object.keys(fruits)); // ['0', '1', '2', '5']
console.log(fruits.length);       // 6
```

Увеличиваем свойство length

```
fruits.length = 10;
console.log(Object.keys(fruits)); // ['0', '1', '2', '5']
console.log(fruits.length);       // 10
```

Однако, уменьшение свойства length приведёт к удалению элементов.

```
fruits.length = 2;
console.log(Object.keys(fruits)); // ['0', '1']
console.log(fruits.length); // 2
```

Более подробно эта тема освещена на странице, посвящённой свойству Array.length.

Создание массива с использованием результата сопоставления

Результатом сопоставления регулярного выражения строке является JavaScript-массив. Этот массив имеет свойства и элементы, предоставляющие информацию о сопоставлении. Подобные массивы возвращаются методами RegExp.exec, String.match и String.replace. Чтобы было проще понять, откуда и какие появились свойства и элементы, посмотрите следующий пример и обратитесь к таблице ниже:

```
// Сопоставляется с одним символом d, за которым следует один
// или более символов b, за которыми следует один символ d
// Запоминаются сопоставившиеся символы b и следующий за ними символ d
// Регистр игнорируется

var myRe = /d(b+)(d)/i;
var myArray = myRe.exec('cdbBdbsbz');
```

Свойства и элементы, возвращаемые из данного сопоставления, описаны ниже:

Свойство/Элемент	Описание	Пример
input	Свойство только для чтения, отражающее оригинальную строку, с которой сопоставлялось регулярное выражение.	cdbBdbsbz
index	Свойство только для чтения, являющееся индексом (отсчёт начинается с нуля) в строке, с которого началось сопоставление.	1
[0]	Элемент только для чтения, определяющий последние сопоставившиеся символы.	dbBd

[1], ...[n]	Элементы только для чтения, определяющие сопоставившиеся подстроки, заключённые в круглые скобки, если те включены в регулярное выражение. Количество возможных подстрок не ограничено.	[1]: bB [2]: d

---

Свойства

Array.length

Значение свойства length конструктора массива равно 1.

Array.prototype

Позволяет добавлять свойства ко всем объектам массива.

Методы

Array.from()

Создаёт новый экземпляр Array из массивоподобного или итерируемого объекта.

Array.isArray()

Возвращает true, если значение является массивом, иначе возвращает false.

Array.observe()

Асинхронно наблюдает за изменениями в массиве, подобно методу Object.observe() для объектов. Метод предоставляет поток изменений в порядке их возникновения.

Array.of()

Создаёт новый экземпляр Array из любого количества аргументов, независимо от их количества или типа.

Экземпляры массива

Все экземпляры массива наследуются от Array.prototype. Изменения в объекте прототипа конструктора массива затронет все экземпляры Array.

Свойства

Array.prototype.constructor

Определяет функцию, создающую прототип объекта.

Array.prototype.length

Отражает количество элементов в массиве.

Методы

Методы изменения

Эти методы изменяют массив:

Array.prototype.copyWithin()

Копирует последовательность элементов массива внутри массива.

Array.prototype.fill()

Заполняет все элементы массива от начального индекса до конечного индекса указанным значением.

Array.prototype.pop()

Удаляет последний элемент из массива и возвращает его.

Array.prototype.push()

Добавляет один или более элементов в конец массива и возвращает новую длину массива.

Array.prototype.reverse()

Переворачивает порядок элементов в массиве — первый элемент становится последним, а последний — первым.

Array.prototype.shift()

Удаляет первый элемент из массива и возвращает его.

Array.prototype.sort()

Сортирует элементы массива на месте и возвращает отсортированный массив.

Array.prototype.splice()

Добавляет и/или удаляет элементы из массива.

Array.prototype.unshift()

Добавляет один или более элементов в начало массива и возвращает новую длину массива.

Методы доступа

Эти методы не изменяют массив, а просто возвращают его в ином представлении.

Array.prototype.concat()

Возвращает новый массив, состоящий из данного массива, соединённого с другим массивом и/или значением (списком массивов/значений).

Array.prototype.includes()

Определяет, содержится ли в массиве указанный элемент, возвращая, соответственно, true или false.

Array.prototype.join()

Объединяет все элементы массива в строку.

Array.prototype.slice()

Извлекает диапазон значений и возвращает его в виде нового массива.

Array.prototype.toSource()

Возвращает литеральное представление указанного массива; вы можете использовать это значение для создания нового массива. Переопределяет метод Object.prototype.toSource().

Array.prototype.toString()

Возвращает строковое представление массива и его элементов. Переопределяет метод Object.prototype.toString().

Array.prototype.toLocaleString()

Возвращает локализованное строковое представление массива и его элементов. Переопределяет метод Object.prototype.toLocaleString().

Array.prototype.indexOf()

Возвращает первый (наименьший) индекс элемента внутри массива, равный указанному значению; или -1, если значение не найдено.

Array.prototype.lastIndexOf()

Возвращает последний (наибольший) индекс элемента внутри массива, равный указанному значению; или -1, если значение не найдено.

Методы обхода

Некоторые методы принимают в качестве аргументов функции, вызываемые при обработке массива. Когда вызываются эти методы, достаётся длина массива, и любой элемент, добавленный свыше этой длины изнутри функции обратного вызова не посещается. Другие изменения в массиве (установка значения или удаление элемента) могут повлиять на результаты операции, если изменённый элемент метод посещает после изменения. Хотя специфическое поведение этих методов в таких случаях хорошо определено, вы не должны на него полагаться, чтобы не запутывать других людей, читающих ваш код. Если вам нужно изменить массив, лучше вместо этого скопируйте его в новый массив.

Array.prototype.forEach()

Вызывает функцию для каждого элемента в массиве.

Array.prototype.entries()

Возвращает новый объект итератора массива Array Iterator, содержащий пары ключ / значение для каждого индекса в массиве.

Array.prototype.every()

Возвращает true, если каждый элемент в массиве удовлетворяет условию проверяющей функции.

Array.prototype.some()

Возвращает true, если хотя бы один элемент в массиве удовлетворяет условию проверяющей функции.

Array.prototype.filter()

Создаёт новый массив со всеми элементами этого массива, для которых функция фильтрации возвращает true.

Array.prototype.find()

Возвращает искомое значение в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или undefined, если такое значение не найдено.

Array.prototype.findIndex()

Возвращает искомый индекс в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или -1, если такое значение не найдено.

Array.prototype.keys()

Возвращает новый итератор массива, содержащий ключи каждого индекса в массиве.

Array.prototype.map()

Создаёт новый массив с результатами вызова указанной функции на каждом элементе данного массива.

Array.prototype.reduce()

Применяет функцию к аккумулятору и каждому значению массива (слева-направо), сводя его к одному значению.

Array.prototype.reduceRight()

Применяет функцию к аккумулятору и каждому значению массива (справа-налево), сводя его к одному значению.

Array.prototype.values()

Возвращает новый объект итератора массива Array Iterator, содержащий значения для каждого индекса в массиве.

Array.prototype[@@iterator]()

Возвращает новый объект итератора массива Array Iterator, содержащий значения для каждого индекса в массиве.

Общие методы массива

Иногда хочется применить методы массива к строкам или другим массивоподобным объектам (например, к аргументам функции). Делая это, вы трактуете строку как массив символов (другими словами, рассматриваете не-массив в качестве массива). Например, в порядке проверки каждого символа в переменной *str* на то, что он является буквой (латинского алфавита), вы пишете следущий код:

