Введение

~ 2 минуты

Глава 3.1. Массивы и их методы

Массивы в JavaScript обладают набором полезных методов, позволяющих более гибко манипулировать их содержимым. С помощью этих методов вы можете перебирать значения массива (без использования циклов), трансформировать массив и делать другие полезные вещи. Методов у массивов больше двух десятков, мы разберём наиболее полезные и часто применяемые методы. Информацию о всех методах массивов вы всегда можете найти в справочнике разработчика MDN.

Методом называют обычную функцию, которая является частью большой сущности, объекта. Например, знакомый вам из тренажёра метод `keks.plot()`. Многие методы массивов, если не сказать большая их часть, в качестве аргумента принимают другую функцию, которая будет вызвана не в момент передачи, а потом. Эта функция, предназначенная для отложенного выполнения, называется колбэком (от англ. callback, перезвонить) или по-русски «функцией обратного вызова».

```
[1, 2, 3].forEach(function () {});
//               ^^^^^^^^^^^^^^ вот он, колбэк
```

Действительно, принцип работы колбэков схож с заказом обратного телефонного звонка. Представьте, что вы звоните заказать пиццу, но срабатывает автоответчик, где приятный голос просит оставаться на линии, пока не освободится оператор, или предлагает заказать обратный звонок. Когда оператор освободится — он перезвонит и примет заказ. В таком случае вместо ожидания ответа оператора мы можем заниматься своими делами, как только нам перезвонят (произойдёт колбэк), мы сможем выполнить задуманное — заказать пиццу.

Так и с колбэками, которые мы передаём в методы массивов. Объявляем мы их сразу:

```
[1, 2, 3].forEach(function () {});
```

А вызов колбэка `function () {}` произойдёт где-то во внутренностях метода `.forEach()`, описанного в движке JavaScript.

~ 1 минута

Глава 3.1.1. Склейка элементов массива

Метод `.join()` приводит все элементы массива к строке и конкатенирует их в одну итоговую строку, разделяя переданным символом — разделителем.

```
const titles = ['Die hard', 'Terminator'];

const message = titles.join('. ');

console.log(message); // 'Die hard. Terminator'
```

Разделитель можно не передавать, по умолчанию это запятая:

```
const titles = ['Die hard', 'Terminator'];

const message = titles.join();

console.log(message); // 'Die hard,Terminator'
```

Если в массиве только один элемент, то он будет приведён к строке и возвращён без разделителя:

```
const titles = ['Die hard'];

const message = titles.join('.');

console.log(message); // 'Die hard', а не 'Die hard.'
```

Обратите внимание, что приведение элементов массива к строке производится по правилам приведения типов. И элементы вроде `undefined` или `null` будут приведены к пустой строке.

~ 2 минуты

Глава 3.1.10. Проверка каждого элемента массива на условие «удовлетворяет хоть один»

Метод `.some()` тоже относится к перебирающим методам массива. То есть это очередной метод, позволяющий перебрать элементы массива, но с одной особенностью. С его помощью можно проверить, присутствует ли в массиве элемент, который удовлетворяет определённому условию. Результатом выполнения метода `.some()` будет булево значение: `true` или `false`.

```
массив.some((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Метод `some` перебирает элементы массива и для каждого элемента вызывает переданную функцию. Метод `.some()` будет вызывать функцию для каждого элемента, пока она не вернёт `true`. Как только это случится, метод `.some()` прервёт работу и вернёт в качестве результата значение `true`.

Если для всех элементов массива переданная функция вернёт `false`, тогда результатом `.some()` станет `false`. Получается, метод `.some()` решает задачу проверки элементов массива на соответствие какому-то условию. Рассмотрим на практическом примере:

```
const numbers = [1, 4, 10, 5];

const isExistsOverFive = numbers.some((value) => {
  return value > 5; // Проверяем каждый элемент, больше ли он, чем 5
}); // Когда some дойдёт до 10, то прекратит работу и вернёт true

const isExistsOverTwenty = numbers.some((value) => {
  return value > 20; // Проверяем каждый элемент, больше ли он, чем 20
}); // some пройдёт все элементы, они все меньше 20, поэтому some вернёт false
```

У метода `.some()` есть одна особенность. Если вызвать его на пустом массиве, то результатом всегда будет `false` вне зависимости от условия:

```
const result = [].some(() => {
  return 1 === 1;
}); // так как массив пустой, some вернёт false
```

~ 1 минута

Глава 3.1.2. Объединение массивов в один новый

Метод `.concat()` используется для склеивания двух и более массивов в один.

```
первый_массив.concat(второй_массив[, третий_массив])
```

Например:

```
const ivanFavoriteFilms = ['Die hard', 'Terminator'];
const mariaFavoriteFilms = ['Kindergarten Cop'];

const favoriteFilms = ivanFavoriteFilms.concat(mariaFavoriteFilms);

console.log(favoriteFilms); // ['Die hard', 'Terminator', 'Kindergarten Cop']
```

Метод не изменяет исходный массив, а возвращает новый.

~ 2 минуты

Глава 3.1.3. Копирование массива или его части

Почему в конце при сравнении `films` и `copyOfFilms` получается `false`, рассказываем в главе:

— Сравнение сложных типов данных

Метод `slice` возвращает копию всех или части элементов исходного массива. Метод не изменяет исходный массив, а возвращает новый. Чтобы получить часть элементов в виде нового массива, нужно передать аргументами диапазон индексов:

```
массив.slice(минимальный_индекс, максимальный_индекс)
```

Например:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

console.log(films.slice(0, 2)); // ['Die hard', 'Terminator']
```

Обратите внимание, что элемент с максимальным индексом переданного диапазона в копию не попадает. Например, элемент с индексом `2` в массиве `films` — это `'Kindergarten Cop'`, но он в копию не попал.

Кстати, если нужна копия элементов с какого-то индекса и до конца, максимальный индекс диапазона можно не указывать:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

console.log(films.slice(1)); // ['Terminator', 'Kindergarten Cop']
```

Кроме использования по прямому назначению — получение части элементов в виде нового массива — `slice` часто используют для создания копии целого массива. Для этого просто не указывают диапазон:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

const copyOfFilms = films.slice();

console.log(copyOfFilms); // ['Die hard', 'Terminator', 'Kindergarten Cop']

console.log(films === copyOfFilms); // false
```

~ 1 минута

Глава 3.1.4. Расположение элементов массива в обратном порядке

Метод массива `.reverse()` «переворачивает» массив с ног на голову, то есть располагает элементы *в том же массиве* в обратном порядке. Рассмотрим на примере массива с числами:

```
const numbers = [0, 1, 2, 3, 4];
const reversedNumbers = numbers.reverse();

console.log(reversedNumbers); // [4, 3, 2, 1, 0]
console.log(numbers); // [4, 3, 2, 1, 0]
// ...
```

Так как элементы переставляются в исходном массиве, результат работы метода — ссылка на этот исходный массив:

```
// ...
console.log(numbers === reversedNumbers); // true
```

Такие методы ещё называют мутирующими, и чтобы избежать неожиданных мутаций, используют метод `.slice()`:

```
const numbers = [0, 1, 2, 3, 4];
const reversedNumbers = numbers.slice().reverse();

console.log(reversedNumbers); // [4, 3, 2, 1, 0]
console.log(numbers); // [0, 1, 2, 3, 4]
console.log(numbers === reversedNumbers); // false
```

~ 3 минуты

Глава 3.1.5. Поиск элементов в массиве

Метод `.find()` позволяет решить одну из самых часто возникающих задач при работе с массивами — осуществить поиск элемента. Раз массивы позволяют хранить наборы значений, то рано или поздно может потребоваться найти значение, которое соответствует определённому условию.

Метод `.find()` как и другие методы аргументом принимает функцию, которая будет вызвана для каждого элемента массива, пока не найдётся элемент, который удовлетворяет условию. Как только такой элемент будет найден, метод `.find()` прекратит работу и вернёт найденный элемент.

```
массив.find((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим метод `.find()` на примере поиска подстроки в массиве:

```
const titles = ['Die hard', 'Terminator'];

const favoriteFilmTitle = titles.find((title) => title.includes('hard'));

console.log(favoriteFilmTitle); // 'Die hard'
```

▶ Что за метод `.includes()`

Но что вернёт функция, если в массиве есть несколько элементов, удовлетворяющих условию? Ответ прост: первый элемент, который соответствует условию. После этого работа метода будет прервана. А если метод `.find()` не найдёт ни одного элемента, удовлетворяющего условию, функция вернёт `undefined`.

findIndex

Метод работает один в один как `.find()`, только результатом вернёт не найденный элемент, а его индекс в массиве.

```
массив.findIndex((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Например:

```
const titles = ['Die hard', 'Terminator'];

const favoriteFilmTitleIndex = titles.findIndex((title) => title.includes('hard'));

console.log(favoriteFilmTitleIndex); // 0
```

~ 2 минуты

Глава 3.1.6. Перебор массива

Глава 3.11.6. Перебор массива

Перебор значений, наверное, самая частая задача при работе с массивами. Если мы храним набор значений, то рано или поздно возникнет необходимость проделать какую-то операцию с элементами этого набора. Для решения этой задачи можно воспользоваться операторами циклов `for` или `while`, или сделать то же самое с помощью метода `.forEach()`.

Метод `.forEach()` позволяет выполнить произвольную функцию однократно для каждого элемента. Попросту говоря: он запускает перебор значений массива и для каждого значения выполняет функцию.

```
массив.forEach((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим пример:

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

fruits.forEach((value, index, array) => {
  console.log(value);
});

// Выведет в консоль:
//   'banana'
//   'apple'
//   'lemon'
//   'orange'
```

В примере выше мы определяем такую функцию с тремя параметрами:

— `value` — текущий элемент массива;

— `index` — порядковый номер (индекс) текущего элемента массива;

— `array` — ссылка на сам массив.

Эти параметры будут доступны при каждом вызове функции.

Если вам не требуется порядковый номер элемента в массиве или как-то взаимодействовать с массивом, то соответствующие параметры в функции можно не определять и воспользоваться более сокращённой записью:

```
массив.forEach((текущий_элемент_массива) => {})
```

Например:

```
fruits.forEach((value) => {
  console.log(value);
});
```

При использовании метода `.forEach()` стоит помнить одну важную деталь: работу метода нельзя остановить. Оператор `break` не поможет. Поэтому если вам требуется перебрать только часть массива, то `.forEach()` следует отодвинуть в сторонку и воспользоваться циклом `for`. Помните об этом.

~ 3 минуты

Глава 3.1.7. Преобразование массива

Преобразование — одна из частых задач при работе с массивами. С помощью метода `.map()` мы можем итерироваться по массиву, изменять его элементы и в результате получить новый массив.

```
массив.map((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим пример. Предположим, у нас есть массив `films` с фильмами. Каждый фильм описан в виде объекта с двумя ключами:

— `id` — идентификатор фильма;

— `title` — название фильма.

Примерно вот так:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];
```

Наша задача заключается в получении массива, который будет содержать только названия фильмов. То есть на выходе мы должны получить массив вида: `['Die hard', 'Terminator']`.

У любой задачи всегда есть минимум два решения. Можно решить её «в лоб» и воспользоваться знаниями о методе `.forEach()`. Алгоритм будет таким: заводим новый массив и начинаем перебирать `films` методом `.forEach()`. В функции, которая будет вызываться для каждого элемента, напишем код для добавления названия фильма в новый массив. На этом задача, можно сказать, решена. Пример такого решения:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];

const titles = [];

films.forEach((film, index) => {
  titles[index] = film.title;
});

console.log(titles); // ["Die hard", "Terminator"]
```

Код работает, однако нам нужно руками заводить пустой массив `titles`, руками в него добавлять элементы. Зачем, если существует метод `.map()`, который может взять эти обязанности на себя:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];

const titles = films.map((film) => {
  return film.title;
});

console.log(titles); // ["Die hard", "Terminator"]
```

Результатом выполнения метода `.map()` будет новый массив, собранный из значений, которые вернёт функция, переданная в качестве параметра методу `.map()`.

Кстати, параметры колбэка у метода `.map()` такие же, как у колбэка `.forEach()` — `текущий_элемент_массива`, `индекс_текущего_элемента`, `ссылка_на_весь_массив`. Выходит, что метод `.map()` похож на `.forEach()`. Только он позволяет не просто перебрать все значения массива, а получить новый массив значений.

Метод `.map()` удобно использовать, когда требуется трансформировать массив, то есть создать новый массив на основе существующего.

~ 5 минут

Глава 3.1.8. Свёртка массива

Может показаться, что `.reduce()` — ещё один метод, позволяющий перебрать содержимое массива, но его основная задача — свернуть массив, то есть из набора значений получить одно. Это значение может быть произвольного типа. За счёт этой возможности метод `.reduce()` становится мощным инструментом, позволяющим решить множество разных задач.

Перед тем как познакомиться с примером, давайте рассмотрим аргументы самого метода:

```
массив.reduce(колбэк_функция[, начальное_значение_результата]);
```

В отличие от других методов массива `.reduce()` может принимать второй аргумент — начальное значение результата, а точнее результирующего значения, того, что мы получим по итогу работы метода. Этот аргумент опциональный (в описании такие оборачиваются `[, ]`), его можно передать:

```
[1, 2, 3].reduce(() => {}, 0);
```

А можно не передавать:

```
[1, 2, 3].reduce(() => {});
```

Что будет, если его не передать, расскажем в конце главы.

Теперь разберём параметры колбэк-функции:

```
(результирующее_значение, текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_сам_массив) => {};
```

Их аж четыре:

— `результирующее_значение` — параметр, через который передаётся предыдущий результат выполнения функции, таким образом этот параметр «кочует» от перебора одного элемента к перебору другого. Та отличительная особенность `.reduce()` от других методов;

— `текущий_элемент_массива` — элемент массива, для которого вызывается колбэк-функция;

— `индекс_текущего_элемента` и `ссылка_на_сам_массив` — тут должно быть понятно без лишних слов.

Рассмотрим применение метода `.reduce()` на практическом примере — подсчёт суммы. Для наглядности опишем задачу так: есть корзина с товарами (массив), где каждый товар представлен объектом с несколькими ключами: `title` (название товара), `quantity` (количество) и `price` (цена). Наша задача заключается в подсчёте общей суммы стоимости всех товаров, и в этом нам поможет метод `.reduce()`:

```
const goods = [
  {
    title: 'Кукуруза',
    quantity: 3,
    price: 99,
  },
  {
    title: 'Корм для кота',
    quantity: 2,
    price: 113,
  },
];

const sum = goods.reduce((total, product) => total + (product.quantity * product.price), 0);

console.log(sum); // 523
```

В этом примере мы посчитали общую сумму стоимости всех товаров и для этого нам потребовалась одна строчка кода, никаких дополнительных переменных как в случае с `.forEach()` или `for`-циклом.

Как это работает? Метод `.reduce()` вызывает переданную функцию для каждого элемента массива. Результат выполнения этой функции доступен на следующей итерации через параметр `total`. Для первой итерации значение `total` будет `0`, потому что мы передали его вторым аргументом в сам `.reduce()`. После завершения всех итераций значение `total` станет результатом выполнения `.reduce()`.

Что происходит внутри колбэк-функции? Она задаёт новое результирующее значение. Не изменяет, а именно задаёт новое! Да, на основе предыдущего, потому что мы прибавляем к результату прошлой итерации произведение количества товара и его стоимости.

Метод `.reduce()` штука сложная, поэтому давайте ещё раз разберём, как будет выполняться этот код, но уже

по шагам.

На первой итерации результирующим значением будет `0` — мы определили его самостоятельно, передав вторым аргументом в метод `.reduce()`. Получается, что на первой итерации выражение будет таким:

```
0 + (3 * 99) = 297;
```

Это мы посчитали стоимость первого товара («Кукуруза»).

На второй итерации мы переходим к следующему товару — «Корм для кота». На этот раз значением `total` будет `297` — результат выполнения функции на прошлой итерации — и к этому значению мы прибавляем стоимость второго товара:

```
297 + (2 * 113) = 523;
```

Поскольку больше элементов в массиве нет, метод `.reduce()` завершит работу и вернёт значение `total`, поэтому результатом выполнения `.reduce()` будет `523`.

Что будет, если не передать начальное значение результата?

Если забыть или нарочно не передать второй аргумент в `.reduce()`, тогда он начнёт обход массива со второго элемента, а начальным значением возьмёт первый элемент:

```
['x', 'y'].reduce((total, current, index) => {
  console.log(total); // 'x'
  console.log(current); // 'y'
  console.log(index); // 1
});
```

Из примера видно, что `.reduce()` начал работу сразу с `'y'`, а результирующим значением для первой проходки взял `'x'`.

Это не ошибка, а заложенное поведение. Удобно, когда вам нужно просуммировать элементы массива — числа:

```
[5, 2, 3].reduce((total, current) => total + current); // 10
```

Результат будет аналогичный «полной» записи:

```
[5, 2, 3].reduce((total, current) => total + current, 0); // 10
```

~ 2 минуты

Глава 3.1.9. Проверка каждого элемента массива на условие «удовлетворяют все»

При помощи метода `.every()` можно проверить, что условию соответствуют все элементы массива. Результатом вызова метода `.every()` будет булево значение: `true` или `false`.

```
массив.every((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Метод `.every()` так же аргументом принимает функцию, которая будет вызываться для каждого элемента до тех пор, пока при проверке условия не вернётся `false`. В таком случае метод `.every()` прекратит выполнение и вернёт `false`. Если для каждого элемента массива будет возвращено значение `true`, то результатом метода `.every()` также станет `true`:

```
const numbers = [11, 12, 13, 15, 100];

const isEveryNumberOverTen = numbers.every((value) => {
  return value > 10;
}); // every вернёт true, потому что все элементы массива больше 10
```

Добавим в массив `numbers` элемент меньше `10`:

```
const numbers = [11, 12, 13, 15, 100, 9];

const isEveryNumberOverTen = numbers.every((value) => {
  return value > 10;
}); // every вернёт false, потому что один элемент массива меньше 10
```

У метода `.every()` есть одна особенность. Если вызвать его на пустом массиве, то результатом всегда будет `true` вне зависимости от условия:

```
const result = [].every(() => {
  return 1 === 1;
}); // every всё равно вернёт true, хотя массив пустой
```

Теория

~ 4 минуты

Глава 3.2. spread-синтаксис

spread-синтаксис или синтаксис расширения, распространения — это способ расширить одну перечисляемую структуру элементами другой. Можно сказать, spread-синтаксис в какой-то степени обратный rest-параметрам. Даже знак используется один и тот же, многоточие `...`

Для примера возьмём массивы. Для объединения массивов существует метод `.concat()`:

```
[1, 2].concat([3, 4]); // [1, 2, 3, 4]
```

С помощью spread-синтаксиса можно обойтись без использования `.concat()`:

```
[1, 2, ...[3, 4]]; // [1, 2, 3, 4]
```

Мы как бы говорим JavaScript: «Отбрось скобки».

Может возникнуть справедливый вопрос, а зачем нам ещё один способ сделать `.concat()`? Дело в том, что spread-синтаксис работает гораздо шире. Метод `.concat()` возвращает массив, а spread откидывает скобки и возвращает содержимое массива. И если место, где это произошло, может принять содержимое — *это важно* — мы получим элементы массива через запятую.

А кто умеет принимать значения через запятую? Ну, кроме объявления массива... Функции! Например, существует метод `Math.max()`, который принимает через запятую значения и возвращает максимальное из них:

```
Math.max(1, 2, Infinity, 100, -3); // Infinity
```

Но что делать, если у нас значения хранятся в массиве:

```
const numbers = [1, 2, Infinity, 100, -3];

Math.max(numbers); // NaN
```

`Math.max()` не умеет работать с массивами, поэтому результат `NaN`.

На помощь приходит spread-синтаксис, который «отбрасывает скобки», а содержимое массива передаёт в `Math.max()`:

```
const numbers = [1, 2, Infinity, 100, -3];

Math.max(...numbers); // Infinity
```

Мысленно можно представить, что произошёл вызов `Math.max(1, 2, Infinity, 100, -3)`.

Также в начале главы не просто так говорится о «перечисляемых структурах», а не просто о массивах. spread-синтаксис можно использовать, например, с объектами вместо `Object.assign()`:

```
// Обе записи равнозначны

Object.assign({ a:1 }, { b:2 }); // { a:1, b:2 }

({ a:1, ...{ b:2 }}); // { a:1, b:2 }
```

Круглые скобки нужны, чтобы JavaScript отличил объявление объекта от блока кода `{}`.

А вот в функцию передать такой объект не получится:

```
console.log(...{ a:1 }); // TypeError: ({a:1}) is not iterable
```

Чтобы понять почему, достаточно мысленно отбросить скобки — `console.log( a:1 )` — согласитесь, странный код.

Также не получится расширить массив содержимым объекта:

```
[1, 2, 3, ...{ a:0 }]; // TypeError: ({a:0}) is not iterable
```

А вот массивом расширить объект получится:

```
({ a:1, ...['b', 'c']}); // { 0:'b', 1:'c', a:1 }
```

▶ Попробуйте ответить «Почему?» самостоятельно, а после посмотрите ответ

Поэтому всегда, всегда учитывайте содержимое и место его вставки, когда используете spread-синтаксис.

Теория

~ 3 минуты

Глава 3.3. Отличия spread-синтаксиса от rest-параметров

Главный вопрос о rest-параметрах и spread-синтаксисе — это как отличить одно от другого, если в обоих случаях используется символ многоточие `...`

В этом поможет контекст, а по-русски — смысл. rest означает «оставшийся». Например «оставшиеся параметры функции»:

```
function doAll (a, b, ...rest) {}

doAll(1, 2, 3, 4); // В rest будет [3, 4]
```

Или «оставшиеся ключи объекта» при деструктуризации:

```
const {a, b, ...rest} = { a:1, b:2, c:3, d:4 }; // В rest будет { c:3, d:4 }
```

А spread означает «расширить», «распустить». Например, мы хотим «распустить элементы массива», чтобы они стали аргументами функции:

```
function doAll (a, b, c) {}

doAll(...[1, 2, 3]);
```

Или когда мы хотим «расширить один массив элементами другого»:

```
[1, ...[2, 3]]; // [1, 2, 3]
```

Нюансы использования

rest-параметры

Здесь ничего нового, rest-параметры всегда должны идти последними и могут быть только одни. Никаких других нюансов.

spread-синтаксис

А вот здесь нужно быть внимательнее. Первое, что отличает синтаксис расширения от rest-параметров, что он не обязательно должен идти последним:

```
[1, ...[2, 3], 4]; // [1, 2, 3, 4]
```

Также можно использовать spread столько раз, сколько захочется, и с вложениями:

```
[1, ...[2, 3], 4, ...[5, ...[6, 7]]]; // [1, 2, 3, 4, 5, 6, 7]
```

А можно составить массив только из spread-синтаксиса:

```
[...[1, 2, 3], ...[4, 5], ...[6, 7]]; // [1, 2, 3, 4, 5, 6, 7]
```

Только нужно помнить, что в случае с массивом, где будет spread, туда элементы и вставятся:

```
[...[2, 3], 1, 4]; // [2, 3, 1, 4]
```

Потому что в массиве элементы упорядочены по индексам. А в случае с объектом такой проблемы нет, потому что пары ключ-значение в объектах не упорядочены:

```
({ ...{ c:3, b:2 }, a:1}); // { a:1, b:2, c:3 }
```

Кстати, порядок по алфавиту тоже не гарантирован, каждый браузер может выстраивать пары в угодном одному ему порядке. Единственное, что гарантируется, что все указанные ключи будут присутствовать.

Резюме

spread-синтаксис и rest-параметры помогают очень быстро вытаскивать отдельные значения из списков или, наоборот, быстро создавать новые списки на основе других списков. spread-синтаксис к тому же мощный инструмент, который применим шире, чем `.concat()` и `Object.assign()`, а ещё он лаконичнее, за что полюбился многим фронтенд-разработчикам и встречается повсеместно.

Теория

~ 5 минут

Глава 4.1. Что такое DOM

DOM (Document Object Model) — это объектная модель документа. Разберём определение по частям:

— Объектная — потому что состоит из объектов;

— Модель — слово в прямом своём значении;

— Документ — имеется в виду веб-страница.

Для примера возьмём простую HTML-страницу:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>DOM</title>
    </head>
    <body>
        <h1>Document Object Model</h1>
        <p>DOM <em>(объектная модель документа)</em> – способ представления разметки страницы в виде связанных между собой объектов</p>
        <p>Каждому элементу на странице – тегу, текстовому блоку, комментарию – в JS ставится в соответствие объект</p>
        <p>Каждый из объектов знает про свой родительский объект, соседние объекты и объекты, расположенные внутри него</p>
        <p>Главный объект, из которого начинают «расти» все остальные элементы DOM-дерева – document.</p>
    </body>
</html>
```

Проблема в том, что браузер понимает HTML, а JavaScript — нет. И чтобы управлять разметкой из JavaScript, например для добавления интерактивности на страницу, нам нужен специальный инструмент. Этим инструментом является DOM.

Образно выражаясь, DOM следует воспринимать как некий словарь для JavaScript к HTML-разметке веб-страницы. DOM описывает HTML-структуру объектами JS (теми самыми, которые в фигурных скобках). То есть всю нашу страницу можно представить в виде объекта `document`. В `document` есть ключ `documentElement`, который соответствует корневому элементу документа — тегу `html`. В `documentElement` лежит `head`, `body` и так далее.

```
const document = {
    documentElement: {
        head: { /*...*/ },
        body: {
            h1: { /*...*/ },
            p: { /*...*/ },
            p: { /*...*/ },
            p: { /*...*/ },
        }
    }
}
```

И тогда доступ к элементу объекта возможен по ключу. Например, к заголовку `h1` по пути `document.documentElement.body.h1`.

Следует помнить, что DOM — это не просто точное отражение разметки, а *сверхточное*, содержащее массу полезных вещей, которые позволяют программисту удобно работать с разметкой.

К таким удобным инструментам в DOM относятся ссылки для быстрого доступа. Например, чтобы получить доступ к узлу `body`, не обязательно обращаться `document.documentElement.body`, а можно сразу — `document.body`. Так же можно поступить, например, с формами — в ключе `document.forms` будут собраны все формы на странице. И таких ссылок для быстрого доступа существует множество.

Однако в DOM присутствуют не только теги. Например, после тега `body` есть перенос строки и табуляция в 4 пробела. Вспомним, как мы пишем код:

— открываем тег `<body>`,

— далее нажимаем клавишу [ENTER] для переноса строки,

— далее в новой строке нажимаем клавишу [TAB] для отступа,

— и далее, например, открываем тег `<h1>`,

— затем, перенос строки, табуляция,

— и, к примеру, в `<h1>` кладём ссылку `<a>`...

```
<body>
    <h1>
        <a href="#" target="_blank">Ссылка</a>
```

С тегом `<a>` тоже всё непросто. У него заданы атрибуты `href` и `target`, а ещё есть содержимое — текст. Поэтому с точки зрения DOM ссылка — это отдельный объект. Кстати, так бывает не всегда. Любой тег может быть представлен как примитив, а может как объект. Потому что DOM экономный.

Теги образуют узлы-элементы, а текст внутри тега образует текстовый узел. Переносы строки, пробелы и табуляция — всё это полноправные текстовые узлы. Даже комментарий является элементом DOM. И вообще всё, что есть в HTML — есть в DOM. Таким образом, HTML-разметка в веб-странице является основой для формирования первоначального, исходного состояния DOM.

Вернёмся к примеру с ссылкой. У неё есть адрес, куда она ведёт, и текст. Это как минимум. Помимо этого там может быть набор атрибутов, например «открыться в новом окне» или «не следить за мной» и так далее. Всё это превратится в поля объекта:

```
{
    /*...*/
    a: {
        href: '#',
        target: '_blank',
        textContent: 'Ссылка'
    }
    /*...*/
}
```

Резюме

DOM — это огромный объект, который имеет древовидную структуру. Часто можно встретить выражение — DOM-дерево.