```
function isLetter(character) {
  return character >= 'a' && character <= 'z';
}

if (Array.prototype.every.call(str, isLetter)) {
  console.log("Строка '" + str + "' содержит только (латинские) буквы!");
}
```

Эта запись довольно расточительна и в JavaScript 1.6 введён общий сокращённый вид:

```
if (Array.every(str, isLetter)) {
  console.log("Строка '" + str + "' содержит только (латинские) буквы!");
}
```

Общие методы также доступны для объекта String.
В настоящее время они не являются частью стандартов ECMAScript (хотя в ES2015 для достижения поставленной цели можно использовать Array.from()). Следующая прослойка позволяет использовать их во всех браузерах:

```
// Предполагаем, что дополнения массива уже присутствуют (для них так же можно использовать polyfill'ы)
(function() {
  'use strict';

  var i,
    // Мы могли построить массив методов следующим образом, однако метод
    //  getOwnPropertyNames() нельзя реализовать на JavaScript:
    // Object.getOwnPropertyNames(Array).filter(function(methodName) {
    //   return typeof Array[methodName] === 'function'
    // });
    methods = [
      'join', 'reverse', 'sort', 'push', 'pop', 'shift', 'unshift',
      'splice', 'concat', 'slice', 'indexOf', 'lastIndexOf',
      'forEach', 'map', 'reduce', 'reduceRight', 'filter',
      'some', 'every'
    ],
    methodCount = methods.length,
    assignArrayGeneric = function(methodName) {
      if (!Array[methodName]) {
        var method = Array.prototype[methodName];
        if (typeof method === 'function') {
          Array[methodName] = function() {
            return method.call.apply(method, arguments);
          };
        }
      }
    };

  for (i = 0; i < methodCount; i++) {
    assignArrayGeneric(methods[i]);
  }
}());
```

Примеры

Пример: создание массива

Следующий пример создаёт массив msgArray с длиной 0, присваивает значения элементам msgArray[0] и msgArray[99], что изменяет длину массива на 100.

```
var msgArray = [];
msgArray[0] = 'Привет';
msgArray[99] = 'мир';

if (msgArray.length === 100) {
  console.log('Длина равна 100.');
}
```

Пример: создание двумерного массива

Следующий код создаёт шахматную доску в виде двумерного массива строк. Затем он перемещает пешку путём копирования символа 'p' в позиции (6,4) на позицию (4,4). Старая позиция (6,4) затирается пустым местом.

```
var board = [
  ['R','N','B','Q','K','B','N','R'],
  ['P','P','P','P','P','P','P','P'],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  ['p','p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r'] ];

console.log(board.join('\n') + '\n\n');

// Двигаем королевскую пешку вперёд на две клетки
board[4][4] = board[6][4];
board[6][4] = ' ';
console.log(board.join('\n'));
```

Ниже показан вывод:

```
R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , , 
 , , , , , , , 
 , , , , , , , 
 , , , , , , , 
p,p,p,p,p,p,p,p
r,n,b,q,k,b,n,r