Теория

~ 4 минуты

Глава 4.2. DOM-дерево

Дерево — это структура. Такое название структуре дано не зря, она действительно напоминает дерево, только перевёрнутое: корень вверху, от корня вниз идут ветки и листья. Каждая часть дерева называется элементом, а в DOM-дереве элемент называется узлом. То есть **узел** — это абсолютно любой элемент дерева.

Узлы бывают родителями — когда у них есть дочерние узлы. И узлы бывают детьми — это узлы, у которых есть родитель.

Продолжая аналогию: у дерева и корень, и ветки, и листья — это узлы. Для тех веток, которые растут прямо из ствола, ствол — это родитель, с другой стороны для ствола ветка — это ребёнок. И такая иерархическая структура идёт от корня до листьев.

Почти все узлы могут быть одновременно и родителями и детьми. Почти, потому что есть такие узлы, как корень и листья. Корень — это тот элемент, с которого начинается дерево. То есть, выше корня ничего нет, и у корня нет родителя. Соответственно, каждый элемент, за исключением корня, имеет одного родителя. А лист — это узел, у которого нет детей. Соответственно, каждый узел, кроме листа, имеет любое количество детей.

« Корень — это узел, у которого есть только потомки, лист — это узел, у которого есть только родитель.

Важные моменты

— Корень может быть только один.

— Родитель может быть только один.

У DOM-дерева корень — это `document`. Узел `html` не может являться корнем, потому что DOM-дерево содержит, кроме тегов, и отступы, и табуляции, и комментарии и так далее. И на одном уровне с `html` могут оказаться ещё узлы, а корень может быть только один, поэтому корень — это `document`. Для примера возьмём такую разметку:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
    </body>
</html>
```

У каждого узла, представленного объектом, есть свойство `parentElement`, которое содержит информацию о родителе элемента.

```
console.log(document.parentElement); // null
```

Потому что у корня не может быть родителей. В свойстве `children` записаны дочерние элементы узла:

```
console.log(document.children); // documentElement — в DOM дочерний элемент document называется не html, а именно documentElement
// Остальные DOM-узлы называются так же, как и теги в HTML
```

Свойство `children` можно воспринимать как массивоподобную коллекцию детей, а значит мы можем написать код,

который будет перебирать всех детей и выводить в консоль структуру.

Чтобы узнать имя текущего элемента, нужно обратиться к свойству `tagName` или `nodeName`. По соглашению, для HTML-документов имя тега всегда возвращается в верхнем регистре, поэтому важно не забывать приводить к общему написанию с помощью метода строки `toLowerCase`.

Выведем структуру узла `html`, включая все дочерние элементы, на примере такой страницы:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Document Object Model</h1>
    </body>
</html>
```

Скрипт для вывода структуры узла `html` будет иметь следующий вид:

```
const html = document.documentElement;
for (let i = 0; i < html.children.length; i++) {
    const child = html.children[i];
    console.log(child.tagName.toLowerCase());
    for (let j = 0; j < child.children.length; j++) {
        const innerChild = child.children[j];
        console.log('|---' + innerChild.tagName.toLowerCase());
    }
}
```

Скрипт в цикле перебирает все дочерние элементы HTML-узла — `documentElement.children`. У каждого ребёнка в свою очередь перебирает его дочерние элементы и выводит имена тегов, приведённые к нижнему регистру.

Результат будет иметь следующий вид:

```
head
  |---meta
body
  |---h1
```

Теория
~ 4 минуты

Глава 4.3. Поиск в деревьях

В HTML все элементы, кроме `document`, являются вложенными: `head` и `body` вложены в `document`, элементы списка `li` могут быть вложены в элемент нумерованного списка `ol` или неупорядоченного списка `ul`. Структура,

которую представляет HTML-документ, а соответственно и DOM, является деревом.

Когда возникает необходимость найти элемент в древовидной структуре, сделать это можно двумя способами:

1. Поиск в глубину (DFS — Depth-First Search)
2. Поиск в ширину (BFS — Breadth-First Search)

Хотя в JavaScript для поиска элемента используется первый способ — поиск в глубину, знать оба полезно.

Для разбора каждого из подходов возьмём следующую древовидную структуру, которая представляет HTML-документ. Соответственно, каждый узел — это HTML-элемент.

Рисунок 1. Схема DOM-дерева

Допустим, наша задача — найти ссылку, то есть HTML-элемент `a`. И поиск в глубину, и поиск в ширину начинается с узла `document`. В `document` мы найдём единственного ребёнка — узел `html`. Так как ни `document`, ни `html` ссылками не являются, мы идём дальше и находим не один, а нескольких дочерних элементов. Разница между поиском в глубину и поиском в ширину заключается в том, каким образом мы продолжим поиск.

Поиск в глубину

Мы находим одного из детей `html` — элемент `head`. Нам опять не повезло найти ссылку, поэтому пойдём вглубь, то есть в единственного ребёнка `head` — элемент `title`. `Title` также не соответствует требованиям поиска, и к тому же не имеет дочерних элементов.

Рисунок 2. Поиск в глубину

Мы упёрлись в тупик. Самое время вернуться назад, в `html`, и пойти по второму доступному пути — в элемент `body`. Далее, по такой же цепочке, мы найдём элементы `h1` и, наконец, `a`.

Рисунок 3. Поиск в глубину, продолжение

Когда имеется несколько доступных путей поиска, при поиске в глубину мы исследуем каждый из них до тех пор, пока не сталкиваемся с тупиком. Из тупика мы возвращаемся на последнюю развилку, исследуем её и так далее.

Поиск в ширину

Теперь попробуем снова найти ссылку, но уже используя поиск в ширину. Вспомним, что начинаем мы из `document`, проваливаемся в `html` и сталкиваемся с распутьем.

В отличие от поиска в глубину, поиск в ширину сначала проверит всех детей элемента `html`, то есть `head` и `body`, на соответствие элементу-ссылке. Не найдя то, что ищет, он пойдёт в их непосредственно дочерние элементы и проверит их. Данный паттерн, проверка слой за слоем — это принцип работы поиска в ширину.

Рисунок 4. Поиск в ширину

Резюме

Поиск в глубину и поиск в ширину — два подхода к обходу древовидных структур. Первый находит узел и проваливается в него, даже если у него есть соседи, которых можно проверить. Поиск в ширину, с другой стороны, сначала проверит все узлы одного уровня, и лишь затем будет спускаться в их непосредственных детей. JavaScript для обхода DOM использует поиск в глубину.

Теория

~ 5 минут

Глава 4.4. Живые и неживые коллекции в JavaScript

Найти несколько DOM-элементов и получить к ним доступ из JavaScript можно разными способами: `querySelectorAll`, `getElementsByTagName`, `children` и так далее. В итоге в каждом случае будет возвращена коллекция — сущность, которая похожа на массив объектов, но при этом им не является, на самом деле это набор DOM-элементов. Стоит учесть, что фактически разные методы возвращают разные коллекции:

— `HTMLCollection` — коллекция непосредственно HTML-элементов.

— `NodeList` — коллекция узлов, более абстрактное понятие. Например, в DOM-дереве есть не только узлы-элементы, но также текстовые узлы, узлы-комментарии и другие, поэтому `NodeList` может содержать другие типы узлов.

При работе с DOM-элементами тип коллекции значительной роли не играет, поэтому для удобства будем рассматривать их как одну сущность — коллекцию.

Во время работы с коллекциями можно столкнуться с поведением, которое покажется странным, если не знать один нюанс — они бывают живыми (динамическими) и неживыми (статическими). То есть либо реагируют на любое изменение DOM, либо нет. Вид коллекции зависит от способа, с помощью которого она получена. Рассмотрим на примере.

Разница между живыми и неживыми коллекциями

Допустим, в разметке есть список книг:

```
<ul class="books">
    <li class="book book--one"></li>
    <li class="book book--two"></li>
    <li class="book book--three"></li>
</ul>
```

Для взаимодействия с книгами получим с помощью JavaScript список всех нужных элементов. Чтобы в дальнейшем увидеть разницу между видами коллекций, используем разные способы поиска элементов — свойство `children` и метод `querySelectorAll`:

```
const booksList = document.querySelector(`.books`);
const liveBooks = booksList.children;

// Выведем все дочерние элементы списка .books
console.log(liveBooks);
```

```
const notLiveBooks = document.querySelectorAll(`.book`);

// Выведем коллекцию, содержащую все элементы с классом book
console.log(notLiveBooks);
```

Пока никакой разницы не видно. В обоих случаях console.log выведет одни и те же элементы. Но что, если попробовать удалить из DOM одну из книг?

```
const booksList = document.querySelector(`.books`);
const liveBooks = booksList.children;

// Удалим первую книгу
liveBooks[0].remove();
// Получим 2
console.log(liveBooks.length);
// Получим элемент book--two, который теперь стал первым в коллекции
console.log(liveBooks[0]);
```

```
const notLiveBooks = document.querySelectorAll(`.book`);

// Удалим первую книгу
notLiveBooks[0].remove();
// Получим 3
console.log(notLiveBooks.length);
// Получим ссылку на удалённый элемент book--one
console.log(notLiveBooks[0]);
```

В первом случае информация о количестве элементов внутри коллекции автоматически обновилась после удаления одного элемента из DOM — эта коллекция живая. Во втором случае в переменной `notLiveBooks` хранится первоначальное состояние коллекции, которое было актуально на момент вызова метода `querySelectorAll`. Эта коллекция неживая, она ничего не знает об изменении DOM. При этом доступна ссылка на удалённый элемент `book--one`, которого фактически больше нет в DOM.