R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , , 
 , , , , , , , 
 , , , ,p, , , 
 , , , , , , , 
p,p,p,p, ,p,p,p
r,n,b,q,k,b,n,r
```

Спецификации

Спецификация	Статус	Комментарий
ECMAScript 1-е издание.	Стандарт	Изначальное определение.
ECMAScript 5.1 (ECMA-262) Определение 'Array' в этой спецификации.	Стандарт	Добавлены новые методы: Array.isArray, indexOf, lastIndexOf, every, some, forEach, map, filter, reduce, reduceRight.
ECMAScript 2015 (6th Edition, ECMA-262) Определение 'Array' в этой спецификации.	Стандарт	Добавлены новые методы: Array.from, Array.of, find, findIndex, fill, copyWithin.
ECMAScript 2016 (ECMA-262) Определение 'Array' в этой спецификации.	Стандарт	Добавлен новый метод: Array.prototype.includes()

Инструкции и объявления. For.

Введение

Выражение **for** создаёт цикл, состоящий из 3 необязательных выражений в круглых скобках, разделённых точками с запятой.

Синтаксис

```
for ([инициализация]; [условие]; [финальное выражение])выражение
```

инициализация

Выражение (в том числе выражения присвоения) или определение переменных. Обычно используется, чтобы инициализировать счётчик. Это выражение может опционально объявлять новые переменные с помощью ключевого слова var. Эти переменные видимы не только в цикле, т.е. в той же области области видимости, что и цикл for. Результат этого выражения отбрасывается.

условие

Выражение, выполняющееся на каждой интерации цикла. Если выражение истинно, цикл выполняется. Условие не является обязательным. Если его нет, условие всегда считается истиной. Если выражение ложно, выполнение переходит к первому выражению, следующему за for.

финальное выражение

Выражение, выполняющееся в конце интерации цикла. Происходит до следующего выполнения условия. Обычно используется для обновления или увеличения переменной счётчика.

выражение

Выражение, которое выполняется, когда условие цикла истинно. Чтоб выполнить множество выражений в цикле, используйте блок ({ ... }) для группировки этих выражений. Чтобы не выполнять никакого выражения в цикле, используйте пустое выражение (;).

Примеры

Использование for

Следующий цикл for начинается объявлением переменной i и задания ей значения 0. Затем проверяет, что i меньше девяти, выполняет выражения внутри цикла и увеличивает i на 1 каждый раз.

```
for (var i = 0; i < 9; i++) {
  console.log(i);
  // ещё какие-то выражения
}
```

Необязательные выражения в for

Все 3 выражения в цикле for не обязательны.
Например, в блоке инициализации не требуется определять переменные:

```
var i = 0;
for (; i < 9; i++) {
   console.log(i);
   // ещё выражения
```

```
}
```

Как и блок инициализации, блок условия не обязателен. Если пропустите это выражение, вы должны быть уверены, что прервете цикл где-то в теле, а не создадите бесконечный цикл.

```
for (var i = 0;; i++) {
  console.log(i);
  if (i > 3) break;
  // тут какой-то код
}
```

Вы можете пропустить все 3 блока. Снова убедитесь, что используете break, чтоб закончить цикл, а также изменить счётчик, так что условие для break было истинно в нужный момент.

```
var i = 0;

for (;;) {
  if (i > 3) break;
  console.log(i);
  i++;
}
```

Использование for без блока выражений

Следующий цикл for вычисляет смещение позиции узла в секции *[финальное выражение]*, и, следовательно, не требует использования выражения внутри цикла или блока, пустое выражение используется вместо этого.

```
function showOffsetPos (sId) {
  var nLeft = 0, nTop = 0;

  for (var oItNode = document.getElementById(sId); // инициализация
      oItNode; // условие
      nLeft += oItNode.offsetLeft, nTop += oItNode.offsetTop, oItNode = oItNode.offsetParent) //
финальное выражение
      /* пустое выражение */ ;

  console.log("Смещение позиции элемента \"" + sId + "\":\n left: " + nLeft + "px;\n top: " + nTop + "px;");
}

// Пример вызова:

showOffsetPos("content");

// Выводит:
// "Смещение позиции элемента "content":
// left: 0px;
// top: 153px;"
```

Замечание: В этом случае, когда вы не используете условие внутри цикла, **точка с запятой ставится сразу после выражения цикла.**

Инструкции и объявления. for...of
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for...of

Сводка

Оператор for...of выполняет цикл обхода итерируемых объектов (включая Array, Map, Set, объект аргументов и подобных), вызывая на каждом шаге итерации операторы для каждого значения из различных свойств объекта.

Синтаксис

```
for (variable of iterable) {
  statement
}
```

variable

На каждом шаге итерации *variable* присваивается значение нового свойства объекта iterable. Переменная variable может быть также объявлена с помощью const, let или var.

iterable

Объект, перечисляемые свойства которого обходятся во время выполнения цикла.

Примеры

Обход Array

```
let iterable = [10, 20, 30];

for (let value of iterable) {
  value += 1;
  console.log(value);
}
// 11
// 21
// 31
```

Можно также использовать const вместо let, если не нужно переназначать переменные внутри блока.