Другие способы получить коллекцию

Кроме `children` и `querySelectorAll` есть другие способы поиска DOM-элементов:

— `getElementsByTagName(tag)` — находит все элементы с заданным тегом,

— `getElementsByClassName(className)` — находит все элементы с заданным классом,

— `getElementsByName(name)` — находит все элементы с заданным атрибутом name.

Все эти методы могут встречаться в старом коде. Они возвращают живые коллекции и используются реже, потому что в большинстве случаев возможности живых коллекций не пригождаются. К тому же `querySelectorAll` в разы удобнее использовать из-за его универсальности.

Как использовать

Для решения большинства задач можно ограничиться неживыми коллекциями. Но если нужно сохранить ссылку на реальное состояние DOM — понадобится живая коллекция. Это удобно в тех случаях, когда программе нужно постоянно манипулировать списком элементов, которые могут регулярно удаляться и добавляться. Хороший пример — задачи в системе учёта задач. С помощью живой коллекции можно хранить именно те задачи, которые фактически существуют в данный момент времени.

Структура и некоторые свойства коллекции имеют много общего с массивом. Например, у неё тоже есть свойство `length`, и элементы коллекции можно перебирать в цикле `for...of`, потому что это перечисляемая сущность. Но, как упоминалось ранее, коллекции не во всём похожи на обычные массивы. С коллекциями не работают такие методы массивов, как `push`, `splice` и другие. Для их использования нужно преобразовать коллекцию в массив — например, с помощью метода `Array.from`:

```
const booksList = document.querySelector(`.books`);
const books = booksList.children;

// Выведет обычный массив с элементами из коллекции books
console.log(Array.from(books));
```

При этом нужно помнить — массив статичен, поэтому при таком преобразовании теряются преимущества живых коллекций.

Теория

~ 4 минуты

Глава 4.5. DOM и разметка

Любое изменение, которое мы вносим в DOM, не является изменением разметки сайта. DOM может изменяться путём воздействия на него через JavaScript; либо же пользователем, путём взаимодействия с интерфейсом. Рассмотрим на примере: допустим, у нас на странице есть форма с двумя флажками, один из которых заранее отмечен:

```
<form>
    <label>
        <input type="checkbox" name="someCheckbox" value="1" checked>
        First checkbox
    </label>
    <label>
        <input type="checkbox" name="someCheckbox" value="2">
        Second checkbox
    </label>
</form>
```

Рисунок 1. Форма с двумя флажками

На данной странице пользователь может каким-либо образом изменять значение данных флажков. Допустим, пользователь снял отметку с первого чекбокса и поставил во втором:

Рисунок 2. Выбран другой флажок

При сохранении или отправке формы мы хотим узнать значение чекбокса, который отметил пользователь, для этого мы можем попробовать использовать следующий код, для того, чтобы найти чекбокс с нужным именем и атрибутом `checked`:

```
document.querySelector(`input[name="someCheckbox"][checked]`).value;
```

Однако, это неверно — когда пользователь взаимодействует с разметкой, используя элементы управления, то он изменяет DOM. То же самое мы можем делать из JavaScript. А в селекторе, который используется в коде выше, идёт привязка к разметке, и в данной ситуации такой селектор вернёт первое поле ввода, так как только у первого

поля ввода в разметке присутствует атрибут `checked`:

Рисунок 3. Поиск выбранного флажка

Чтобы такого не происходило, мы можем использовать псевдоклассы (как в CSS):

```
document.querySelector(`input[name="someCheckbox"]:checked`).value;
```

Тогда селектор найдёт поле ввода, которое выбрано на данный момент:

Рисунок 4. Поиск элемента в DOM

Помните, что любая манипуляция с DOM не меняет разметку. В этом плане ввести в заблуждение могут инструменты разработчика. Если проинспектировать страницу, то во вкладке «Elements» можно увидеть, казалось бы, разметку:

Рисунок 5. Вкладка Elements

Однако на самом деле это DOM, просто браузер помогает нам смотреть на DOM, как на разметку. Чтобы увидеть разметку, нужно из контекстного меню на странице выбрать пункт «View Page Source»:

Рисунок 6. Открыть исходный код страницы

Тогда в открывшемся окне мы увидим разметку:

Рисунок 7. Исходный код страницы

Разметка для DOM — это начальное состояние, то есть то состояние, которое появляется после загрузки страницы. После этого в ход вступают пользователь, JavaScript, какие-либо сторонние библиотеки, и DOM может измениться и уже не соответствовать разметке.

Теория

~ 5 минут

Глава 4.6. Шаблоны и данные

Начнём с определений. **Шаблон** — некоторая оболочка для данных, разметка, любой способ отобразить информацию. Шаблон никогда не несёт содержательной информации.

Данные — информация, которую вводит пользователь, присылает сервер или которая может быть сгенерирована компьютером.

Данные не должны повторять шаблон, они должны описывать параметры сущностей, которыми мы оперируем. Простой способ отделить шаблон от данных — попробовать изменить одно или другое.

Например, использовать иной способ отображения данных (отобразить товары в линейку вместо списка) или изменить отображаемую информацию (описать не утюг, а пылесос). Допустим, у нас есть структура, описывающая логотип:

```
const header = {
    logo: {
        src: `logo.png`,
        width: 100,
        height: 30
    }
};
```

С первого взгляда можно подумать, что это данные, однако, такую информацию неправильно хранить как данные. Эта информация описывает логотип, расположенный в шапке, она не приходит с сервера, не вводится пользователем, возможно, эта информация никогда не поменяется. Поэтому данная информация не является данными — это шаблон, который описывает, как некоторая сущность должна выглядеть. Данными могут быть: название компании, адрес, телефон и тому подобное.

Зачем отделять данные и шаблоны

Допустим, что у нас есть список некоторых продуктов интернет-магазина:

```
const products = [
    {name: `Утюг`},
    {name: `Чайник`},
    {name: `Пылесос`},
    {name: `Стиральная машина`},
    {name: `Кухонный комбайн`},
    {name: `Автомобиль`}
];
```

Дизайнер говорит нам, что в интерфейсе мы должны для некоторых пользователей показывать товары списком, а для других (кто хочет) сеткой. Неужели нам придётся заводить два одинаковых списка продуктов? Нет! Как раз список продуктов, приведённый выше, является данными, а способы показа товаров являются шаблоном.

Данные не должны меняться в зависимости от того, каким образом они должны отображаться. Шаблон отвечает за то, куда вставить данные — в элемент списка или элемент сетки.

Создание DOM элементов

DOM-элементы можно создать несколькими способами:

— **На основе разметки** — в специальные места разметки в тексте разметки подставляются данные;

— **На основе строк;**

— **Компилируемые шаблоны** — использование сторонних библиотек, способных переводить некоторый язык в разметку;

— **На основе DOM-API**:

  — На основе шаблонного элемента (`template` из WebComponents);

  — С помощью обёрток над шаблонами (Incremental DOM).

Рассмотрим разделение на шаблон и данные на примере. Допустим, у нас на странице есть следующий блок, показывающий нескольких волшебников:

Рисунок 1. Несколько волшебников

Как можно заметить, волшебники выглядят однотипно, то есть способ их отображения един и не меняется. Однако, некоторые параметры у них отличаются: имя и цвет мантии.

В данном примере разметка для каждого из волшебников является шаблоном и не несёт в себе информации, а лишь служит обёрткой для отображения данных. Разметка для каждого из волшебников (шаблон) имеет следующий вид:

```
<template id="similar-wizard-template">
  <div class="setup-similar-item">
    <div class="setup-similar-content">
      <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 62 86" class="setup-similar-wizard">
        <g class="wizard">
          <use xlink:href="#wizard-coat" class="wizard-coat"></use>
          <use xlink:href="#wizard-head" class="wizard-head"></use>
          <use xlink:href="#wizard-eyes" class="wizard-eyes"></use>
          <use xlink:href="#wizard-hands" class="wizard-hands"></use>
        </g>
      </svg>
    </div>
    <p class="setup-similar-label"></p>
  </div>
</template>
```

В данном шаблоне нет конкретных данных, лишь способ отображения. В данный шаблон в нужные места можно подставить данные, нужные нам. Данные для нашего примера будут иметь следующий вид:

```
const wizards = [
    {
        name: `Дамблдор`,
        coatColor: `rgb(241, 43, 107)`
    },
    {
        name: `Волдеморт`,
        coatColor: `rgb(215, 210, 55)`
    },
    {
        name: `Доктор`,
        coatColor: `rgb(101, 137, 164)`
    },
    {
        name: `Гарри`,
        coatColor: `rgb(127, 127, 127)`
    }
];
```

Как видно, данные описывают параметры, изменяющиеся у различных волшебников.

Резюме

Важно уметь выделять данные и шаблоны. К данным можно отнести ту информацию, которая не влияет на способ отображения, соответственно, к шаблонам нужно отнести то, что описывает, как некоторые однотипные элементы могут быть отображены.

Простой способ отделить шаблон от данных — попробовать заменить их: изменить способ отображения данных или отображаемую информацию.

Теория

~ 4 минуты

Глава 5.1. Синхронные и асинхронные операции

Когда рассказывают про программы и алгоритмы, их часто сравнивают с инструкциями, написанными для человека. Например, вот так выглядит инструкция по приготовлению каши:

1. Насыпал кашу.
2. Включил плиту.
3. Помешиваю, помешиваю, помешиваю кашу.
4. Каша готова.