```
let iterable = [10, 20, 30];

for (const value of iterable) {
  console.log(value);
}
// 10
// 20
```

```
// 30
```

Обход String

```
let iterable = 'boo';

for (let value of iterable) {
  console.log(value);
}
// "b"
// "o"
// "o"
```

Обход TypedArray

```
let iterable = new Uint8Array([0x00, 0xff]);

for (let value of iterable) {
  console.log(value);
}
// 0
// 255
```

Обход Map

```
let iterable = new Map([['a', 1], ['b', 2], ['c', 3]]);

for (let entry of iterable) {
  console.log(entry);
}
// ['a', 1]
// ['b', 2]
// ['c', 3]

for (let [key, value] of iterable) {
  console.log(value);
}
// 1
// 2
// 3
```

Обход Set

```
let iterable = new Set([1, 1, 2, 2, 3, 3]);

for (let value of iterable) {
  console.log(value);
}
// 1
// 2
// 3
```

Обход объекта arguments

```
(function() {
```

```
  for (let argument of arguments) {
    console.log(argument);
  }
})(1, 2, 3);

// 1
// 2
// 3
```

Обход DOM коллекций

Обход DOM коллекций наподобие NodeList: следующий пример добавляет класс read параграфам, являющимся непосредственными потомками статей:

```
// Примечание: работает только на платформах, где
// реализован NodeList.prototype[Symbol.iterator]
let articleParagraphs = document.querySelectorAll('article > p');

for (let paragraph of articleParagraphs) {
  paragraph.classList.add('read');
}
```

Закрытие итераторов

В циклах for...of аварийный выход осуществляется через break, throw или return. Во всех вариантах итератор завершается.

```
function* foo(){
  yield 1;
  yield 2;
  yield 3;
};

for (let o of foo()) {
  console.log(o);
  break; //  итератор закрывается, возврат
}
```

Обход генераторов

Вы можете выполнять обход генераторов, вот пример:

```
function* fibonacci() { // функция-генератор
    let [prev, curr] = [0, 1];
    for (;;) {
        [prev, curr] = [curr, prev + curr];
        yield curr;
    }
}

for (let n of fibonacci()) {
    // ограничивает последовательность на 1000
```

```
    if (n > 1000)
      break;
    console.log(n);
}
```

Не пытайтесь повторно использовать генератор

Генераторы нельзя использовать дважды, даже если цикл for...of завершится аварийно, например, через оператор break . При выходе из цикла генератор завершается, и любые попытки получить из него значение обречены.

```
var gen = (function *(){
  yield 1;
  yield 2;
  yield 3;
})();
for (let o of gen) {
  console.log(o);
  break;  // Завешение обхода
}

// Генератор нельзя повторно обойти, следующее не имеет смысла!
for (let o of gen) {
  console.log(o); // Не будет вызван
}
```

Обход итерируемых объектов

Кроме того, можно сделать обход объекта, явно реализующего iterable:

```
var iterable = {
  [Symbol.iterator]() {
    return {
      i: 0,
      next() {
        if (this.i < 3) {
          return { value: this.i++, done: false };
        }
        return { value: undefined, done: true };
      }
    };
  }
};

for (var value of iterable) {
  console.log(value);
}
// 0
// 1
// 2
```

Различия между for...of и for...in

Оба оператора, и for...in и for...of производят обход объектов . Разница в том, как они это делают.
Для for...in обход перечисляемых свойств объекта осуществляется в произвольном порядке.
Для for...of обход происходит в соответствии с тем, какой порядок определен в итерируемом объекте.
Следующий пример показывает различия в работе циклов for...of и for...in при обходе Array.

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];
iterable.foo = 'hello';

for (let i in iterable) {
  console.log(i); // выведет 0, 1, 2, "foo", "arrCustom", "objCustom"
}

for (let i in iterable) {
  if (iterable.hasOwnProperty(i)) {
    console.log(i); // выведет 0, 1, 2, "foo"
  }
}

for (let i of iterable) {
  console.log(i); // выведет 3, 5, 7
}
```

Разберемся шаг за шагом в вышеописанном коде.

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];
iterable.foo = 'hello';
```

Каждый объект унаследует метод objCustom и каждый массив Array унаследует метод arrCustom благодаря созданию их в Object.prototype и Array.prototype.
Объект iterable унаследует методы objCustom и arrCustom из-за наследования через прототип.

```
for (let i in iterable) {
  console.log(i); // выведет 0, 1, 2, "foo", "arrCustom", "objCustom"
}
```

Цикл выводит только перечисляемые свойства объекта iterable, в порядке их создания. Он не выводит **значения** 3, 5, 7 и hello поскольку они **не являются** перечисляемыми. Выводятся же **имена свойств и методов**, например arrCustom и objCustom. Если вы еще не совсем поняли, по каким свойствам осуществляется обход, вот дополнительное объяснение того, как работает array iteration and for...in .

```
for (let i in iterable) {
  if (iterable.hasOwnProperty(i)) {
    console.log(i); // выведет 0, 1, 2, "foo"
  }
}
```

Цикл аналогичен предыдущему, но использует hasOwnProperty() для проверки того, собственное ли это свойство объекта или унаследованое. Выводятся только собственные свойства. Имена 0, 1, 2 и foo принадлежат только экземпляру объекта (**не унаследованы**). Методы arrCustom и objCustom не выводятся, поскольку они **унаследованы**.

```
for (let i of iterable) {
  console.log(i); // выведет 3, 5, 7
}
```

Этот цикл обходит iterable и выводит те значения итерируемого объекта, которые определены в способе его перебора, т.е. не свойства объекта, а значения массива 3, 5, 7 .

Array.prototype.includes()

Метод **includes()** определяет, содержит ли массив определённый элемент, возвращая в зависимости от этого true или false.

Синтаксис

```
arr.includes(searchElement[, fromIndex = 0])
```

Параметры

searchElement

Искомый элемент.

fromIndex Необязательный

Позиция в массиве, с которой начинать поиск элемента searchElement. При отрицательных значениях поиск производится начиная с индекса array.length + fromIndex по возрастанию. Значение по умолчанию равно 0.

Возвращаемое значение

Boolean.

Примеры

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
[1, 2, NaN].includes(NaN); // true
```

fromIndex больше или равен длине массива

Если fromIndex больше или равен длине массива, то возвращается false. При этом поиск не производится.

```
var arr = ['a', 'b', 'c'];

arr.includes('c', 3);   // false
arr.includes('c', 100); // false
```

Вычисленный индекс меньше нуля 0

Если fromIndex отрицательный, то вычисляется индекс, начиная с которого будет производиться поиск элемента searchElement. Если вычисленный индекс меньше нуля, то поиск будет производиться во всём массиве.

```
// длина массива равна 3
// fromIndex равен -100
// вычисленный индекс равен 3 + (-100) = -97

var arr = ['a', 'b', 'c'];

arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
```

Использование includes() в качестве общих метода

includes() специально сделан общим. Он не требует, чтобы this являлся массивом, так что он может быть применён к другим типам объектов (например, к массивоподобным объектам). Пример ниже показывает использование метода includes() на объекте arguments.

```
(function() {
  console.log([].includes.call(arguments, 'a')); // true
  console.log([].includes.call(arguments, 'd')); // false
})('a','b','c');
```

Полифилл

```
// https://tc39.github.io/ecma262/#sec-array.prototype.includes
if (!Array.prototype.includes) {
  Object.defineProperty(Array.prototype, 'includes', {
    value: function(searchElement, fromIndex) {

      if (this == null) {
        throw new TypeError('"this" is null or not defined');
      }

      // 1. Let O be ? ToObject(this value).
      var o = Object(this);

      // 2. Let len be ? ToLength(? Get(O, "length")).
```

```
    var len = o.length >>> 0;

    // 3. If len is 0, return false.
    if (len === 0) {
      return false;
    }

    // 4. Let n be ? ToInteger(fromIndex).
    //    (If fromIndex is undefined, this step produces the value 0.)
    var n = fromIndex | 0;

    // 5. If n ≥ 0, then
    //  a. Let k be n.
    // 6. Else n < 0,
    //  a. Let k be len + n.
    //  b. If k < 0, let k be 0.
    var k = Math.max(n >= 0 ? n : len - Math.abs(n), 0);

    function sameValueZero(x, y) {
      return x === y || (typeof x === 'number' && typeof y === 'number' && isNaN(x) && isNaN(y));
    }

    // 7. Repeat, while k < len
    while (k < len) {
      // a. Let elementK be the result of ? Get(O, ! ToString(k)).
      // b. If SameValueZero(searchElement, elementK) is true, return true.
      if (sameValueZero(o[k], searchElement)) {
        return true;
      }
      // c. Increase k by 1.
      k++;
    }

    // 8. Return false
    return false;
  }
 });
}
```

Если требуется поддержка устаревших движков JavaScript, которые не поддерживают Object.defineProperty, наилучшим решением будет вообще не делать полифилл для методов Array.prototype, так как не получится сделать их неперечисляемыми.

Array.prototype.flat()

Метод **flat()** возвращает новый массив, в котором все элементы вложенных подмассивов были рекурсивно "подняты" на указанный уровень depth.

Синтаксис

```
var newArray = arr.flat(depth);
```

Параметры

depth Необязательный

На сколько уровней вложенности уменьшается мерность исходного массива. По-умолчанию 1.

Возвращаемое значение

Новый массив с объединенными в него подмассивами.

Примеры

Упрощение вложенных массивов

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);
// [1, 2, 3, 4, 5, 6]

var arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Упрощение и "дырки" в массивах

Метод flat удаляет пустые слоты из массива:

```
var arr4 = [1, 2, , 4, 5];
arr4.flat();
// [1, 2, 4, 5]
```

Альтернативы

reduce и concat

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();

//В одномерный массив
arr1.reduce((acc, val) => acc.concat(val), []);// [1, 2, 3, 4]

//или
const flatSingle = arr => [].concat(...arr);
//для глубокого упрощения используем рекурсивно reduce и concat
var arr1 = [1,2,3,[1,2,3,4, [2,3,4]]];

function flattenDeep(arr1) {
   return arr1.reduce((acc, val) => Array.isArray(val) ? acc.concat(flattenDeep(val)) : acc.concat(val), []);
}
flattenDeep(arr1);// [1, 2, 3, 1, 2, 3, 4, 2, 3, 4]

//не рекурсивное упрощение с использованием стэка
var arr1 = [1,2,3,[1,2,3,4, [2,3,4]]];
function flatten(input) {
  const stack = [...input];
  const res = [];
  while (stack.length) {
    // забираем последнее значение
    const next = stack.pop();
    if (Array.isArray(next)) {
      // добавляем к массиву элементы не модифицируя исходное значение
      stack.push(...next);
    } else {
      res.push(next);
    }
  }
  //разворачиваем массив, чтобы восстановить порядок элементов
  return res.reverse();
}
flatten(arr1);// [1, 2, 3, 1, 2, 3, 4, 2, 3, 4]
//рекурсивно упрощаем массив
function flatten(array) {
  var flattend = [];
  (function flat(array) {
    array.forEach(function(el) {
      if (Array.isArray(el)) flat(el);
      else flattend.push(el);
    });
  })(array);
  return flattend;
}
```

Array.prototype.sort()

Сводка

Метод **sort()** *на месте* сортирует элементы массива и возвращает отсортированный массив. Сортировка не обязательно устойчива (англ.). Порядок сортировки по умолчанию соответствует порядку кодовых точек Unicode.

Синтаксис

```
arr.sort([compareFunction])
```

Параметры

compareFunction

Необязательный параметр. Указывает функцию, определяющую порядок сортировки. Если опущен, массив сортируется в соответствии со значениями кодовых точек каждого символа Unicode, полученных путём преобразования каждого элемента в строку.

Возвращаемое значение

Отсортированный массив. Так как массив сортируется *на месте* не нужно создавать новую переменную.

Описание

Если функция сравнения compareFunction не предоставляется, элементы сортируются путём преобразования их в строки и сравнения строк в порядке следования кодовых точек Unicode. Например, слово "Вишня" идёт перед словом "бананы". При числовой сортировке, 9 идёт перед 80, но поскольку числа преобразуются в строки, то "80" идёт перед "9" в соответствии с порядком в Unicode.