Или схематично:

Рисунок 1. Синхронное приготовление завтрака

Человек, следующий этой инструкции, будет выполнять её пункты по очереди. Начнёт с первого пункта, завершив его, перейдёт ко второму, и так далее. Однако, есть кашу всухомятку довольно грустно, поэтому добавим к ней чай.
В синхронной схеме чай мы приготовим после того, как каша будет готова:

3. …
4. Каша готова.
5. Налил воду в чайник.
6. Поставил чайник на плиту.
7. Дождался, пока вода закипит.
8. Залил чай кипятком.
9. Подождал, пока чай заварится.
10. Чай готов.

Что можно сказать про эту инструкцию? Ну, во-первых, она действительно позволяет приготовить завтрак. Во-вторых, человеку, который решит ей воспользоваться, придётся встать пораньше, если он рассчитывает успеть на работу. Инструкция довольно долгая. Сначала нужно дождаться, пока приготовится каша, потом — пока вскипит чайник. Да и к моменту, когда чай будет готов, каша наверняка уже остынет. Разумно ли это? Конечно же, нет! Заменим кастрюлю на мультиварку, и чайник на электрический, тогда ждать понадобится намного меньше, потому что готовить можно одновременно.

Перепишем инструкцию с учётом новой техники:

1. Насыпал кашу.
2. Включил мультиварку.
3. Налил воды в чайник.
4. Включил чайник.
5. Приготовил тосты.

6. Накрыл на стол.
7. …

Рисунок 2. Асинхронное приготовление завтрака

В этом случае пока каша готовится, мы успеем накрыть на стол. Пока мы накрываем на стол, вероятно, закипит чайник — мы сможем заварить чай, а тут и каша готова. Приятного аппетита.

Рисунок 3. Асинхронное приготовление завтрака (завершение)

Согласитесь, оптимальная схема, и поспать с утра можно подольше. Однако мы незаметно отказались от одного свойства, которым обладала предыдущая инструкция — от последовательности.

Мы не можем знать наверняка, что мультиварка подаст сигнал раньше, чем чайник вскипит. Обратное тоже с уверенностью утверждать нельзя. Скорее всего, каша будет готовиться дольше, но может случиться по-всякому. Вдруг у исполнителя очень мощная мультиварка и очень старый чайник. А значит — мы не знаем, в каком порядке и когда завершится то или иное действие.

Второе важное различие: за приготовлением каши в кастрюле целиком и полностью следим мы, и только мы решаем, когда каша будет готова. В случае с мультиваркой готовность каши зависит от некоего внешнего фактора (мультиварки).

Если говорить на программистском языке, приготовление каши в кастрюле — череда синхронных операций. Синхронные операции производятся одна за другой в предсказуемом порядке. А приготовление каши в мультиварке включает в себя асинхронную операцию. Особенность асинхронных операций, что мы не можем предугадать, в каком порядке они завершатся и сколько времени займёт каждая.

«Но какое это имеет отношение к веб-программированию?» — спросит нетерпеливый читатель. Самое прямое: браузер — это тоже мультиварка. Разумеется, в переносном смысле. Браузер не умеет варить овсяную кашу (по крайней мере, текущие веб-стандарты этого не предусматривают), зато он умеет выполнять асинхронные операции.

Мы даём ему команду загрузить файл, но не знаем, в какой момент произойдёт загрузка. Мы создаём на странице кнопку, но неизвестно, когда пользователь на неё нажмёт. Точно так же, как и при варке каши, мы можем отдать команду и заниматься своими делами. И когда всё будет готово, браузер подаст сигнал — опять же, совсем как мультиварка.

Теория
~ 7 минут

Глава 5.2. Функции обратного вызова (колбэки)

При изучении программирования мы привыкаем мыслить последовательно: строки кода выполняются по порядку. Для многих языков это утверждение верно на 100%, но всё начинает меняться, когда речь заходит про асинхронное программирование. Самым простым и часто используемым способом достижения асинхронности в коде являются колбэки. Это одна из важнейших тем программирования на JavaScript, ни одна более-менее серьёзная программа не обойдётся без применения колбэк-функций.

▶ Ещё раз, что такое колбэк-функция, если забыли

Как писать код для колбэков

Посмотрим на колбэки с практической стороны. Выше мы сказали, что колбэки неразрывно связаны с асинхронностью и позволяют «запланировать» действие, которое будет совершено после выполнения какого-то другого, возможно длительного действия. Пример с заказом пиццы это прекрасно иллюстрирует. Давайте посмотрим, как это может выглядеть в коде, но для начала взглянем на синхронный код:

```
// Приготовление пиццы. Длительный процесс
const newPizza = makePizza('pepperoni');
// Читаем книгу пока готовят пиццу
readBook();

// Съедаем пиццу
eatPizza(newPizza);
```

Что в этом коде больше всего бросается в глаза? Правильно — последовательность. Здесь представлен синхронный код, который будет выполняться последовательно:

1. Мы ждём, пока для нас приготовят пиццу «Пепперони».
2. Затем мы читаем книгу.
3. Наконец-таки откладываем книгу в сторону и ужинаем пиццей.

Проблема видна невооружённым глазом — пока готовится пицца, мы вынуждены ждать и ничего не делать. Строка `readBook()` будет выполнена только **после** приготовления пиццы. Фактически мы начнём читать книгу после приготовления пиццы, а не **во время** готовки.

Само собой, в реальном мире вместо выпекания пиццы может быть любой долгий процесс, например, запрос на получение данных с сервера.

Такой запрос не выполняется мгновенно: браузеру понадобится время, чтобы найти IP-адрес сервера, установить соединение, передать запрос, дождаться ответа и т. д. Все эти действия занимают разное количество времени. Временные задержки будут постоянно отличаться и зависеть от скорости соединения с сетью, времени выполнения запроса на сервере и некоторых других факторов.

Синхронные запросы к серверу будут блокировать дальнейшее выполнение веб-приложения, и это уже очень плохо. Представьте, что каждый раз при отправке запроса к серверу интерфейс вашего приложения становится полностью недоступным.

Эту проблему решает асинхронность, и длительные операции лучше выполнять именно асинхронно. В этом варианте мы как бы откладываем длительную операцию «на потом» и вместо ожидания завершения выполняем другой код. В этой схеме прозрачно всё, кроме вопроса: «Как выполнить код после завершения асинхронной операции?». Ответ прост — функции обратного вызова.

В JavaScript функции являются объектами высшего порядка. Это означает, что функции можно передавать в другие функции в виде параметров или возвращать в виде результата выполнения.

Рассмотрим пример:

```
const foo = function () {
  return 'Hello, world!';
}

// Вызываем функцию и выводим результат в консоль
console.log(foo()); // Hello, world

// Выводим функцию в консоль без вызова
```

```
// Выводим функцию в консоль без вызова
console.log(foo); // ƒ () { return 'Hello, world!'; }
```

В первом случае мы вызываем функцию `foo` при помощи круглых скобок и выводим результат выполнения в консоль. Во втором примере мы не делаем вызов функции (обратите внимание на отсутствие круглых скобок), и в консоль выводится содержимое функции. Выходит, нам ничего не мешает передать функцию в виде параметра для других функций:

```
const runIt = function (fn) {
  return fn(); // Вызываем функцию, переданную в качестве параметра
}

console.log(runIt(foo)); // Hello, world
```

Мы передали функцию `foo` в виде параметра и вызывали её внутри функции `runIt`. Вызов функции мы сделали стандартным образом — применяя круглые скобки.

Что в итоге? Мы передали ссылку на функцию в виде параметра и вызвали её внутри другой функции. В этом и заключается идея колбэков: мы передаём в виде параметров функции, которые будут вызваны «когда-нибудь потом».

И снова пицца

Вернёмся к примеру с приготовлением пиццы. Попробуем поэкспериментировать с кодом и перевести его на асинхронные рельсы. Напомню, наша задача — попросить приготовить пиццу, и читать книгу, пока пицца не будет готова.

```
const makePizza = function (title, cb) {
  console.log(`Заказ на приготовление пиццы «${title}» получен. Начинаем готовить…`);

  // setTimeout — встроенная функция для отложенного вызова колбэка.
  // Интерфейс у неё прост: первым аргументом нужно передать колбэк, а вторым — задержку (таймаут) в милисекундах.
  // Ближе с setTimeout вы познакомитесь позже, в отдельном материале раздела.
  setTimeout(cb, 3000);
}

const readBook = function () {
  console.log('Читаю книгу «Колдун и кристалл»…');
}

const eatPizza = function () {
  console.log('Ура! Пицца готова, пора подкрепиться.');
}

makePizza('Пеперонни', eatPizza);
readBook();
```

Это рабочий код, попробуйте выполнить его в консоли и посмотреть на результат вывода. Он будет таким:

```
Заказ на приготовление пиццы «Пепперони» получен. Начинаем готовить…
Читаю книгу «Колдун и кристалл»…

// Здесь будет пауза

Ура! Пицца готова, пора подкрепиться.
```

Функция `makePizza` выполняется мгновенно, и сразу за ней последовал вызов readBook. Пока мы читали книгу

Функция `makePizza` выполнилась мгновенно, и сразу за ней последовал вызов `readBook`. Пока мы читали книгу, приготовилась пицца, и произошёл вызов функции `eatPizza` из функции `makePizza`.

Резюме

Как видите, ничего сверхъестественного в колбэках нет. Это обычная функция, которая будет выполнена не сейчас, а когда-нибудь потом. «Когда-нибудь» — не преувеличение. Мы не можем сказать, в какой момент времени это случится, но можем сказать, после какой именно функции — после выполнения функции приготовления пиццы.