```
var fruit = ['арбузы', 'бананы', 'Вишня'];
fruit.sort(); // ['Вишня', 'арбузы', 'бананы']

var scores = [1, 2, 10, 21];
scores.sort(); // [1, 10, 2, 21]

var things = ['слово', 'Слово', '1 Слово', '2 Слова'];
things.sort(); // ['1 Слово', '2 Слова', 'Слово', 'слово']
// В Unicode, числа находятся перед буквами в верхнем регистре,
// а те, в свою очередь, перед буквами в нижнем регистре.
```

Если функция сравнения compareFunction предоставлена, элементы массива сортируются в соответствии с её возвращаемым значением. Если сравниваются два элемента a и b, то:

- Если compareFunction(a, b) меньше 0, сортировка поставит a по меньшему индексу, чем b, то есть, a идёт первым.

- Если compareFunction(a, b) вернёт 0, сортировка оставит a и b неизменными по отношению друг к другу, но отсортирует их по отношению ко всем другим элементам. Обратите внимание: стандарт ECMAscript не гарантирует данное поведение, и ему следуют не все браузеры (например, версии Mozilla по крайней мере, до 2003 года).
- Если compareFunction(a, b) больше 0, сортировка поставит b по меньшему индексу, чем a.
- Функция compareFunction(a, b) должна всегда возвращать одинаковое значение для определённой пары элементов a и b. Если будут возвращаться непоследовательные результаты, порядок сортировки будет не определён.

Итак, функция сравнения имеет следующую форму:

```
function compare(a, b) {
  if (a меньше b по некоторому критерию сортировки) {
    return -1;
  }
  if (a больше b по некоторому критерию сортировки) {
    return 1;
  }
  // a должно быть равным b
  return 0;
}
```

Для числового сравнения, вместо строкового, функция сравнения может просто вычитать b из a. Следующая функция будет сортировать массив по возрастанию:

```
function compareNumbers(a, b) {
  return a - b;
}
```

Метод sort можно удобно использовать с функциональными выражениями (и замыканиями):

```
var numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) {
  return a - b;
});
console.log(numbers); // [1, 2, 3, 4, 5]
```

Объекты могут быть отсортированы по значению одного из своих свойств.

```
var items = [
  { name: 'Edward', value: 21 },
  { name: 'Sharpe', value: 37 },
  { name: 'And', value: 45 },
  { name: 'The', value: -12 },
  { name: 'Magnetic' },
  { name: 'Zeros', value: 37 }
];
items.sort(function (a, b) {
  if (a.name > b.name) {
    return 1;
  }
  if (a.name < b.name) {
    return -1;
  }
```

```
  // a должно быть равным b
  return 0;
});
```

Пример: создание, отображение и сортировка массива

В следующем примере создаётся четыре массива, сначала отображается первоначальный массив, а затем они сортируются. Числовые масивы сортируются сначала без, а потом с функцией сравнения.

```
var stringArray = ['Голубая', 'Горбатая', 'Белуга'];
var numericStringArray = ['80', '9', '700'];
var numberArray = [40, 1, 5, 200];
var mixedNumericArray = ['80', '9', '700', 40, 1, 5, 200];

function compareNumbers(a, b) {
  return a - b;
}

// снова предполагаем, что функция печати определена
console.log('stringArray:', stringArray.join());
console.log('Сортировка:', stringArray.sort());

console.log('numberArray:', numberArray.join());
console.log('Сортировка без функции сравнения:', numberArray.sort());
console.log('Сортировка с функцией compareNumbers:', numberArray.sort(compareNumbers));

console.log('numericStringArray:', numericStringArray.join());
console.log('Сортировка без функции сравнения:', numericStringArray.sort());
console.log('Сортировка с функцией compareNumbers:', numericStringArray.sort(compareNumbers));

console.log('mixedNumericArray:', mixedNumericArray.join());
console.log('Сортировка без функции сравнения:', mixedNumericArray.sort());
console.log('Сортировка с функцией compareNumbers:', mixedNumericArray.sort(compareNumbers));
```

Этот пример произведёт следующий вывод. Как показывает вывод, когда используется функция сравнения, числа сортируются корректно вне зависимости от того, являются ли они собственно числами или строками с числами.

```
stringArray: Голубая,Горбатая,Белуга
Сортировка: Белуга,Голубая,Горбатая

numberArray: 40,1,5,200
Сортировка без функции сравнения: 1,200,40,5
Сортировка с функцией compareNumbers: 1,5,40,200

numericStringArray: 80,9,700
Сортировка без функции сравнения: 700,80,9
Сортировка с функцией compareNumbers: 9,80,700

mixedNumericArray: 80,9,700,40,1,5,200
Сортировка без функции сравнения: 1,200,40,5,700,80,9
```

Сортировка с функцией compareNumbers: 1,5,9,40,80,200,700

Пример: сортировка не-ASCII символов

Для сортировки строк с не-ASCII символами, то есть строк с символами акцента (e, é, è, a, ä и т.д.), строк, с языками, отличными от английского: используйте String.localeCompare. Эта функция может сравнивать эти символы, чтобы они становились в правильном порядке.