Теория
~ 6 минут

Глава 5.3. Функции обратного вызова на практике

В прошлом разделе мы на примере заказа пиццы разобрались с тем, зачем нужны колбэки. Сейчас посмотрим на то, как работают запросы к серверу, как получать данные через AJAX.

Практически любое приложение на JavaScript взаимодействует с сервером. Нам постоянно приходится получать данные с помощью методологии AJAX. Попробуем запросить информацию о произвольном аккаунте с GitHub и вывести её в консоль. Такая программа может выглядеть так:

```
const loadData = function(url, cb) {
  const xhr = new XMLHttpRequest();
  xhr.open(`GET`, url);
  xhr.responseType = `json`;
  // После получения данных выполним cb.
  xhr.addEventListener(`load`, cb);
  xhr.send();
}

loadData(`https://api.github.com/users/AntonovIgor`, function (evt) {
  const response = evt.currentTarget.response;
  console.log(response);
});
```

В этом коде мы делаем запрос к серверу с помощью объекта `XMLHttpRequest`. Для удобства мы создали функцию `loadData`, которая принимает два параметра: адрес сервера, который отдаст нам набор информации, и функцию обратного вызова. Вызов этой функции произойдёт сразу после получения данных от сервера. Если вас смутило наименование параметра `cb` — не переживайте. Это общепринятый вариант обозначения функций обратного вызова (сокращение от callback). При виде cb становится очевидно, что в этот параметр должна передаваться ссылка на функцию.

Мы не будем детально рассматривать содержимое функции `loadData`. Подробно про взаимодействие с сервером расскажем в одном из следующих разделов учебника. Если коротко, то эта функция делает запрос к серверу и вызывает нашу функцию после получения ответа.

Обратите внимание, как мы вызываем функцию `loadData`. В первый параметр мы передаём адрес сервера, а во второй — ссылку на функцию. В этот раз мы не описывали функцию отдельно, а сделали это прямо «в параметре» — создали анонимную функцию. Подобный подход часто применяется, когда функция нужна один раз.

В таких ситуациях нет смысла писать отдельный код — достаточно создать анонимную функцию в нужном месте.

Наша функция сработает сразу после получения ответа от сервера (произойдёт событие `load`) и выведет ответ сервера в консоль.

Колбэки повсюду

Даже если вам не требуется работать с сервером, довольно высоки шансы столкнуться с функциями обратного вызова.

Не верите? Хорошо, а как насчёт стандартных возможностей вроде перебирающих методов массивов: `.forEach()`, `.every()`, `.some()`, `.reduce()`, `.filter()`, или функции сортировки `.sort()`, или методе `.addEventListener()` (второй аргумент — функция, которая будет вызвана при наступлении события). Список можно продолжать до бесконечности.

Всякий раз, когда вы пишете код, похожий на этот, вы применяете функции обратного вызова:

```
[1, 2, 3, 4].forEach(function(it) {
  console.log(it)
});
```

Аналогично с установкой обработчиков событий. Каждый раз, когда вы подписываетесь на событие с помощью `addEventListener`, через её второй параметр вы определяете функцию обратного вызова, которая сработает при наступлении события. В мире JavaScript колбэки повсюду.

Как передать параметры в колбэк-функцию

Это очень важный вопрос, и в начале понимания сути колбэк-функций он может загнать в тупик.

Проблема в том, что мы привыкли передавать параметры явно: пишем имя функции, открываем круглые скобки и передаём значения. Этот подход применяется постоянно, но с колбэк-функциями дело обстоит иначе. Если мы укажем круглые скобки, то произойдёт вызов функции, а нам требуется передать ссылку на неё. Как поступить в этом случае?

Для этого нужно вспомнить про замыкания. Функции в JavaScript могут возвращать в качестве результата выполнения другие функции. А уже у них будет доступ к родительским областям видимости.

С таким подходом легко решить задачу передачи параметров в колбэк-функцию.

Код заказа пиццы, который мы написали в первой части, выглядел так:

```
const makePizza = function (title, cb) {
 console.log(`Заказ на приготовление пиццы «${title}» получен. Начинаем готовить…`);
 setTimeout(cb, 3000);
}

const readBook = function () {
 console.log(`Читаю книгу «Колдун и кристалл»…`);
}

const eatPizza = function () {
  console.log(`Ура! Пицца готова, пора подкрепиться.`);
}

makePizza(`Пеперонни`, eatPizza);
readBook();
```

Давайте модифицируем его и добавим для функции `eatPizza` параметр `drink`, через который будем передавать

напиток:

```
const makePizza = function (title, cb) {
 console.log(`Заказ на приготовление пиццы «${title}» получен. Начинаем готовить…`);
 setTimeout(cb, 3000);
}

const readBook = function () {
 console.log(`Читаю книгу «Колдун и кристалл»…`);
}

const eatPizza = function (drink) {
 return function() {
   console.log(`Ура! Пицца готова, пора подкрепиться и запить ${drink}.`);
 }
}

makePizza(`Пепперони`, eatPizza(`Coca-Cola`));
readBook();
```

Первое, что мы сделали — внесли изменения в функцию `eatPizza`. Теперь она принимает параметр `drink` и возвращает новую функцию. В теле новой функции происходит вывод информации в консоль. Помимо текста, который у нас был, мы добавили вывод информации о напитке (`drink`). Теперь самое интересное. Функция `eatPizza` перестаёт быть функцией обратного вызова, вместо неё эту роль будет исполнять функция, которую возвращает `eatPizza`.

Мы изменили тело функции, и теперь нам требуется обновить вызов `makePizza`. Вторым параметром мы указываем не ссылку на `eatPizza`, а вызов функции, передав информацию о напитке. Получается, на место второго параметра будет передана новая функция, полученная в результате выполнения `eatPizza`.

Если представленный пример вызывает затруднение — обязательно перечитайте теорию замыканий.

Резюме

Колбэк-функции просты в применении и открывают много возможностей, но с большой силой приходит большая ответственность. Легко потеряться во вложенности колбэк-функций и познать все прелести ада функций обратного вызова. Любым подходом нужно пользоваться с осторожностью. Как не попасть в ад обратных вызов, мы поговорим в одной из ближайших статей.

Теория

~ 5 минут

Глава 5.4. Действия браузера по умолчанию

Возможно, вам уже когда-то встречалась в коде такая строка — `evt.preventDefault()`. Например, в интерактивных курсах по JavaScript. Давайте подробно разберём, зачем она нужна.

При разработке таких типичных элементов интерфейса, как форма или попап, часто нужно изменить поведение браузера по умолчанию. Допустим, при клике по ссылке мы хотим, чтобы открывался попап, но вместо этого браузер

будет автоматически переходить по адресу, указанному в атрибуте href. Или вот другая проблема — мы хотим перед отправкой формы проверять корректность введённых данных, но после нажатия на кнопку submit форма каждый раз будет отправляться на сервер, даже если там куча ошибок. Такое поведение браузера нам не подходит, поэтому мы научимся его переопределять.

Объект события и метод preventDefault

Событие — это какое-то действие, произошедшее на странице. Например, клик, нажатие кнопки, движение мыши, отправка формы и так далее. Когда срабатывает событие, браузер создаёт объект события `Event`. Этот объект содержит всю информацию о событии. У него есть свои свойства и методы, с помощью которых можно эту информацию получить и использовать. Один из методов как раз позволяет отменить действие браузера по умолчанию — `preventDefault()`.

`Event` можно передать в функцию-обработчик события и в ней указать инструкции, которые должны быть выполнены, когда оно сработает. При передаче объекта события в обработчик обычно используется сокращённое написание — `evt`.

Пример: когда ссылка — не ссылка

Ранее мы уже говорили о попапе, который должен появляться при клике на ссылку — давайте разберём этот кейс на практике. Так будет выглядеть разметка в упрощённом виде:

```
<a class="click-button" href="pop-up.html">Клик</a>

<div class="content">
  <!-- Здесь содержимое попапа -->
</div>
```

Мы хотим при клике на ссылку `click-button` добавлять элементу с классом `content` класс `show`. Он сделает попап видимым, поменяв значение свойства `display` с none на `block`. Напишем логику добавления этого класса с помощью JavaScript:

```
// Находим на странице кнопку и попап
const button = document.querySelector('.click-button');
const popup = document.querySelector('.content');

// Навешиваем на кнопку обработчик клика
button.onclick = function (evt) {
  // Отменяем переход по ссылке
  evt.preventDefault();

  // Добавляем попапу класс show, делая его видимым
  popup.classList.add('show');
};
```

Если мы уберём строку `evt.preventDefault()`, вместо попапа откроется отдельная страница `pop-up.html`, адрес которой прописан в атрибуте `href` у ссылки. Такая страница нужна, потому что мы хотим, чтобы вся функциональность сайта была доступна, если скрипт по какой-то причине не будет загружен. Именно поэтому мы изначально реализовали кнопку с помощью тега `a`, а не `button`. Но у нас с JavaScript всё в порядке, поэтому вместо отдельной страницы мы открыли попап, отменив действие браузера по умолчанию.

Пример: проверка формы перед отправкой

Разберём следующий кейс — отправку формы при нажатии на кнопку `submit`. Допустим, мы хотим перед отправкой проверять введённые данные, потому что в поле ввода обязательно должно быть значение 'Кекс' и никакое другое. Разметка формы:

```
<form class="form" action="#" method="post">
  <input class="name" type="text" id="name" name="name">
  <label for="name">Введите имя</label>

  <button type="submit">Готово!</button>
</form>
```

При нажатии на кнопку «Готово» сработает событие отправки формы `submit`, и форма отправится вне зависимости от корректности введённого значения, поэтому мы должны перехватить отправку.

```
// Находим на странице форму и инпут
const form = document.querySelector('.form');
const name = document.querySelector('.name');

// Навешиваем на форму обработчик отправки
form.onsubmit = function(evt) {
  // Проверяем введённое значение на соответствие
  if (name.value !== 'Кекс') {
    // Если значение не подходит, отменяем автоматическую отправку формы
    evt.preventDefault();
    // И выводим предупреждение в консоль
    console.log('Вы не Кекс!');
  }
};
```

Здесь мы не дали отправить форму при неверно введённом значении. Но если всё в порядке, условие не выполнится, и форма будет отправлена как обычно.