```
var items = ['réservé', 'premier', 'cliché', 'communiqué', 'café', 'adieu'];
items.sort(function (a, b) {
  return a.localeCompare(b);
});

// items равен ['adieu', 'café', 'cliché', 'communiqué', 'premier', 'réservé']
```

Пример: сортировка с помощью map

Функция сравнения (compareFunction) может вызываться несколько раз для каждого элемента в массиве. В зависимости от природы функции сравнения, это может привести к высоким расходам ресурсов. Чем более сложна функция сравнения и чем больше элементов требуется отсортировать, тем разумнее использовать map для сортировки. Идея состоит в том, чтобы обойти массив один раз, чтобы извлечь фактические значения, используемые для сортировки, во временный массив, отсортировать временный массив, а затем обойти временный массив для получения правильного порядка.

```
// массив для сортировки
var list = ['Дельта', 'альфа', 'ЧАРЛИ', 'браво'];

// временный массив содержит объекты с позицией и значением сортировки
var mapped = list.map(function(el, i) {
return { index: i, value: el.toLowerCase() };
});

// сортируем массив, содержащий уменьшенные значения
mapped.sort(function(a, b) {
  if (a.value > b.value) {
    return 1; }
  if (a.value < b.value) {
    return -1; }
  return 0;
});

// контейнер для результа
var result = mapped.map(function(el) {
  return list[el.index];
});
```

Список вопросов для самоконтроля.

1. В чем разница между null и undefined?

2. Для чего используется оператор "&&"?

3. Для чего используется оператор "||"?

4. Является ли использование унарного плюса (оператор "+") самым быстрым способом преобразования строки в число?

5. Что такое DOM?

6. Что такое распространение события (Event Propogation)?

7. Что такое всплытие события (Event Bubbling)?

8. Что такое погружение события (Event Capturing)?

9. В чем разница между методами event.preventDefault() и event.stopPropagation()?

10. Как узнать об использовании метода event.preventDefault()?

11. Почему obj.someprop.x приводит к ошибке?

12. Что такое цель события или целевой элемент (event.target)?

13. Что такое текущая цель события (event.currentTarget)?

14. В чем разница между операторами "==" и "==="?

15. Почему результатом сравнения двух похожих объектов является false?

16. Для чего используется оператор "!!"?

17. Как записать несколько выражений в одну строку?

18. Что такое поднятие (Hoisting)?

19. Что такое область видимости (Scope)?

20. Что такое замыкание (Closures)?

21. Какие значения в JS являются ложными?

22. Как проверить, является ли значение ложным?

23. Для чего используется директива «use strict»?

24. Какое значение имеет this?

25. Что такое прототип объекта?

26. Что такое IIFE?

27. Для чего используется метод Function.prototype.apply?

28. Для чего используется метод Function.prototype.call?

29. В чем разница между методами call и apply?

30. Для чего используется метод Function.prototype.bind?

31. Что такое функциональное программирование и какие особенности JS позволяют говорить о нем как о функциональном языке программирования?

32. Что такое функции высшего порядка (Higher Order Functions)?

33. Почему функции в JS называют объектами первого класса (First-class Objects)?

34. Как бы Вы реализовали метод Array.prototype.map?

35. Как бы Вы реализовали метод Array.prototype.filter?

36. Как бы Вы реализовали метод Array.prototype.reduce?

37. Что такое объект arguments?

38. Как создать объект, не имеющий прототипа?

39. Почему в представленном коде переменная b становится глобальной при вызове функции?

40. Что такое ECMAScript?

41. Что нового привнес в JS стандарт ES6 или ECMAScript2015?

42. В чем разница между ключевыми словами «var», «let» и «const»?

43. Что такое стрелочные функции (Arrow Functions)?

44. Что такое классы (Classes)?

45. Что такое шаблонные литералы (Template Literals)?

46. Что такое деструктуризация объекта (Object Destructuring)?

47. Что такое модули (Modules)?

48. Что такое объект Set?

49. Что такое функция обратного вызова (Callback Function)?

50. Что такое промисы (Promises)?

51. Что такое async/await?

52. В чем разница между spread-оператором и rest-оператором?

53. Что такое параметры по умолчанию (Default Parameters)?

54. Что такое объектная обертка (Wrapper Objects)?

55. В чем разница между явным и неявным преобразованием или приведением к типу (Implicit and Explicit Coercion)?

56. Что такое NaN? Как проверить, является ли значение NaN?

57. Как проверить, является ли значение массивом?

58. Как проверить, что число является четным, без использования деления по модулю или деления с остатком (оператора "%")?