Неотменяемые события

Не для всех событий можно отменить действие по умолчанию. Например, событие прокручивания страницы `scroll` проигнорирует попытки отменить его. Чтобы узнать, можно отменить действие по умолчанию или нет, нужно обратиться к свойству `cancelable` объекта `Event`. Оно будет равно `true`, если событие можно отменить, и `false` — в обратном случае.

```
document.onscroll = function(evt) {
  // В консоль выведется false
  console.log(evt.cancelable);
  // Отмена не сработает
  evt.preventDefault();
};
```

В статье мы разобрали базовые примеры, когда может понадобиться отмена действия браузера по умолчанию. В реальной разработке вы будете сталкиваться с такой необходимостью довольно часто — при сложной валидации форм, предотвращении ввода пользователем неверных символов, создании самописного меню вместо стандартного (при клике правой кнопкой мыши) и так далее.

Теория

~ 3 минуты

Глава 5.5. Фазы событий

Работать с событиями несложно: достаточно подписаться на нужное событие с помощью `addEventListener` и подготовить функцию обратного вызова с кодом. При наступлении события эта функция выполнится. Рассмотрим небольшой пример:

```
<body>
  <form>
    <button type="submit" />
  </form>
</body>
```

При нажатии на кнопку событие `click` произойдёт почти на всех элементах. Порядок того, на каком элементе оно произойдёт первым, зависит от фазы события. В настоящее время в стандарте закреплено три фазы: захват, целевое событие и всплытие. Фаза целевого события, как правило, не используется.

Фаза захвата

Другие названия: погружение, перехват, capturing. На этой фазе, когда пользователь нажмёт на кнопку, событие произойдёт сначала на `body`, затем на `form`, и только потом оно опустится до `button`.

Отследить это можно, навесив обработчики одного и того же события (например, `click`) на каждый из тегов и добавив к ним вывод `console.log()` или `alert` — код, позволяющий идентифицировать элемент. В этом случае на стадии захвата сначала выведется последовательно результат обработчика на `body`, следом `form`, и в конце `button`.

Рисунок 1. Фаза захвата

Фаза всплытия

Вторая фаза называется всплытие (bubbling). Это та самая фаза, которая используется по умолчанию в addEventListener и в устаревшем способе добавления обработчика через.onclick.

Здесь наступление событий работает с точностью наоборот. Когда пользователь кликает по кнопке, сначала отработает обработчик на самой кнопке, затем на её первом родителе `form` и только потом дойдёт очередь до `body`. После этого (вне примера), событие поднимется до `html`, `document`, а в некоторых случаях и `window`.

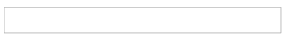

Рисунок 2. Фаза всплытия

Порядок фаз

— Захват

— Событие на элементе (испускание, отправка, dispatch)

— Всплытие (*если возможно*)

Резюме

Важно запомнить: захват произойдёт в любом случае, а всплытие только если возможно. Всплытие возможно для *почти* всех событий, кроме `focus/onfocus`, `blur`, `mouseleave`, `mouseenter`.

Теория

~ 2 минуты

Глава 5.6. Делегирование событий

Делегирование событий — это подход, который сокращает количество однотипных обработчиков в коде.

Разберёмся на примере. Представим, что наша задача звучит так: обрабатывать клик на каждом элементе списка одинаковым обработчиком. Можно пройтись по каждому элементу и добавить обработчики:

```
<ul>
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
</ul>
```

```
// script.js
const onListItemClick = function (evt) {
  // Действия
}
```

Минусы решения

— Для каждого обработчика выделяется место в оперативной памяти. Если элементов станет много, то количество однотипных обработчиков может повлиять на быстродействие сайта

— Если в список добавятся новые элементы, то придётся за ними следить и вешать на них обработчики вручную, что увеличивает количество поддерживающего кода

— Хороший программист следует принципу **DRY** — Don’t repeat yourself, избегает лишних повторений и не плодит одинаковые сущности без надобности

Делегирование

Вместо добавления одинаковых обработчиков на каждый элемент списка, добавим один обработчик на родительском элементе, то есть родителю *делегируется* (поручается) обработка событий его дочерних элементов:

```
<ul> <!-- onListClick() -->
  <li>...</li>
  <li>...</li>
  <li>...</li>
  <li>...</li>
  <li>...</li>
</ul>
```

```
// script.js
```

```
const onListClick = function (evt) {
  if (evt.target.nodeName === 'LI') {
    // Действия
  }
}
```

Как это работает

1. В момент клика на элементе `li`, ссылка на него записывается в свойство target интерфейса Event
2. Начинается стадия всплытия
3. Всплытие доходит и срабатывает на `ul`. Обработчик проверяет, является ли `evt.target` элементом списка и выполняет код.

Резюме

Используя делегирование событий, мы избавились от однотипных обработчиков и сделали нашу вёрстку легко масштабируемой: можем добавить сколько угодно дополнительных элементов списка и ничего не сломается.

Теория

~ 5 минут

Глава 5.7. Пропуск кадров и устранение дребезга

Каждый разработчик желает, чтобы его приложение работало быстро, поэтому старается его оптимизировать. Подходов к оптимизации приложений множество: можно упрощать и оптимизировать код, кэшировать данные, уменьшать вес ресурсов — и так далее. В этой главе мы разберём только два способа оптимизации кода, а точнее его работы. Оба эти подхода основаны на сокращении вызовов функции.

Устранение дребезга (debounce)

Подход, смысл которого заключается в выполнении последнего вызова функции из череды вызовов. Последний вызов определяется по указанной задержке между вызовами: если задержка больше, чем пауза между вызовами, значит тот, что был до паузы — последний. Таким образом мы исключаем все предыдущие, ненужные вызовы, а значит сокращаем трудозатраты нашей программы.

Серые полоски — череда вызовов, красные — реальные вызовы

«Устранение дребезга» отлично подходит для обработки пользовательских событий (например, ввод текста или клики), реакция на которые очень дорога (например, запрос на сервер или сложный расчёт).

> «Дребезг» или «Мигание» — эффект, связанный с мгновенной реакцией на пользовательские события, что приводит к миганию элементов интерфейса из-за частой перерисовки

Представим ситуацию, что вы разрабатываете поисковую строку. Требуется, чтобы при вводе в строку показывались подходящие подсказки (умное автодополнение):

Подсказки в Яндексе

Список подсказок нужно получать с сервера.

Давайте рассуждать, что произойдёт внутри программы? При вводе каждого символа «под капотом» выполнится запрос на сервер. Сервер, получив запрос, составит список подходящих подсказок и вернёт их обратно. И так на каждый символ!

А что, если пользователь — магистр слепой печати 10-пальцевым методом, и вводит символы очень быстро? Вы получите кучу лишних запросов на сервер, и когда пользователь уже введёт слово «погода», в худшем случае вы покажете подсказки только для «по» или «пог», а в лучшем уже для «погода», но частая перерисовка списка подсказок будет «дребезжать».

Идеальная ситуация для «устранения дребезга». Всё, что нужно сделать, это отображать подсказки, если пользователь закончил ввод. Или сделал паузу дольше указанной нами задержки. Тогда запросы на сервер будут уходить реже, а подсказки станут точнее. Profit!

Пропуск кадров (throttling)

Подход, смысл которого заключается в выполнении вызова функции из череды вызовов не чаще, чем 1 раз в указанный интервал времени: если мы укажем интервал 5 секунд, сколько бы вызовов функции за это время не произошло, сработает только один. Таким образом мы исключаем промежуточные, ненужные вызовы, а значит также сокращаем трудозатраты нашей программы.

Серые полоски — череда вызовов, красные — реальные вызовы

«Пропуск кадров» помогает в случае, когда «устранение дребезга» не подходит. То есть когда нужно совершить не только вызов по факту окончания череды действий, но и редкие промежуточные вызовы.

> Пропуск кадров — это пропуск именно лишних кадров, тормозящих производительность, но не влияющие на плавность отрисовки. Иными словами *частота кадров снижается, но остаётся постоянной*

Представим ситуацию, что вы разрабатываете индикатор скролла, чтобы пользователь понимал, какую часть текста он уже прочёл:

Пример индикатора скролла от w3schools

Если изменять ширину индикатора на каждое событие скролла, а оно происходит чаще чем 1 раз в секунду, мы получим излишнюю точность, которую обычным глазом пользователь не заметит, а вот трудозатраты на перерисовку ширины будут большие.

Идеальная ситуация для «пропуска кадров». Всё, что нужно сделать, это изменять ширину не чаще одного раза в N секунд, пока пользователь скроллит. Тогда перерисовка будет происходить реже, а визуально всё так и останется плавным. Profit!

Резюме

При разработке фронтенда вам постоянно придётся сталкиваться с оптимизацией при помощи пропуска кадров или устранения дребезга. Но! Помните, что не стоит искать изначально супер оптимизированное решение, самое главное — решить задачу, а уже после оптимизировать работающий код *при необходимости*.