59. Как определить наличие свойства в объекте?

60. Что такое AJAX?

61. Как в JS создать объект?

62. В чем разница между методами Object.freeze и Object.seal?

63. В чем разница между оператором «in» и методом hasOwnProperty?

64. Какие приемы работы с асинхронным кодом в JS Вы знаете?

65. В чем разница между обычной функцией и функциональным выражением?

66. Как в JS вызвать функцию?

67. Что такое запоминание или мемоизация (Memoization)?

68. Как бы Вы реализовали вспомогательную функцию запоминания?

69. Почему typeof null возвращает object? Как проверить, является ли значение null?

70. Для чего используется ключевое слово «new»?

Список источников

1. Бангал, Шэм ActionScript. Основы / Шэм Бангал. - М.: Символ-плюс, 2015. - 455 с.
2. Бер, Бибо jQuery. Подробное руководство по продвинутому JavaScript / Бибо Бер. - М.: Символ-плюс, 2015. - 243 с.
3. Вуд, Кит Расширение библиотеки jQuery / Кит Вуд. - М.: ДМК Пресс, 2018. - 184 с.
4. Дронов, В. JavaScript в Web-дизайне / В. Дронов. - М.: БХВ-Петербург, 2017. - 880 с.
5. Дронов, Владимир JavaScript. Народные советы / Владимир Дронов. - М.: БХВ-Петербург, 2015. - 928 с.
6. Дунаев, Вадим HTML, скрипты и стили / Вадим Дунаев. - М.: БХВ-Петербург, 2015. - 816 с.
7. Дунаев, Вадим HTML, скрипты и стили Уцененный товар (№1) / Вадим Дунаев. - М.: БХВ-Петербург, 2015. - 816 с.
8. Закас, Н. JavaScript для профессиональных веб-разработчиков / Н. Закас. - М.: Питер, 2015. - 831 с.
9. Клименко, Роман Веб-мастеринг на 100% / Роман Клименко. - М.: Питер, 2015. - 747 с.
10. Климов, Александр JavaScript на примерах / Александр Климов. - М.: БХВ-Петербург, 2018. - 336 с.

11. Крокфорд, Д. JavaScript. Сильные стороны / Д. Крокфорд. - М.: Питер, 2018. - 199 с.
12. Макфарланд, Дэвид JavaScript и jQuery. Исчерпывающее руководство (+ DVD-ROM) / Дэвид Макфарланд. - М.: Эксмо, 2017. - 688 с.
13. Макфарланд, Дэвид JavaScript. Подробное руководство / Дэвид Макфарланд. - М.: Эксмо, 2015. - 608 с.
14. Машнин, Тимур Web-сервисы Java / Тимур Машнин. - М.: БХВ-Петербург, 2017. - 560 с.
15. Минник, Крис JavaScript для чайников / Крис Минник , Ева Холланд. - М.: Вильямс, 2016. - 320 с.
16. Николас, Закас JavaScript. Оптимизация производительности / Закас Николас. - М.: Символ-плюс, 2016. - 482 с.
17. Никсон, Р. Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML5 / Р. Никсон. - М.: Питер, 2015. - 228 с.
18. Османи, Эдди Разработка Backbone.js приложений / Эдди Османи. - М.: Питер, 2018. - 366 с.
19. Роббинс, Дженнифер HTML5, CSS3 и JavaScript. Исчерпывающее руководство (+ DVD-ROM) / Дженнифер Роббинс. - М.: Эксмо, 2018. - 528 с.
20. Симпсон, Кайл ES6 и не только / Кайл Симпсон. - М.: Питер, 2017. - 336 с.
21. Слепцова, Л. Д. JavaScript. Самоучитель (+ CD-ROM) / Л.Д. Слепцова, Ю.М. Бидасюк. - М.: Вильямс, 2017. - 448 с.
22. Сухов, К. К. Node.js. Путеводитель по технологии / К.К. Сухов. - М.: ДМК Пресс, 2015. - 369 с.
23. Флэнаган, Дэвид JavaScript. Карманный справочник / Дэвид Флэнаган. - М.: Вильямс, 2015. - 320 с.
24. Херман, Д. Сила JavaScript. 68 способов эффективного использования JS / Д. Херман. - М.: Питер, 2016. - 907 с.
25. Хэррон, Дэвид Node.js Разработка серверных веб-приложений на JavaScript / Дэвид Хэррон. - М.: ДМК Пресс, 2016. - 862 с.
26. Хэррон, Дэвид Node.js. Разработка серверных приложений на JavaScript / Дэвид Хэррон. - М.: ДМК Пресс, 2016. - 144 с.
27. Чекко, Рафаэлло Графика на JavaScript / Рафаэлло Чекко. - М.: Питер, 2017. - 733 с.
28. Шпильман, Сью JSTL. Практическое руководство для JSP-программистов / Сью Шпильман. - М.: КУДИЦ-Образ, 2018. - 272 с.
29. Штефен, Вальтер Создание приложений для Windows 8 с использованием HTML5 и JavaScript / Вальтер Штефен. - М.: ДМК Пресс, 2015. - 871 с.
30. Эспозито, Дино Разработка приложений для Windows 8 на HTML5 и JavaScript / Дино Эспозито. - М.: Питер, 2015. - 210 с.

Учебное издание

С.Е.Попов

JavaScript.Основы программирования.
Учебно-методическое пособие

Техническое исполнение – В. М. Гришин
Книга печатается в авторской редакции

Лицензия на издательскую деятельность
ИД № 06146. Дата выдачи 26.10.01.
Формат 60 x 84 /16. Гарнитура Times. Печать трафаретная.
Печ.л. 4,4 Уч.-изд.л. 4,1
Тираж 300 экз. (1-й завод 1-25 экз.). Заказ 106

Отпечатано с готового оригинал-макета на участке оперативной полиграфии
Елецкого государственного университета им. И. А. Бунина

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«Елецкий государственный университет им. И.А. Бунина»
399770, г. Елец, ул. Коммунаров, 28,1